

高品質のCプリプロセッサ MCPP

松井 潔

kmatsui@t3.rim.or.jp

2006 年 7 月 7 日

概要

長い歴史を持つCのプリプロセッサ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みられる。

その上、プリプロセッサには処理系依存の仕様がしばしばあり、そのため portability の確保を目的とするプリプロセッサが逆に portability を損なう結果も生んでしまっている。また、プリプロセッサレベルのデバッグはコンパイルフェーズではできないのでプリプロセッサが支援すべきところであるが、そうした機能は既存のプリプロセッサにはほとんど実装されていない。これらの問題はいずれも、多くの処理系でプリプロセッサがコンパイラのオマケのような存在だったことが背景となっていると考えられる。この状況はC90以降も現在までずっと尾を引いてきているのである。

MCPP はこれらの問題を解決することを目指して開発されたものである。MCPP は free でかつ portable なCプリプロセッサであり、C/C++ プリプロセッサの徹底的なテストと評価をする検証セットが付属している。これを適用すると MCPP は抜群の成績を示す。MCPP は規格の仕様をほぼ完全に実装しているとともに、豊富で的確な診断メッセージとデバッグ用の情報を出力する `#pragma` を持ち、ソースのプリプロセッサ上の問題をほぼすべてチェックすることができる。

1 はじめに

長い歴史を持つCのプリプロセッサ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言わざるをえない。

この状況の背景となっているのは、C90以前のCプリプロセッサの仕様がはなはだあいまいなものであることである。C90でCのプリプロセッサ仕様は、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。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C99⁵⁻⁷でも解決されなかったのである。さらに現実の処理系の多くは原則をあいまいにしたまま規格にある個々の仕様を接ぎ木することを繰り返して、それによって問題を長く引きずってきていると考えられる。プリプロセッサがコンパイラに従属する存在だったことが、さらにその背景を成している。

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 を欠いた不必要に処理系依存のコード等、プリプロセッサレベルの問題を持つものが少なくない。Cにプリプロセッサ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、プリプロセッサが逆に portability を損なう結果も生んでしまっているのである。

さらに、プリプロセッサがコンパイラの「プリ」プロセスであるという性格から、コンパイルフェーズ

ではプリプロセッサ・ディレクティブやマクロは消えてしまっているために、デバッグが困難になるという問題が引き起こされる。プリプロセッサレベルのデバッグはプリプロセッサが支援してほしいところであるが、そうした機能を持つプリプロセッサは少なくとも私がテストした処理系には存在しない。

私は久しく以前から 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 `cpp V.2.0` として、1998/11 に `cpp V.2.2` として公開していたが、V.2.3 への update の途中で情報処理推進機構 (IPA) の 2002 年度と 2003 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 がリリースされた¹⁰。その後、V.2.5, V.2.6 がリリースされている。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 はおそらく世界一優れた C プリプロセッサである。私が勝手にそう言っているだけではなく、「検証セット」を並行して作成して、それによって検証済みであるところが特徴である。

また、MCPP は診断メッセージが豊富であり、デバッグ用のいくつかの `#pragma` ディレクティブも持っている。これを使うことでソースのプリプロセッサ上の問題点をほぼすべてチェックすることができ、ソースの portability の向上に役立てることができる。

MCPP はどの処理系にも容易に移植できるプリプロセッサであり、コンパイラ本体から独立したフェーズとしてのプリプロセッサの portability を確保することができる。また、プリプロセッサの原則を明確にしてプログラム構造を組み立てており、仕様が明確である。

本稿では、まず MCPP の概要を述べ、C プリプロセッサの基礎的規定に簡単に触れたあとで、検証セットを紹介し、他のプリプロセッサとの比較データを示す。そして、処理系のバグと問題点の例を示す。次いで、MCPP による現実のソースプログラムのチェックを取り上げ、さらに、C プリプロセッサの原則とその実装方法について論ずる。

