ソースチェックに威力を発揮するＣプリプロセッサ

松井　潔
kmatsui@t3.rim.or.jp

2004 年 3 月 20 日

概要

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みられる。また、既存のプリプロセッサはどれも寡黙すぎて、ソースチェックの能力が不十分である。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 は正確であるとともに、豊富で的確な診断メッセージを持ち、ソースのプリプロセス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

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 として公開している。このプリプロセッサは V.2.3 への update の途中で、情報処理振興事業協会（現・情報処理推進機構）(IPA) の「平成１４年度未踏ソフトウェア創造事業」に採択された。さらに「平成１５年度未踏ソフトウェア創造事業」にも継続して採択された [1]。この cpp を他の cpp と区別するために MCPP と呼ぶことにする。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」を並行して作製して、それによって検証ず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る。これを使うことでソースのプリプロセス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。

本稿では、まず MCPP と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、処理系のバグの例を示す。次いで、MCPP によるソースチェックを取り上げ、その意義を述べる。さらに、Ｃプリプロセスの原則とその実装方法について論ずる。次に V.2.3, V.2.4 への update の成果を述べ、最後に今後の update の計画に触れる。

2　MCPP の概要

MCPP は次のような特徴を持っている。

1. きわめて正確である。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 [2–5] はもちろんのこと、C99 [6, 7], C++98 [8] 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・中国・台湾・韓国の多様な 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。16 ビットシステムでも使えるように作られているので、メモリが少なくても動作する。
7. Portable なソースである。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、DOS/Windows 系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。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. 標準モード（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私が整理した自称 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。
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。
    （a）README: configure と make の方法を説明。
    （b）mcpp-summary.pdf: サマリ文書。この文書。
    （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。

表 1: 検証セット V.1.4 の項目数と配点

	項目数	最低点	最高点
C90 規格合致度	173	-162	463
C99 規格合致度	20	0	98
C++98 規格合致度	9	0	26
品質:診断メッセージ	47	0	74
品質:その他	16	-40	113
計	265	-202	774

3 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と同時に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。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。私は MCPP 開発の一環として、プリプロセス検証セットを作製し、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.1.4 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。$K\&R^{1st}$ と C90 との共通仕様を正しく実装していれば 0 点、それさえも実装できていなければマイナス点、C90 以降の新しい仕様を正しく実装していればプラス点をつけるようになっている。「規格合致度」には診断メッセージとドキュメントの評価も含まれる。C99, C++98 の「規格合致度」というのは、C90 にない新たな規定に関するものである。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。

表 2: 各種プリプロセッサの検証結果

ＯＳ	処理系	プリプロセッサ (版数)	規格合致度	総合評価	注
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 (改造版 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

検証セット V.1.4 をいくつかの処理系に適用した結果のサマリを表 2 に示す。処理系は古い順に並べてある。

*1 Martin Minow による DECUS cpp のオリジナル版 (1985/01) [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 J. Roskind による UNIX, OS/2, MS-DOS 用 shareware である JRCPP の MS-DOS - OS/2 用試用版 (1990/03)。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 1993 年のものの日本語版 (1994/12)。 [11]

*4 GNU C 2.7.1 / cpp (1995/12) を DJ 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]

*5 LSI C-86 / cpp のきだあきらによる改造版 (1996/02)。 [13]

*6 筆者による free software の V.2.0 (1998/08)。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 日本語版 (2000/08)。 [14]

*8 VineLinux 2.6, FreeBSD 4.4, CygWIN 1.3.10 で使われている GNU C 2.95.3 (2001/03)。 [15]

*9 GNU C 3.2R のソース (2002/08) を筆者が VineLinux 2.6, FreeBSD 4.7 上でコンパイルしたもの。 [15]

*10 Thomas Pornin による portable な free software (2003/01)。Stand-alone のプリプロセッサ。 [16]

*11 Microsoft (2003/04)。 [17]

