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

2002年8月19日

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。現在、これをV.2.3にupdateする作業をしているところである。

このcpp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。

また、このcpp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、まず私のcpp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。次いで、このcpp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。最後に、このcppの今後のupdateの計画に触れる。

2　私のプリプロセッサの概要

私のcpp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 [1–4]はもちろんのこと、C99 [5, 6], C++98 [7]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なテスト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. 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。
6. Portable なソースである。 Cpp 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 cpp 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。
7. 標準モード（C90, C99, C++98 対応）の cpp 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 cpp 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。
8. オープンソースである。
9. 詳細なドキュメントが付属している。
   （a）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。
   （b）実装用ドキュメント。任意の処理系に実装する方法。
   （c）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。

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 cpp 開発の一環として、プリプロセス検証セットを作製し、cpp 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.1.2 をいくつかの処理系に適用した結果は、その cpp_test.doc [5] 章に報告している。

現在開発中の検証セット V.1.3 は表 1 のようにテスト項目が 261 に及んでいる。うち動作テストが 227 項目、ドキュメントや品質の評価が 34 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。

検証セット V.1.3 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。

	項目数	最低点	最高点
C90 規格合致度	173	-162	448
C99 規格合致度	19	0	96
C++98 規格合致度	8	0	24
品質:診断メッセージ	45	0	64
品質:その他	16	-40	111
計	261	-202	743

表 1: 検証セット V.1.3 の項目数と配点

ＯＳ	処理系	実行プログラム (版数)	(1)	(2)	注
OS-9/6x09	Microware C/6809	DECUS cpp	256	318	*1
MS-DOS		JRCPPCHK (V.1.00B)	400	449	*2
WIN32	Borland C++ V.4.02J	cpp32	412	458	*3
DJGPP V.1.12 M4	GNU C 2.7.1	cpp	452	556	*4
MS-DOS	LSI C-86 V.3.30c	cpp (改造版 beta13)	342	400	*5
FreeBSD, WIN32, etc.	GNU C, Borland C, etc.	cpp (V.2.0)	502	655	*6
WIN32	Borland C++ V.5.5	cpp32	412	465	*7
WIN32	LCC-Win32 V.3.6	lcc	392	479	*8
Linux, etc		ucpp (V.0.7)	423	491	*9
Linux, FreeBSD	GNU C 2.95.3	cpp	476	581	*10
Linux, FreeBSD, etc.	GNU C, LCC-Win32, etc.	cpp (V.2.3)	562	717	*11
				(1) 規格合致度 (2) 総合評価	

表 2: 各種プリプロセッサの検証結果

*1 Martin Minow による DECUS cpp のオリジナル版 (1985/06) の shift-JIS に対応した OS-9/09 への移植版 (1989/04)。 [8]

*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。 [9]

*3 1993 年のものの日本語版 (1994/12)。 [10]

*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 SJIS に対応。 [11]

*5 LSI C-86 / cpp のきだあきらによる改造版 (1996/02)。 [12]

*6 松井 潔による 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, OS-9/09 / Microware C 等に対応。各種の動作モードの cpp 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)。 [13]

*8 Jacob Navia 等による shareware (2000/09)。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。 [14]

*9 Thomas Pornin による portable な free software (2000/10)。Stand-alone のプリプロセッサ。 [15]

*10 VineLinux 2.5, FreeBSD 4.4 で使われている GNU C 2.95.3 (2001/03)。 [16]

*11 松井 潔による free software の V.2.3 (開発中)。Linux / GNU C, CygWin, LCC-Win32 等への対応が追加されている。

このように、松井版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。４年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.2.3 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。

松井版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 の新しい仕様の多く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（-pedantic は C90 に対応するものであり、C99, C++98 には使えないという問題もある)。
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 -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)。

松井版 cpp 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。

4.1 プリプロセッサによるソースチェックの影響力

私の cpp 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 の kernel および libc ソースに適用した結果は、cpp V.2.2 の cpp22_man.doc [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、現在開発中の cpp V.2.3 を Linux の glibc 2.1.3 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、かつては UNIX 系のオープンソースにしばしば見られたものの、最近で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。

4.2 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。

4.2.1 行をまたぐ文字列リテラル

```
#define ELF_MACHINE_RUNTIME_TRAMPOLINE asm ("\
    .text
    .globl _dl_runtime_resolve
    etc. ...
");
```

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、それを使ってこう書くことができる。

```
#define ELF_MACHINE_RUNTIME_TRAMPOLINE asm ("\
    .text\n\
    .globl _dl_runtime_resolve\n\
    etc. ...\n\
");
```

ディレクティブ行に限らない一般性のある書き方は、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。

```
#define ELF_MACHINE_RUNTIME_TRAMPOLINE asm ("\t"    \
    ".text\n\t"                                     \
    ".globl _dl_runtime_resolve\n\t"                \
    "etc. ...\n");
```

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。

4.2.2 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コメント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。

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。この形であればディレクティブ行も、#include 以外なら挟んで問題ない。

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る。

4.2.3 `defined' に展開されるマクロ

```
#define HAVE_MREMAP defined(__linux__) && !defined(__arm__)
```

というマクロ定義があり、

```
#if HAVE_MREMAP
```

というふうに使われている。しかし、#if 式中でマクロ展開の結果に defined 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、このマクロはまずこう置換され、

```
defined(__linux__) && !defined(__arm__)                    (1)
```

__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こうなる。

```
defined(1) && !defined(__arm__)
```

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 (1) 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。このマクロはこう書けば問題ない。

```
#if defined(__linux__) && !defined(__arm__)
#define HAVE_MREMAP 1
#endif
```

4.2.4 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。

```
#define CHAR_CLASS_TRANS SWAPU16
```