2 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++ のプリプロセッサの 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^{8,9} 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。
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。Tokenization をトレースしたり、マクロ展開をトレースしたり、マクロ定義の一覧を出力したりすることができる。
5. Multi-byte character の処理は日本の EUC-JP, shift-JIS, ISO-2022-JP、中国の GB-2312、台湾の Big-5、韓国の KSC-5601 (KSX 1001)、および UTF-8 の各 encoding に対応している。Shift-JIS, ISO-2022-JP, Big-5 の場合、コンパイラ本体が漢字を認識しない処理系では、MCPP がそれを補う。
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。メモリが少なくても動作する。
7. Portable なソースである。C90, C99, C++98 のどれに準拠する処理系でもコンパイルできる広い portability を持っている。MCPP をコンパイルする時に、ヘッダファイルにある設定を書き換えることで、UNIX 系、Windows のいくつかの処理系で、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。処理

系から独立して動作する stand-alone 版のプリプロセッサを作ることでもある。

8. 標準モード(C90, C99, C++98 対応)の仕様のほか、 $K\&R^{1st}$ の仕様やいわゆる Reiser cpp モデルのオプションもあり、規格の問題点を私が整理した自称 post-Standard モードのオプションまである。
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。GCC の test-suite がインストールされていれば、'make check' コマンド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。BSD 形式の LICENSE のもとにソース・ドキュメント・検証セットのすべてが公開されている。
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。英語版は未踏ソフトウェアのプロジェクトで、翻訳会社のハイウェル(東京)¹⁹ に日本語版からの翻訳を委託し、それに私が修正を加えて作成したものである。その後の update は私が行ってきている。
 - (a) INSTALL: configure と make の方法を説明。
 - (b) mcpp-summary.pdf: サマリ文書。この文書。
 - (c) mcpp-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) mcpp-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。

3 プリプロセッサの基礎的規定

本題に入る前に、C/C++ プリプロセッサの基礎的な規定に手短に触れておく。

3.1 プリプロセッサの手順

プリプロセッサの手順は $K\&R^{1st}$ 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、主旨は変わっていない。

C++98 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

表 1 検証セット V.1.5.1 の項目数と配点

		項目数	最高点
規格 準拠度	K&R	31	166
	C90	140	432
	C99	20	98
	C++98	9	26
品質	診断メッセージ	47	74
	その他	18	164
計		265	960

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。

3.2 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、用語の違いを除けば C90, C99, C++98 で実質的にすべて同じであり、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。

4 プリプロセス検証セットによる各種プリプロセッサの検証

プリプロセッサの開発でもう一つ問題となるのは、プリプロセッサの動作や品質の検証である。多くの処理系は「規格準拠」と称しているが、その根拠が明らかにされているものは少ない。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。

私は 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る。

検証セット V.1.5.1 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り、「C90」というのは C90 で新たに規定された仕様に関する項目である。C99, C++98 の「規格準拠度」というのは、C90 になく新たな規定に関するものである。「規格準拠度」のテスト項目は規格の全仕様を網羅している。

また、「品質:診断メッセージ」というのは、規格で要求されていない診断メッセージに関する評価である。「品質:その他」というのは、実行時オプション・`#pragma`・`multi-byte character` への対応・速度等々の各種品質の評価である。

品質の評価や配点には主観の入る余地があり、また次元の異なる評価項目を合計して 1 本の物差しで表すことには矛盾があるが、実際の使用感に近い評価が得られると考えている。

検証セット V.1.5.1 を次のようないくつかの処理系に適用してみた。その結果を表 2 および図 1 に示す。処理系は古い順に並べてある。

MCPP はずば抜けた成績である。V.2.4 以降では、規格準拠性は C++98 での `multi-byte character` 等の UCN への変換という奇怪な仕様を実装していない以外は完全であり、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デバッグ用の `#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 GCC (GNU C) / `cpp (cc1)` である。このプリプロセッサは C90 規格に準拠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、次のよ

