高品質のCプリプロセッサ **mcpp**

松井　潔
kmatsui@t3.rim.or.jp

2007 年 2 月 26 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。

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スが逆に portability を損なう結果も生んでしまっている。また、プリプロセス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** はオープンソースの 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

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。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[5–7] 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いて、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。

さらに、プリプロセスがコンパイルの「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

ではプリプロセス・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。プリプロセス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見当たらない。

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。

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スの portability を確保することができる。また、プリプロセス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プリプロセス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プリプロセス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[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、（もし可能なら）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 compiler-independent 版のプリプロセッサを作ることもできる。また、何らかの他のメインプログラムからサブルーチンとして呼び出されるようにコンパイルすることもできる。

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 testsuite 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.html: 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.html: 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.html: 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 プリプロセスの基礎的規定

本題に入る前に、C/C++ プリプロセスの基礎的な規定に手短に触れておく。

3.1 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、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。

表 1　検証セット V.1.5.3 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

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.1.5.3 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 にない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.1.5.3 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、次のような問題がある。

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	366	20	6	556	18	72	646
GCC 2.95.3 [*4]	2001/03	166	404	56	6	632	24	113	769
GCC 3.2 [*5]	2002/08	166	419	86	20	691	32	117	840
ucpp 1.3 [*6]	2003/01	166	384	88	9	647	25	88	760
Visual C 2003 [*7]	2003/04	156	394	43	15	608	21	83	712
LCC-Win32 2003-08 [*8]	2003/08	158	376	18	6	558	19	84	661
Wave 1.0.0 [*9]	2004/01	140	338	53	18	549	21	79	649
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	38	120	846
Visual C 2005 [*12]	2005/09	160	399	65	17	641	20	77	738
LCC-Win32 2006-03 [*13]	2006/03	156	374	22	6	558	22	85	665
GCC 4.1.1 [*14]	2006/05	166	417	87	20	690	38	120	848
mcpp 2.6.3 [*15]	2007/02	166	432	98	22	718	74	136	928
最高点		166	432	98	26	722	74	164	960

1. 診断メッセージが不十分である。
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。
4. 規格と矛盾する独自仕様が多い（拡張仕様は -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。

[*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: 筆者によるオープンソース・ソフトウェアの 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 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 なオープンソース・ソフトウェア。Compiler-independent なプリプロセッサ。[14]

[*7] Visual C 2003: Visual C++ .net 2003。Microsoft。

[*8] LCC-Win32 2003-08: Jacob Navia の開発した処理系。プリプロセス部分は Dennis Ritchie が C90 対応のプリプロセッサとして書いたものを元にしている。

[*9] Wave 1.0.0: Hartmut Kaiser によるオープンソース・ソフトウェア。Paul Mensonides 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 2003-08, 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。

[*12] Visual C 2005: Visual C++ 2005 Express Edition。Microsoft。[15]

[*13] LCC-Win32 2006-03: LCC-Win32 の 2006/03 のバージョン。[16]

[*14] GCC 4.1.1: Linux 上で筆者がコンパイルしたもの。[13]

[*15] **mcpp** 2.6.3: V.2.4 以降、GCC 3.3-4.1, Visual C++ 2005 への対応が追加されている。MinGW/GCC にも移植された。[18]

