規格準拠性に優れたCプリプロセッサMCPP

松井 潔†

長い歴史を持つCのプリプロセス仕様には多くの混乱があった．C90以降は各処理系の仕様が規格を中心に収束してきているが，「規格準拠」をうたう処理系が間違った動作をすることがいまだに稀ならず見られる．コンパイラ本体と比べると，プリプロセッサはかなり未成熟な分野である．MCPPはfreeでかつportableなCプリプロセッサであり，C90，C99，C++98のほぼすべての仕様を正確に実装している．そして，C/C++プリプロセッサの体系的なテストと評価をする検証セットが付属していて，それによって検証ずみであることが特徴である．本稿では，多くの優れた特徴を持つMCPPについて規格準拠性を中心に紹介し，さらにCプリプロセスとその規格の歴史的背景と将来展望を論考する．

MCPP: A C Preprocessor with the Highest Conformance

Kiyoshi Matsui†

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. MCPP is a free portable C preprocessor and implements precisely almost all the specifications of C90, C99 and C++98. It provides a validation suite to make systematic tests and evaluation of C/C++ preprocessors. Its behaviors have been completely verified using the validation suite. In this article, I present MCPP focusing on the conformance among it's many superior features. In addition, I discuss the historical background and the future perspective of C preprocessing and it's Standard.

1. はじめに

長い歴史を持つCのプリプロセス仕様には多くの混乱があった．C90（C89）[1)~4)]以降は各処理系の仕様が規格を中心に収束してきているが，「規格準拠」をうたう処理系が間違った動作をすることがいまだに稀ならず見られる．コンパイラ本体と比べると，プリプロセッサはかなり未成熟な分野だといわざるをえない．

この状況の背景となっているのは，C90以前のCプリプロセスの仕様がはなはだあいまいなものだったことである．C90でCのプリプロセス仕様は，その原則にまでさかのぼって初めて全面的に定義されたのであったが，しかし，C90にも歴史的な負の遺産を清算しきれなかった部分があり，それらの問題はC99[5),6)]でも解決されなかったのである．さらに現実の処理系は原則をあいまいにしたまま個々の仕様を接ぎ木することによって，問題を長く引きずってきていると考えられる．

こうした事情を背景として，Cで書かれたソースプログラムには，portabilityを欠いた処理系依存のコード等，プリプロセスレベルの問題を持つものが少なくない．Cにプリプロセスというフェーズがあるのはportabilityの確保が大きな目的の1つであるが，プリプロセスが逆にportabilityを損なう結果も生んできてしまっているのである．

筆者は久しく以前からCプリプロセッサを開発してきた．その成果はすでに1998/08にcppV.2.0として，1998/11にcppV.2.2として公開していたが，情報処理振興事業協会（IPA）の平成14年度と平成15年度の「未踏ソフトウェア創造事業」に採択され，そのプロジェクトの成果としてV.2.3，V.2.4が公開された[7)]．このcppを他のcppと区別するためにMCPPと呼ぶ．Matsui CPPの意味である．

MCPPは最も規格準拠性の高いCプリプロセッサである．筆者が勝手にそう言っているだけではなく，「プリプロセス検証セット」を並行して作成して，それによって検証ずみであるところが特徴である．

本稿は多くの優れた特徴を持つMCPPについて，その概要を紹介したうえで，規格準拠性とその検証方

† 陽和病院相談室
Counseling Room, Yohwa Hospital

法を述べ，さらに C プリプロセスとその規格の歴史的背景と将来展望を論考するものである．

本稿は次のように構成されている．

2 章：C/C++プリプロセスの基礎的規定に触れる．

3 章：MCPP の概要を紹介する．

4 章：プリプロセス検証セットを紹介する．

5 章：検証セットによる各種プリプロセッサの検証結果を述べる．

6 章：プリプロセッサのバグの例を示す．

7 章：MCPP によるソースチェックの例を示す．

8 章：C プリプロセスの歴史的背景を概観し，将来を展望する．

9 章：まとめ．

2. プリプロセスの基礎的規定

本論に入る前に，C/C++プリプロセスの基礎的な規定に少し触れておく．

2.1 プリプロセスの手順

プリプロセスの手順は $K\&R^{1st}$ ではまったく記載されていなかったために，多くの混乱の元となっていた．C90 では次のような translation phases というものが規定されて，これが明確にされた．

(1) ソースファイルの文字を必要ならソース文字セットに map する．Trigraph の置換をする．

(2) `<backslash><newline>` の sequence を削除する．それによって物理行を接続して論理行にする．

(3) Preprocessing token と white spaces とに分割する．コメントは one space character に置換する．改行コードは保持する．

(4) 各種の preprocessing directive を実行し，マクロ呼び出しを展開する．#include directive があれば，指定されたファイルについて phase (1) から phase (4) を再帰的に処理する．

(5) ソース文字セットから実行時文字セットへの変換をする．同様に，文字定数中と文字列リテラル中の escape sequence を変換する．

(6) 隣接する文字列リテラルを連結する．

(7) Preprocessing token を token に変換し，コンパイルする．

(8) リンクする．

その後，C99 では phase (4) に `_Pragma()` operator の処理が付け加えられた．そのほか若干の字句が追加されたり修正されたりしたが，大筋は変わっていない．

C++98[8] では，phase (7) と (8) の間に instantiation 処理の phase が挿入されたほか，phase (1) で basic source character set に含まれない文字は universal character name（UCN）というものに変換し，phase (5) でこれを実行時文字セットに再変換するという規定が追加された．

これらの translation phases のうち，通常は phase (4) までをプリプロセスと呼んでいる．

2.2 診断メッセージとドキュメント

診断メッセージとドキュメントに関する規定は C90, C99，C++98 ですべて同じであり，次のように規定されている．

Violation of syntax rule（構文規則違反）または violation of constraint（制約違反）を含む translation unit に対しては，処理系は診断メッセージを出さなければならない．診断メッセージの出し方は implementation-defined（処理系定義）である．