*12 Jacob Navia 等による shareware (2003/08)。ソース付き。プリプロセス部分のソースは Dennis Ritchie その人が 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 (2004/03)。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]

このように、MCPP はずば抜けた成績である。動作の正確さ、診断メッセージの豊富さと的確さ、ドキュメントの詳細さ、portability、どれをとっても抜群である。５年半前のバージョンである V.2.0 でさえも、まだそれを超えるものが見当たらない。V.2.4 ではさらに update され改良されている。自分で作って自分でテストしているのであるから当然とも言えるが、これだけ多角的なテストであればかなりの客観性があろう。

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、それでもまだかなり不足している。
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 な仕様が隠れていることが問題である。
4. 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 / cpplib は GNU C 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 GNU C / cpp に負け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。

4　プリプロセッサのバグと誤仕様の例

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。

```
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　コメントを生成するマクロ

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。

4.2 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++ 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。

4.3 オーバーフローが診断されないもの

example-6 はオーバーフローの発生する式であるが、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。

5 プリプロセッサによるソースチェックはなぜ必要か

ところで、プリプロセッサはC処理系の一部にすぎない。いくら世界一でも、プリプロセッサだけ作って何の役に立つのだろうか。

Cのプリプロセスは文字通りソースの「前処理」であるが、コンパイラ本体に対するオマケのようなものとして扱われてきたきらいがある。しかし、この「前処理」の存在理由は、readability とメンテナンス性を改善することであり、多様なシステムに対応するための portability の確保であり、要するにソースをより人間（プログラマ）にとって扱いやすくすることである。コーディングとメンテナンスの作業に対する影響は大きい。逆に、「前処理」が誤用されると、readability も portability もかえって損なわれることになる。この意味で、プリプロセッサによるソースチェックは重要なのである。

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、一つには 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。

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みられる。これは処理系そのものの検証が不十分なためである。その背景には、多くのCプリプロセッサの開発が C90 以前のものをベースとして、Cプリプロセスの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たという経過があるのではないかと考えられる。正確なCプリプロセッサの開発のためには、原則を明確にしてソースプログラムを書き直すことが必要であり、また、その検証をするソフトウェアの整備が不可欠である。

また、処理系固有の仕様はすべてドキュメントに記載することが規格で定められているように、詳細で正確なドキュメントの作成は処理系の不可欠の部分を成す。しかし、この点も不十分な処理系が多い。この仕様のあいまいさが、portability を欠いたソースプログラムを生む一つの背景ともなっていると考えられる。

さらに、規格自体にも、歴史的な事情からくる矛盾やあいまいさがいくつかあり、問題を複雑にしている。十分枯れた分野のようにも見えるCプリプロセスであるが、まだ問題は収束したとは言えないのである。

他方で、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るので、処理系の他の部分と異なり、portable なプリプロセッサというものが成立しやすい。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熟しているのである。

5.1 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 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 GNU C / 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。

5.2 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。

5.2.1 行をまたぐ文字列リテラル

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。

