Oracle® Solaris Studio 12.2: C ユーザーガイド

Part No: 821–2496
2010 年 9 月

ORACLE®

110412@25097

目次

はじめに

『Oracle Solaris Studio 12 Update 2 C ユーザーズガイド』では、Oracle Solaris Studio C コンパイラ、**cc** に関する環境とコマンド行オプションについて説明します。このマニュアルは、C 言語に関する実用的な知識を持ち、Solaris Studio C コンパイラの効率的な使用法を学ぼうとしているプログラマを対象に書かれています。Solaris または Linux オペレーティング環境に関する全般的な知識があることも想定されています。

サポートされるプラットフォーム

Oracle Solaris Studio のこのリリースは、SPARC および x86 ファミリ (UltraSPARC、SPARC64、AMD64、Pentium、Xeon EM64T) プロセッサアーキテクチャーを使用するシステムをサポートしています。ご使用の Oracle Solaris オペレーティングシステムのバージョンに対するシステムのサポート状況は、ハードウェア互換性リスト (`http://www.sun.com/bigadmin/hcl`) を参照してください。ここには、すべてのプラットフォームごとの実装の違いについて説明されています。

このドキュメントでは、x86 関連の用語は次のものを指します。

- 「x86」は、64 ビットおよび 32 ビットの x86 互換製品を指します。
- 「x64」は、AMD 64 または EM64T システムで、特定の 64 ビット情報を指します。
- 「32 ビット x86」は、x86 ベースシステムで特定の 32 ビット情報を指します。

サポートされるシステムについては、ハードウェアの互換性に関するリストを参照してください。

Solaris Studio ドキュメントへのアクセス方法

ドキュメントには、次の場所からアクセスできます。

- ドキュメントは、次に示すマニュアル索引のページからアクセスできます。`http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation`。

- IDE、パフォーマンスアナライザ、dbxtool、および DLight の全コンポーネントのオンラインヘルプは、これらのツール内の「ヘルプ」メニューだけでなく、F1 キー、および多くのウィンドウやダイアログにある「ヘルプ」ボタンを使用してアクセスできます。

アクセシブルな製品ドキュメント

ドキュメントは、技術的な補足をすることで、ご不自由なユーザーの方々にとって読みやすい形式のドキュメントを提供しております。アクセシブルなドキュメントは次の表に示す場所から参照することができます。

ドキュメントの種類	アクセシブルな形式と格納場所
マニュアル	HTML 形式。docs.sun.com にある Oracle Solaris Studio 12.2 Collection - Japanese から選択
『Oracle Solaris Studio 12.2 リリースの新機能』(以前はコンポーネントの README ファイル)	HTML 形式。docs.sun.com にある Oracle Solaris Studio 12.2 Collection - Japanese から選択
マニュアルページ	`man` コマンドを使用して Oracle Solaris 端末に表示されます。
オンラインヘルプ	HTML 形式。IDE、dbxtool、DLight、およびパフォーマンスアナライザの「ヘルプ」メニュー、「ヘルプ」ボタン、および F1 キーを使用して表示
リリースノート	HTML 形式。docs.sun.com にある Oracle Solaris Studio 12.2 Collection - Japanese から選択

関連するサードパーティの **Web** サイトリファレンス

このマニュアルには、詳細な関連情報を提供するサードパーティの URL が記載されています。

注 – このマニュアルで紹介するサードパーティ Web サイトが使用可能かどうかについては、Oracle は責任を負いません。このようなサイトやリソース上、またはこれらを経由して利用できるコンテンツ、広告、製品、またはその他の資料についても、Oracle は保証しておらず、法的責任を負いません。また、このようなサイトやリソースから直接あるいは経由することで利用できるコンテンツ、商品、サービスの使用または依存が直接のあるいは関連する要因となり実際に発生した、あるいは発生するとされる損害や損失についても、Oracle は一切の法的責任を負いません。

開発者向けのリソース

http://www.oracle.com/technetwork/server-storage/solarisstudio を参照して、次の頻繁に更新されるリソースを確認してください。

- リソースは頻繁に更新されます。
- ソフトウェアのドキュメント、およびソフトウェアとともにインストールされる一連のドキュメント
- Oracle Solaris Studio ツールを使用して行う開発タスク全体を順を追って説明するチュートリアル
- サポートレベルに関する情報
- http://forums.sun.com/category.jspa?categoryID=113 にあるユーザーフォーラム

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P–1　表記上の規則

字体または記号	意味	例
`AaBbCc123`	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	`.login` ファイルを編集します。 `ls -a` を使用してすべてのファイルを表示します。 `system%`
`AaBbCc123`	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	`system%` **`su`** `password:`
AaBbCc123	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、`rm` *filename* と入力します。
『』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。
「」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第 5 章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。

表 P–1 表記上の規則 (続き)

字体または記号	意味	例
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合に、継続を示します。	`sun% grep '^#define \` `XV_VERSION_STRING'`

コード例は次のように表示されます。

- C シェル

  `machine_name% command y|n [filename]`

- C シェルのスーパーユーザー

  `machine_name# command y|n [filename]`

- Bourne シェルおよび Korn シェル

  `$ command y|n [filename]`

- Bourne シェルおよび Korn シェルのスーパーユーザー

  `# command y|n [filename]`

[ ] は省略可能な項目を示します。上記の例は、*filename* は省略してもよいことを示しています。

| は区切り文字 (セパレータ) です。この文字で分割されている引数のうち 1 つだけを指定します。

キーボードのキー名は英文で、頭文字を大文字で示します (例: Shift キーを押します)。ただし、キーボードによっては Enter キーが Return キーの動作をします。

ダッシュ (-) は 2 つのキーを同時に押すことを示します。たとえば、Ctrl-D は Control キーを押したまま D キーを押すことを意味します。

ドキュメント、サポート、およびトレーニング

追加リソースについては、次の Web サイトを参照してください。

- ドキュメント (http://docs.sun.com)
- サポート (http://www.oracle.com/us/support/systems/index.html)
- トレーニング (http://education.oracle.com) – 左のナビゲーションバーにある Sun リンクをクリックします。

ご意見の送付先

ドキュメントの品質や使いやすさに関するご意見やご提案をお待ちしています。間違いやその他の改善すべき箇所がありましたら、http://docs.sun.com で「Feedback」をクリックしてお知らせください。ドキュメント名とドキュメントの Part No.、および、可能な場合は章、節、ページ番号を記載してください。返答が必要な場合はお知らせください。

Oracle 技術ネットワーク (http://www.oracle.com/technetwork/index.html) では、Oracle ソフトウェアに関するさまざまなリソースを提供しています。

- 技術上の問題やソリューションについては、ディスカッションフォーラム (http://forums.oracle.com) を参照してください。
- 実践的なステップ・バイ・ステップのチュートリアルについては、Oracle By Example (http://www.oracle.com/technology/obe/start/index.html) を参照してください。
- サンプルコードのダウンロードについては、サンプルコード (http://www.oracle.com/technology/sample_code/index.html) を参照してください。

◆ ◆ ◆ 第 1 章

1 C コンパイラの紹介

この章では、Oracle Solaris Studio C コンパイラに関する基本的な情報を紹介します。

1.1 Solaris Studio 12 Update 2 リリースでの Version 5.11 の新機能

C コンパイラの現在のリリースには、次の新機能と変更された機能があります。

ABI の変更により再コンパイルが必要に
C コンパイラの変更により、複雑な型が含まれる構造体が 64 ビットモードの SPARC プロセッサで渡されて返される方法が修正されます。以前は、これらの構造体の値は間違ったレジスタで渡されて返され、gcc により作成されるバイナリと互換性がないバイナリが作成されることがありました。この変更は、C コンパイラに実装されている既存の ABI の要素に影響を与えるため、複雑なフィールドを持つ構造体を使用するソースファイルがアプリケーション内にある場合は、間違った応答が返される可能性をなくすため、そのアプリケーションのソースベース全体を再コンパイルする必要があります。32 ビットの SPARC プロセッサ向けと、32 ビットまたは 64 ビットの x86 プロセッサ向けのコンパイルは、この変更の影響を受けません。

- SPARC-V9 ISA の SPARC VIS3 バージョンのサポートが追加されました。`-xarch=sparcvis3` オプションを使用してコンパイルすると、SPARC-V9 命令セットの命令、UltraSPARC および UltraSPARC-III 拡張機能、積和演算 (FMA) 命令、および VIS (Visual Instruction Set) バージョン 3.0 をコンパイラが使用できるようになります。(259 ページの「B.2.74 -xarch=*isa*」)
- x86 ベースのシステムに基づく `-xvector` オプションのデフォルト値が `-xvector=simd` に変更されました。(329 ページの「B.2.157 -xvector[=*a*]」)
- AMD SSE4a 命令セットのサポートが `-xarch=amdsse4a` オプションで使用できるようになりました。(259 ページの「B.2.74 -xarch=*isa*」)

- `-traceback` オプションを使用すると、重大なエラーが発生した場合に実行可能ファイルでスタックトレースを出力できます。(251 ページの「B.2.62 `-traceback[={ %none|common|signals_list}]`」)
- `-mt` オプションが `-mt=yes` または `-mt=no` に変更されています。(249 ページの「B.2.50 `-mt[={ yes|no}]`」)
- `#warning` コンパイラ指令により、指令内のテキストが警告として発行され、コンパイルが続行されます。(45 ページの「2.10 警告とエラー」)
- 新しいプラグマ `does_not_read_global_data`、`does_not_write_global_data`、および `no_side_effect` が追加されました。(45 ページの「2.11 プラグマ」)
- ヘッダーファイル `mbarrier.h` を使用できるようになりました。このヘッダーファイルは、SPARC プロセッサと x86 プロセッサでマルチスレッドコードのさまざまなメモリーバリアー組み込み関数を定義します。(97 ページの「3.9 メモリーバリアー組み込み関数」)
- `-xprofile=tcov` オプションが拡張されて、オプションのプロファイルディレクトリパス名がサポートされるようになりました。また、`tcov` 互換のフィードバックデータも生成できます。(312 ページの「B.2.136 `–xprofile=p`」)
- このリリースでは、`-xMD` オプションと `-xMMD` オプションにより記述された依存関係ファイルにより、既存のファイルがすべて上書きされます。(291 ページの「B.2.111 `-xMD`」)

1.2 x86 の特記事項

x86 Solaris プラットフォーム用にコンパイルを行う場合に注意が必要な、重要な事項がいくつかあります。

`-xarch` を `sse`、`sse2`、`sse2a`、または `sse3` 以降に設定してコンパイルしたプログラムは、必ずこれらの拡張と機能を提供するプラットフォームでのみ実行してください。

Solaris 9 4/04 以降の Solaris OS リリースは、Pentium 4 互換プラットフォームでは SSE/SSE2 に対応しています。これより前のバージョンの Solaris OS は SSE/SSE2 に対応していません。`-xarch` で選択した命令セットが、実行中の Solaris OS で有効ではない場合、コンパイラはその命令セットのコードを生成またはリンクできません。

コンパイルとリンクを個別に行う場合は、必ずコンパイラを使ってリンクし、同じ `-xarch` 設定で正しい起動ルーチンがリンクされるようにしてください。

x86 の 80 ビット浮動小数点レジスタが原因で、x86 での演算結果が SPARC の結果と異なる場合があります。この差を最小にするには、`-fstore` オプションを使用するか、ハードウェアが SSE2 をサポートしている場合は `-xarch=sse2` でコンパイルします。

Solaris と Linux でも、固有の数学ライブラリ (たとえば、`sin(x)`) が同じではないため、演算結果が異なることがあります。

1.3 バイナリの互換性の妥当性検査

Solaris システムの Solaris Studio 11 以降では、Solaris Studio コンパイラによってコンパイルされたプログラムのバイナリには、そのコンパイル済みバイナリによって想定されている命令セットを示すアーキテクチャーハードウェアフラグが付いています。実行時にこれらのマーカーフラグが検査され、実行しようとしているハードウェアで、そのバイナリが実行できることが確認されます。

これらのアーキテクチャーハードウェアフラグを含まないプログラムを、適切な機能または命令セット拡張に対応していないプラットフォームで実行すると、セグメント例外、または明示的な警告メッセージなしの不正な結果が発生することがあります。

このことは、`.il` インラインアセンブリ言語関数を使用しているプログラムや、SSE、SSE2、SSE2a、SSE3 の命令、およびより新しい命令と拡張機能を利用している `__asm()` アセンブラコードにも当てはまります。

1.4 64 ビットプラットフォーム用のコンパイル

ILP32 32 ビットモデル用にコンパイルするには、`–m 32` オプションを使用します。ILP64 64 ビットモデル用にコンパイルするには、`–m64` オプションを使用します。

C 言語の int、long、およびポインタデータ型を指定する ILP32 モデルは、すべて 32 ビット拡張です。long およびポインタデータ型を指定する LP64 モデルは、すべて 64 ビット拡張です。Solaris および Linux OS は、LP64 メモリーモデルの大きなファイルや配列もサポートします。

`–m64` を使用してコンパイルを行う場合、結果の実行可能ファイルは、64 ビットカーネルを実行する Solaris OS または Linux OS の 64 ビット UltraSPARC または x86 プロセッサでのみ動作します。コンパイル、リンク、および 64 ビットオブジェクトの実行は、64 ビット実行をサポートする Solaris または Linux OS でのみ行うことができます。

1.5 準拠規格

このマニュアルで使用される C99 という用語は、ISO/IEC 9899:1990 の C プログラミング言語を表します。C90 という用語は、ISO/IEC 9899:1990 の C プログラミング言語を意味します。

Solaris プラットフォーム上では、このコンパイラは、C99 規格に完全に準拠しています (`-xc99=all,lib` –Xc を指定した場合)。

このコンパイラはまた、『Programming Language - C (ISO/IEC 9899:1999)』規格にも準拠しています。

このコンパイラは従来の K&R C (Kernighan と Rithcie、つまり ANSI C の前段階) もサポートしているため、ISO C への移行が容易に行えます。

C90 の実装固有の動作については、366 ページの「D.1.17 _Pragma」を参照してください。

C99 機能の詳細は、表 C–6 を参照してください。

1.6 C Readme ファイル

C コンパイラの readme ファイルでは、コンパイラに関する重要な情報について説明しています。これは、『Oracle Solaris Studio 12.2 リリースの新機能』ガイドの一部となりました。次の内容が含まれています。

- マニュアルの印刷後に判明した情報
- 新規および変更された機能
- ソフトウェアの非互換性
- 問題および解決方法
- 制限および互換性の問題

『新機能』ガイドを参照するには、`http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation` にある Oracle Solaris Studio 12.2 のドキュメントページにアクセスしてください。

1.7 マニュアルページ

オンラインマニュアル (`man`) ページは、コマンド、関数、サブルーチン、およびそれらの集合について即座に参照できるドキュメントを提供します。

次のコマンドを実行することにより、C コンパイラのマニュアルページを表示できます。

```
example% man cc
```

Cのドキュメント全体を通して、マニュアルページのリファレンスは、トピック名とマニュアルのセクション番号で表示されます。cc(1) を表示するには、man cc と入力します。たとえばieee_flags(3M) など、ほかのセクションを表示するには、man コマンドに -s オプションを使用します。

```
example% man -s 3M ieee_flags
```