処理系定義の事項についてはすべて，処理系はその仕様をドキュメントに記載しなければならない．

3. MCPP の概要

MCPP は次のような特徴を持っている．

(1) 規格準拠性がきわめて高い．C，C++のプリプロセスの reference model となるものを目指して作ってある．C90 はもちろんのこと，C99, C++98 に対応する実行時オプションも持っている．

(2) C，C++プリプロセッサそのものの詳細かつ網羅的なテストと評価をする検証セットが付属している．

(3) 診断メッセージが豊富で親切である．診断メッセージは百数十種に及び，問題点を具体的に指摘する．それらは数種のクラスに分けられており，実行時オプションでコントロールすることができる．診断メッセージの約 2/3 は規格で要求されているものであるが，そのほかにも portability の問題等に関する多くの診断メッセージが用意されている．

(4) デバッグ用の情報を出力する各種の#pragma ディレクティブを持っている．

(5) Multi-byte character の処理は日本・中国・台湾・韓国の各種 encoding，および UTF-8 という多様な encoding に対応している．

(6) 速度も遅いほうではないので，デバッグ時だけでなく日常的に使うことができる．

(7) Portable なソースである．MCPP をコンパイルするときに，ヘッダファイルにある設定を書

き換えることで，処理系付属のプリプロセッサに代替して使えるプリプロセッサが生成されるようになっている．C90，C99，C++98 のどれに準拠する処理系でもコンパイルでき，C90 以前のいわゆる $K\&R^{1st}$ の処理系でさえもコンパイルできる広い portability を持っている．

(8) Standard モード（C90，C99，C++98 対応）のプリプロセッサのほか，$K\&R^{1st}$ の仕様やいわゆる Reiser モデルのもの等，各種仕様のプリプロセッサを生成することができる．規格そのものの問題点を筆者が整理した自称 post-Standard モードまである．

(9) UNIX 系のシステムでは configure スクリプトによって MCPP の実行プログラムを自動生成することができる．GNU C の testsuite がインストールされていれば，'make check' によって検証セットのうちの動作テストの testcase の大半を自動実行することもできる．

(10) オープンソースである．

(11) 詳細なドキュメントが付属している．次の文書についてそれぞれ日本語版と英語版が用意されている．

- (a) README： configure と make の方法を説明.
- (b) mcpp-summary.pdf： サマリ文書.
- (c) manual.txt： 実行プログラム用マニュアル．使い方，仕様，診断メッセージの意味．ソースの書き方も示唆.
- (d) porting.txt： 実装用ドキュメント．任意の処理系に実装する方法.
- (e) cpp-test.txt： 検証セット解説．規格の解説を兼ねる．規格そのものの矛盾点も指摘し，代案を提示している．検証セットをいくつかの処理系に適用した結果を報告している．

4. プリプロセス検証セット

プリプロセッサの開発と同時にもう 1 つ問題となるのは，プリプロセッサの動作や品質の検証である．処理系が誤動作したり品質が悪かったりするのでは論外であるが，実際にテストしてみると，かなりの問題が見つかるものである．処理系の開発には検証システムの整備が不可欠である．しかし，多くの処理系ではランダムなサンプルによるテストしか行っていないと見られる．

他方で，プリプロセッサでアルゴリズムの最も難しいのはマクロ展開であるが，マクロ展開のバグというのは比較的少ないものである．これは規格書にマクロ展開の examples が掲載されているからだと考えられる．各処理系はこれをクリアしたうえで，「規格準拠」と称する版をリリースしていると推測される．しかしながら，規格書にあるサンプルはこの数個のマクロだけであり，他のプリプロセス仕様についてはサンプルは掲載されていない．

表 1 検証セット V.1.4.2 の項目数と配点
Table 1 Number of test items and scores covered by Validation Suite V.1.4.2.

			項目数	最高点
規格準拠度	K&R		31	166
	C90	N	73	284
		I	2	4
		E	50	112
		D	13	32
	C99		20	98
	C++98		9	26
品質	診断メッセージ		47	74
	その他		20	171
計			265	967

筆者は MCPP 開発の一環として，プリプロセス検証セットを作成し，MCPP とともに公開している．これはきわめて多面的な評価項目を持ち，プリプロセッサのできるだけ客観的で網羅的なテストをするものであり，次のような方針で作成されている．

(1) 規格にあるプリプロセス仕様をすべて網羅する．

(2) 規格にない事項でも，バグや portability のチェックに有効だと考えられるものは網羅する．

(3) テスト用サンプル（testcase）は原則としてできるだけシンプルなものとする．テストしようとする事項はできるだけ小さく分割する．それによってテストの客観性を高める．

(4) プリプロセス後の出力を直接見て判定することを原則とし，コンパイルと実行によるテストを副次的に使う．

どの処理系にも使える testcase を書くのはときに困難であるが，1 つの事項について複数の testcase を用意する等の方法で，ほとんどの場合は対処できている．

一方，テストの自動実行には多くの制約があり，実用的で portable な方法は存在しない．しかし，GNU C と MCPP に関しては，動作テストの大半の testcase について，GNU C の testsuite の一部として自動テストのできる版を用意している．

検証セット V.1.4.2（未公開版）は **表 1** のようにテスト項目が 265 に及んでいる．うち動作テストが 230 項目，ドキュメントや品質の評価が 35 項目を占めてい

表 2 テストしたプリプロセッサ
Table 2 Preprocessors tested.