```
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)
```

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。

5.2.2 プリプロセスを要する *.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。

5.2.3 '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、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。

5.2.4 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。

5.2.5 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。

6 MCPP の実装方法

私が MCPP の実装にあたって目標としたのは「概要」で述べた諸点である。これは完全主義的な目標であるため、かなりの時間がかかってしまったが、ほぼ達成できてきている。

正しいソースを正しく処理することはもちろんであるが、間違ったソースや怪しげなソースに的確な診断メッセージを出すことも重視し、十分なメッセージ群を用意した。

ドキュメントも重視し、undocumented な仕様のないよう、規格書にある共通仕様以外の全仕様をドキュメントに記載した。規格については、検証セットのドキュメントに網羅的な解説を記載した。ドキュメントを書くことで仕様を明確に意識することができ、MCPP のデバッグと検証セットの充実が進んだという効果も大きい。

多くの処理系に実装してきたことも、MCPP 自身のソースの portability を広げ、動作チェックを徹底するために役立っている。

また、網羅的な検証セットの作製と並行して開発してきたことも、バグのないプリプロセッサを作る結果につながってきている。他のプリプロセッサを見ると、例えば LCC-Win32 のプリプロセス部分は Ritchie の書いたソースを使っているが、これには #if 式の評価などにかなりのバグがある。いかに Ritchie であっても、検証セットなくしてはバグは免れないのである。GNU C / cpp は長い間に多くの人々にデバッグされてきてほぼ bug free となっているが、十分な検証セットがあればもっと早くバグがとれたはずである。それほどポピュラーでないプリプロセッサの場合は、検証セットなしにはデバッグは不可能と言って良い。

6.1 トークンベースの原則

さらに MCPP の内部的な実装方法に立ち入ると、私が重視したのはまず「トークンベース」の処理である。

MCPP 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。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 以降もプリプロセッサがそれを看過していたり、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、混乱が長く続いてきているのである。その上、規格自体にも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。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、プリプロセスが逆に portability を損なう結果も生んでしまっているのである。

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。

例えば 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、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。トークンベースのプリプロセスでは、サボらずにいちいちトークンを確かめなければならないのである。

さらに私は、規格そのものの不規則性を整理して「トークンベース」の原則を徹底させた自称 post-Standard モードのプリプロセスも実装している。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。

6.2 関数型マクロの関数的展開

MCPP の実装ではマクロ展開ルーチンも意を注いだところであり、既存のプログラムにとらわれず、明快なプログラム構造とすることを心掛けた。

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。5.2.4 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。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。

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、こうした論理の矛盾は何一つ解決されていない。言語の進化の過程ではありがちなことであるが、長い歴史を持つからこそ、かえって収束が遅れてしまうという面があるようである。

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。5.2.3 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（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）。

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 「未踏ソフトウェア」の成果

私は V.2.2 を公開した後、しばらくの間は時間がとれず、MCPP の開発は停滞してしまった。しかし、「平成１４年度未踏ソフトウェア」に新部 裕・プロジェクトマネージャによって採択されたのを機に、仕事を減らして時間を確保した。

7.1 V.2.3

2003/02 にリリースされた MCPP V.2.3 と検証セット V.1.3 の主な成果としては次のようなものがある。

7.1.1 GNU C 3.2 への対応