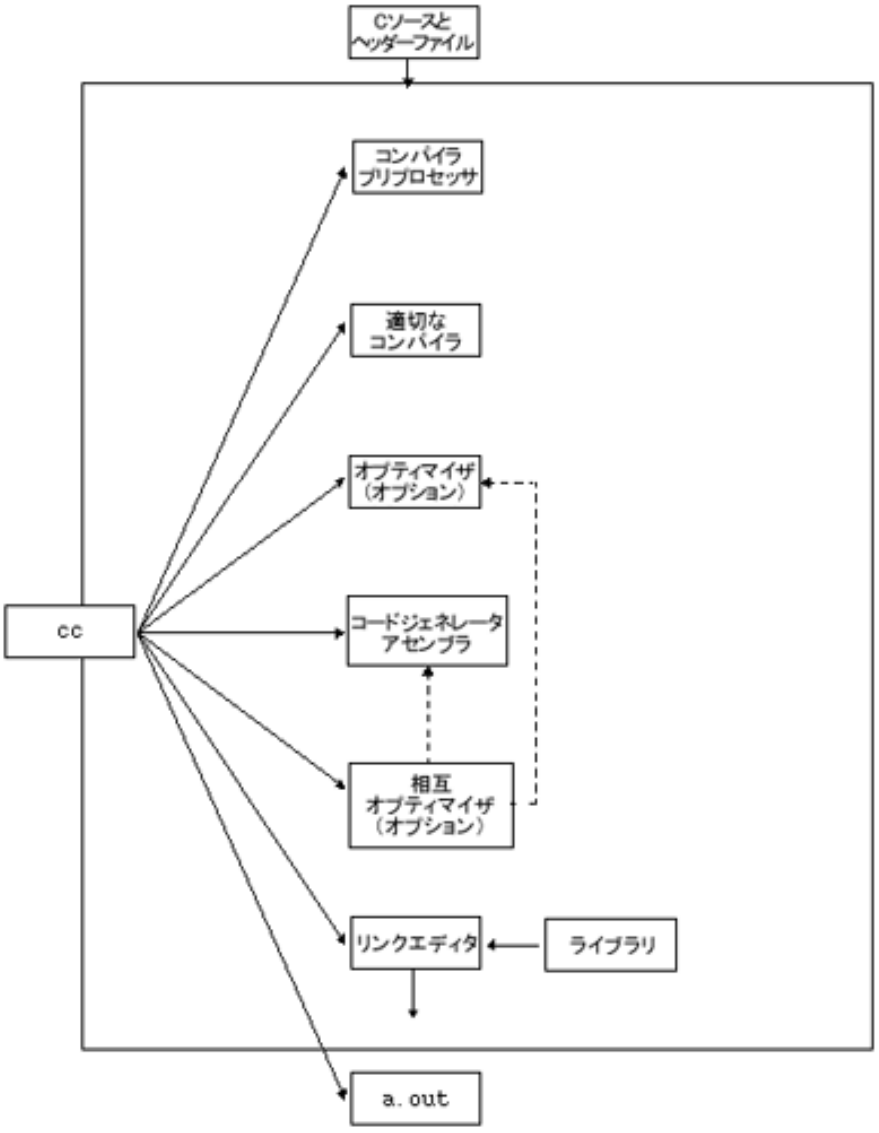
1.8 コンパイラの構成

Cコンパイルシステムはコンパイラ、アセンブラ、およびリンカーから構成されます。cc コマンドは、コマンド行オプションでほかの指定をしないかぎり、この3つの構成要素をそれぞれ自動的に起動します。

表 A–15 では、cc コマンドで使用できるオプションについて説明しています。

次の図にCコンパイルシステムの構成を示します。

図 1–1　C コンパイルシステムの構成



次の表は、コンパイルシステムの構成要素を要約したものです。

表 1–1　C コンパイルシステムの構成要素

コンポーネント	内容の説明	使用時の注意
cpp	プリプロセッサ (前処理系)	-Xs のみ
acomp	コンパイラ (-Xs 以外のモードではプリプロセッサが組み込まれている)	
ssbd	静的同期バグ検出	*(SPARC)*
iropt	コードオプティマイザ	-O、 -xO2、 -xO3、 -xO4、 -xO5、 -fas
fbe	アセンブラ	
cg	コード生成、インライン機能、アセンブラ	
ipo	内部手続きオプティマイザ	
postopt	ポストオプティマイザ	*(SPARC)*
ir2hf	中間コード翻訳	(x86)
ube	コードジェネレータ	(x86)
ld	リンカー	
mcs	コメントセクションの操作	-mr

1.9　C 関連のプログラミングツール

C プログラムの開発、保守、改良を行うときに役立つツールは多数あります。本書では、C にもっとも密接な 2 つのツール、cscope と lint について説明します。また、ツールごとにマニュアルページが用意されています。

◆ ◆ ◆ 第 2 章

2 C コンパイラ実装に固有の情報

この章では、C コンパイラに固有の部分について説明します。言語の拡張と環境に分けて説明します。

C コンパイラは、新しい ISO C 規格である ISO/IEC 9899-1999 で規定されている C 言語のいくつかの機能と互換性があります。従来の C 規格である ISO/IEC 9889-1990 規格 (および修正 1) と互換性のあるコードをコンパイルすることを希望する場合は、`-xc99=none` を使用します。その結果、コンパイラは ISO/IEC 9899-1999 規格による拡張を無視します。

2.1 定数

ここでは、Solaris Studio C コンパイラの定数に関する情報を掲載します。

2.1.1 整数定数

次の表に示すように、10 進数、8 進数、16 進数の定数に接尾辞を付けて型を示すことができます。

表 2–1 データ型の接尾辞

接尾辞	種類
`u` または `U`	`unsigned`
`l` または `L`	`long`
`ll` または `LL`	`long long` (`-xc99=none` や `-Xc` モードでは使用できません)
`lu`、`LU`、`Lu`、`lU`、`ul`、`uL`、`Ul`、`UL` のいずれか	`unsigned long`

表 2–1 データ型の接尾辞 (続き)

接尾辞	種類
llu、LLU、LLu、llU、ull、ULL、uLL、Ull のいずれか	unsigned long long (-xc99=none や -Xc モードでは使用できません)

-xc99=all を指定する場合、定数の大きさに応じて、次のリストから値が表現できる最初の型を使用します。

- int
- long int
- long long int

long long int で表現できる値の最大値を超えると、コンパイラは警告を発行します。

-xc99=none を指定すると、コンパイラが接尾辞を持たない定数の型を割り当てる場合、定数の大きさに応じて、次の中から値が表現できる最初の型を使用します。

- int
- long int
- unsigned long int
- long long int
- unsigned long long int

2.1.2 文字定数

エスケープシーケンスの発生しない複数バイト文字セットの値は、各文字の示す数値から派生しています。たとえば定数 '123' の持つ値は次のようになります。

0	'3'	'2'	'1'

あるいは 0x333231 です。

-Xs オプション使用の場合、あるいは ISO でないほかの C では、この値は次のようになります。

0	'1'	'2'	'3'

または 0x313233。

2.2 リンカースコープ指示子

次の宣言指示子を使用して、extern シンボルの宣言と定義を隠せます。これらの指示子を使うと、リンカースコープのマップファイルは使用しなくてすみます。また、コマンド行で -xldscope を指定して、変数スコープのデフォルト設定を制御することもできます。詳細については、286 ページの「B.2.101 -xldscope={*v*}」 を参照してください。

表 2–2 宣言指示子

値	意味
__global	シンボルは大域リンカースコープを持ち、このリンカースコープはもっとも制限の少ないリンカースコープです。シンボルに対する参照はすべて、シンボルを定義する最初の動的モジュール内の定義に結合します。このリンカースコープは、extern シンボルの現在のリンカースコープです。
__symbolic	シンボルはシンボリックリンカースコープを持ち、このリンカースコープは大域リンカースコープよりも制限されたリンカースコープです。リンクしている動的モジュール内のシンボルに対する参照はすべて、モジュール内に定義されたシンボルに結合します。モジュールの外側では、シンボルは大域と同じです。このリンカースコープはリンカーオプション -Bsymbolic に対応します。リンカーの詳細については、「ld(1)」を参照してください。
__hidden	シンボルは隠蔽リンカースコープを持ち、隠蔽リンカースコープは、シンボリックリンカースコープや大域リンカースコープよりも制限されたリンカースコープです。動的モジュール内の参照はすべて、そのモジュール内の定義に結合します。シンボルはモジュールの外側では認識されません。

オブジェクトまたは関数は、より制限された指示子で再宣言することはできますが、制限のよりゆるやかな指示子で再宣言することはできません。シンボルは一度定義したら、異なる指示子で宣言することはできません。

__global はもっとも制限の少ないスコープです。__symbolic はより制限されたスコープです。__hidden はもっとも制限の多いスコープです。

2.3 スレッドローカルな記憶領域指示子

スレッドローカルの変数を宣言して、スレッドローカルな記憶領域を利用します。スレッドローカルな変数の宣言は、通常の変数宣言に変数指示子 __thread を加えたものから成ります。詳細については、325 ページの「B.2.151 -xthreadvar[= *o*]」 を参照してください。

__thread 指示子は、コンパイル対象のソースファイルにあるスレッド変数の最初の宣言に含める必要があります。

__thread 指示子を使用できるのは、静的記憶領域を持つオブジェクトの宣言内だけです。スレッド変数を静的に初期化する方法は、静的記憶領域のほかのオブジェクトの場合と同じです。

__thread 指示子で宣言する変数は、__thread 指示子なしで宣言する場合と同じリンカー結合を持っています。初期設定子のない宣言など、一時的な定義が含まれます。

スレッド変数のアドレスは定数ではありません。したがって、スレッド変数のアドレス演算子 (&) は実行時に評価され、現在のスレッドのスレッド変数のアドレスが返されます。結果的に、静的記憶領域のオブジェクトはスレッド変数のアドレスに動的に初期化されます。

スレッド変数のアドレスは、対応するスレッドの有効期間の間は安定しています。変数の有効期間内は、プロセス内の任意のスレッドがスレッド変数のアドレスを自由に使用できます。スレッドが終了したあとは、スレッド変数のアドレスを使用できません。スレッドの終了後は、そのスレッドの変数のアドレスはすべて無効となります。

2.4 浮動小数点 (非標準モード)

IEEE 754 のデフォルトの浮動小数点演算機能は「無停止」であり、アンダーフローは「段階的」です。ここでは概要を説明します。詳細については『数値計算ガイド』を参照してください。

「無停止」とは、ゼロによる除算、浮動小数点のオーバーフロー、不正演算例外などが生じても実行を停止しないことを意味します。たとえば次の式で、x はゼロ、y は正の数であるとします。

```
z = y / x;
```

デフォルトでは、z の値は +Inf になりますが、プログラムの実行は続けられます。ただし、-fnonstd オプションを使用した場合は、このコードによってプログラムが終了します (コアダンプなど)。

次に、段階的アンダーフローの動作を説明するために、次のようなコードを例として考えます。

```
x = 10;
for (i = 0; i < LARGE_NUMBER; i++)
x = x / 10;
```

ループを 1 回通ると x は 1 になり、2 回目で 0.1、3 回目で 0.01 と続き、やがてはマシンによって値を表現できる許容範囲の下限に到達します。次にループを実行すると、どうなるのでしょうか。

表現可能な最小の数が 1.234567e-38 であると仮定します。

次にループを実行すると、仮数部から「盗んだ」ものを指数部に「与える」ことによって数値が修正され、新しい値 1.23456e-39 になります。その次はさらに、1.2345e-40 と続いていきます。これがデフォルト動作である「段階的アンダーフロー」です。非標準モードでは、仮数部から「盗む」ことをせず、通常は単に x をゼロに設定します。

2.5 値としてのラベル

C コンパイラは、計算型 goto 文として知られる C の拡張機能を認識します。計算型 goto 文を使用すると、実行時に分岐先を判別することができます。次のように演算子 '&&' を使用して、ラベルのアドレスを取得し、void * 型のポインタに割り当てることができます。

```
void *ptr;
...
ptr = &&label1;
```

あとに続く goto 文は、ptr により label1 に分岐できます。

```
goto *ptr;
```

ptr は実行時に計算されるため、ptr は有効範囲内にある任意のラベルのアドレスを取得でき、goto 文はこのアドレスに分岐することができます。

ジャンプテーブルを実装するには、計算型 goto 文を次の方法で使用します。

```
static void *ptrarray[] = { &&label1, &&label2, &&label3 };
```

これで、次のようにインデックスを指定して配列要素を選択できます。

```
goto *ptrarray[i];
```

ラベルのアドレスは、現在の関数スコープからしか計算できません。現在の関数以外のラベルについてアドレスを取得しようとすると、予測できない結果になります。

ジャンプテーブルは switch 文と同様の働きをしますが、両者にはいくつかの相違点があり、ジャンプテーブルではプログラムフローの追跡がより困難になる可能性があります。顕著な相違点は、switch 文のジャンプ先は必ず予約語から順方向にあることです。計算型 goto 文を使用してジャンプテーブルを実装すると、順方向と反対方向の両方に分岐することができます。

```
#include <stdio.h>
void foo()
{
```

```
  void *ptr;

  ptr = &&label1;

  goto *ptr;

  printf("Failed!\n");
  return;

  label1:
  printf("Passed!\n");
  return;
}

int main(void)
{
  void *ptr;

  ptr = &&label1;

  goto *ptr;

  printf("Failed!\n");
  return 0;

  label1:
  foo();
  return 0;
}
```

次の例では、プログラムフローの制御にジャンプテーブルを使用しています。

```
#include <stdio.h>

int main(void)
{
  int i = 0;
  static void * ptr[3]={&&label1, &&label2, &&label3};

  goto *ptr[i];

  label1:
  printf("label1\n");
  return 0;

  label2:
  printf("label2\n");
  return 0;

  label3:
  printf("label3\n");
  return 0;
}

%example: a.out
%example: label1
```

また、計算型 goto 文は、スレッド化されたコードのインタプリタとしても使用されます。高速ディスパッチを行うために、インタプリタ関数の範囲内にあるラベルアドレスを、スレッド化されたコードに格納することができます。

2.6 long long データ型

-xc99=none を指定してコンパイルを行う場合、Solaris Studio C コンパイラにはデータ型 long long および unsigned long long があり、これらはデータ型 long と類似しています。-m32 を使用してコンパイルすると、long long データ型には 64 ビットの情報が格納され、long には 32 ビットの情報が格納されます。-m64 を使用してコンパイルすると、long データ型には 64 ビットが格納されます。long long データ型は -Xc モードでは使用できません (警告が発行されます)。

2.6.1 long long データ型の入出力

long long データ型を出力または入力するには、変換指定子の前に ll の接頭辞を付けてください。たとえば、long long データ型を持つ変数 llvar を符号付き 10 進形式で出力するには、次のように指定します。

```
printf("%lld\n", llvar);
```

2.6.2 通常の算術変換

2 項演算子によっては、両方のオペランドの型を共通の型にするために変換することがあります。この時、結果の型も共通の型となります。この変換は通常の算術変換と呼ばれます。

- どちらか一方のオペランドが long double 型である場合、もう一方のオペランドは long double に変換されます。
- 一方のオペランドが double 型を持つ場合、もう一方のオペランドは double に変換されます。
- 一方のオペランドが float 型を持つ場合、もう一方のオペランドは float に変換されます。
- これ以外の場合は、汎整数拡張が両方のオペランドで実行されます。次の規則が適用されます。
  - 一方のオペランドが unsigned long long int 型を持つ場合、もう一方の演算子は unsigned long long int に変換されます。
  - 一方のオペランドが long long int 型を持つ場合、もう一方の演算子は long long int に変換されます。
  - 一方のオペランドが unsigned long int 型を持つ場合、もう一方の演算子は unsigned long int に変換されます。

- SPARC V9 でコンパイルする場合のみ、`cc -xc99=none` を指定するときは、一方のオペランドが `long int` 型を持ち、もう一方が `unsigned int` 型を持つ場合、両オペランドは `unsigned long int` に変換されます。
- 一方のオペランドが `long int` 型を持つ場合、もう一方のオペランドは `long int` に変換されます。
- 一方のオペランドが `unsigned int` 型を持つ場合、もう一方のオペランドは `unsigned int` に変換されます。
- これ以外の場合、両オペランドは `int` 型になります。

2.7 Switch 文内の Case 範囲

標準 C では、switch 文内にある case のラベルに、ただ 1 つの値を関連付けることができます。Solaris Studio C では、case 範囲として知られる、一部のコンパイラに見られる拡張を許可しています。

case 範囲は、値範囲を指定し、個別の case のラベルに関連付けます。case 範囲の構文は、次のとおりです。

`case` *low* **`...`** *high*`:`

case 範囲は、*low* ～ *high* で指定された範囲内にある各値に対して case ラベルを指定した場合とまったく同じ動作をします (*low* と *high* が等しい場合は、case 範囲はただ 1 つの値を指定します)。下限と上限の値は、C 規格の要件に準拠している必要があります。つまり、これらの値は、有効な整数型の定数式であることが必要です (C 規格 6.8.4.2)。case 範囲と case ラベルは、自由に混在することができ、単一の switch 文内で複数の case 範囲を指定できます。