表 2 各種プリプロセッサの検証結果

処理系	年/月	規格準拠度					品質		総合 評価
		K&R	C90	C99	C++ 98	計	診断 msg	その 他	
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	368	20	6	558	18	72	646
GCC 2.95.3 ^{*4}	2001/03	166	404	56	6	632	23	113	768
GCC 3.2 ^{*5}	2002/08	166	419	86	20	691	33	117	841
ucpp 1.3 ^{*6}	2003/01	166	384	88	9	647	25	88	760
Visual C 2003 ^{*7}	2003/04	156	394	43	15	610	20	83	711
LCC-Win32 3.2 ^{*8}	2003/08	158	376	18	6	558	18	94	670
Wave 1.0.0 ^{*9}	2004/01	140	338	53	18	549	20	79	648
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	39	120	847
GCC 4.0.2 ^{*12}	2005/09	166	417	87	20	690	32	122	844
Visual C 2005 ^{*13}	2005/09	160	399	65	17	641	19	77	737
LCC-Win32 3.8 ^{*14}	2006/03	158	374	22	6	558	20	95	673
MCPP 2.6 ^{*15}	2006/06	166	432	98	22	718	74	134	926
最高点		166	432	98	26	722	74	164	960

うな問題がある。

^{*1} DECUS cpp: Martin Minow による DECUS cpp のオリジナル版¹¹ を筆者が若干の修正を加えて Linux / GCC でコンパイルしたもの。

^{*2} 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 / GCC 2.7, WIN32 / Borland C 4.0, MS-DOS / Turbo C 2.0 等に対応していた。各種の仕様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 Linux / GCC でコンパイルしたものの標準モードを使用。

^{*3} Borland C 5.5: Borland C++ 5.5, Borland, 日本語版。¹²

^{*4} GCC 2.95.3: VineLinux 3.2, CygWIN 1.3.10 にバンドルされているもの。

^{*5} GCC 3.2: ソースから筆者が Linux 上でコンパイルしたもの。

^{*6} ucpp 1.3: Thomas Pornin による portable な free software。Stand-alone のプリプロセッサ。¹⁴

^{*7} Visual C 2003: Visual C++ .net 2003, Microsoft。

^{*8} LCC-Win32 3.2: Jacob 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の。

^{*9} Wave 1.0.0: Hartmut Kaiser による free software。Paul Menssonides らによる Boost C++ preprocessor library というものを使って実装されている。WIN32 版

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、それでもまだかなり不足している。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. デバッグ情報を出力する機能はほとんどない。
3. ドキュメントが不足しており、仕様の不明確な部分や隠れ仕様が存在する。

の実行プログラムでテスト。¹⁷

^{*10} MCPP 2.4: V.2.0 以降、Linux, FreeBSD / GCC (2.95, 3.2), CygWIN 1.3.10, LCC-Win32 3.2, Borland C 5.5, Visual C++ .net 2003 等への対応が追加された。

^{*11} GCC 3.4.3: Linux 上で筆者がコンパイルしたもの。¹³

^{*12} GCC 4.0.2: openSUSE Linux 10.0 にバンドルされているもの。

^{*13} Visual C 2005: Visual C++ 2005 Express Edition, Microsoft。¹⁵

^{*14} LCC-Win32 3.8: LCC-Win32 の V.3.8。¹⁶

^{*15} MCPP 2.6: MCPP V.2.6, V.2.4 以降、GCC 3.3-4.0, Visual C++ 2005 への対応が追加されている。¹⁸