No.	OS	処理系	プリプロセッサ（版数）	年/月
1	Linux		DECUS cpp	1985/01
2	MS-DOS		JRCPPCHK（V.1.00B）	1990/03
3	WIN32	Borland C++ V.4.02J	cpp32	1994/12
4	DJGPP V.1.12 M4	GNU C 2.7.1	cpp	1995/12
5	MS-DOS	LSI C-86 V.3.30c	cpp（改造版 beta13）	1996/02
6	FreeBSD DJGPP 1.12 WIN32 MS-DOS	GNU C 2.7.2 GNU C 2.7.1 Borland C 4.0 LSI C-86 3.3	MCPP（V.2.0）	1998/08
7	WIN32	Borland C++ V.5.5	cpp32	2000/08
8	Linux, FreeBSD	GNU C 2.95.3	cpp0	2001/03
9	Linux, FreeBSD	GNU C 3.2R	cpp0	2002/08
10	Linux, etc		ucpp（V.1.3）	2003/01
11	WIN32	Visual C++ .net 2003	cl	2003/04
12	WIN32	LCC-Win32 V.3.2	lcc	2003/08
13	Linux, FreeBSD CygWIN WIN32 WIN32 WIN32	GNU C 2.95.3, 3.2 GNU C 2.95.3 Visual C++ .net 2003 Borland C 5.5 LCC-Win32 3.2	MCPP（V.2.4.1）	2004/03

る．各項目はウェイトを付けて配点されている．各項目とも最低点はすべて 0 点である．「規格準拠度」には診断メッセージとドキュメントの評価も含まれる．「規格準拠度」のうち「K&R」というのは，$K\&R^{1st}$ と C90 との共通の仕様に関するものである．C99，C++98 の「規格準拠度」というのは，C90 にない新たな規定に関するものである．「C90 規格準拠度」の N，I，E，D というのはそれぞれ次のような意味である．

- N：（normal）正しいソースを正しく処理する．
- I：（implementation-defined）処理系定義の文字定数を正しく処理する．
- E：（erroneous）構文規則または制約に違反するソースに適切な診断メッセージを出す．
- D：（document）ドキュメントに処理系定義事項について必要な記載がある．

また，「品質：診断メッセージ」というのは，規格では要求されていない portability 等に関する診断メッセージの評価である．

「品質：その他」というのは，実行時オプション・#pragma・multi-byte character encoding への対応・速度等々の各種品質の評価である．これらの項目の中には評価に主観の入るものが多い．しかし，これらの項目は実際の使い勝手を大きく左右するものであり，プリプロセッサの総合的な評価のためには重要な事項であるので，あえて検証セットに含めている．

この検証セットは多面的な評価をするのが 1 つの特徴であるが，異質な評価項目にウェイトを付けた配点をして 1 本の物差しのように表示する方法に無理があるのは確かである．にもかかわらずウェイトを付けた配点をしているのは，ユーザにとっての使用感を重視するからである．

そして，できるだけ客観性を高めるために，評価項目を小さく分割し，採点基準を明示している．さらに，評価に主観の入りやすい項目には抑制的に配点しているが，そのためウェイトの付け方が中途半端になってしまったきらいもある．

5. 検証セットによる各種プリプロセッサの検証

検証セット V.1.4.2 を**表 2** のようないくつかの処理系に適用してみた．これらの処理系は次のようなものである．処理系は古い順に並べてある．

---

*1 DECUS cpp： Martin Minow による DECUS cpp のオリジナル版[9] を筆者が若干の修正を加えて Linux/GNU C 3.2 でコンパイルしたもの．

*2 JRCPPCHK： Roskind による UNIX，OS/2，MS-DOS 用 shareware である JRCPP の MS-DOS-OS/2 用試用版．具体的な処理系には対応しない stand-alone のプリプロセッサ[10]．

*3 Borland C 4.0： 1993 年のものの日本語版[11]．

*4 DJGPP 1.12： GNU C 2.7.1/cpp を Delorie が DOS extender である GO32 に移植したもの．日本語版への移植で shift-JIS に対応[12]．

表 3 各種プリプロセッサの検証結果
Table 3 Validation results of each preprocessor.

No.	処理系略称	規格準拠度								品質		総合評価
		K&R	C90				C99	C++	計	診断	その	
			N	I	E	D		98		msg	他	
1	DECUS cpp	150	162	2	74	4	0	0	392	15	76	483
2	JRCPPCHK	160	266	2	92	22	16	4	562	31	60	653
3	Borland C 4.0	166	246	4	90	24	20	6	556	13	71	640
4	DJGPP 1.12	166	282	2	106	12	30	6	604	23	114	741
5	LSI C-86 b13	160	248	2	64	8	14	4	500	9	84	593
6	MCPP 2.0	166	284	4	112	32	46	8	652	65	130	847
7	Borland C 5.5	166	250	4	92	18	20	6	556	18	73	647
8	GNU C 2.95.3	166	284	4	104	12	56	6	632	23	111	766
9	GNU C 3.2	166	284	4	108	24	86	20	692	35	115	842
10	ucpp 1.3	166	276	2	98	6	88	9	645	25	88	758
11	Visual C 2003	156	272	4	100	18	43	15	608	20	85	713
12	LCC-Win32 3.2	160	270	2	92	10	18	6	558	18	96	672
13	MCPP 2.4.1	166	284	4	112	32	98	22	718	74	151	943
	最高点	166	284	4	112	32	98	26	722	74	171	967

*5 LSI C-86 cpp b13： LSI C-86/cpp の，きだあきらによる改造版[13]．

*6 MCPP 2.0： 筆者による free software の V.2.0．DECUS cpp をベースとして書き直したもの．FreeBSD/GNU C 2.7，DJGPP V.1.12，WIN32/Borland C 4.0，MS-DOS/Turbo C 2.0，LSI C-86 3.3 等に対応．各種の動作モードのプリプロセッサを生成することができるが，このテストでは 32 ビットシステムでの標準版を使用．

*7 Borland C 5.5： 日本語版[14]．

*8 GNU C 2.95.3： VineLinux 2.6，FreeBSD 4.4，CygWIN 1.3.10 で使われているもの[15]．

*9 GNU C 3.2： ソースから筆者が VineLinux 2.6，FreeBSD 4.7 上でコンパイルしたもの[15]．

*10 ucpp 1.3： Pornin による portable な free software．Stand-alone のプリプロセッサ[16]．

*11 Visual C 2003： Microsoft/Visual C++ .net 2003[17]．

*12 LCC-Win32 3.2： Navia による shareware．ソース付き．プリプロセス部分のソースは Dennis Ritchie が C90 対応のプリプロセッサとして書いたもの[18]．