GNU C 3.2 のプリプロセッサを MCPP に置き換えた上で、GNU C 3.2 を使って GNU C 3.2 自身をリコンパイルし、それに testsuite を適用して結果を検証した。MCPP が GNU C 3 / cpp と実用上はほぼ十分な互換性を持っていることが確かめられた。

また、プリプロセスに MCPP を使った結果、GNU C 3.2 のソースのプリプロセス上の問題がチェックできた。GNU C 3.2 のソースは glibc 2.1 等と比べると、はるかに問題の少ないことがわかった。これはプリプロセッサが GNU C 2 のものから一新されたことによる影響が大きいと考えられる。この効果は今後、GNU C 以外のソフトウェアのソースにも波及してゆくことが予想される。

GNU C 3 のプリプロセッサは GNU C 2 と比べて、ソースが一新されるとともに、ドキュメントも一新され、testsuite の testcases も飛躍的に増えている。5.2 でとりあげた諸問題についても、行をまたぐ文字列リテラルのような traditional な書き方は obsolete あるいは deprecated とされてウォーニングが出るようになり、ドキュメントの記載も増え、MCPP と比肩できる品質となってきている。しかし、まだウォーニングやドキュメントが不足している。

7.1.2 検証セットの GNU C / testsuite への対応

私の検証セットのうち動作テストの testcases には V.1.3 で、GNU C / cpp の testsuite として使える edition が追加された。これを Linux と FreeBSD で、それぞれ GNU C 2.95 / cpp, GNU C 3.2 / cpp, MCPP という３つのプリプロセッサに適用できることを確認した。GNU C 2.9x, 3.x の大半のバージョンに使えると思われる。

従来の testsuite の testcases にはかなりの片寄りがあるので、私の検証セットのような網羅的な testcases が追加されるのは、意味のあることだと思われる。

また、従来の testsuite の testcases は単一の処理系しか想定しておらず、GNU C 3 の testcases は GNU C 2 / cpp にさえも適用できないものが多いが、私の検証セットの testsuite 用 edition は GNU C 2 / cpp, GNU C 3 / cpp, MCPP という３つのプリプロセッサのテストができるように書かれた。すなわち、GNU C / testsuite では DejaGnu, Tcl 等のツールが使われているので、それらのツールの正規表現の機能を活用すれば、処理系によるプリプロセス出力の spacing の差異や診断メッセージの差異を吸収することができるのである。DejaGnu, Tcl の正規表現の処理にはかなりの癖や欠陥があり、使うには種々の工夫が必要であるが、工夫すれば複数の処理系について一通りの自動テストが可能であることがわかった。

ただ、testsuite では実行時オプションを処理系によって変えることはできない。複数の規格が事実上併存している現在の状況では `-std=` というオプ

ションで規格のバージョンを明示する必要があるが、このオプションは GNU C の古いバージョンには存在しない。したがって、私の testsuite の適用範囲は GNU C 2.9x 以降と MCPP V.2.3 以降ということになる。

7.1.3 英語版ドキュメントの作成

MCPP V.2.3 および検証セット V.1.3 の全ドキュメントの日英翻訳を、有限会社ハイウェル（東京）[20] に委託した。３人のバイリンガルの担当者によって翻訳され、技術的な内容については私が修正を加えて、英語版ドキュメントとして仕上げた。

7.2 V.2.4 prerelease

MCPP は「平成１５年度未踏ソフトウェア」にも伊知地 宏・PM によって採択された。2003/11 に公開された MCPP V.2.4-prerelease では次のような成果が得られた。

7.2.1 Visual C++ .net 2003 への対応

市販の処理系で最もユーザの多い Visual C++ をとりあげ、これに検証セットを適用するとともに、MCPP を移植した。

Visual C++ .net 2003 のプリプロセッサは最新のバージョンであるにもかかわらず、プリプロセスの手順が恣意的であること、文字ベースの処理が混在していること等、根本的な欠陥を持っていることが明らかになった。

MCPP を Visual C++ に移植したが、この処理系ではプリプロセッサがコンパイラに吸収されているので、プリプロセッサを MCPP に置き換えることができない。したがって、MCPP の使い方としては、MCPP で処理してから Visual C++ の cl.exe を起動するように makefile を書くことになる。

統合開発環境 (IDE) でも、通常の「プロジェクト」では MCPP を使うことはできない。しかし、MCPP を使う makefile を先に書いておいて、IDE で「メイクファイルプロジェクト」というものを作成すると、その makefile がそのまま使われ、ソースレベル・デバッグを含めた IDE のほぼ全機能を使うことができる。

7.2.2 Configure スクリプトの作成

UNIX 系のシステムでは、MCPP のコンパイルを configure スクリプトによって自動的にできるようにした。

MCPP はターゲット処理系のプリプロセッサと置換して使えるのが特徴であるが、そのためには一般のアプリケーションプログラムと異なり、ターゲット処理系の実装に関する多くの調査が必要である。configure.ac ファイルを書くこと自体がターゲット処理系に関する知識を必要とする。

MCPP の configure ではターゲット処理系が GNU C の場合は、MCPP が処理系のプリプロセッサに取って代わって動くようになるところまで完全に自動的に実行することができる。私が GNU C を知っているからである。GNU C の testsuite がインストールされていれば、make check で検証セットの testsuite edition を実行することもできる。

しかし、UNIX 系システムでの他の処理系については私は経験がないので、configure ではいくつかのオプションを用意し、ユーザに必要な事項を指示してもらうようにした。処理系のプリプロセッサと置き換えて使うためには、さらに若干のソースコードを書き足す「移植」の作業が必要である。