プログラミングの例:

```
enum kind { alpha, number, white, other };
enum kind char_class(char c);
{
     enum kind result;
     switch(c) {
         case 'a' ... 'z':
         case 'A' ... 'Z':
             result = alpha;
             break;
         case '0' ... '9':
             result = number;
             break;
         case ' ':
         case '\n':
         case '\t':
         case '\r':
         case '\v':
             result = white;
```

```
            break;
        default:
            result = other;
            break;
    }
    return result; }
```

case ラベルに関する既存の要件に対してエラー条件を追加:

- low の値が high の値より大きい場合、コンパイラはエラーメッセージを生成してコードを拒否します (他のコンパイラの動作は一貫していないので、他のコンパイラでコンパイルした場合にプログラムが異なる方法で動作するわけではないことを保証する唯一の方法は、エラー状況を発生させることです)。
- ある case ラベルの値が、switch 文の中ですでに使用された case 範囲内の範囲に含まれている場合、コンパイラはエラーメッセージを生成してコードを拒否します。
- case 範囲どうしの範囲が重複している場合、コンパイラはエラーメッセージを生成してコードを拒否します。

case 範囲の終点 (上限または下限) が数値リテラルである場合、省略記号 (`...`) の前後に半角スペースを記述し、ピリオドのいずれかが小数点として扱われることを防止してください。

次に例を示します。

```
case 0...4;   // error
case 5 ... 9; // ok
```

2.8 表明 (assertion)

次の書式で指定します。

`#assert` *predicate (token-sequence)*

token-sequence は、表明の名前空間 (マクロ定義用の空間から分離されている) にある述語と関連付けられます。述語は識別子トークンでなければいけません。

`#assert` *predicate*

これは述語が存在していることを表明しますが、それにトークン列を関連付けることはしません。

コンパイラは、次のような事前定義された述語をデフォルトとして提供しています (`-Xc` モードを除く)。

```
#assert system (unix)
#assert machine (sparc)
```

```
#assert machine (i386)(x86)
#assert cpu (sparc)
#assert cpu (i386)(x86)
```

lint は、次のような事前定義された述語をデフォルトとして提供しています (-Xc モードを除く)。

```
#assert lint (on)
```

表明は #unassert を使用して削除できます。この場合、assert と同じ構文が使用されます。引数なしで #unassert を使用すると述語に対するすべての表明が削除され、表明を指定すればその表明だけが削除されます。

表明は、次の構文を持つ #if 文でテストすることができます。

```
#if #predicate(non-empty token-list)
```

たとえば次のように指定して、事前定義された述語 system をテストすることができます。

```
#if #system(unix)
```

これは真と評価されます。

2.9 サポートされる属性

次の属性 (__attribute__ ((*keyword*))) は、互換性のためにコンパイラにより実装されます。

- **always_inline** — #pragma inline および -xinline と同等です。
- **noinline** — #pragma no_inline および -xinline と同等です。
- **pure** — #pragma does_not_write_global_data と同等です。
- **const** — #pragma no_side_effect と同等です。
- **malloc** — #pragma returns_new_memory と同等です。
- **aligned** — #pragma align とほぼ同等ですが、構造体と共用体に限定されます。
- **constructor** — #pragma init と同等です。
- **destructor** — #pragma fini と同等です。
- **alias** — 名前を、宣言された関数または変数名の別名にします。
- **weak** — #pragma weak と同等です。
- **noreturn** — #pragma does_not_return と同等です。
- **packed** — #pragma pack() と同等です。

- **visibility** — 37 ページの「2.2 リンカースコープ指示子」で説明されているように、リンカースコープを提供します。構文は、`__attribute__((visibility("`*visibility_type*`")))` です。ここで、*visibility_type* は次のいずれかです。
  - `default` — `__global` リンカースコープと同じ
  - `hidden` — `__hidden` リンカースコープと同じ
  - `internal` — `__symbolic` リンカースコープと同じ

2.10 警告とエラー

`#error` および `#warning` プリプロセッサディレクティブを使用すると、コンパイル時の診断を生成できます。

`#error` *token-string*	エラー診断 *token-string* を発行して、コンパイルを終了します。
`#warning` *token-string*	警告診断 *token-string* を発行してコンパイルを続行します。

2.11 プラグマ

複数行形式の前処理:

`#pragma` *pp-tokens*

各処理系が定義した処理を指定します。

次の `#pragmas` はコンパイルシステムに認識されます。認識されなかったプラグマは無視されます。`-v` オプションを使用すると、認識されなかったプラグマについて警告が出されます。

2.11.1 align

`#pragma align` *integer* (*variable*[, *variable*])

整列プラグマで指定した変数のメモリーはデフォルト値によらず、すべて *integer* バイト境界に揃えられます。ただし、次の制限があります。

- *integer* は、1 〜 128 の範囲にある 2 のべき乗、つまり、1、2、4、8、16、32、64、128 のいずれかでなければいけません。
- *variable* には大域または静的な変数を指定します。自動変数は指定できません。
- 指定された境界がデフォルトより小さい場合は、デフォルトが優先します。

- プラグマ行は、その行に指定される変数の宣言よりも先になければいけません。先にないと無視されてしまいます。
- プラグマ行で記述されているが、そのあとで宣言されていない変数は無視されます。次に例を示します。

```
#pragma align 64 (aninteger, astring, astruct)
int aninteger;
static char astring[256];
struct astruct{int a; char *b;};
```

2.11.2 c99

`#pragma c99("implicit" | " no%implicit")`

このプラグマは、暗黙的な関数宣言の診断を制御します。`c99` プラグマの値が `"implicit"` に設定されている場合 (引用符を使用することに注意)、コンパイラが暗黙的な関数宣言を検出すると、警告が生成されます。`c99` プラグマの値が `"no%implicit"` に設定されている場合 (引用符を使用することに注意)、プラグマの値がリセットされるまで、コンパイラは暗黙的な関数宣言をそのまま受け入れます。

`-xc99` オプションの値は、このプラグマに影響を与えます。`-xc99=all` の場合はプラグマが `#pragma c99("implicit")` に設定され、`-xc99=none` の場合はプラグマが `#pragma c99("no%implicit")` に設定されます。

デフォルトでは、このプラグマは `c99=("implicit")` に設定されます。

2.11.3 does_not_read_global_data

`#pragma does_not_read_global_data (` *funcname* [, *funcname*] `)`

リストに指定したルーチンが直接にも間接にも大域データを読み取らないことを表明します。この表明により、そうしたルーチンへの呼び出し前後のコードをさらに最適化することができます。具体的には、代入文やストア命令をそうした呼び出しの前後に移動することができます。

指定した関数は、このプラグマの前にプロトタイプまたは空のパラメータリストで宣言する必要があります。大域アクセスに関する表明が真でない場合は、プログラムの動作は未定義になります。

2.11.4 does_not_return

`#pragma does_not_return (` *funcname* [, *funcname*] `)`

指定した関数への呼び出しが復帰しないことをコンパイラに表明します。この表明により、コンパイラは、指定された関数への呼び出しが戻らないと仮定して最適化を行うことができます。たとえば、レジスタの存続期間が呼び出し元で終了する場合は、さらに最適化率を高めることができます。

指定した関数が復帰した場合は、プログラムの動作は未定義になります。次の例に示すように、このプラグマは、指定した関数をプロトタイプまたは空のパラメータリストで宣言したあとでのみ使用できます。

```
extern void exit(int);
#pragma does_not_return(exit)

extern void __assert(int);
#pragma does_not_return(__assert)
```

2.11.5 does_not_write_global_data

`#pragma does_not_write_global_data (`*funcname [, funcname]*`)`

リストに指定したルーチンが直接にも間接にも大域データを書き込まないことを表明します。この表明により、そうしたルーチンへの呼び出し前後のコードをさらに最適化することができます。具体的には、代入文やストア命令をそうした呼び出しの前後に移動することができます。

指定した関数は、このプラグマの前にプロトタイプまたは空のパラメータリストで宣言する必要があります。大域アクセスに関する表明が真でない場合は、プログラムの動作は未定義になります。

2.11.6 error_messages

`#pragma error_messages (on|off|default,` *tag… tag*`)`

このエラーメッセージプラグマは、ソースプログラムの中から、C コンパイラおよび lint が発行するメッセージを制御可能にします。C コンパイラでは、警告メッセージに対してのみ有効です。C コンパイラの `-w` オプションを使用すると、このプラグマは無効になり、すべての警告メッセージが抑止されます。

- `#pragma error_messages (on,` *tag… tag*`)`

  `on` オプションは、先行する `#pragma error_messages` オプション (`off` オプションなど) をその時点で無効にして、`-erroff` オプションも無効にします。

- `#pragma error_messages (off,` *tag… tag*`)`

`off` オプションは、C コンパイラまたは lint プログラムが指定トークンから始まる特定のメッセージを発行することを禁止します。この特定のエラーメッセージに対するプラグマの指定は、別の `#pragma error_messages` によって無効にされるか、コンパイルが終了するまで有効です。

- `#pragma error_messages (default,` *tag… tag*`)`

  `default` オプションは、指定タグについて、先行する `#pragma error_messages` 指令を無効にします。

2.11.7 fini

`#pragma fini (`*f1[, f2…,fn]*

main() ルーチンを呼び出したあと、<*f1*> から <fn> (終了関数) までの関数を呼び出します。この種の関数は、型が `void` で引数はあってはいけません。プログラム制御下でプログラムが終了したとき、またはこのプログラムを含む共有オブジェクトがメモリーから除去されたときに呼び出されます。次の「初期化関数」の場合と同様、終了関数もリンクエディタによって処理された順に実行されます。

大域プログラム状態が初期化関数の影響を受ける場合には注意が必要です。たとえばシステムライブラリ初期化関数を使用したときに何が発生するかがインタフェースに明示的に規定されていないかぎり、システムライブラリ初期化関数によって変更される可能性がある `errno` の値などの大域状態情報をすべて取得し、復元する必要があります。

このような関数は **`#pragma fini`** 指令の中に登場するたびに、1 回呼び出されます。

2.11.8 hdrstop

```
#pragma hdrstop
```

同じプリコンパイル済みヘッダーファイルを共有すべき各ソースファイルの活性文字列 (viable prefix) の最後を識別するために、hdrstop プラグマを最後のヘッダーファイルのあとに置く必要があります。たとえば次のファイルがあるとします。

```
example% cat a.c
#include "a.h"
#include "b.h"
#include "c.h"
#include <stdio.h>
#include "d.h"
.
.
.
```

```
example% cat b.h
#include "a.h"
#include "b.h"
#include "c.h"
```

ソースファイルの活性文字列は c.h で終わっているので、各ファイルの c.h のあとに #pragma hdrstop を挿入します。

#pragma hdrstop は、cc コマンドで指定されるソースファイルの活性文字列の最後にのみ使用できます。#pragma hdrstop をインクルードファイル内に指定しないでください。

2.11.9 ident

`#pragma ident` *string*

実行可能プログラムの `.comment` セクション内に任意の *string* を格納します。

2.11.10 init

`#pragma init (`*f1[, f2…,fn]* `)`

`main()` を呼び出す前に、*f1* から *fn* までの関数 (初期化関数) を呼び出します。この種の関数は、型が `void` で引数はあってはいけません。実行開始時にプログラムのメモリーイメージを構成しているときに呼び出されます。共有オブジェクトの中の初期値設定子は、その共有オブジェクトをメモリー内へ持っていく動作の間、つまりプログラムの開始中、または `dlopen()` のような動的ロード動作中に実行されます。初期化関数の呼び出しを順序付ける方法は、それがリンクエディタによって動的または静的に処理される順序に依存します。

大域プログラム状態が終了関数の影響を受ける場合には注意が必要です。たとえばシステムライブラリを終了関数として使用したときに発生する事柄は、インタフェースに明示的に規定されていません。そういった情報が無い状態で、終了関数として使われたシステムライブラリが変更した可能性がある `errno` の値などの大域状態情報をすべて捕捉して、復元する必要があります。

このような関数は **`#pragma init`** 指令の中に登場するたびに、1 回呼び出されます。

2.11.11 inline

`#pragma [no_]inline (`*funcname[, funcname]*`)`

指定したルーチン名のインライン化を制御します。このプラグマはファイル全体に対して有効です。このプラグマでは、大域的なインライン化のみ制御可能であり、呼び出し元固有の制御を行うことはできません。

`#pragma inline` は、現在のファイル内にある、指定したルーチンに一致する呼び出しをインライン化するようコンパイラに指示します。この指示は、無視されることがあります。たとえば、関数本体が別のモジュールに存在していて、crossfile オプションが使用されていない場合などです。

`#pragma no_inline` は、現在のファイル内にある、指定したルーチンに一致する呼び出しをインライン化しないようコンパイラに指示します。

次の例に示すように、`#pragma inline` および `#pragma no_inline` は、関数がプロトタイプまたは空のパラメータリストで宣言されたあとでのみ使用できます。

```
static void foo(int);
static int bar(int, char *);
#pragma inline(foo, bar)
```

`-xldscope`、`-xinline`、`-xO`、`-xcrossfile` も参照してください。

2.11.12 int_to_unsigned

`#pragma int_to_unsigned` (*funcname*)

`-Xt` モードまたは `-Xs` モードで `unsigned` の型を返す関数が、戻り値の型 `int` を持つように変更します。

2.11.13 MP serial_loop

(SPARC) `#pragma MP serial_loop`

注 – 従来の Sun 固有の MP プラグマは推奨されず、サポートされません。代わりに、OpenMP 3.0 の仕様で規定された API をサポートします。標準命令への移植については、『OpenMP API ユーザーズガイド』を参照してください。

詳細は、90 ページの「3.8.3.1 直列プラグマ」を参照してください。

2.11.14 MP serial_loop_nested

(SPARC) `#pragma MP serial_loop_nested`

注 – 従来の Sun 固有の MP プラグマは推奨されず、サポートされません。代わりに、OpenMP 3.0 の仕様で規定された API をサポートします。標準命令への移植情報については、『Solaris Studio OpenMP API ユーザーズガイド』を参照してください。

詳細は、90 ページの「3.8.3.1 直列プラグマ」を参照してください。

2.11.15 MP taskloop

(SPARC) `#pragma MP taskloop`

注 – 従来の Sun 固有の MP プラグマは推奨されず、サポートされません。代わりに、OpenMP 3.0 の仕様で規定された API をサポートします。標準命令への移植については、『OpenMP API ユーザーズガイド』を参照してください。

詳細は、90 ページの「3.8.3.2 並列プラグマ」を参照してください。

2.11.16 nomemorydepend

(SPARC) `#pragma nomemorydepend`

ループのどの繰り返しでもメモリーの依存がないと指示します。つまり、ループのどの繰り返しの中でも、同じメモリーの参照は必要がないと指示します。このプラグマを指定すると、コンパイラ (パイプライナ) はループの 1 回の繰り返しの中で、より効率的に命令をスケジュールすることができます。ループの繰り返しの中でメモリーの依存があると、プログラムの実行結果は未定義になります。コンパイラはこの情報をレベル 3 以上の最適化に利用します。

このプラグマのスコープは、プラグマから始まり、次のブロックの始まりか現在のブロック内のプラグマに続く最初の for ループ、または現在のブロックの終わりのいずれか先に達したところで終わります。プラグマは、スコープの終端に到達した時点で最初に見つかった for ループに適用されます。

2.11.17 no_side_effect

`#pragma no_side_effect(`*funcname[, funcname…]* `)`

funcname には、現行の翻訳単位内の関数名を指定します。関数は、プラグマの前にプロトタイプまたは空のパラメータリストで宣言する必要があります。またプラグ

マはその関数の定義より前に指定されていなければいけません。指定した *funcname* に対し、プラグマはその関数に一切の副作用がないことを宣言します。つまり、*funcname* は渡された引数にだけ依存する結果の値を返します。*funcname* および呼び出された子孫については、次のことがいえます。