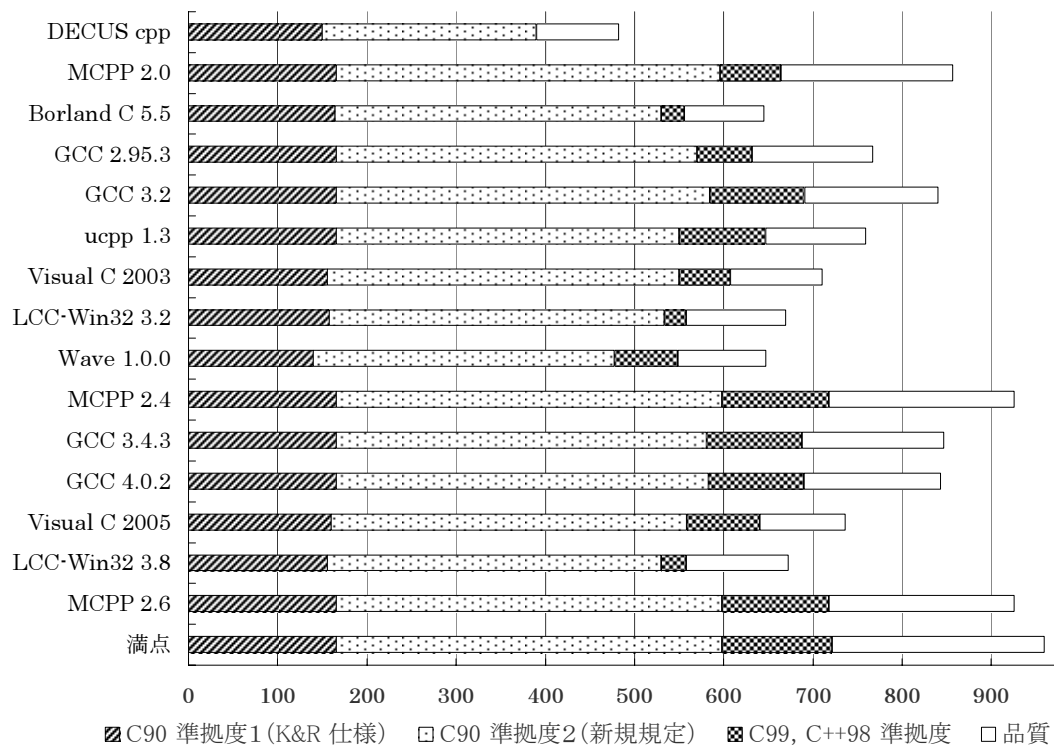


図 1 各種プリプロセッサの検証結果

4. 規格と矛盾する独自仕様がいくつ (拡張仕様は #pragma で実装すべきものである)。

GCC V.3 / cc1 は GCC V.2 / cpp 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 GCC / cc1 に劣るのは速度くらいである。

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 の仕様はまだ半分しか実装していないものが多い。
2. 診断メッセージ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. ドキュメントの不十分な処理系が大半である。

6. Multi-byte character は 1、2 種の encoding にしか対応していない処理系が多い。

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。

5 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次に示す。

5.1 コメントを生成するマクロ

図 2 の example-1 は Visual C の Platform SDK 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 2003, 2005 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

```

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

```

図 2 プリプロセッサのバグの例

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解されコメントが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など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 のプリプロセッサの実装である。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. この例では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. プリプロセッサの手順 (translation phases) が恣意的であり、論理的な一貫性がない。

5.1.1 MCPP の診断メッセージ

Visual C 用の MCPP で <windows.h> をプリプロセッサするといくつかのウォーニングが出るが、図 2 の example-2 のマクロについては図 3 のような診断メッセージが出る (紙面のつごうで行は適宜改行している)。

まず問題のマクロ呼び出しのあるソースファイル

名と行番号・診断メッセージの本文が表示され、次いでそのマクロがどこでどう定義されているかが示される。さらに、ソースファイルのインクルード元がネストをさかのぼって順次表示される。どこに問題があるのかが一目瞭然である。

5.2 スキップされるはずの式でエラーになるもの

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、いまは見掛けなくなっている。しかし、example-4 は Visual C 2003, 2005 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。これは必ずしも間違っているとは言えないが、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言えないが、間に合わせ的な診断メッセージであり品質は良くない。

5.3 オーバーフローが診断されないもの

example-6 は C90 ではオーバー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。値がラップラウンドして符合や大小の判定が逆転しても、診断メッセージは出ないことになる。GCC と Borland C は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、出さない場合も多く、一貫していない。

```

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"
...