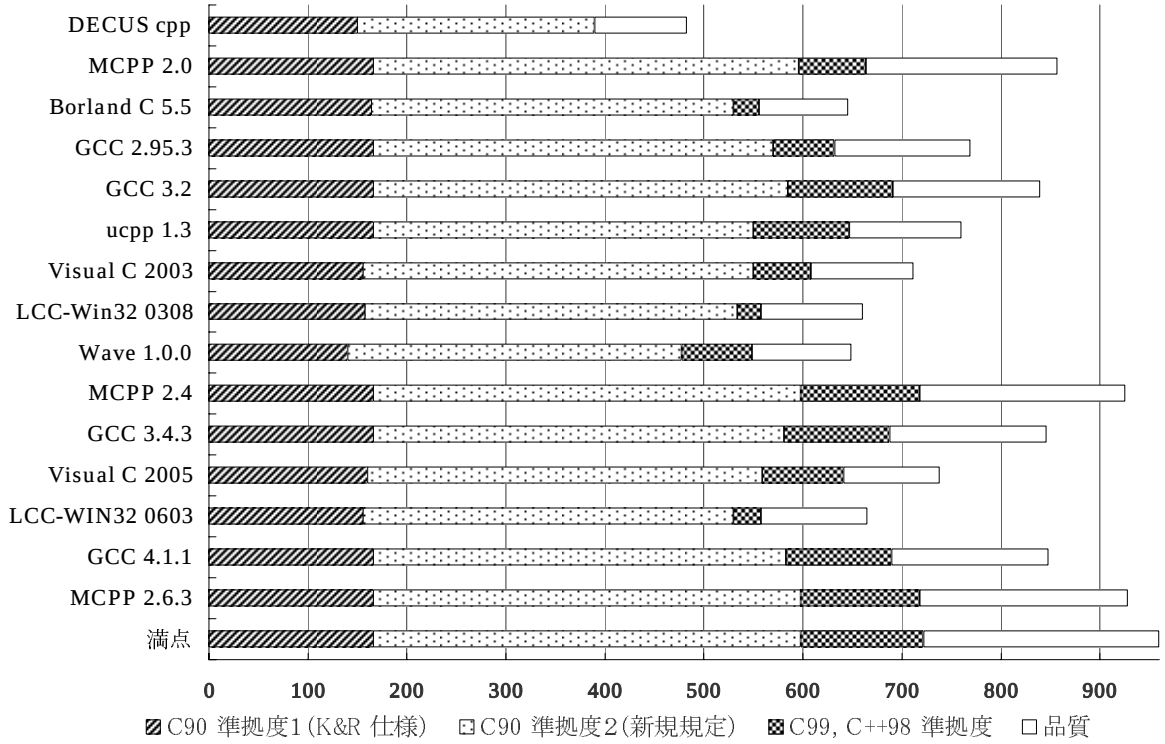

図 1　各種プリプロセッサの検証結果

#pragma で実装すべきものである)。

GCC V.3,4 / 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、

```
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

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. 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. プリプロセスの手順 (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> をプリプロセス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が、出さない場合も多く、一貫していない。

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

```
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

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。

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 2.2.2 (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.html 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 (2000/02) 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、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。

最近、glibc 2.4 (2006/03) のソースを **mcpp** でチェックしてみたところ、glibc 2.1.3 に見られた portability の問題点は、6 年経ってもほとんど解消されていないことがわかった。「行をまたぐ文字

列リテラル」のような一部の問題は消失したものの、それ以外の GCC の local な仕様に依存するソースはむしろ大幅に増加しているのである。

6.2 glibc のソースを例に

glibc 2.4 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。

6.2.1 プリプロセスを要する *.S ファイル

*.S ファイルというのは、アセンブラソースの中に `#if, #include` 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。マクロが埋め込まれているものもある。

アセンブラソースはCの token sequence の形を成していないので、これをCプリプロセッサで処理するのは危険がある。プリプロセッサは C では文字列リテラルや文字定数以外では使わない %, $ 等の文字もそのまま通し、space の有無も原則としてそのまま維持しなければならない。

図 4 の example-1 はある *.S ファイルの一部である。`#ifdef SHARED` は C の directive であるが、各行の後半の # で始まる部分はコメントのつもりなのである。しかし、`# column.` のように行の最初の non-white-space-character が # の場合は、無効な directive と構文上、区別がつかない。`# + DW_EH_PE_sdata4` に至っては C では構文エラーになるものである。

同じソースには example-2 のような行もあるが、これはマクロ呼び出しである。マクロの定義はここでは省略するが、このマクロ呼び出しは example-3 のように展開されることが期待されている。ここで `pthread_cond_wait@@GLIBC_2.3.2` という部分は ## 演算子によっていくつかの部分が連結されて生成されるものであるが、こういう @ を含む token (pp-token) はCには存在しない。

このソースには標準Cのプリプロセスでは文法エラーや undefined となる部分がふんだんにあり、エラーにならない部分でさえも、アセンブラソースとして期待する結果になることはまったく保証されない。

アセンブラ用のコードをCで処理するのであれば、asm() 関数を使って、アセンブラコードの部分を文字列リテラルに埋め込み、*.S ではなく *.c ファイルとすべきである。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。しかし、マクロが埋め込まれているものは通常は asm() では対処できない。この種のソースは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。

6.2.2 GCC 仕様の可変引数マクロ

GCC には C99 以前から独自の仕様の可変引数マクロがある。図 5 の example-1 のようなものである。さらに GCC 3 では example-2 のような書き方も実装された。GCC 2.95.3 以降では C99 の仕様の可変引数マクロも実装されているが、これは example-3 のようになる。

しかし、GCC 仕様はどちらもきわめて特殊なものであり、Cf99 のものと 1 対 1 には対応しない。C99 では可変引数に対応する実引数は少なくとも 1 つ必要であるのに対して、GCC 仕様では実引数の欠如も許容され、しかも、## はここではトークン連結演算子ではなく、実引数が欠如している場合にその直前のコンマを削除するという特殊な機能を持っている。古い GCC 仕様に至っては `args...` といった「名前付き可変引数」を使うが、こういうトークンはCには存在しない。

そして、glibc で使われているのはすべて古い GCC 仕様の可変引数マクロであり、C99 仕様のものはおろか GCC 3 仕様のものさえも使われていない。実際のマクロ呼び出しでは可変引数が欠如していてコンマが削除されるものがきわめて多い。すなわち、古い仕様がいつまでも使われているのである。