- 呼び出し時点で呼び出し側が認識できるプログラム状態の一部に、読み出しまたは書き込みのためにアクセスすることはありません。
- 入出力を実行しません。
- 呼び出し時点で認識できるプログラム状態のどの部分も変更しません。

コンパイラはこの情報を、その関数を用いる最適化に利用することができます。関数に副作用があると、この関数を呼び出すプログラムの実行結果は未定義になります。コンパイラはこの情報をレベル 3 以上の最適化に利用します。

2.11.18 opt

`#pragma opt` *level* (*funcname[, funcname]*)

funcname には、現行の翻訳単位内で定義された関数名を指定します。*level* の値は、指定した関数に対する最適化レベルです。0、1、2、3、4、5 いずれかの最適化レベルを割り当てることができます。*level* を 0 に設定すると、最適化を無効にできます。関数は、プラグマの前にプロトタイプまたは空のパラメータリストで宣言する必要があります。プラグマは、最適化する関数の定義を処理する必要があります。

プラグマ内に指定される関数の最適化レベルは、`-xmaxopt` の値に下げられます。`-xmaxopt=off` の場合、プラグマは無視されます。

2.11.19 pack

`#pragma pack(`*n*`)`

`#pack(`*n*`)` 構造体または共用体のメンバーの境界整列を制御します。デフォルトでは、構造体または共用体のメンバーは、char 型なら 1 バイト、short 型なら 2 バイト、整数なら 4 バイトというように、その自然境界で整列されます。*n* を指定する場合には、すべての構造体または共用体のメンバーに対してもっとも厳密な自然境界整列となる 2 の乗数にします。ゼロは受け入れられません。

このプラグマを使用すると、構造体または共用体のメンバーの境界整列を指定できます。たとえば、`#pragma pack(2)` を指定すると、int、long、long long、float、double、long double、pointer が、それぞれの自然境界ではなく、2 バイト境界に整列されます。

n がプラットフォームでもっとも厳密な整列を指示する値 (-m32 の x86 では 4、–m32 の SPARC では 8、-m64 の SPARC では 32) か、それより大きな値の場合は、自然境界整列が有効になります。*n* が省略された場合も、メンバーは自然境界整列に戻ります。

#pragma pack(*n*) 指令は、その指令から次の pack 指令の間のすべての構造体または共用体の定義に適用されます。別の翻訳単位で同じ構造体または共用体に対して異なる #pragma pack の定義が行われている場合、プログラムは予期しない形でコンパイルに失敗することがあります。特に、#pragma pack(*n*) は、事前にコンパイルされたライブラリのインタフェースを定義するヘッダーをインクルードする前には使用しないでください。#pragma pack(*n*) は、プログラムコード内の境界整列を変更するすべての構造体または共用体の直前に挿入することをお勧めします。そして、その構造体の直後に #pragma pack( ) を続けてください。

#pragma pack を使用する場合、構造体または共用体自身の整列条件は、その構造体または共用体でより厳密に境界整列されるメンバーの整列条件と同一です。したがって、その構造体または共有体の任意の宣言は pack の境界整列となります。たとえば、char 型だけの struct は整列の制限はありませんが、double 型を含む struct は 8 バイトの境界上に並びます。

注 – #pragma pack を使用して構造体または共用体のメンバーを自然境界以外の境界で整列させると、通常、これらのフィールドへのアクセスが発生した場合に SPARC 上でバスエラーが起きます。このエラーを避けるには、-xmemalign オプションも指定します。このようなプログラムをコンパイルする最適な方法については、293 ページの「B.2.116 -xmemalign=*ab*」を参照してください。

2.11.20 pipeloop

#pragma pipeloop(*n*)

このプラグマは、引数 *n* に正の整定数または 0 を受け入れます。このプラグマは、ループがパイプライン化可能で、ループによる依存の最小の依存距離が *n* であることを指定します。距離が 0 の場合、そのループは実質的には Fortran 形式の doall ループで、ターゲットプロセッサ上でパイプライン処理するべきであることを意味します。距離が 0 より大きい場合、コンパイラは *n* 回だけの連続繰り返しでパイプラインを試みます。コンパイラはこの情報をレベル 3 以上の最適化に利用します。

このプラグマのスコープは、プラグマから始まり、次のブロックの始まりか現在のブロック内のプラグマに続く最初の for ループ、または現在のブロックの終わりのいずれか先に達したところで終わります。プラグマは、スコープの終端に到達した時点で最初に見つかった for ループに適用されます。

2.11.21 rarely_called

#pragma rarely_called(*funcname[, funcname]*)

指定した関数があまり使用されないというヒントをコンパイラに与えます。このヒントにより、コンパイラは、プロファイル収集段階に負担をかけることなく、ルーチンの呼び出し元でプロファイルフィードバック方式の最適化を行うことができます。このプラグマはヒントの提示ですので、コンパイラは、このプラグマに基づく最適化を行わないこともあります。

指定した関数は、このプラグマの前にプロトタイプまたは空のパラメータリストで宣言する必要があります。#pragma rarely_called の例を次に示します。

```
extern void error (char *message);
#pragma rarely_called(error)
```

2.11.22 redefine_extname

#pragma redefine_extname *old_extname new_extname*

このプラグマにより、オブジェクトコード中で外部定義された *old_extname* の名前が *new_extname* に置換されます。この結果、リンク時にリンカーは *new_extname* だけを認識します。関数定義、初期設定子、または式のいずれかとして *old_extname* を最初に使用したあと、#pragma redefine_extname が指定されていると、結果は未定義になります。-Xs のモードではこのプラグマはサポートされていません。

#pragma redefine_extname を使用できる場合、コンパイラは、事前に定義されたマクロの __PRAGMA_REDEFINE_EXTNAME の定義を使用して、#pragma redefine_extname の有無に関係なく機能する移植可能なコードを作成できるようにします。

#pragma redefine_extname の目的は、関数名を変更できない場合に関数インタフェースを効率的に再定義する手段を提供することにあります。関数名を変更できない場合とは、たとえば、既存のプログラムとの互換性を保つために、ライブラリ内に、関数の古い定義と、新しいプログラムで使用する 新しい定義の両方を維持する必要がある場合です。ライブラリに新しい名前で新しい関数定義を追加した場合に、このようなことが必要になります。新旧の名前と定義が存在する関数を宣言するヘッダーファイルで #pragma redefine_extname を使用すると、その関数が使用されるときは、必ずその関数の新しい定義でリンクされるようになります。

```
#if    defined(__STDC__)

#ifdef __PRAGMA_REDEFINE_EXTNAME
extern int myroutine(const long *, int *);
#pragma redefine_extname myroutine __fixed_myroutine
#else /* __PRAGMA_REDEFINE_EXTNAME */
```

```
static int
myroutine(const long * arg1, int * arg2)
{
    extern int __myroutine(const long *, int*);
    return (__myroutine(arg1, arg2));
}
#endif /* __PRAGMA_REDEFINE_EXTNAME */

#else /* __STDC__ */

#ifdef __PRAGMA_REDEFINE_EXTNAME
extern int myroutine();
#pragma redefine_extname myroutine __fixed_myroutine
#else /* __PRAGMA_REDEFINE_EXTNAME */

static int
myroutine(arg1, arg2)
    long *arg1;
    int *arg2;
{
    extern int __fixed_myroutine();
    return (__fixed_myroutine(arg1, arg2));
}
#endif /* __PRAGMA_REDEFINE_EXTNAME */

#endif /* __STDC__ */
```

2.11.23 returns_new_memory

`#pragma returns_new_memory (`*funcname[, funcname]*`)`

指定した関数の戻り値が呼び出し元のどのメモリーとも別名処理されないことを表明します。つまり、この呼び出しでは、新しいメモリー位置が返されます。この情報により、オプティマイザはポインタ値をより正確に追跡し、メモリー位置を明確化することができます。この結果、スケジューリング、パイプライン化、ループの並列化が改善されます。表明が偽の場合には、プログラムの動作は未定義になります。

次の例に示すように、このプラグマは、指定した関数をプロトタイプまたは空のパラメータリストで宣言したあとでのみ使用できます。

```
void *malloc(unsigned);
#pragma returns_new_memory(malloc)
```

2.11.24 unknown_control_flow

`#pragma unknown_control_flow (`*name;[, name;]*`)`

#pragma unknown_control_flow 指令は、呼び出し元のフローグラフを変更する手続きを説明するために使用されます。通常、この指令には setjmp() のような関数の宣言が伴います。Solaris のシステム上では、インクルードファイル <setjmp.h> に次の指定が含まれています。

```
extern int setjmp();
#pragma unknown_control_flow(setjmp)
```

setjmp() のような特性を持つほかの関数も、同様に宣言する必要があります。

原則として、この属性を認識するオプティマイザは、制御フローグラフに適切な境界を挿入できます。これによって、setjmp() を呼び出す関数内で関数呼び出しを安全に処理しながら、影響を受けないフローグラフ部分のコードを最適化することが可能になります。

指定した関数は、このプラグマの前にプロトタイプまたは空のパラメータリストで宣言する必要があります。

2.11.25 unroll

#pragma unroll (*unroll_factor*)

このプラグマは、引数 *unroll_factor* に正の整定数を受け入れます。展開係数が 1 以外の場合、指定されたループを指定の係数で展開するよう、コンパイラに指示します。コンパイラは可能なかぎり、その展開係数を使用します。展開係数が 1 の場合、ループを展開してはいけないことをコンパイラに指定します。コンパイラはこの情報をレベル 3 以上の最適化に利用します。

このプラグマのスコープは、プラグマから始まり、次のブロックの始まりか現在のブロック内のプラグマに続く最初の for ループ、または現在のブロックの終わりのいずれか先に達したところで終わります。プラグマは、スコープの終端に到達した時点で最初に見つかった for ループに適用されます。

2.11.26 warn_missing_parameter_info

#pragma [no_]warn_missing_parameter_info

#pragma warn_missing_parameter_info を指定すると、パラメータ型情報が含まれない関数宣言を持つ関数の呼び出しに対して警告が発生します。次の例を考えてみましょう。

```
example% cat -n t.c
     1    #pragma warn_missing_parameter_info
     2
```

```
     3    int foo();
     4
     5    int bar () {
     6
     7       int i;
     8
     9       i = foo(i);
    10
    11       return i;
    12    }
% cc t.c -c -errtags
"t.c", line 9: warning: function foo has no prototype (E_NO_MISSED_PARAMS_ALLOWED)
example%
```

#pragma no_warn_missing_parameter_info は、それ以前の #pragma warn_missing_parameter_info を無効にします。

デフォルトでは、#pragma no_warn_missing_parameter_infoは有効です。

2.11.27 weak

#pragma weak *symbol1 [= symbol2]*

弱い大域シンボルを定義します。このプラグマは主にソースファイルの中でライブラリを構築するために使用されます。リンカーは弱いシンボルを解決できなくてもエラーメッセージを表示しません。

#pragma weak *symbol*

これは *symbol* を弱いシンボルとして定義しています。*symbol* の定義が見つからなくても、リンカーはメッセージ等を出さなくなります。

#pragma weak *symbol1 = symbol2*

これは *symbol1* を、*symbol2* の別名である弱いシンボルと定義します。この形式のプラグマは、ソースファイルまたはそこにインクルードされたヘッダーファイルのいずれかで、*symbol2* を定義した同じ変換ユニットの中にかぎり使用できます。それ以外で使用された場合は、コンパイルエラーになります。

プログラムが *symbol1* を呼び出しますが、それがプログラム中で定義されていない場合には、*symbol1* がリンクライブラリ中の弱いシンボルになっていると、リンカーはライブラリにある定義を使用します。しかし、プログラム自身が *symbol1* を定義してある場合、プログラムでの定義が優先され、ライブラリに存在する *symbol1* の弱い大域定義は使用されません。プログラムが直接 *symbol2* を呼び出すと、ライブラリにある定義が使用されます。*symbol2* の定義が重複して行われるとエラーになります。

2.12 事前に定義されている名前

事前定義に関する現在のリストは、**cc**(1) のマニュアルページを参照してください。

次の識別子は、オブジェクトに似たマクロとして事前に定義されています。

表 2–3 定義済み識別子 __STDC__

展開後	次のモードでコンパイルする場合
1	-Xc
0	-Xa、-Xt
未定義	-Xs

__STDC__ が未定義 (#undef __STDC__) の場合、警告が発行されます。__STDC__ は、-Xs モードでは定義されていません。

2.13 errno の値の保持

-fast を使用すると、コンパイラは errno 変数を設定しない同等の最適化コードを使用して呼び出しを浮動小数点関数に自由に置き換えることができます。さらに、-fast はマクロ __MATHERR_ERRNO_DONTCARE も定義します。このマクロを使用すると、コンパイラは errno の妥当性の確認を無視できます。この結果、浮動小数点関数の呼び出しのあとに errno の値に依存するユーザーコードにより、一貫しない結果が生成される可能性があります。

この問題を解決する 1 つの方法は、-fast を使用してそのようなコードをコンパイルしないことです。ただし、-fast の最適化が必要で、コードが浮動小数点ライブラリの呼び出しのあとに正しく設定される errno の値に依存している場合は、次のオプションを使用してコンパイルしてください。

```
-xbuiltin=none -U__MATHERR_ERRNO_DONTCARE -xnolibmopt -xnolibmil
```

これを、コマンド行で -fast のあとに使用することで、コンパイラはそのようなライブラリ呼び出しを最適化しなくなり、errno が確実に正しく処理されるようになります。

2.14 拡張機能

C コンパイラは、C 言語に多数の拡張機能を実装しています。

2.14.1 _Restrict キーワード

C コンパイラは、C99 規格の restrict キーワードの同義語として _Restrict キーワードをサポートします。restrict キーワードは、-xc99=all を指定する場合にしか使用できませんが、_Restrict キーワードは、-xc99=none と -xc99=all のどちらを指定する場合でも使用できます。

サポートしている C99 機能の詳細は、表 C–6 を参照してください。

2.14.2 __asm キーワード

__asm キーワード (先頭の 2 つの下線に注意) は、asm キーワードと同義語です。__asm キーワードではなく asm を使用し、-Xc モードでコンパイルを実行する場合、コンパイラは警告メッセージを表示します。-Xc モードで __asm を使用する場合、警告メッセージは表示されません。__asm 文の書式は次のようになります。

```
__asm("string");
```

ここで、*string* は有効なアセンブリ言語文です。

この文は、与えられたアセンブラテキストをアセンブリファイルに直接出力します。関数スコープではなくファイルスコープで宣言された基本的な asm 文は、「グローバル asm 文」と呼びます。ほかのコンパイラはこの文を「トップレベル」asm 文と呼びます。

グローバル asm 文は、指定された順に出力されます。つまり、互いに対して相対的な順序を維持し、周囲の関数との相対的な位置を維持します。

より高い最適化レベルでは、参照されていないと考えられる関数をコンパイラが削除することがあります。グローバル asm 内ではどの関数が参照されているかをコンパイラが把握しないので、関数が誤って削除される可能性があります。

テンプレートとオペランドの仕様を提供する、拡張された asm 文は、グローバルにできないことに注意してください。**__asm** および **__asm__** はキーワード **asm** の同義語であり、互いに入れ替えて使用できます。

2.14.3 __inline と __inline__

__inline および __inline__ はキーワード inline の同義語であり、互いに入れ替えて使用できます (C 規格、6.4.1 節)。

2.14.4 __builtin_constant_p()

__builtin_constant_p はコンパイラの組み込み関数です。この関数は単一の数値引数を取り、引数がコンパイル時の定数として既知である場合は 1 を返します。戻り値 0 は、その引数がコンパイル時の定数であるかどうかをコンパイラが判定できないことを意味します。この組み込み関数の標準的な使用法は、マクロ内で手動のコンパイル時最適化を行うことです。