```

図3 MCPP の診断メッセージの例

6 プリプロセッサによるソースチェックはなぜ必要か

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、プリプロセッサ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、一つには C90 以前のプリプロセッサ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。もう一つは、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。

6.1 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ッサ上の問題点を、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 2.2.2 (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 mcpp-manual.txt の 3.9 節で報告している。Libc にはまったくと言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 (GNU LIBC) 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わかった。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 なプリプロセッサ仕様を使ったものと、GCC / cpp の独自の仕様や undocumented な仕様を前提としたものが多い。GCC / 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時々見られることが問題である。システムのヘッダファイルにさえも時に問題がある。

他方で、コメントのネストは規格違反であるが、1990 年代中ごろまでは UNIX 系のオープンソースにしばしば見られたものの、その後は見かけなくなっている。これは GCC / cpp がコメントのネストを認めなくなったためであろう。プリプロセッサ

はソースに大きな影響を与えるのである。

6.2 glibc のソースを例に

glibc 2.1.3 のソースを例にとって、プリプロセッサ上の問題点の一端を見てみたい。

6.2.1 行をまたぐ文字列リテラル

図 4 の 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ッサ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。

6.2.2 プリプロセッサを要する *.S ファイル

アセンブラソースの中に `#if` 等のプリプロセッサディレクティブや 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ッサ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リプロセッサを使うべきものである

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。

6.2.3 'defined' に展開されるマクロ

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、GCC / 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。^{*16}

ちなみに MCPP では example-11 はエラーとなるが、次のように `#pragma` 行を挿入すると、マクロ展開のプロセスが表示され、何が問題なのかを把握することができる。

```
#pragma MCPP debug expand token
#if HAVE_MREMAP
#pragma MCPP end_debug
```

6.2.4 関数型マクロとして展開されるオブジェクト型マクロ

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) の定義が時々ある。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` が高いとも言える。example-15 のようなオブジェ