可変引数マクロは C99 の仕様が無理がなく portable でもある。マクロ呼び出しで可変引数が不要の場合は `0` や `NULL` といった無害な引数を使うのが良い。

```
example-1
        .byte   8               # Return address register
                                # column.
    #ifdef SHARED
        .uleb128 7              # Augmentation value length.
        .byte   0x9b            # Personality: DW_EH_PE_pcrel
                                # + DW_EH_PE_sdata4

example-2
    versioned_symbol (libpthread, __pthread_cond_wait, pthread_cond_wait,
                      GLIBC_2_3_2)

example-3
    .symver __pthread_cond_wait, pthread_cond_wait@@GLIBC_2.3.2
```

図 4　glibc のソースの例 1: ∗.S ファイル

```
example-1
    #define libc_hidden_proto(name, attrs...)   hidden_proto (name, ##attrs)

example-2
    #define libc_hidden_proto(name, ...)    hidden_proto (name, ## __VA_ARGS__)

example-3
    #define libc_hidden_proto(name, ...)    hidden_proto (name, __VA_ARGS__)
```

図 5　glibc のソースの例 2: GCC 仕様の可変引数マクロ

```
example-1
    #define _G_HAVE_ST_BLKSIZE defined (_STATBUF_ST_BLKSIZE)

example-2
    #if _G_HAVE_ST_BLKSIZE

example-3
    defined (_STATBUF_ST_BLKSIZE)

example-4
    #define   _STATBUF_ST_BLKSIZE

example-5
    defined ()

example-6
    defined (0)

example-7
    #if defined (_STATBUF_ST_BLKSIZE)
    #define _G_HAVE_ST_BLKSIZE 1
    #endif

example-8
    #if defined (_STATBUF_ST_BLKSIZE)

example-9
    #pragma MCPP debug expand token
    #if _G_HAVE_ST_BLKSIZE
    #pragma MCPP end_debug
```

図 6　glibc のソースの例 3: defined に展開されるマクロ

```
example-1
    #define _IO_close close_not_cancel

example-2
    #define close_not_cancel(fd) __close_nocancel (fd)

example-3
    #define _IO_close(fd)   close_not_cancel(fd)
```

図 7　glibc のソースの例 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

6.2.3 ‘defined’ に展開されるマクロ

図 6 の example-1 のようなマクロ定義があり、example-2 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-3 のように置換され、_STATBUF_ST_BLKSIZE が example-4 のように定義されている場合、最終的な展開結果は普通は example-5 のようになる。_STATBUF_ST_BLKSIZE が定義されていなければ example-6 のようになる。どちらも #if 式としては、もちろん syntax error である。

実際、GCC でも #if 行でなければこうなるが、ところが GCC は #if 行では example-3 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。このマクロ定義は example-7 のように書けば問題ない。あるいは _G_HAVE_ST_BLKSIZE というマクロを使わず、example-2 の #if 行を example-8 のように書くことである。

ちなみに **mcpp** では example-2 はエラーとなるが、example-9 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。

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。図 7 の example-1 のようなオブジェクト型マクロの定義があり、close_not_cancel の定義を見ると example-2 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-3 のように書いたほうが良い。

6.2.5 GCC の動作の細部に依存するソース

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 の動作の細部に依存しているものが散見される。プリプロセス後の出力の行頭やトークン間の space の数まで GCC の動作を前提としているものさえある。

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一部であるが、GCC がデフォルトでウォーニングを出していれば、こうしたソース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。

オープンソース・ソフトウェアのプリプロセス上の portability の問題の多くは、90 年代中ごろまでは C90 以前の UNIX 上の処理系の traditional な

仕様をひきずったものであったが、現在はそうした極端に古い書き方のソースは大幅に減少し、代わって GCC の local な仕様に依存したものが大半を占めるようになっている。

そして、glibc では数年を経過してもそうした部分は減少せず、むしろ増加する傾向を見せている。大規模なソフトウェアでは多くのソースファイルが絡み合うために、書き直しが困難になり、新しいソースもそれに合わせて書かれるためだと思われる。GCC のほうも仕様の変更は影響が大きくなるために、簡単にはできなくなる。GCC も glibc もどこかで思い切った整理が必要だと思われる。

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-test.html の 2.7 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ようである。

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.html の 2.7.6 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。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。**mcpp** 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現されている。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.html 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。

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。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

mcpp V.2.6 は 2005/03 にリリースした V.2.5 に次のような updates を加えたものであり、2006/07 にリリースされた。

1. Compiler-independent 版の動作をどの処理系にも依存しないようにした。
2. 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。
3. GCC V.4.0, Visual C++ 2005 に対応させた。

その後、2007/02 までに V.2.6.1, V.2.6.2, V.2.6.3 がリリースされている。その間に MinGW への移植、GCC 版の GCC との互換性の強化、**mcpp** をサブルーチンとして使うサブルーチン版の実装、テキストファイルのドキュメントの html への変換、多くのバグフィックス、等々の改良が加えられた。

8.2　V.2.7 の計画

mcpp 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. **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。
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. 検証セットには **mcpp** 固有の仕様をテストす

るための一連の testcases を追加する。

9 おわりに

最高度の規格準拠性を持ち、かつプリプロセスレベル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、プリプロセスに関しては十分なものが存在しなかったので、プリプロセス検証セットを **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ス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/devel/`

[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/`