*13 MCPP V.2.4.1： V.2.0 以降，Linux/GNU C（2.95.3，3.2），FreeBSD/GNU C（2.95.4，3.2），CygWin 1.3.10，LCC-Win32 3.2，Borland C 5.5，Visual C++ .net 2003，Plan 9 ed.4/pcc 等への対応が追加されている[19]．

これらの処理系の検証結果を **表 3** および **図 1** に示す．

MCPP はずば抜けた成績である．V.2.4.1 では，規格準拠性は C++98 の extended character -> UCN の変換という奇妙な仕様を実装していない以外はすべて正しく実装している．診断メッセージの豊富さと的確さ，ドキュメントの網羅性，豊富な実行時オプションと#pragma，多様な multi-byte character encoding への対応，portability 等々の規格外の品質では他の処理系との差がさらに大きい．

MCPP の次に優れているのは，このリストでは GNU C/cpp である．この cpp は C90 規格に合致した正しいソースを処理する分には，ほとんど問題がない．しかし，C99，C++98 の新しい仕様のいくつかが未実装であることは時間とともに解決されてゆくとしても，それ以外にも次のような問題がある．

(1) 診断メッセージが不十分である．-pedantic -Wall オプションを指定することで多くの問題はチェックできるが，それでもまだ不足している．ことに規格で要求されていない事項に関する診断メッセージはごく少ない．
(2) ドキュメントが少なく，仕様の不明確な部分や隠れ仕様が多い．ことに V.2 では -traditional オプションを指定しなくても traditional な仕様が隠れていて，それを期待するソースが黙って通ってしまうことが問題である．
(3) 規格と矛盾する独自仕様が多い（拡張仕様は #pragma で実装すべきものである）．

GNU C V.3/cpp0 は GNU C V.2/cpp0 に比べるとこれらの点で大幅に改善されたが，まだ不十分である．

MCPP が GNU C/cpp に負けるのは速度くらいで

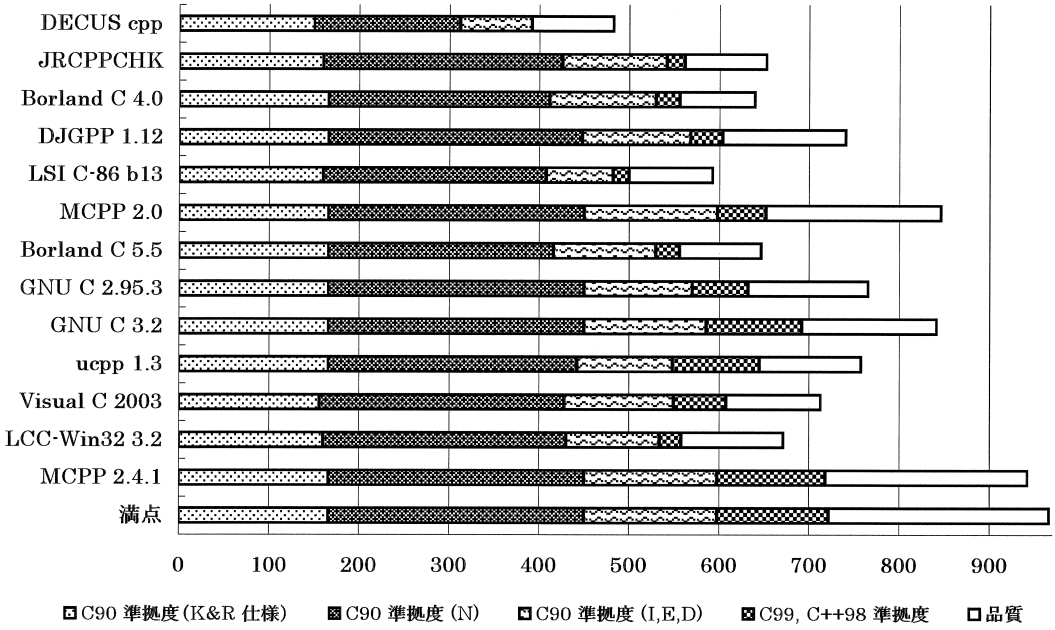

図 1 各種プリプロセッサの検証結果
Fig. 1 Validation results of each preprocessor.

ある．

他のプリプロセッサにはさらに問題が多い．多くのプリプロセッサに共通して見られる傾向としては，次のようなものがある．

(1) C99，C++98 の仕様はまだ半分以下しか実装していないものが多い．
(2) 診断メッセージとドキュメントの不足している処理系が多い．
(3) 規格外事項に関する診断メッセージはほとんど出さない処理系が大半である．
(4) 診断メッセージの質の悪いものも目立つ．
(5) Multi-byte character は 1，2 種の encoding にしか対応していない処理系が大半である．

そのほか，処理系ごとに意外なバグが 1 つや 2 つはたいてい発見される．

6. プリプロセッサのバグと誤仕様の例

検証セットまたは MCPP によって検出された，各種プリプロセッサのバグないし誤仕様のほんの一部の例を**図 2** に示す．

なお，この 6 章と次の 7 章では，対応する検証セットの testcase の番号を記しておく．番号の先頭の 1 文字はそれぞれ次のような分類を意味している．

(1) n：(normal) 正しいソースを正しく処理するか．
(2) e：(erroneous) 構文規則または制約に違反するソースに適切な診断メッセージを出すか．
(3) u：(undefined) Undefined behavior を引き起こすソースに適切な診断メッセージを出すか．