なお、DOS/Windows 系の処理系は（CygWIN 以外は）configure の対象とならないので、従来と同じように各処理系ごとに専用の差分ファイルと makefile を用意している。

7.2.3 その他

MCPP のオプションと #pragma を追加し、マクロの再定義と展開に関する診断メッセージを改良した。また、古いコードを削除するなどして、ソースを整理した。

これらに伴って、日本語版のドキュメントを更新した。

これまで MCPP は明確な LISENCE を示してこなかったが、このバージョンから BSD 形式のものを表示するようにした。

7.3 V.2.4

2004/02 には次のような updates を加えた MCPP V.2.4 が公開された。

7.3.1 Multi-byte character の多様な encoding に対応

Multi-byte character の encoding は V.2.0 以来、日本の EUC-JP, shift-JIS、中国の GB 2312、台湾の Big5、韓国の KS C 5601 (KSX 1001) という、いずれも 1 文字が 2 バイトの encoding に対応してきたが、新たにより複雑な encoding にも対応する枠組みを作り、手始めに ISO-2022-JP1, UTF-8 への対応を実装した。

そして、32 ビット以上のシステムではこれらの encoding をすべて実装し、環境変数と実行時オプションによってデフォルトの encoding を変更できるようにした。また、Visual C++ にある `#pragma setlocale` というディレクティブも実装し、ソース中でも encoding を変更できるようにした。さらに、コンパイラ本体がこれらの encoding を処理できない場合は、MCPP の側でその欠陥を補う機能まで実装できるようにした。

かつては multi-byte character の encoding はシステムによって 1 つに固定されていたが、多言語に対応したプログラムの必要が増大するにつれて、処理系も多様な encoding への対応が必要になっている。環境変数・実行時オプション・#pragma によって手軽に各種の encoding が使えることは、ソフトウェア開発者にとって重要である。

7.3.2 Plan 9 に移植

MCPP を Plan 9 edition 4 / pcc に移植した。このシステムでは Ken Thompson の書いたコンパイラと Dennis Ritchie の書いたプリプロセッサが使われているが、MCPP は Ritchie の cpp と置き換えて使うことになる。もちろん、MCPP のほうが格段に高品質である。また、このシステムでは multi-byte character encoding としては UTF-8 が使われているが、MCPP がそれに十分対応していることを確認した。

7.3.3 ドキュメントの更新

これに伴って、ドキュメントを日本語版・英語版ともに update した。英語版ドキュメントは前年度と同様にハイウェルに翻訳を委託し、それを筆者が修正して仕上げた。

7.3.4 世界に公開

英語版ドキュメントとともにパッケージングして、gnu.org および freebsd.org に提起した。

8 V.2.4.1

「未踏ソフトウェア」のプロジェクトは 2004/02 で終了したが、その後、2004/03 には MCPP V.2.4.1 がリリースされた。これは再帰的マクロの展開方法を修正したものである。

9 V.2.5 の計画

MCPP V.2.5 では次のような updates を計画している。

1. Multi-byte character の encoding は容易に追加できるようになったので、ISO-2022-* のいくつかの encoding を追加する。
2. MCPP 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。
3. GNU C 3 / cpp のソースプログラムと test-suite に検討を加える。
4. 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。
5. ドキュメントには texinfo 版と html 版を用意して、検索性を改善する。

10 おわりに

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年弱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。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。

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

平成１４年度未踏ソフトウェアのプロジェクトでは、新部 PM によって m17n.org に CVS repository, ftp site, web page が用意された。開発中の revision を含めて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。

参考文献, URL

[1] 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/

[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. 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] DJ Delory, 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] Thomas Pornin, ucpp V.1.3.
http://pornin.nerim.net/ucpp/

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

[18] Jacob Navia, LCC-Win32 V.3.2.
http://www.q-software-solutions.com/lccwin32/

[19] 松井　潔, MCPP V.2.4.1.
http://www.m17n.org/mcpp/

[20] 有限会社ハイウェル.
http://www.highwell.net/