2.14.5 __FUNCTION__ と __PRETTY_FUNCTION__

__FUNCTION__ と __PRETTY_FUNCTION__ は定義済みの識別子であり、字句を包含する関数の名前が含まれています。これらは、c99 の定義済みの識別子である __func__ に相当する機能です。Solaris プラットフォームでは、__FUNCTION__ と __PRETTY_FUNCTION__ は **-Xs** および **-Xc** の各モードで使用できません。

2.15 環境変数

ここでは、コンパイルや実行時環境を制御する環境変数について説明します。

2.15.1 OMP_DYNAMIC

スレッド数の動的な調整を無効または有効にします。

2.15.2 OMP_NESTED

入れ子の並列化を有効または無効にします。

2.15.3 OMP_NUM_THREADS

実行中に使用するスレッド数を設定します。

2.15.4 OMP_SCHEDULE

実行スケジュールのタイプとチャンクサイズを設定します。

2.15.5 PARALLEL

プログラムをマルチプロセッサ上で実行する場合に、使用するプロセッサ の数を指定します。対象マシンに複数のプロセッサが搭載されている場合は、スレッドは個々のプロセッサにマップできます。この例では、プログラムを実行すると、2 個のスレッドが生成され、各スレッド上でプログラムの並列化された部分が実行されるようになります。

2.15.6 SUN_PROFDATA

`-xprofile=collect` コマンドが実行頻度のデータを格納しているファイルの名前を制御します。

2.15.7 SUN_PROFDATA_DIR

`-xprofile=collect` コマンドが実行頻度データファイルを配置するディレクトリを制御します。

2.15.8 SUNW_MP_THR_IDLE

各ヘルパースレッドのタスク終了状態を制御します。 `spin` *n*`s`、または `sleep` *n*`ms` と設定できます。デフォルトは `sleep` です。詳細については、『OpenMP API ユーザーズガイド』を参照してください。

2.15.9 TMPDIR

`cc` は通常 `/tmp` ディレクトリに一時ファイルを作成します。環境変数 `TMPDIR` を設定すると、別のディレクトリを指定することができます。`TMPDIR` が有効なディレクトリ名でない場合は、`/tmp` が使用されます。`-xtemp` オプションと環境変数 `TMPDIR` では、-xtemp が優先されます。

Bourne シェルの場合は次のように入力します。

```
$ TMPDIR=dir; export TMPDIR
```

C シェルの場合は次のように入力します。

```
% setenv TMPDIR dir
```

2.16 インクルードファイルを指定する方法

インクルードC コンパイラシステムに付属している標準的なヘッダーファイルのいずれかをインクルードするには、次の形式を使用します。

```
#include <stdio.h>
```

山括弧 (<>) を使用すると、プリプロセッサがが、システム内にあるヘッダーファイル用の標準的な場所でヘッダーファイルを検索するようになります。通常は /usr/include ディレクトリです。

ユーザーが自分のディレクトリに格納したヘッダーファイルの場合は、次のように書式が異なります。

```
#include "header.h"
```

書式 #include "foo.h" (二重引用符を使用) の文に対し、コンパイラは、次の順番でインクルードファイルを検索します。

1. 現在のディレクトリ (つまり、「インクルード」するファイルを含むディレクトリ)
2. -I オプションで命名されたディレクトリ
3. /usr/include ディレクトリ

ヘッダーファイルがインクルードされたソースファイルと同じディレクトリにない場合は、cc コマンドで -I オプションを使用して、ヘッダーファイルが格納されているディレクトリのパスを指定してください。たとえば次のように、ソースファイル mycode.c の中で stdio.h と header.h をインクルードしたとします。

```
#include <stdio.h>
#include "header.h"
```

この header.h が ../defs ディレクトリに格納されている場合は、次のコマンドを実行します。次のようなコマンド行があるとします。

```
% cc– I../defs mycode.c
```

この場合、プリプロセッサが header.h を検索する順序は、最初が mycode.c を含むディレクトリ、次が ../defs ディレクトリ、最後が標準の場所となります。stdio.h については最初が ../defs、次が標準の場所となります。相違点は、現ディレクトリを検索するのは名前を二重引用符で囲んだヘッダーファイルを検索する場合だけであることです。

-I オプションは 1 つの cc コマンド行の中で複数回指定することができます。指定したディレクトリをプリプロセッサが検索する順序は、コマンド行での指定順序と同じです。cc の複数のオプションを、同じコマンド行で指定できます。

```
% cc– o prog– I../defs mycode.c
```

2.16.1 -I- オプションによる検索アルゴリズムの変更

新しい -I- オプションは、デフォルトの検索規則に対する制御をさらに強化します。この節で説明しているように、コマンド行の最初の -I- だけが有効です。-I- オプションをコマンド行に配置すると、次のような効果があります。

#include "foo.h" 形式のインクルードファイルの場合、次の順序でディレクトリを検索します。

1. -I オプションで指定されたディレクトリ内 (-I- の前後)

2. コンパイラで提供される C++ ヘッダーファイル、ANSI C ヘッダーファイル、および特殊な目的のファイルのディレクトリ。

3. /usr/include ディレクトリ内。

#include <foo.h> 形式のインクルードファイルの場合、次の順序でディレクトリを検索します。

1. -I のあとに指定した -I- オプションで指定したディレクトリ内。

2. コンパイラで提供される C++ ヘッダーファイル、ANSI C ヘッダーファイル、および特殊な目的のファイルのディレクトリ。

3. /usr/include ディレクトリ内。

次の例は、prog.c のコンパイル時に -I- を使用した結果を示しています。

```
prog.c
#include "a.h"

#include <b.h>

#include "c.h"


c.h
#ifndef _C_H_1

#define _C_H_1

int c1;

#endif


int/a.h
#ifndef _A_H

#define _A_H

#include "c.h"
```

```
int a;

#endif


int/b.h
#ifndef _B_H

#define _B_H

#include <c.h>

int b;

#endif
int/c.h
#ifndef _C_H_2

#define _C_H_2

int c2;

#endif
```

次のコマンドでは、#include "foo.h" 形式のインクルード文のカレントディレクトリ (インクルードしているファイルのディレクトリ) のデフォルトの検索動作を示します。inc/a.h の #include "c.h" 文を処理する際、プリプロセッサは、inc サブディレクトリから c.h ヘッダーファイルを読み取ります。prog.c の #include "c.h" 文を処理する際、プリプロセッサは、prog.c を含むディレクトリから c.h ファイルを取り込みます。-H オプションがインクルードファイルのパスを印刷するようにコンパイラに指示していることに注意してください。

```
example% cc -c -Iinc -H prog.c
inc/a.h
            inc/c.h
inc/b.h
            inc/c.h
c.h
```

次のコマンドでは、-I- オプションの影響を示します。プリプロセッサは、書式 #include "foo.h" の文を処理する際に、インクルードするディレクトリを最初に検索しません。代わりに、コマンド行に配置されている順番で、-I オプションで命名されたディレクトリを検索します。inc/a.h 内にある #include "c.h" 文を処理する際、プリプロセッサは /c.h ヘッダーファイルを、inc/c.h ヘッダーファイルの代わりに取り込みます。

```
example% cc -c -I. -I- -Iinc -H prog.c
inc/a.h
            ./c.h
inc/b.h
            inc/c.h
./c.h
```

2.16.1.1 警告

コンパイラがインストールされている位置の `/usr/include`、 `/lib`、`/usr/lib` を検索ディレクトリに指定しないでください。

詳細は、245 ページの「B.2.37 -I[-|*dir*]」の節を参照してください。

2.17 フリースタンディング環境でのコンパイル

Solaris Studio C コンパイラは、標準 C ライブラリとリンクされたプログラムのコンパイルや、標準 C ライブラリおよびほかの実行時サポートライブラリが含まれる実行環境での実行をサポートします。C 標準では、そのような環境はホスト環境と呼ばれます。標準ライブラリ関数を提供しない環境は、フリースタンディング環境と呼ばれます。

コンパイルされたコードから呼び出される特定の実行時サポート関数は通常、標準 C ライブラリでのみ使用可能なため、C コンパイラはフリースタンディング環境での一般的なコンパイルをサポートしません。問題は、コンパイラによるソースコードの変換で、関数呼び出しが存在しないソースコード構造体に実行時サポート関数への呼び出しが導入される場合があることです。これらの関数は通常フリースタンディング環境で使用できません。次の例を考えてみましょう。

```
% cat -n lldiv.c
     1    void
     2    lldiv(
     3        long long *x,
     4        long long *y,
     5        long long *z)
     6    {
     7        *z = *x / *y ;
     8    }
% cc -c -m32 lldiv.c
% nm lldiv.o | grep " U "
 0x00000000 U __div64
% cc -c -m64 lldiv.c
% nm lldiv.o | grep " U "
```

この例では、ソースファイル `lldiv.c` がコンパイルされ、**-m32** オプションを使用して 32 ビットプラットフォームで実行されると、行 7 の文の変換が、`__div64` という名前の実行時サポート関数への外部参照となります。この関数は、標準 C ライブラリの 32 ビットバージョンでのみ使用できます。

同じソースファイルが **-m64** オプションを使用して 64 ビットプラットフォームでコンパイルされると、コンパイラはターゲットマシンの 64 ビット算術命令セットを使用するため、標準 C ライブラリの 64 ビットバージョンで実行時サポート関数が不要になります。

一般的な事例では、フリースタンディング環境を対象にしてCコンパイラを使用することはできませんが、特定のフリースタンディング環境 (つまり Solaris カーネルとデバイスドライバ) ではコードをコンパイルするためにコンパイラを使用できます (警告が生成されます)。

デバイスドライバなど、Solaris カーネルで実行されるコードは、外部関数呼び出しがカーネル内で使用可能な関数のみ参照するように記述する必要があります。これを可能にするため、次のガイドラインが推奨されます。

1. ユーザーモードでのみ実行されるライブラリのヘッダーファイルは含めないでください。
2. 同じ関数がカーネルに存在していることがわかっているのでない限り、標準 C ライブラリまたはほかのユーザーモードライブラリ内の関数を呼び出さないでください。
3. 浮動小数点型または C99 複合型を使用しないでください。
4. ランタイムサポートライブラリに関連付けられたコンパイラオプションを使用しないでください (**-xprofile**、**-xopenmp** など)

   特定のコンパイラオプションに関連付けられた再配置可能なオブジェクトファイルは、**cc**(1) マニュアルページの FILES 節で説明されています。C コンパイラオプションに関連付けられた実行時サポートライブラリは、関連するオプションを説明している箇所に記載されています。

前述のように、ソースコードの変換の結果、コンパイラにより実行時サポート関数への呼び出しが生成されることがあります。Solaris カーネルの特定の事例では、カーネルが浮動小数点型や複合型、数学ライブラリ関数、または実行時サポートライブラリに関連付けられたコンパイラオプションを使用しないため、呼び出される実行時サポート関数のセットが一般的な事例より小さくなります。

次の表に、C コンパイラによるソースコードの変換の結果、Solaris カーネルで実行するためにコンパイルされたコードで呼び出される可能性のある実行時サポート関数を示します。この表に、ソースコードの変換で呼び出しが生成されるプラットフォーム、呼び出される関数の名前、関数呼び出しを引き起こすソース構造体またはコンパイラ機能を示します。C コンパイラをサポートする Solaris のすべてのバージョンで 64 ビットカーネルが実行されるため、64 ビットプラットフォームだけがリストに示されています。

32 ビット命令セット用にコンパイルすると、命令セットに固有の制限があるため、追加のマシン固有のサポート関数が呼び出されることがあります。

関数	64 ビットプラットフォーム	参照元
__align_cpy_*n*	SPARC	大きい構造体を返します。*n* は 1、2、4、8、または 16 です。
_memcpy	x86	大きい構造体を返します。

関数	64 ビットプラットフォーム	参照元
_memcpy	x86 および SPARC	ベクトル化。
_memmove	x86 および SPARC	ベクトル化。
_memset	x86 および SPARC	ベクトル化。

一部のバージョンのカーネルは、**_memmove()**、**_memcpy()**、または **_memset()** を提供しませんが、ユーザーモードルーチン **memmove()**、**memcpy()**、および **memset()** のカーネルモードアナログは提供する点に注意してください。

追加情報については、『Writing Device Drivers』および『SPARC Compliance Definition, version 2.4』を参照してください。

◆ ◆ ◆ 第 3 章

3 C コードの並列化

Oracle Solaris Studio C コンパイラは、メモリー共有型マルチプロセッサ、マルチコア、またはマルチスレッドのシステム上で実行するコードを最適化できます。 コンパイルされたコードは、システムの複数のプロセッサを使用して並列して実行できます。自動的な並列化と明示的な並列化という両方の手法が使用可能です。この章では、このコンパイラの並列化機能を利用する方法について説明します。

3.1 概要

C コンパイラは、並列化しても安全であると判断したループに対して並列コードを生成します。通常、これらのループは、独立して実行可能な繰り返しを持っています。繰り返しが実行される順番や、並列に実行するかどうかといったことなどは、ループの実行結果に影響はありません。すべてではありませんが、ほとんどのベクトル処理用ループはこのような種類のループです。

C では別名が存在する (複数の変数が同一の実体である / を指す) 可能性があるため、並列化の安全性を判断することは困難です。コンパイラの作業を容易にするため、Solaris Studio C にはプラグマおよび追加のポインタ修飾子が用意されており、プログラマは認識できてもコンパイラが判定できない別名情報をコンパイラに渡します。詳細については、第 5 章「型に基づく別名解析」を参照してください。

3.1.1 使用例

次の例は、C を並列化し、制御する方法を示しています。

```
% cc -fast -xO4 -xautopar example.c -o example
```

この例では、通常の方法で実行できる `example` という実行可能ファイルが生成されます。マルチプロセッサ上で実行する場合は、264 ページの「B.2.75 -xautopar」 を参照してください。

3.2 OpenMP に対する並列化

C コンパイラは、共有メモリーの並列化に OpenMP API をネイティブで受け入れます。この API は、一連の並列化プラグマで構成されます。OpenMP API の仕様の情報は、OpenMP Web サイト (http://www.openmp.org/ (http://www.openmp.org)) にあります。

コンパイラの OpenMP サポートと OpenMP プラグマの認識を有効にするには、`-xopenmp` オプションを使用してコンパイルします。`-xopenmp` を使用しないと、コンパイラは OpenMP プラグマをコメントとして扱います。299 ページの「B.2.123 -xopenmp[= *i*]」 を参照してください。

詳細は、『*Solaris Studio OpenMP API* ユーザーズガイド』を参照してください。

3.2.1 OpenMP の実行時の警告の処理

OpenMP 実行時システムは、軽度のエラーに対し警告を発行できます。次の関数を使用すると、それらの警告を処理するコールバック関数を登録できます。

```
int sunw_mp_register_warn(void (*func) (void *) )
```

この関数のプロトタイプにアクセスするには、`<sunw_mp_misc.h>` に対する `#include` プリプロセッサ指令を発行します。

関数を登録したくない場合、環境変数 `SUNW_MP_WARN` を `TRUE` に設定すると、警告メッセージが `stderr` に送られます。`SUNW_MP_WARN` の詳細については、71 ページの「3.3.3 SUNW_MP_WARN」 を参照してください。

この実装の OpenMP に固有の情報については、『Solaris Studio OpenMP API ユーザーズガイド』を参照してください。

3.3 環境変数

並列化された C には、関連する環境変数として次の 4 つが存在します。

- `PARALLEL` または `OMP_NUM_THREADS`
- `SUNW_MP_THR_IDLE`
- `SUNW_MP_WARN`
- `STACKSIZE`

3.3.1 PARALLEL または OMP_NUM_THREADS

マルチプロセッサ実行を活用できる場合は、`PARALLEL` 環境変数を設定します。`PARALLEL` 環境変数には、プログラムの実行に使用できるプロセッサの数を指定します。次は、`PARALLEL` を 2 に設定する例を示しています。

```
% setenv PARALLEL 2
```

対象マシンに複数のプロセッサが搭載されている場合は、スレッドは個々のプロセッサにマップできます。この例では、プログラムを実行すると、2 個のスレッドが生成され、各スレッド上でプログラムの並列化された部分が実行されるようになります。