6.1 コメントを生成するマクロ

(testcase 番号：u.concat)

example-1 は Visual C++.net 2003 のシステムヘッダに実際に出てくるマクロ定義であり，example-2 のように使われている．これは `_VARIANT_BOOL` が `//` に展開されて，その結果，この行がコメントアウトされることを期待しているものである．そして，実際に Visual C の cl.exe ではそういう結果になる．

しかし，`//` はトークン (preprocessing-token) ではない．また，マクロの定義や展開は，ソースがトークンに分割されコメントが 1 個のスペースに変換されたあとのフェーズで処理されるものであり，したがってコメントを生成するマクロ等というものはありえないのである．このマクロは `//` に展開されたところで，`//` は有効な preprocessing-token ではないので結果は undefined となるはずのものである．

このマクロは論外であるが，それ以上に問題なのは，これをコメントとして処理してしまう Visual C/cl.exe のプリプロセスの実装である．この例には，このプリプロセッサの次のような深刻な問題が露呈している．

(1) トークンベースではなく文字ベースの処理がされている．
(2) プリプロセスの手順 (translation phases) が恣意的であり，論理的な一貫性がない．

```
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
```

図 2 プリプロセッサのバグの例
Fig. 2 Examples of preprocessor bug.

6.2 スキップされるはずの式でエラーになるもの

（testcase 番号：n.13.7）

example-3 と example-4 はいずれも正しい式である．分母が 0 でない場合だけ割り算を実行するように用心深く書かれた式である．ところが，MACRO_0 が 0 であるにもかかわらず割り算を実行してエラーになる処理系がある．example-3 はかつてはエラーになる処理系はよくあったが，いまは見かけなくなっている．しかし，example-4 は Visual C++ではエラーになってしまう．式の評価に関する C の基本的な仕様が正しく実装されていないのである．

一方，Borland C 5.5 では example-3 でも example-4 でもウォーニングが出る．これ自体は必ずしも間違っているとはいえないが，しかし，この処理系は example-5 のような正真正銘の 0 除算でもまったく同じウォーニングを出す．すなわち，正しいソースと間違ったソースとの区別がつかない．Turbo C では本当の 0 除算でも正しい式でもどちらも同じエラーになっていたが，Borland C はその診断メッセージをウォーニングに格下げしただけなのである．規格に違反しているとはいえないが，間に合わせ的な診断メッセージであり品質は良くない．

6.3 オーバフローが診断されないもの

（testcase 番号：e.14.10）

example-6 は C90 ではオーバフローの発生する定数式であるが，これには何の診断メッセージも出さない処理系が大半である．Borland C と ucpp だけがこの種のソースに対しては場合によってウォーニングを出すが，出さない場合も多く，一貫していない．

7. MCPP によるソースチェックの例

次に，MCPP による現実のソースコードのチェックを，glibc 等を例に見ることにする．

C で書かれたソースプログラムには，プリプロセスのレベルでの問題を持っているものが少なくない．特定の処理系でコンパイルできることをもって良しとしてしまっているものの portability を欠いているもの，不必要にトリッキーな書き方をしているもの，C90 以前の特定の処理系の仕様をいまだにあてにしているもの，等々である．こうしたソースの書き方は portability と readability そしてメンテナンス性を損なうものであり，悪くすればバグの温床ともなりかねない．そうしたソースをより portable で明快な形で書き直すことは，多くの場合，簡単なことなのであるが，見過ごされている場合も多い．

そうしたソースが多く存在する背景となっているのは，1 つには C90 以前のプリプロセス仕様がはなはだあいまいだったことである．これが，C99 が決まった今となっても尾を引いている．もう 1 つは，既存のプリプロセッサが寡黙すぎることである．プリプロセッサが怪しげなソースを黙って通すために，問題が見過ごされてしまうのである．

7.1 プリプロセッサによるソースチェックの影響力

MCPP を処理系付属のプリプロセッサと置き換えて使うことで，ソースプログラムのプリプロセス上の問題点を，潜在的なバグや規格違反から portability の問題まで，ほぼすべて洗い出すことができる．

これを FreeBSD 2.2.2R（1997/05）の kernel および libc ソースに適用した結果は，MCPP V.2.0 以来，そのマニュアル文書 manual.txt の 3.9 節で報告している．Libc にはまったくといってよいほど問題がなかったが，kernel には全体からみればごく一部のソースではあるものの，いくつかの問題が発見された．問題のソースの多くは，4.4BSD-lite にあったものではなく，FreeBSD への実装と拡張の過程で新しく書かれたものであった．

その後，当時開発中であった MCPP V.2.3 を Linux の glibc 2.1.3（2000/09）に適用してみたところ，こちらにも少なからぬ問題点のあることが分かった．これらの問題には，UNIX 系システムに古くから存在するいわゆる traditional なプリプロセス仕様を使ったものと，GNU C/cpp の独自の仕様や undocumented な仕様を前提としたものが多い．GNU C/cpp がこれらを，少なくともデフォルトの設定では黙って通してしまうことが，こうした感心しない書法のソースを温存させ，そればかりか新たに生み出す結果になっていると考えられる．こうした書法は古いソースに多いとは限らず，むしろ新しいソースにときどき見られるこ

```
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)
```

図 3 glibc のソースの例
Fig. 3 Examples of glibc source.