^{*16} glibc 2.2 ではこのマクロは修正された。

```

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 4 glibc のソースの例

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、少なくともこの場合はないと思われる。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。

6.2.5 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 GCC 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。

そのほか glibc を make するときに使われる script やツールの中には、不必要に GCC / cpp (cc1) の動作の細部に依存しているものが散見される。

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、GCC / 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。

7 C プリプロセスの原則と MCPP の実装方法

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 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。しかし、現実の処理系の多く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。その上、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. 引数付きマクロの呼び出し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通りの「プリ」プロセスであり、処理系依存の部分の必要性はほとんどない。

これらは MCPP の実装の原則でもある。

7.1 トークンベースの処理

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。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）。

MCPP のプログラム構造は「トークンベースのプリプロセス」という原則で組み立てられており、traditional な文字ベースのプリプロセスとは発想を異にする。他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のである。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。

例えば Borland C 5.5 や Visual 2003, 2005 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一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、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。

7.2 関数型マクロの関数的展開

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、まだ収束したとは言えない状況にある。この問題については私の検証セットの cpp-test.txt の

1.7.6 節で詳細に論じているところである。

混乱の元は一つには、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。もう一つは、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。6.2.4 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、迷走を続けてきている。

7.3 マクロ展開と他の処理との分離

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。GCC 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テ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。6.2.3 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 (GCC 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)。

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り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。

7.4 ポータブルなCプリプロセッサ

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一つであるが、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として規定されており、それによってかなりの 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 で用意されたと言える。MCPPE もこの状況を動機の1つとして開発が始められたものである。現在はコンパイラにプリプロセッサを吸収した処理系が増えてきているが、コンパイラとプリプロセッサが複雑にからみあうプログラム構造は決して良いことではない。プリプロセッサが独立したプログラムであることは、コンパイラに依存する部分を少なくして独立したフェーズとしてのプリプロセスの portability を確保するために有効なことなのである。

C90 のプリプロセス規定には以上のような諸原則が体现されている。しかし、残念ながら同時に上記のような矛盾も存在しており、その解決は後の規格が課題とすべきものであった。ところが、C99では新しい機能がいくつか付け加えられたものの、これらの論理の矛盾は何一つ解決されなかったのである。そればかりか、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。C++98にはさらに問題が多い(これらの問題については、cpp-test.txt で詳細に検討している)。

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言える。現在は再び仕様が拡散しかけている時であり、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。

MCPPE 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。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、これは私の期待する単純明快な仕様の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 の高いソースだと言える。

8 現バージョンと update 計画

8.1 V.2.6

MCPPE V.2.6 は 2005/03 にリリースした V.2.5 に次のような updates を加えたものである。

1. Stand-alone 版の動作をどの処理系にも依存しないようにした。
2. 処理系専用版はその処理系専用のディレクトリにだけインストールされるようにした。
3. 従来 Standard モードと pre-Standard モードによって実行プログラムが2つに分かれていたのを1つに統合し、実行時オプションで区別するようにした。
4. GCC V.4.0, Visual C++ 2005 に対応させた。
5. pre-C90 の処理系および MS-DOS 上の処理系用のコードを削除した。
6. その他、いくつかの小さな改良を加えた。
7. これらに合わせてドキュメントを大幅に書き直した。

8.2 V.2.7 の計画

MCPPE の update は作業が大幅に遅れているが、現在は V.2.7 で次のようなことを計画している。

1. MCPPE の診断メッセージを別ファイルに分

離し、各国語版の診断メッセージを随時追加できるようにする。

2. 問題のあるソースを portable なものに自動的に書き換えるオプションを実装する。
3. テキスト版のドキュメントを html 版に移して、検索性を改善する。

9 おわりに

最高度の規格準拠性を持ち、かつプリプロセッサのソースチェックに役立つ高品質のCプリプロセッサを目指して MCPP を開発してきた。処理系の開発には検証システムが不可欠であるが、プリプロセッサに関しては十分なものが存在しなかったので、プリプロセッサ検証セットを MCPP と並行して開発した。それによって、他のプリプロセッサに対する優位性を示すことができた。また、コンパイラ本体から独立したプリプロセッサが、処理系依存の仕様を最小にし、独立したフェーズとしてのプリプロセッサを実現するために有効であることを示した。さらに、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を組み立てることが最重要であることを論じた。

私が 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、世界一正確で品質の優れた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。そして、英語版ドキュメントとともに国際的な評価の場に出せる状態となった。「未踏ソフトウェア」では、その成果によって最高級の評価も受けることができた。熟年のアマチュアプログラマとして、非力ながらもよくやったと自分では納得している。

プロジェクトの終了後も MCPP の update 作業を続けており、今後も続けてゆくつもりである。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。

参考文献, URL

- [1] ISO/IEC. *ISO/IEC 9899:1990(E) Programming Languages – C*. 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. *ISO/IEC 9899:1999(E) Programming Languages – C*. 1999.
- [6] ISO/IEC. *ibid. Technical Corrigendum 1*. 2001.
- [7] ISO/IEC. *ibid. Technical Corrigendum 2*. 2004.
- [8] ISO/IEC. *ISO/IEC 14882:1998(E) Programming Languages – C++*. 1998.
- [9] ISO/IEC. *ISO/IEC 14882:2003(E) Programming Languages – C++*. 2003.
- [10] 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/>
- [11] Martin Minow, DECUS cpp.
<http://www.isc.org/index.pl?sources/develop>
- [12] Borland Software Corp., ボーランド株式会社, Borland C++ Compiler
<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.

<http://spirit.sourceforge.net/>

[18] 松井 潔, MCPP.

<http://mcpp.sourceforge.net/>

[19] 有限会社ハイウェル.

<http://www.highwell.net/>