というオブジェクト型マクロの定義があり、SWAPU16 の定義を見るとこうなっているというものである。

```
#define SWAPU16(w) ((((w) >> 8) & 0xff) | (((w) & 0xff) << 8))
```

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書いたほうが良いであろう。

```
#define CHAR_CLASS_TRANS(w) SWAPU16(w)
```

4.2.5 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 / cpp の 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。

5 Ｃｐｐの実装方法

私が cpp 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、ほぼ達成できたと思っている。

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。

多くの処理系に実装してきたことも、cpp 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 cpp 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。

5.1 トークンベースの原則

さらに cpp 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。

私の cpp 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。

私の cpp 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 / 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 cpp 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。

5.2 関数型マクロの関数的展開

私の cpp 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（骨格を書いたのは１０年前のことであるが）。

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.doc [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。4.2.4 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、引数内のマクロは先に展開されてから置換リスト中のパラメータ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、引数をパラメータと置き換えてから、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。

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。これは必要に応じて途中まで戻って再走査し、対象を後ろにずらしながら再走査を繰り返してゆくものである。GNU C / 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、この中からプリプロセスディレクティブの処理ルーチン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。この構造は、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。4.2.3 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 / 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)。

私の cpp 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。

6 今後のｕｐｄａｔｅ計画

私のプリプロセッサと検証セットは４年前のバージョンでも、プリプロセスに関してやるべきこと・やって意味のあることの大半をやったつもりであるが、それでもまだ多くの課題が残っている。それは、既存のプリプロセッサとの互換性を高めて使い勝手を良くすることや、文書の英語版を作成すること等であり、この cpp を世に出してゆくために必要な諸整備が中心である。現在、cpp V.2.3 と検証セット V.1.3 では次のような updates が進行中である。

1. 正式決定した C99 に合わせる（これは cpp では実行時オプションにとどめ、デフォルトの仕様にはしないが)。
2. GNU C 用は GNU C / cpp との互換性を向上させる。この互換性というのは多くは後ろ向きのものであり、規格準拠性や明快さとは矛盾するが、これがないと GNU C / cpp と置き換えて使うには不便である。そこで、実行時オプションと #pragma によってこれを実装し、それを使うソースに対してウォーニングは出すがエラーにはしないという仕様で両立を図る。
3. 他の処理系用の版も含めて、エラーの一部をウォーニングに格下げしたり、ウォーニングのクラスを増やしたりして、使い勝手（互換性）の向上を図る。
4. これまで対応してきた処理系のバージョンを update する。GNU C 3.x にも対応させる。また、GNU C 3.x / cpp のソースと test-suite を検討し、その開発に何らかの形でフィードバックを図る。
5. 対応する処理系を増やす。Linux / GNU C にも対応させる（FreeBSD / GNU C とほとんど同じ設定ですむ)。CygWin, LCC-Win32 等も追加する。また、従来は free の処理系が中心であったが、今後は市販の主要な処理系をいくつか購入して、私の cpp を対応させたい。各処理系付属のプリプロセッサのテストも update または追加する。
6. 若干のバグフィックスをする。
7. ドキュメントも update する。
8. Linux 用は rpm、FreeBSD 用は ports, package の形でもまとめる。
9. ドキュメントの英訳を翻訳会社に委託したい。処理系の大半がアメリカ製となっている現在では、この種のソフトウェアが普及するカギは英語のドキュメントの存在だと思われるからである。 英語のドキュメントを含む Linux の rpm や FreeBSD の ports が distribution に収録されることを期待したい。

7 おわりに

私の cpp は、Martin Minow による DECUS cpp (1985) という古い public domain software から出発したものである。今ではソースの量は３倍になり、規格の仕様に合わせてプログラム構造

を全面的に作り直しており、DECUS cpp のソースは痕跡をとどめる程度になっている。ドキュメントや検証セットはすべて私が新しく書いたものである。

私が DECUS cpp をいじり始めたのは 1992 年のことである。実に１０年間もこの cpp をいじってきていることになる。しかし、いくら完成度を高めても、その成果を世に問う機会がなく、残念な思いをしてきた。

このたび、情報処理振興事業協会 (IPA) の平成１４年度「未踏ソフトウェア創造事業」に新部プロジェクトマネージャーによって採択され、ようやく世に出る機会を与えられた。この機会にできるだけ多くの方に私のプリプロセッサを知ってもらいたく、Linux Conference で発表させていただくことにした。

私のプリプロセッサと検証セットの旧版は vector と @nifty で公開している。

```
http://www.vector.co.jp/
```

をアクセスし、Yahoo! で 「松井 潔」をキーワードとして vector 内をサーチすると、登録されている私のソフトがすべて出てくる。

@nifty では「C言語フォーラム (FC)」の Lib2 にある。

なお、cpp V.2.3 開発のための web page を近日中に開設し、そこでオープンな形で開発を進めてゆく予定である。多くの方のコメント・感想・討論・開発参加をお願いしたい。

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

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

[9] J. Roskind, JRCPPCHK. 現在は所在不明

[10] Borland International Inc., ボーランド株式会社. *Borland C++ V.4.0*. 1994.

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

[12] きだあきら, LSI C-86 / cpp 改造版 beta13.
かつては *C Magazine* 誌の付録 CD-ROM に入っていたが....

[13] Borland Software Corp., ボーランド株式会社., Borland C++ Compiler 5.5
http://www.borland.co.jp/cppbuilder/freecompiler/
[14] Jacob Navia., LCC-Win32.
http://www.q-software-solutions.com/lccwin32/
[15] Thomas Pornin., ucpp. http://pornin.nerim.net/ucpp/
[16] Free Software Foundation, GCC. http://gcc.gnu.org/