とが問題である．システムのヘッダファイルにさえも，ときに問題がある．

7.2 glibc のソースを例に

VineLinux 2.1（i386）で使われている glibc 2.1.3 のソースを例にとって，プリプロセス上の問題点の一端を見てみたい．サンプルを**図 3** に示す．

7.2.1 行をまたぐ文字列リテラル

（testcase 番号：u.1.9）

example-7 のようなものである．この traditional な仕様を使う必要はないはずであるが，いまだに使われている．Makefile によって生成されるものもある．この例はプリプロセスディレクティブ行なので行接続が必要であり，それを使って example-8 のように書くことができる．ディレクティブ行に限らない一般性のある書き方は，example-9 のように文字列リテラルの連結の機能を使うものである．ディレクティブ行でなければ，行接続はもちろん不要である．

他のソースファイルではこの形になっているものが多いのであるが，なぜか古い書き方も残っている．

7.2.2 プリプロセスを要する *.S ファイル

（testcase は特に用意していない）

アセンブラソースの中に`#if` 等のプリプロセスディレクティブや C のコメントが挟まっているものである．アセンブラは C とは構文が異なるので，これを C プリプロセッサで処理するのは危険がある．さらに `#APP`，`#NO_APP` といったアセンブラ用の行まで混じっているものもあるが，これは無効なプリプロセスディレクティブと構文上，区別がつかない．

example-9 のように，できるだけ asm() 関数を使って，アセンブラソースの部分を文字列リテラルに埋め込み，∗.S ではなく ∗.c ファイルとするのが良いと思われる．この形であれば，文字列リテラルの行が並んでいる途中にディレクティブ行を（#include 以外なら）挟んでも問題ない．

アセンブラソースの中にマクロが埋め込まれているものもあるが，これは asm() では対処できない．この種のソースは C のソースではなく，本来はアセンブラ用のマクロプロセッサを使うべきものであろう．C プリプロセッサはそのために流用されているのであり，好ましいことではない．

7.2.3 ‘defined’ に展開されるマクロ

（testcase 番号：u.1.19）

example-10 のようなマクロ定義があり，example-11 のように使われている．しかし，#if 式中でマクロ展開の結果に `defined` というトークンが出てくるのは，規格では undefined である．そのことは別としても，このマクロはまず example-12 のように置換され，`__linux__` が 1 に定義されていて `__arm__` が定義されていない場合，最終的な展開結果は普通は example-13 のようになる．`defined(1)` というのは #if 式としては，もちろん syntax error である．

実際，GNU C/cpp でも #if 行でなければこうなるが，ところが #if 行では example-12 で展開をやめてしまって，これを #if 式として評価するのである．一貫しない仕様であり，この書き方には portability がない．このマクロは example-14 のように書けば問題ない．

7.2.4 関数型マクロとして展開されるオブジェクト型マクロ

（testcase は特に用意していない）

この例は規格準拠性の問題ではない．しかし，C プリプロセスの歴史を考えるには示唆的な例であるので，ここでとりあげておく．

展開すると関数型マクロ（function-like macro）の名前になるオブジェクト型マクロ（object-like macro）の定義がときどきある．このマクロの呼び出しは後続するトークン列を取り込んで，関数型マクロとして展開されることになる．マクロ展開のこの仕様は C90 以前からの伝統的なものであり，C90 でも公認されたものである．その意味では portability が高いともいえる．example-15 のようなオブジェクト型マクロの定義があり，`SWAPU16` の定義を見ると example-16 のようになっているというものである．

しかし，オブジェクト型マクロと見えて実は関数型マクロとして展開されるマクロというのは，少なくとも readability が悪い．そういう書き方をするメリットもないと思われる．この書き方の背景にあるのは，エディタによる一括置換の発想であり，C の関数型マクロの書き方としては感心しない．これは初めから関数型マクロとして example-17 のように書いたほうが良い．

7.2.5 Undocumented な環境変数の仕様

これは C のソースではなく Makefile の問題であるが，`SUNPRO_DEPENDENCIES` なる環境変数が定義されていると，-dM オプションの出力先がその定義されたファイル名になるという GNU C 2/cpp では undocumented な仕様を使うものがある．`DEPENDENCIES_OUTPUT` という類似の環境変数もあり，こちらはドキュメントにあるが，どちらも必要性は疑問である．

以上のほかにもいくつかの問題点がある．その多くは，より明快な形で書くことが簡単にできるものである．Glibc の数千本のソースファイルから見れば問題のソースはごく一部であるが，GNU C/cpp がウォーニングを出していれば，こうしたソースは書き直されるか，あるいは初めから別の書き方になっていたと思われるのである．

8. C プリプロセスの歴史的背景と展望

ところで,「規格準拠性」が高いということは処理系としてはいわば当然のことである．しかし，C90 のような 10 数年を経て広く受け入れられている規格に対する準拠性さえも十分に満たすプリプロセッサがいまだにほとんど存在しないという事実は，C プリプロセスというものの特殊な歴史的状況を表している．

MCPP とその検証セットで洗い出されるプリプロセス上の多くの問題の底流にあるのは，C プリプロセスの原則に関する混乱である．C90 以前は C プリプロセスの原則や仕様はきわめてあいまいなものであった．C90 によって C プリプロセスは初めて全面的に定義されたのであり，そこでは「C プリプロセスとは何か」という原則にまでさかのぼって検討されたうえで仕様が規定されたのであった．しかし，各処理系はこの仕様を消化するのにひどく手間どっている．古いプリプロセッサのソースを元に，原則を明確にしないままバージョンアップを繰り返してきているからではないかと推測される．そのうえ，C90 自体にも歴史的な背景から来る中途半端な規定や矛盾がいくつか存在し，C99 でも改められていないことが，問題をいっそう複雑にしている．

C90 のプリプロセス規定から，次のような原則を抽出することができるであろう．

(1) 「トークンベース」の処理を原則とする．
(2) 引数付きマクロの展開は関数呼び出しをモデルとして文法が整理されている．
(3) マクロの定義と展開は多くのプリプロセスディレクティブの処理のうちの 1 つであり，他の処理に優先するものではない．
(4) 実行時環境からは独立した文字どおりの「プリ」プロセスであり，処理系依存の部分は比較的少ない．

これらの諸点について，以下に見てゆく．

8.1 「文字ベース」の処理から「トークンベース」の処理へ

C のプリプロセスは「トークンベース」を原則とするものであるが，C90 以前にはあいまいであったために，文字ベースのテキスト処理の発想が入り込んでいた．中にはいわゆる Reiser 型 cpp のように，文字ベースの処理を原則とするプリプロセッサさえ存在した．C90 以降もそうした処理を期待するソースをプリプロセッサが看過していたり，プリプロセッサ自身に文字ベースの処理が混入していたりすることによって，問題が長く続いてきているのである．さらに C90 自体にも，#演算子の規定や header-name トークンの規定に文字ベースのなごりが残っている（この問題については cpp-text.txt の 1.7 節で詳細に検討している）．