PARALLEL と **OMP_NUM_THREADS** のどちらかを使用できます。これらは同等です。

3.3.2 SUNW_MP_THR_IDLE

現在のところ、プログラムの初期実行を行うスレッドが結合スレッドを作成します。作成されたこれらの結合スレッドは、プログラムの並列部分 (並列ループ、並列領域など) の実行に加わり、プログラムの順次実行部分が実行される間スピン待ち状態を維持します。これらの結合スレッドは、プログラムが終了するまで休眠または停止することはありません。並列化されたプログラムを 1 つのシステム上で実行する場合は、結合スレッドをスピン待ちにすると最高のパフォーマンスが得られます。ただし、スピン待ちのスレッドはシステム資源を使用します。

SUNW_MP_THR_IDLE 環境変数は、各スレッドが並列ジョブの分担部分を終えたあとの各スレッドの状態を制御するために使用してください。

```
% setenv SUNW_MP_THR_IDLE value
```

*value*には、spin または sleep[*n* s|*n* ms] のどちらかを指定できます。デフォルトは sleep であり、スレッドを *n* 単位のスピン待ちにしたあと、休眠させます。待ち時間の単位は秒 (s、デフォルトの単位) かミリ秒 (ms) で、1s は 1 秒、10ms は 10 ミリ秒を意味します。引数を取らずに sleep を指定すると、スレッドは並列化タスクの完了直後にスリープ状態に入ります。sleep、sleep0、sleep0s、および sleep0ms はすべて同等です。*n* 単位が到着する前に新しいジョブが到着すると、スレッドはスピンを停止し、新しいジョブを開始します。

もう 1 つの選択肢は spin です。並列化タスクの完了したスレッドは、新しい並列化タスクが到着するまでスピン (busy-wait) します。

SUNW_MP_THR_IDLE に不正な値が含まれているか、設定されていない場合は、デフォルトとして sleep が使用されます。

3.3.3 SUNW_MP_WARN

この環境変数を TRUE に設定すると、OpenMP そのほかの並列化ランタイムシステムから発行された警告メッセージを出力できます。

```
% setenv SUNW_MP_WARN TRUE
```

sunw_mp_register_warn() を使用して、警告メッセージを処理する関数を登録してある場合、SUNW_MP_WARN は警告メッセージを出力しません。これは、環境変数を TRUE に設定してある場合も同様です。関数を登録しておらず、SUNW_MP_WARN を TRUE に設定してある場合、SUNW_MP_WARN は警告メッセージを stderr に出力します。関数を登録しておらず、SUNW_MP_WARN を設定していない場合、警告メッセージは発行されません。sunw_mp_register_warn() の詳細については、70 ページの「3.2.1 OpenMP の実行時の警告の処理」を参照してください。

3.3.4 STACKSIZE

プログラムを実行すると、マスタースレッドにはメインメモリースタックが、各スレーブスレッドには個別のスタックが保持されます。スタックとは、サブプログラムが呼び出されている間、引数と自動変数を保持するために使用される一時的なメモリーアドレス空間です。

メインスタックのデフォルトサイズは、およそ 8M バイトです。現在のスタックサイズの確認と設定には、limit コマンドを使用します。次に例を示します。

```
% limit
cputime unlimited
filesize unlimited
datasize 2097148 kbytes
stacksize 8192 kbytes <- current main stack size
coredumpsize 0 kbytes
descriptors 256
memorysize unlimited
% limit stacksize 65536 <- set main stack to 64Mb
```

マルチスレッド化されたプログラムの各スレーブスレッドは、それ自体のスレッドスタックを持ちます。このスタックはマスタースレッドのメインスタックに似ていますが、各スレッド固有のものです。スレッドのスタックには、スレッド固有の配列とその (スレッドに対して局所的な) 変数が割り当てられます。

スレーブスレッドはすべて、同じスタックサイズを持ちます。デフォルトのスタックサイズは、32 ビットアプリケーションの場合は 4M バイト、64 ビットアプリケーションの場合は 8M バイトです。このサイズは、STACKSIZE 環境変数で設定します。

```
% setenv STACKSIZE 16483 <- Set thread stack size to 16 Mb
```

並列化されたコードでは、通常、スレッドのスタックサイズをデフォルト値より大きな値に設定する必要があります。

時折、スタックサイズを増やす必要があるという警告メッセージがコンパイラによって表示されることがあります。しかし、通常 (とりわけスレッド固有 / 局所の配列が関わる場合)、設定すべきサイズは試行錯誤でしか把握できません。スタックサイズがスレッドを実行するには小さすぎる場合、プログラムはセグメント例外を生成して終了します。

STACKSIZE 環境変数の設定は、Solaris *pthreads* API を使用しているプログラムに影響しません。

3.3.5 並列コードでの restrict の使用

並列化された C では、キーワード restrict を使用できます。キーワード restrict を適切に使用すると、コードシーケンスを並列化できるかどうかを判別するために必要なデータの別名をオプティマイザが認識する場合に有効です。詳細については、357 ページの「D.1.2 C99 のキーワード」を参照してください。

3.4 データの依存性と干渉

C コンパイラは、プログラム中のループを解析して、ループの各繰り返しを安全に並列実行できるかどうかを判断します。この解析の目的は、ループ中の任意の 2 個の繰り返しが、互いに干渉しないかどうかを調べることです。通常、干渉は、ある繰り返しが書き込みを行なっている変数に対して、別の繰り返しが読み込みを行うと発生します。次に示すプログラムの一部を考えてみましょう。

例 3–1 依存性を持つループ

```
for (i=1; i < 1000; i++) {
    sum = sum + a[i]; /* S1 */
}
```

73 ページの「3.4 データの依存性と干渉」では、2 個の連続した繰り返しである i および i+1 が、同じ変数 sum に書き込みと読み込みを実行しています。したがって、このような 2 個の繰り返しを並列に実行するには、なんらかの方法で変数をロックすることが必要になります。ロックをしないと、2 個の連続した繰り返しを安全に並列実行することができなくなります。

ところが、このロック機構を使用すると、オーバーヘッドが発生してプログラムの実行を遅くすることになります。C コンパイラは73 ページの「3.4 データの依存性と干渉」のループの並列化を行いません。73 ページの「3.4 データの依存性と干渉」には、2 個の連続したループの繰り返しにデータの依存関係があるからです。別の例を考えてみましょう。

例 3–2 依存性を持たないループ

```
for (i=1; i < 1000; i++) {
    a[i] = 2 * a[i]; /* S1 */
}
```

この場合、ループ中の各繰り返しでは、異なる配列の要素が参照されています。したがって、ループ中の繰り返しを実行する順番を守る必要がありません。また、異なる繰り返しでアクセスするデータが互いに干渉しないため、ロックを使用せずに並列実行することが可能になります。

ループ内の 2 個の異なる繰り返しで、同じ変数を参照していないかどうかを判断するためにコンパイラが実行する解析を、「データ依存性解析」といいます。1 回でも変数に書き込みを実行している場合には、データ依存性によって並列化することができなくなります。コンパイラが実行する依存性解析の結果、次のいずれかの解答が得られます。

- 依存性があります。この場合には、ループを安全に並列実行できません。前述の73 ページの「3.4 データの依存性と干渉」がこれに該当します。
- ループ内に依存性がありません。ループを任意の数のプロセッサを使用して並列に実行することができます。前述の73 ページの「3.4 データの依存性と干渉」がこれに該当します。
- 依存性を確認できません。コンパイラは、安全に実行することを重視して、ループに並列実行できないような依存関係が存在するものと仮定して、ループを並列化しません。

73 ページの「3.4 データの依存性と干渉」ではループの 2 個の繰り返しで配列 a の同じ要素に書き込まれるかどうかは、配列 b に重複する要素が存在するかどうかによって決まります。コンパイラがこの事実を確認できないかぎり依存性があるものと判断され、ループは並列化されません。

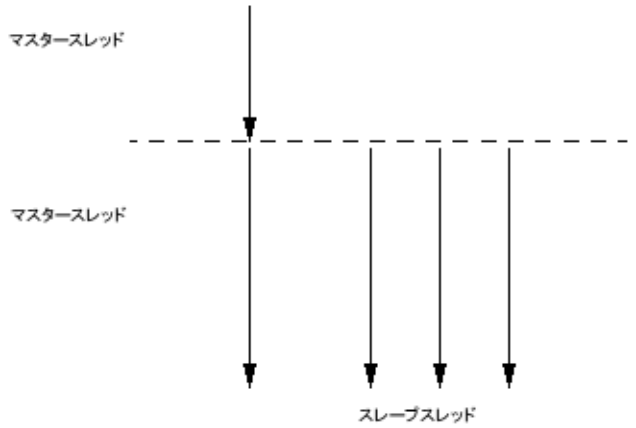
例 3–3　依存性の有無を確認できないループ

```
for (i=1; i < 1000; i++) {
    a[b[i]] = 2 * a[i];
}
```

3.4.1 並列実行モデル

ループの並列実行は、Solaris スレッドによって実行されます。プログラムの初期実行を行うスレッドをマスタースレッドといいます。プログラムの起動時に、マスタースレッドによって複数のスレーブスレッドが生成されます (次の図を参照)。プログラムの終了時には、すべてのスレーブスレッドが終了されます。オーバーヘッドを最小限に抑えるために、スレーブスレッドの生成は 1 回だけ実行されます。

図 3–1 マスタースレッドとスレーブスレッド



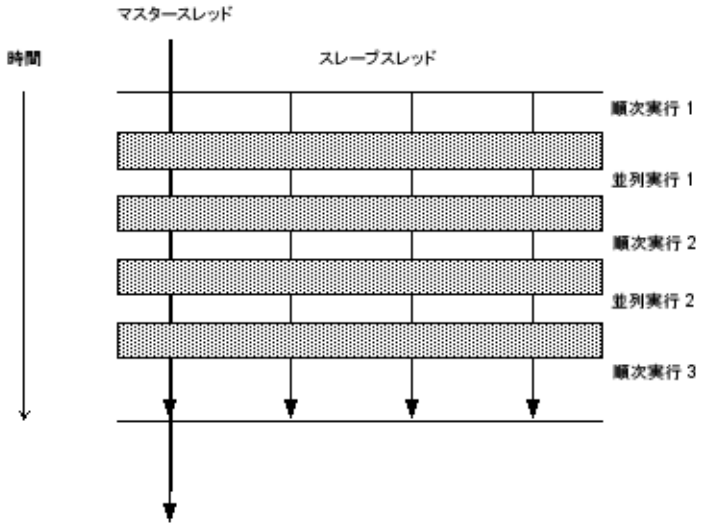
起動後、マスタースレッドによってプログラムの実行が開始されますが、スレーブスレッドはアイドル状態で待機します。マスタースレッドが並列ループを検出すると、ループの異なる繰り返しがスレーブおよびマスタースレッドに割り当てられ、ループの実行が開始されます。それぞれのスレッドが実行を終了すると、残りのスレッドの終了と同期が取られます。この同期を取る点を「バリア」といいます。すべてのスレッドが分担した実行を終了してバリアに達するまで、マスタースレッドは残りのプログラムを実行することができません。スレーブスレッドは、バリアに達すると、ほかの並列化された部分が検出されるまで待ち状態になり、マスタースレッドがプログラムの実行を続行します。

この処理では、次に説明するオーバーヘッドが発生します。

- 同期と作業を分散するためのオーバーヘッド
- バリア同期でのオーバーヘッド

一般的な並列ループの中には、並列化で得られるメリットよりオーバーヘッドの方が多くなってしまうものがあります。このようなループでは、実行速度が大きく低下することがあります。次の図ではループが並列化されていますが、水平の棒で示されたバリアは相当なオーバーヘッドを示しています。バリア間の作業は、図中に示すとおりに 1 つずつ実行される (順次実行) か、あるいは同時に実行 (並列実行) されます。ループの並列実行に必要な時間は、バリアの位置でマスタースレッドとスレーブスレッドの同期を取るために必要な時間よりはるかに短くてすみます。

図 3–2 ループの並列実行



3.4.2 固有スカラーと固有配列

データの依存性が存在してもコンパイラがループを並列化できる場合があります。次の例を考えてみましょう。

例 3–4 依存性があるが並列化可能なループ

```
for (i=1; i < 1000; i++) {
    t = 2 * a[i];           /* S1 */
    b[i] = t;               /* S2 */
}
```

この例では、配列 a と b が重なりあっていないと仮定すると、2 回の繰り返しの間に、変数 t による明らかなデータ依存性が存在します。繰り返しの 1 回目と 2 回目に注目すると、次のような文が実行されることになります。

例 3–5 繰り返し 1 と 2

```
t = 2*a[1];  /* 1 */
b[1] = t;    /* 2 */
t = 2*a[2];  /* 3 */
b[2] = t;    /* 4 */
```

文 1 および 3 によって変数 t が変更されるので、これらを並列実行することはできません。しかし、変数 t は常に同じ繰り返しの中で計算されて使用されるので、コンパイラは繰り返しごとに変数 t のコピーを使用することができます。したがって、このような変数による異なる繰り返し間での干渉を回避することができま

す。実際に変数 t は、繰り返しを実行する各スレッドに固有の変数として使用されます。これを説明した例を、次に示します。

例 3–6 各スレッドに固有の変数としての変数 t

```
for (i=1; i < 1000; i++) {
    pt[i] = 2 * a[i];       /* S1 */
    b[i] = pt[i];           /* S2 */
}
```

76 ページの「3.4.2 固有スカラーと固有配列」は73 ページの「3.4 データの依存性と干渉」と基本的に同じ例ですが、各スカラー変数参照 t が配列参照 pt に置き換えられています。各繰り返しでは、pt の異なる要素が使用されるので、任意の 2 個の繰り返し間でのデータ依存性がなくなります。ただし、この方法では、大きな配列を余分に生成することになります。実際には、コンパイラによってスレッドごとに 1 個の変数だけが割り当てられ、その変数をループの実行で使用します。つまりこのような変数は、スレッドごとに固有であるといえます。

コンパイラは、配列変数を固有化してループを並列実行することもできます。次の例を考えてみましょう。

例 3–7 配列変数を使用した並列化可能なループ

```
for (i=1; i < 1000; i++) {
    for (j=1; j < 1000; j++) {
            x[j] = 2 * a[i];        /* S1 */
            b[i][j] = x[j];         /* S2 */
    }
}
```

76 ページの「3.4.2 固有スカラーと固有配列」では、外側のループの異なる繰り返しによって、配列 x の同じ要素が変更されるので、外側のループを並列化することはできません。しかし、外側のループを実行するそれぞれのスレッドに配列 x 全体のスレッド固有のコピーが存在すれば、外側の任意の 2 個のループ間で干渉が発生しません。これを説明した例を次に示します。

例 3–8 スレッド固有配列を使用した並列化可能なループ

```
for (i=1; i < 1000; i++) {
    for (j=1; j < 1000; j++) {
            px[i][j] = 2 * a[i];    /* S1 */
            b[i][j] = px[i][j];     /* S2 */
    }
}
```

スレッド固有スカラー変数の場合と同様に、配列をすべての繰り返しに対して展開する必要はありません。システムで実行されるスレッドの数に対してのみ展開すればよいことになります。これはコンパイラによって自動的に行われ、各スレッドのスレッド固有領域にオリジナルの配列がコピーされます。

3.4.3 ストアバック変数の使用

変数のスレッド固有化は、プログラムの並列化を向上させる上で便利な方法です。しかし、スレッド固有変数がループの外側で参照される場合には、その値が正しいことを保証することが必要になります。次の例を考えてみましょう。

例 3–9 ストアバック変数を使用した並列ループ

```
for (i=1; i < 1000; i++) {
    t = 2 * a[i];           /* S1 */
    b[i] = t;               /* S2 */
}
x = t;                      /* S3 */
```

78 ページの「3.4.3 ストアバック変数の使用」では、文 S3 で参照されている変数 t の値が、ループを終了したときの最終結果になります。変数 t がスレッド固有化され、ループの実行が終了したあと、t の正しい値をオリジナルの変数に戻すことが必要になります。この操作をストアバック (書き戻し) といいます。これは、繰り返しの最後における t の値をオリジナルの変数 t に書き込むことで実現できます。多くの場合、この操作はコンパイラによって自動的に行われます。しかし、最終値を簡単に計算できないこともあります。