既存のプリプロセッサでは，トークンベースの処理を意図しながらも，そこに文字ベースの処理が紛れ込んでしまっていることが多いようである．プリプロセッサのバグの何割かはこれによるものだと思われる．

たとえば Borland C 4.0，5.5/cpp32 では，マクロ展開によって生成されたトークンが前後のトークンとくっついて 1 つになってしまうことがある．これは中途半端なトークン処理の例である．また，GNU C 2/cpp を含めて，マクロ展開によって illegal なトークンが生成されても，何のウォーニングも出さないプリプロセッサは多い．これはプリプロセスの結果に対するトークンチェックを怠っているからである．

8.2 「引数付きマクロ」から「関数型マクロ」へ

引数のないマクロの展開は単純なものであるが，引数付きマクロの展開には歴史的にさまざまな仕様があり，混乱があった．C90 で一応の整理がされたが，まだ収束したとはいえない状況にある．この問題については cpp-test.txt の 1.7.6 項で詳細に論じているところである．

混乱の元は 1 つには，エディタによるテキストの一括置換と同じテキストベースの発想である．もう 1 つは，マクロ呼び出しに際して，その置換リストが別の引数付きマクロの呼び出しの前半部分を構成する場合，再走査時に後続するトークン列を巻き込んで展開されるという伝統的な仕様である．7.2.4 項で言及したのはその最も害の少ない例である．この仕様は，たまたま C プリプロセッサの伝統的な実装方法がそうした欠陥を持っていたことによるものと考えられる．意図しない「バグのような仕様」だったのではないだろうか．しかし，これが種々の変則的マクロを誘発する結果となったのである．

C90 はこの混乱の続いていた引数付きマクロについて，「関数型マクロ（function-like macro）」という名前を付けて，関数呼び出しに似せて仕様を整理した．すなわち，関数呼び出しと相互に置き換えて使うことができる形を意図したのである．それによって，引数内のマクロが先に展開されてから置換リスト中のパラメータが対応する引数と置き換えられること，引数中のマクロの展開はその引数の中で完結しなければならないことが明確にされた（C90 以前には，パラメータを引数と置き換えてから，再走査時に展開する実装が多かったと思われる）．

ところが C90 は他方で，再走査時に後続するトークン列を取り込むというバグのような仕様を公認してしまった．これは function-like の原則をぶちこわすものであり，これによって混乱がその後も続くことになったのである．同時に C90 は，マクロ展開で無限再帰が発生することを防ぐために，同名のマクロは再走査時には再置換しないという規定を付け加えている．しかし，「後続するトークン列」の取り込みを認めたために，再置換禁止の範囲はどこまでかという疑義を解消することができず，corrigendum が出たり再訂正されたりと，C99 に至るまで迷走を続けてきている．

8.3 「マクロプロセッサ」から「C プリプロセッサ」へ

多くの C プリプロセッサの実装では，マクロ再走査時に置換リストと後続テキストとを連続して読み込むことを原則とする，伝統的なプログラム構造をとっているようである．これはマクロ呼び出しが置換リストに置き換えられると，その先頭に戻って再走査し，対象を後ろにずらしながら次のマクロ呼び出しをサーチして，再走査を繰り返してゆくものである．

この伝統的プログラム構造の背景にあるのは，C プリプロセッサがマクロプロセッサから生まれたという歴史的事情である．GNU C 2/cpp のように，マクロ再走査ルーチンがプリプロセッサの事実上のメイン

ルーチンとなっていて，これが対象を後ろにずらしながらテキストを入力ファイルの終わりまで「再走査」してゆき，この中からプリプロセスディレクティブの処理ルーチンまでも呼び出されるという組み立てになっているものもある．これはマクロプロセッサの構造であるが，マクロ展開と他の処理とが混交しやすいという問題を持っている．7.2.3 項で見た#if 行でマクロ展開の仕様が変わってしまうのは，その一例である（GNU C 2/cpp は#if 行では内部的に `defined` を特殊なマクロとして扱っている．何でもマクロで実装するのはマクロプロセッサの特徴である）．

MCPP の実装では，標準モードおよび post-Standard モードのマクロ展開ルーチンは伝統モードのものとはまったくの別ルーチンとして書いている．そして，マクロ展開ルーチンはマクロ展開だけをやり，他のことはやらない．また，他のルーチンはマクロ展開はすべてマクロ展開ルーチンに任せて，その結果だけを受け取る．

マクロ展開ルーチンは繰返しではなく再帰構造で組み立て，同名マクロの再置換を防ぐ簡単な歯止めを付けている．関数型マクロの展開は function-like の原則を徹底させ，再走査はマクロ呼び出しの中で完結することを原則としている．Post-Standard モードでは，マクロ展開はすっきりとこれでおしまいである．そして，標準モードでは規格の不規則な規則に対応するためにあるトリックを設けて，必要なときだけ例外的に後続のトークン列を取り込むようにしている．このほうがプログラム構造が明快になり，変則的なマクロを捕捉してウォーニングを出すことが容易になるからである．

8.4 処理系の付属物からポータブルなプリプロセッサへ

C にプリプロセスというフェーズがあるのは portability の確保が大きな目的の 1 つであるが，プリプロセッサが処理系のオマケのような存在である場合が多く，その仕様がまちまちであることによって，プリプロセスが逆に portability を損なう結果をしばしば生んできた．

それに対して，C90 ではプリプロセスは実行時環境からほぼ独立したフェーズであり，かなりの portability が保証されている．そればかりでなく，プリプロセッサそのものについても，処理系の他の部分と異なり，portable に書くことが可能となった．各処理系が高品質で portable な同一のプリプロセッサを使うという形態さえ，ありえないことではない．Portable なソースのための portable なプリプロセッサが現れる条件は C90 で用意されたといえる．

しかし，残念ながら C90 にも上記のような矛盾が存在しており，その解決が後の規格の課題として残されることとなった．

8.5 C99 と C++98

ところが，C90 から C99 への規格の更新ではプリプロセスにも新しい機能がいくつか付け加えられたものの，上記のような論理の矛盾は何 1 つ解決されなかったのである．そればかりか，UCN のように，追加された機能によって仕様の明快さが損なわれた部分さえある．言語の進化の過程ではありがちなことであるが，「あれも欲しい」「これも欲しい」という要望に応えることで，論理の一貫性や言語仕様の単純明快さを損なう方向へと動いてしまったのは残念なことである．

C++98 に至っては，keyword の存在しないプリプロセスにいくつかの事実上の keyword を持ち込んだり，multi-byte character の encoding には unicode に変換可能なものを強要したりと，かなり問題の多いものとなっている．

結局，C プリプロセスの歴史では，C90 が不十分ながらも言語仕様の根本にまで踏み込んで規定した唯一の動きだったといえる．現在は再び仕様が拡散しかけているときであり，再度踏み込んだ整理が必要になっている．その方向は，C90 で中途半端に終わったいくつかの原則を徹底させることであろう．

MCPP は「トークンベース」「関数型マクロの関数的展開」「マクロ処理と他の処理との分離」「portable なプリプロセス」等の原則を中心として組み立てられた C プリプロセッサである．規格準拠モードではその上にいくつかの修飾を加えることで規格に対応させているが，それが最終的な目標ではない．規格そのものの不規則性を整理してこれらの原則を徹底させた自称 post-Standard モードのプリプロセスもあり，これは筆者の期待する将来の単純明快な仕様の C プリプロセッサを実装してみたものである．このモードで問題の検出されないソースは，プリプロセス上はきわめて portability の高いソースだといえる．

9. おわりに

規格準拠性にすぐれた C プリプロセッサ MCPP を C プリプロセス検証セットと並行して開発し，他の C プリプロセッサに対する優位性を示すことができた．検証システムの整備が優れた処理系の開発のために重要であることを示した．また，C プリプロセッサの実装では，原則を明確にしてプログラム構造を組み立てることが最重要であることを主張した．さらに，C

プリプロセスの原則について，歴史を振り返って論考した．

筆者が DECUS cpp をいじり始めたのは 1992 年のことである．それから 10 年の歳月が経過した末に，MCPP は「未踏ソフトウェア」に採択されて，ようやく世に出る機会を与えられた．2 年近くにわたる仕上げによって，世界一規格準拠性が高く，品質の優れた，かつ portable な C プリプロセッサを開発することができたつもりである．

MCPP は V.2.0 以来，vector と@nifty で公開してきた．

未踏ソフトウェアのプロジェクトでは，m17n.org に CVS repository，FTP site，WWW page が用意された．MCPP の最新版はここに置かれている[19]．

多くの C プログラマのコメントと開発参加をいただければ幸いである．

謝辞 MCPP と検証セットの開発は，情報処理振興事業協会（現・情報処理推進機構）（IPA）の「未踏ソフトウェア創造事業」のプロジェクトとして，新部裕プロジェクトマネージャ（平成 14 年度）・伊知地宏プロジェクトマネージャ（平成 15 年度）の助言と IPA の援助を受けて遂行された．また，本稿は伊知地 PM および情報処理学会プログラミング研究会の皆さんのコメントを得て書き直された．記して謝意を表する．

参 考 文 献

1) ISO/IEC: *Programming Languages–C*, ISO/IEC 9899:1990 (E) (1990).
2) ISO/IEC: *ibid. Technical Corrigendum 1* (1994).
3) ISO/IEC: *ibid. Amendment 1: C integrity* (1995).
4) ISO/IEC: *ibid. Technical Corrigendum 2* (1996).
5) ISO/IEC: *Programming Languages–C*, ISO/IEC 9899:1999 (E) (1999).
6) ISO/IEC: *ibid. Technical Corrigendum 1* (2001).
7) 情報処理推進機構：未踏ソフトウェア創造事業. http://www.ipa.go.jp/jinzai/esp/
8) ISO/IEC: *Programming Languages–C++*, ISO/IEC 14882:1998 (E) (1998).
9) Minow, M.: DECUS cpp. http://sources.isc.org/devel/lang/cpp-1.0.txt
10) Roskind, J.: JRCPPCHK. 現在は所在不明.
11) Borland International Inc.（ボーランド株式会社）：Borland C++ V.4.0 (1994).
12) Delorie, D.: DJGPP 1.12 m4. http://www.vector.co.jp/soft/dos/prog/se016978.html
13) きだあきら：LSI C-86/cpp 改造版 beta13（かつて C Magazine 誌の付録 CD-ROM に入っていた）.
14) Borland Software Corp.（ボーランド株式会社）: Borland C++ Compiler 5.5. http://www.borland.co.jp/cppbuilder/freecompiler/
15) Free Software Foundation: GCC V.2.95. V.3.2. http://gcc.gnu.org/
16) Pornin, T.: ucpp V.1.3. http://pornin.nerim.net/ucpp/
17) Microsoft Corporation: Visual C++.net 2003 (2003).
18) Navia, J.: LCC-Win32 V.3.2. http://www.q-software-solutions.com/products/lccwin32/
19) 松井 潔：MCPP V.2.4.1. http://www.m17n.org/mcpp/

(平成 16 年 3 月 30 日受付)
(平成 16 年 8 月 3 日採録)

松井 潔

昭和 15 年生．昭和 46 年東京大学社会学系大学院文化人類学専門課程修士課程修了．同年秋川病院就職．昭和 48 年陽和病院就職．精神病院の臨床心理の仕事に従事してきた．