例 3–10 ストアバック変数を使用できないループ

```
for (i=1; i < 1000; i++) {
    if (c[i] > x[i] ) {         /* C1 */
            t = 2 * a[i];           /* S1 */
            b[i] = t;               /* S2 */
    }
}
x = t*t;                        /* S3 */
```

正しく実行した場合、文 S3 の t の値は、一般的にはループの最後における t の値にはなりません。最後の繰り返しで、C1 が真の場合に限って、最後の t の値に等しくなります。すべての場合における t の最終値を計算することは、非常に困難です。このような場合には、コンパイラはループを並列化しません。

3.4.4 縮約変数の使用

ループの繰り返し間に本当の依存性が存在すると、依存性の原因となっている変数を簡単にスレッド固有化できない場合があります。このような状況は、たとえば、変数がある繰り返しから別の繰り返しで累積計算されているような場合に発生します。

例 3–11　並列化されるかどうか不明なループ

```
for (i=1; i < 1000; i++) {
    sum += a[i]*b[i]; /* S1 */
}
```

78 ページの「3.4.4 縮約変数の使用」では、ループで 2 個の配列のベクトル積を計算して、共通変数 sum を求めています。このループを単純な方法で並列化することはできません。ここでは、文 S1 の計算式に結合の法則を適用し、各スレッドに対して psum[i]というスレッド固有変数を割り当てることができます。変数 psum[i]のコピーはそれぞれ 0 に初期化します。各スレッドは、スレッド固有の変数 psum[i] に自分で計算した部分和を代入します。バリアに達したら、すべての部分和を合計してオリジナルの変数 sum に代入します。この例では、和の縮約をしているので、変数 sum を縮約変数といいます。しかし、スカラー変数を縮約変数にした場合には、丸め誤差が累積されて、sum の最終値に影響する可能性があることに注意してください。コンパイラは、ユーザーによる明確な指示がされた場合に、この操作を実行します。

3.5　処理速度の向上

実行時間の大部分を占めるプログラム部分が並列化できない場合、速度の向上は期待できません。これは、基本的にアムダールの法則の結果から言えることです。たとえば、プログラム実行の 5% 部分に相当するループしか並列化できない場合、全体的に速度を向上できる限界は 5% です。しかし、実際には、負荷の量と並列実行に伴うオーバーヘッドによって、まったく速度が向上しないこともあります。

したがって、一般的な規則として、プログラムの並列化される部分が大きくなればなるほど、大幅な速度の向上を期待できます。

それぞれの並列ループには、起動時と終了時にわずかなオーバーヘッドがあります。起動時のオーバーヘッドには作業を分散するためのものがあり、終了時には、バリアでの同期によるものがあります。ループによって実行される作業量が比較的小さい場合には、速度の向上を期待できません。実際にループの実行が遅くなることもあります。したがって、プログラム実行の大部分が小さな並列ループから構成されている場合には、全体の実行速度は上がらず、かえって遅くなることがあります。

コンパイラは、いくつかのループ変換を実行することで、ループの規模を大きくしようとします。この変換には、ループの交換およびループの融合が含まれます。したがって、一般的には、プログラム中の並列化部分が少ない場合や小さな並列化部分に分割される場合には、速度の向上を期待できません。

プログラムサイズが大きくなると、プログラムの並列度が向上することがあります。たとえば、あるプログラムが順次実行する部分がプログラムサイズの 2 乗に増加し、並列化可能な部分が 3 乗に増加するものとします。このプログラムでは、並

列化された部分の作業量が順次実行する部分よりも速い勢いで増加します。したがって、資源の限界に達しないかぎり、ある時点で速度向上の効果が明確に表れます。

一般に並列Cの能力を有効利用するには、コンパイル指令を実験したり、プログラムの大きさやプログラムを再構成するといった調整を行う必要があります。

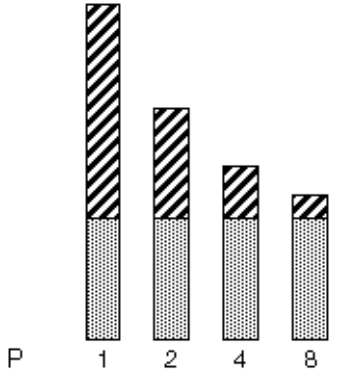
3.5.1 アムダールの法則

決まったサイズの問題の処理速度の向上度は、一般にアムダールの法則によって予測されます。アムダールの法則は単純に、特定の問題に対して並列化がもたらす速度の向上は、問題の逐次処理部分によって制限されると説明しています。次の式は、逐次処理領域で費やされる時間の割合がFであり、時間のうち残りの割合が*P*個のプロセッサの間で一様に費やされる状況における問題の速度向上について説明します。この式からわかるように、式の2番目の項の値がゼロになると、値が決まっている1番目の項によって全体的な速度の向上度が制限されます。

$$\frac{1}{S} = F + \frac{(1-F)}{P}$$

次の図に、この概念を示します。灰色の部分がプログラム中の逐次処理部分を表現しています。この部分は1、2、4、8プロセッサ各々の場合で一定であるのに対し、斜線部分がプログラムの並列処理部分で、複数のプロセッサ間で一様に分割され、処理時間が短くなっています。

図3–3 固定された問題の処理速度の向上

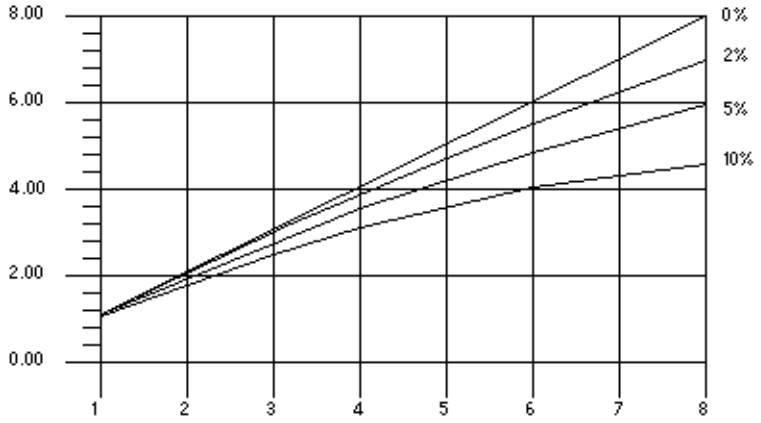


プロセッサの数が増加するにつれ、各プログラムの並列処理部分の所要時間は減少していますが、各プログラムの逐次処理部分は同じままです。

ただし実際には、複数のプロセッサに作業を分散し通信するためのオーバーヘッドが存在します。このようなオーバーヘッドは、プロセッサの数に対して一定であったり、そうでなかったりします。

次の図には、プログラムに逐次処理部分がそれぞれ 0%、2%、5%、10% 含まれる場合の理想的な速度向上が示されています。この図では、オーバーヘッドは想定されていません。

図 3–4　アムダールの法則による処理速度向上の曲線



グラフは、オーバーヘッドがないと想定した場合、0%、2%、5%、および 10% の逐次処理部分を含むプログラムでの理想的な速度向上を示しています。横軸はプロセッサの数、縦軸は速度を表しています。

3.5.1.1　オーバーヘッド

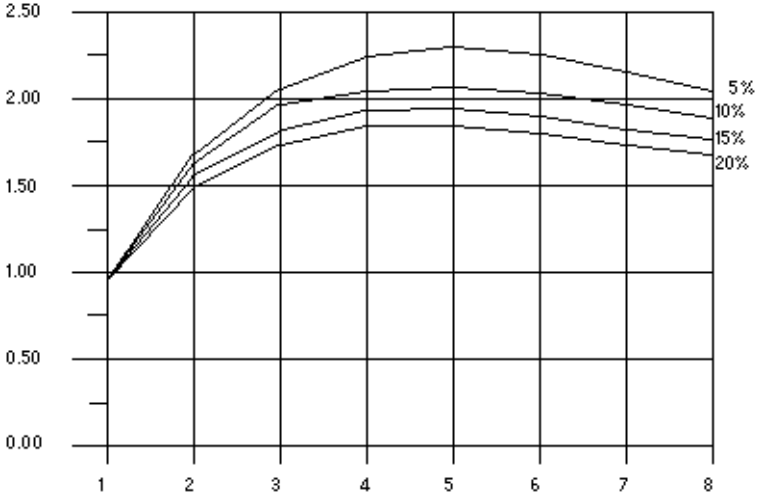
モデルにオーバーヘッドの影響を取り入れると、速度向上の曲線は大幅に変わります。ここでは、説明上 2 つの部分、つまり、プロセッサの数には無関係な固定部分と、使用されるプロセッサの 2 乗で増加する可変部分から成るオーバーヘッドを想定します。

$$\frac{1}{S} = \frac{1}{F + \left(1 - \frac{F}{P}\right) + K_1 + K_2 P^2}$$

S 分の 1 は、{F プラス (1 マイナス P 分の F) プラス K1 プラス K2 P 二乗} 分の 1 に等しいです。

この式で K_1 と K_2 は一定の係数です。この仮定では、速度向上の曲線は次の図のようになります。この場合、速度の向上にピーク点があることに注目してください。ある点を越えると、プロセッサを増加させてもパフォーマンスが下がり始めます。

図 3–5 オーバーヘッドがある場合の速度向上の曲線



グラフは、すべてのプログラムで 5 プロセッサのときにもっとも処理速度が早く、8 プロセッサまで増えると徐々に遅くなることを示しています。横軸はプロセッサの数、縦軸は速度を表しています。

3.5.1.2 ガスタフソンの法則

アムダールの法則では、実際の問題を並列化するときの速度向上の効果を正しく予測できません。プログラムの逐次処理部分に費やされる時間の割合は、問題のサイズに依存することがあります。つまり、問題のサイズが増加すると、速度向上の可能性が大きくなる場合があります。例を使って説明します。

例 3–12 問題サイズの拡大により速度向上の可能性が大きくなることがある

```
/*
* initialize the arrays
*/
for (i=0; i < n; i++) {
    for (j=0; j < n; j++) {
            a[i][j] = 0.0;
            b[i][j] = ...
            c[i][j] = ...
    }
}
/*
* matrix multiply
*/
for (i=0; i < n; i++) {
    for(j=0; j < n; j++) {
            for (k=0; k < n; k++) {
                a[i][j] = b[i][k]*c[k][j];
            }
    }
}
```

理想的にオーバーヘッドがゼロで、2 番目に入れ子にされたループが並列に実行されると仮定すると、問題のサイズが小さい場合 (すなわち n の値が小さい) と、プログラムの順次実行部分と並列実行部分の大きさがそれほど違わないことがわかります。ところが、n が大きくなると、並列実行部分に費やされる時間が順次実行の部分に対するものよりも早い勢いで大きくなります。この問題の場合は、問題のサイズが大きくなるにつれてプロセッサの数を増やす方法が有効です。

3.6 負荷バランスとループのスケジューリング

並列ループの繰り返しを複数のスレッドに分散する作業を「ループのスケジューリング」といいます。速度を最大限に向上させるには、作業をスレッドに均等に分散することによって、オーバーヘッドがあまり発生しないようにすることが重要です。コンパイラは、異なる状況に合わせて、いくつかの種類のスケジューリングをすることができます。

3.6.1 静的 (チャンク) スケジューリング

ループの個々の繰り返しが実行する作業が同じである場合には、システムの複数のスレッドに均一に作業を分散すると効果があります。この方法を静的スケジューリングといいます。

例 3–13 静的スケジューリングに向いたループ

```
for (i=1; i < 1000; i++) {
    sum += a[i]*b[i];       /* S1 */
}
```

静的スケジューリング (チャンクスケジューリングともいう) では、各スレッドは同じ回数の繰り返しを実行します。たとえばスレッドの数が 4 であれば、前述の例では、各スレッドで 250 回の繰り返しが実行されます。割り込みが発生しないものと仮定し、各スレッドが同じ早さで作業を進行していくと、すべてのスレッドが同時に終了します。

3.6.2 セルフスケジューリング

各繰り返しで実行する作業が異なる場合、静的スケジューリングでは、一般に、よい負荷バランスを得ることができなくなります。静的スケジューリングでは、すべてのスレッドが、同じ回数の繰り返しを処理します。マスタースレッドを除くすべてのスレッドは、実行を終了すると、次の並列部分が検出されるまで待つことになります。残りのプログラムの実行はマスタースレッドが行います。セルフスケジューリングでは、各スレッドが異なる小さな繰り返しを処理し、割り当てられた処理が終了すると、同じループのさらに別の繰り返しを実行することになります。

3.6.3 ガイド付きセルフスケジューリング

ガイド付きセルフスケジューリング (Guided Self Scheduling、GSS) では、各スレッドは、連続した小数の繰り返しをいくつか受け持ちます。各繰り返しで作業量が異なるような場合には、GSS によって、負荷のバランスが保たれるようになります。

3.7 ループの変換

コンパイラは、プログラム中のループを並列に実行できるようにするために、ループ再構成のための変換を数回実行します。この変換のいくつかは、シングルプロセッサ上でのループの実行速度も向上させます。コンパイラが実行する変換を次で説明します。

3.7.1 ループの分散

ループには、並列に実行できる文とできない文とが存在することがあります。通常、並列実行できない文はごく少数です。ループの分散によって、これらの文を別のループに移動し、並列実行可能な文だけから成るループを作ります。これを次の例で説明します。

例 3–14 ループの分散に適したコード

```
for (i=0; i < n; i++) {
    x[i] = y[i] + z[i]*w[i];              /* S1 */
    a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0;  /* S2 */
```

例 3–14 ループの分散に適したコード (続き)

```
    y[i] = z[i] - x[i];                    /* S3 */
}
```

配列 x、y、w、a、z が重なりあっていないと仮定すると、文 S1 および S3 を並列実行することはできますが、文 S2 はできません。このループを異なる 2 個のループに分割すると次のようになります。

例 3–15 分散されたループ

```
/* L1: parallel loop */
for (i=0; i < n; i++) {
    x[i] = y[i] + z[i]*w[i];              /* S1 */
    y[i] = z[i] - x[i];                   /* S3 */
}
/* L2: sequential loop */
for (i=0; i < n; i++) {
    a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /* S2 */
}
```

この変換のあと、前述のループ L1 には並列実行を妨害する文が含まれていないので、これを並列実行できるようになります。ところが、2 番目のループ L2 は元のループの並列実行できない部分を引き継いだままです。

ループの分散は、常に効果があって安全に実行できるとはかぎりません。コンパイラは、この効果と安全性を確認するための解析を実行します。

3.7.2 ループの融合

ループが小さい、すなわちループでの作業量が少ないと、大幅にパフォーマンスを向上させることはできません。これは、ループでの作業量に比べて、並列ループを起動するときのオーバーヘッドが大きくなるためです。このような状況では、コンパイラはループの融合を使用して、いくつかのループを 1 つの並列ループに融合し、ループを大きくします。同じ回数の繰り返しを行うループが隣接していると、ループの融合は簡単にしかも安全に行われます。次の例を考えてみましょう。

例 3–16 作業量の少ないループ

```
/* L1: short parallel loop */
for (i=0; i < 100; i++) {
    a[i] = a[i] + b[i];        /* S1 */
}
/* L2: another short parallel loop */
for (i=0; i < 100; i++) {
    b[i] = a[i] * d[i];        /* S2 */
}
```

この例では、2 個の小さなループが隣どうしに記述されていて、次のように安全に融合することができます。

例 3–17 融合された 2 つのループ

```
/* L3: a larger parallel loop */
for (i=0; i < 100; i++) {
    a[i] = a[i] + b[i];       /* S1 */
    b[i] = a[i] * d[i];       /* S2 */
}
```

これによって、並列ループの実行によるオーバーヘッドを半分にすることができます。ループの融合は、別の場合にも役に立ちます。たとえば、同じデータが 2 個のループで参照されている場合には、この 2 個のループを融合すると、参照を局所的なものにすることができます。

ただし、ループの融合は常に安全に実行できるとはかぎりません。ループの融合によって、元々存在していなかったデータの依存関係が生成されると、実行結果が正しくなくなることがあります。次の例を考えてみましょう。

例 3–18 安全でない融合の例

```
/* L1: short parallel loop */
for (i=0; i < 100; i++) {
    a[i] = a[i] + b[i];      /* S1 */
}
/* L2: a short loop with data dependence */
for (i=0; i < 100; i++) {
    a[i+1] = a[i] * d[i];    /* S2 */
}
```

85 ページの「3.7.2 ループの融合」でループの融合が実行されると、文 S2 から S1 に対するデータの依存性が生成されます。実際に、文 S1 の右辺にある `a[i]` の値が、文 S2 で計算されるものになります。これはループが融合されないと起こりません。コンパイラは、ループの融合を実行すべきかどうかを判断するために解析を行い、安全性と有効性を確認します。場合によっては、任意の数のループを融合できることがあります。このような方法でループの作業量を多くすると、並列実行が十分に有効であるようなループを生成することができます。

3.7.3 ループの交換

入れ子になっているループのもっとも外側のループを並列化すると、発生するオーバーヘッドが小さいために、一般に大きな効果が期待できます。しかし、そのループに依存性がある場合は、並列化することは安全ではありません。次の例で説明します。

例 3–19 並列化できない入れ子のループ

```
for (i=0; i <n; i++) {
    for (j=0; j <n; j++) {
            a[j][i+1] = 2.0*a[j][i-1];
    }
}
```

この例では、添字変数 i を持つループは、連続する 2 つの繰り返しで依存関係があるために、並列化することができません。ただし、2 つのループを交換することができるため、交換すると並列ループ (j のループ) が今度は外側のループになります。

例 3–20 交換されたループ

```
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
            a[j][i+1] = 2.0*a[j][i-1];
    }
}
```

この結果生成されたループでは並列作業の分散に対するオーバーヘッドが 1 回で済むのに対して、元のループでは、n 回必要でした。コンパイラは、これまで説明したように、ループの交換をするかどうか決定するための解析を行い、安全性と有効性を確認します。

3.8 別名と並列化

ISO C の別名を使用すると、ループを並列化できなくなることがあります。別名とは、2 個の参照が記憶領域の同じ位置を参照する可能性のある場合に発生します。次の例を考えてみましょう。

例 3–21 同じ記憶領域への参照を持つループ

```
void copy(float a[], float b[], int n) {
    int i;
    for (i=0; i < n; i++) {
            a[i] = b[i]; /* S1 */
    }
}
```

変数 a および b は引数であるため、次のように呼ばれる場合には、a および b が重なりあった記憶領域を参照している可能性があります。次のようなルーチン copy が呼び出される例を考えてみましょう。

```
copy (x[10], x[11], 20);
```

呼び出された側では、copy ループの連続した 2 回の繰り返しが、配列 x の同じ要素を読み書きしている可能性があります。しかし、ルーチン copy が次のように呼び出された場合には、実行される 20 回の繰り返しループで、重なりあう可能性がなくなります。

```
copy (x[10], x[40], 20);
```

一般的に、ルーチンがどのように呼び出されるかをコンパイラが知っていないかぎり、この状況を正しく解析することは不可能になります。ANSI/ISO C では、ANSI C に対する拡張キーワードを装備することで、このような別名の問題に対して指示することが可能になっています。詳細については、88 ページの「3.8.2 制限付きポインタ」を参照してください。

3.8.1 配列およびポインタの参照

別名の問題の一因は、配列参照とポインタ計算演算を定義できる C 言語の性質にあります。効率的にループを並列化するためには、プラグマを自動的または明示的に使用して、配列として配置されているすべてのデータを、ポインタではなく C の配列参照の構文を使用して参照する必要があります。ポインタ構文が使用されると、コンパイラはループの異なる繰り返し間でのデータの関係を解析できなくなります。そのため、安全性を考慮してループを並列化しなくなります。

3.8.2 制限付きポインタ

コンパイラが効率よくループを並列化できるようにするには、左辺値が記憶領域の特定の領域を示していなければいけません。別名とは、記憶領域の決まった位置を示していない左辺値のことです。オブジェクトへの 2 個のポインタが別名であるかどうかを判断することは困難です。これを判断するにはプログラム全体を解析することが必要であるため、非常に時間がかかります。次の関数 `vsq()` を考えてみましょう。

例 3–22　2 個のポインタを使用したループ

```
void vsq(int n, double * a, double * b) {
    int i;
    for (i=0; i<n; i++) {
            b[i] = a[i] * a[i];
    }
}
```

ポインタ `a` および `b` が異なるオブジェクトをアクセスすることをコンパイラが知っている場合には、ループ内の異なる繰り返しを並列に実行することができます。しかし、ポインタ `a` および `b` でアクセスされるオブジェクトが重なりあっていれば、ループを安全に並列実行できなくなります。コンパイル時に関数 `vsq()` を単純に

解析するだけでは、a および b によるオブジェクトのアクセスが重なりあっているかどうかを知ることはできません。この情報を得るには、プログラム全体を解析することが必要になります。

制限付きポインタを使ってオブジェクトを明確に区別すると、コンパイラによるポインタ別名の解析が実行可能になります。次に、vsq() の関数パラメータを制限付きポインタとして宣言した例を示します。

```
void vsq(int n, double * restrict a, double * restrict b)
```

ポインタ a および b が制限付きポインタとして宣言されているので、a および b で示された記憶領域が区別されていることがわかります。この別名情報によって、コンパイラはループの並列化を実行することができます。

キーワード restrict は volatile に似た型修飾子で、ポインタ型に対して有効です。なお、restrict は -Xs モードでコンパイルする場合を除き、-xc99=all を使用する場合に有効なキーワードです。ソースコードを変更しない場合があります。その場合、次のコマンド行オプションを使用して、ポインタ型の値をとる関数の引数を restrict ポインタとして扱うように指定できます。

-xrestrict=[*func1*,...,*funcn*]

関数リストが指定されている場合、指定された関数内のポインタパラメータは制限付きとして扱われます。指定されていない場合は、C ファイル全体のすべてのポインタパラメータが制限付きとして扱われます。たとえば、-xrestrict=vsq を使用すると、前述の関数 vsq() についての最初の例では、ポインタ a および b がキーワード restrict によって修飾されます。

restrict を正しく使用することはとても重要です。区別できないオブジェクトを指しているポインタを制限付きポインタにしてしまうと、ループを正しく並列化することができなくなり、不定な動作をすることになります。たとえば、関数 vsq() のポインタ a および b が重なりあっているオブジェクトを指している場合には、b[i] と a[i+1] などが同じオブジェクトである可能性があります。このとき a および b が制限付きポインタとして宣言されていなければ、ループは順次実行されます。a および b が間違って制限付きであると宣言されていれば、コンパイラはループを並列実行するようになりますが、この場合 b[i+1] の結果は b[i] をを計算したあとでなければ得られないので、安全に実行することはできません。

3.8.3 明示的な並列化およびプラグマ

すでに述べたように、並列化の適用や有効性をコンパイラだけで決めるには、情報が不十分なことがあります。コンパイラはプラグマをサポートしており、コンパイラだけでは不可能なループの並列化を効率よく実行することができます。この節の残りの部分で説明する従来の Sun 固有の MP プラグマは、OpenMP 規格に準拠するため非推奨になりました。標準命令については、『OpenMP API ユーザーズガイド』を参照してください。

3.8.3.1 直列プラグマ

注 – 従来の Sun 固有の MP プラグマは推奨されず、サポートされません。代わりに、OpenMP 3.0 規格で規定された API をサポートします。標準命令への移植については、『OpenMP API ユーザーズガイド』を参照してください。

直列プラグマには 2 通りあり、どちらも `for` ループに適用されます。

- `#pragma MP serial_loop`
- `#pragma MP serial_loop_nested`

`#pragma MP serial_loop` プラグマは、次に存在する `for` ループを自動的に並列化しないことをコンパイラに指示します。

`#pragma MP serial_loop_nested` プラグマは、次に存在する `for` ループ、およびその `for` ループの中で入れ子になっている `for` ループを自動的に並列化しないことをコンパイラに指示します。

これらのプラグマのスコープは、そのプラグマから始まり、次のブロックの始まりか現在のブロック内のプラグマに続く最初の for ループ、または現在のブロックの終わりのいずれか先に達したところで終わります。

3.8.3.2 並列プラグマ

注 – 従来の Sun 固有の MP プラグマは推奨されず、サポートされません。代わりに、OpenMP 3.0 規格で規定された API をサポートします。標準命令への移植については、『OpenMP API ユーザーズガイド』を参照してください。

並列プラグマは 1 つだけあります。`#pragma MP taskloop [`*options*`]`

`MP taskloop` プラグマは、オプションとして、次の引数を取ることができます。

- `maxcpus` *(プロセッサ数)*
- `private` *(スレッド固有変数リスト)*
- `shared` *(共有変数リスト)*
- `readonly` *(読み取り専用変数リスト)*
- `storeback` *(ストアバック変数リスト)*
- `savelast`
- `reduction` *(縮約変数リスト)*
- `schedtype` *(スケジューリング型)*

このプラグマのスコープは、プラグマから始まり、次のブロックの始まりか現在のブロック内のプラグマに続く最初の for ループ、または現在のブロックの終わりのいずれか先に達したところで終わります。プラグマは、スコープの終端に到達した時点で最初に見つかった for ループに適用されます。

オプションは、1 つの MP taskloop pragma に 1 つだけ指定できます。ただし、プラグマは蓄積されて、スコープ内の最初に見つかった for ループに適用されます。

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop shared(a,b)
#pragma MP taskloop storeback(x)
```

これらのオプションは、for ループの前に複数回指定できます。オプションが衝突を起こす場合には、コンパイラによって警告メッセージが出力されます。

for ループの入れ子

MP taskloop プラグマは、現在のブロック内にある次の for ループに適用されます。Sun ANSI/ISO C によって並列化された for ループに入れ子は存在しません。

並列化の適切性

MP taskloop プラグマは、for ループを並列化するように指示します。

不規則なフロー制御や、一定しない増分による繰り返しを持った for ループに対しては、正当な並列化を実行できません。たとえば、setjmp、longjmp、exit、abort、return、goto、labels、break を含んだ for ループは並列化に適しません。

特に重要なこととして、繰り返し間の依存性を持った for ループでも、明示的に並列化できる点に注意してください。すなわち、このようなループに対して MP taskloop プラグマが指定されていると、for ループが並列化に適していないと判断されないかぎり、単にこの指示に従って並列化を実行してしまいます。このような明示的な並列化を行なった場合は、不正確な結果が発生しないかを確認してください。

1 つの for ループに対して serial_loop または serial_loop_nested と taskloop の両方のプラグマが指定されている場合には、最後の指定が優先的に使用されます。

次の例を考えてみましょう。

```
#pragma MP serial_loop_nested
    for (i=0; i<100; i++) {
   # pragma MP taskloop
      for (j=0; j<1000; j++) {
      ...
 }
}
```

この例では、i ループは並列化されませんが、j ループは並列化されます。

プロセッサの数

#pragma MP taskloop maxcpus (プロセッサ数) は、指定が可能であれば、現在のループに対して使用されるプロセッサの数を指定します。

maxcpus に指定する値は正の整数でなければいけません。maxcpus が 1 であれば、指定されたループは直列に実行されます。なお、maxcpus を 1 に指定した場合には、serial_loop プラグマを指定したことと同等になる点に注意してください。また、maxcpus の値か PARALLEL 環境変数のどちらか小さい方の値が使用されます。環境変数 PARALLEL が指定されていない場合には、この値に 1 が指定されているものとして扱われます。

1 つの for ループに複数の maxcpus プラグマが指定されている場合には、最後に指定された値が優先的に使用されます。

変数の分類

ループに使用される変数は、private、shared、reduction、または readonly のどれかに分類されます。1 つの変数は、これらの種類のうち 1 つにのみ属します。変数の種類を reduction または readonly にするには、明示的にプラグマで指示しなければいけません。#pragma MP taskloop reduction および #pragma MP taskloop readonly を参照してください。変数を private または shared にするには明示的にプラグマを使用するか、または次のスコープの規則に基づいて決まります。

スレッド private 変数と shared 変数のデフォルトのスコープの規則

スレッド private 変数は、for ループのある繰り返しを処理するためにそれぞれのプロセッサが専用に使用する値を保持します。別の言い方をすれば、for ループのある繰り返しでスレッド private 変数に割り当てられた値は、for のループの別の繰り返しを処理しているプロセッサからは見えません。これに対して shared 変数は、for ループの繰り返しを処理しているすべてのプロセッサから現在の値にアクセスできる変数のことです。ループのある繰り返しを処理しているプロセッサが shared 変数に代入した値は、そのループの別の繰り返しを処理しているプロセッサからでも見ることができます。共有変数を参照しているループを #pragma MP taskloop 指令によって明示的に並列化する場合には、値の共有によって正確性に問題が起きないことを確認しなければいけません (競合条件の確認など)。明示的に並列化されたループの共有変数へのアクセスおよび更新では、コンパイラによる同期はとられません。

明示的に並列化されたループの解析において、変数がスレッド private と shared のどちらであるかを決定するために、次の「デフォルトのスコープの規則」が使用されます。

- 変数がプラグマによって明示的に分類されていない場合には、その変数がポインタまたは配列として宣言されていて、かつループ内では配列構文を使用して参照しているかぎり、その変数はデフォルトで shared 変数として分類されます。これ以外の場合は、スレッド private 変数として分類されます。
- ループのインデックス変数は常にスレッド private 変数として扱われ、また常に storeback 変数です。

明示的に並列化された for ループ内で使用されているすべての変数を、shared、private、reduction、または readonly として明示的に指定し、「デフォルトのスコープの規則」が適用されないようにしてください。

コンパイラは、共有変数に対するアクセスの同期を一切実行しないので、たとえば、配列参照を含んだループに対して MP taskloop プラグマを使用する前には、十分な考察が必要になります。このように明示的に並列化されたループで、繰り返し間でのデータ依存性がある場合には、並列実行を行うと正しい結果を得られないことがあります。コンパイラによって、このような潜在的な問題を検出し、警告メッセージを出力することもできますが、一般的にこれを検出することは非常に困難です。なお、共有変数に対する潜在的な問題を持ったループでも、明示的に並列化を指示されると、コンパイラはこの指示に従います。

private 変数

```
#pragma MP taskloop private (スレッド固有変数)
```

このプラグマは、現在のループでスレッド固有変数として扱われる必要のあるすべての変数を指定するために使用します。ループで使用されている別の変数は、それ自体が明確にshared、readonly、または reduction であることが指定されていないかぎり、デフォルトのスコープの規則に従って、shared またはスレッド private のどちらかに分類されます。

スレッド private 変数は、ループのある繰り返しを処理するためにそれぞれのプロセッサが専用に使用する値を保持します。別の言い方をすれば、ループのある繰り返しを処理しているプロセッサによってスレッド private 変数に代入された値は、そのループの別の繰り返しを処理しているプロセッサから見ることはできません。スレッド private 変数には、ループの繰り返しの開始時に初期値は代入されず、繰り返し内で最初に使用される前に、その繰り返し内で値が代入されなければいけません。値が設定される前にその値を参照するように明確に宣言されたスレッド private 変数を持つループを実行すると、その動作は保証されません。

shared 変数

```
#pragma MP taskloop shared (共有変数リスト)
```

このプラグマは、現在のループでスレッド shared 変数として扱われる必要のあるすべての変数を指定するために使用します。ループで使用されている別の変数は、それ自体が明確にスレッド private、 readonly、storeback、または reduction であることが指定されていないかぎり、デフォルトのスコープの規則に従って、shared またはスレッド private のどちらかに分類されます。

shared 変数とは、ある for ループの繰り返しを処理しているすべてのプロセッサから現在の値を見ることのできる変数のことです。ループのある繰り返しを処理しているプロセッサが shared 変数に代入した値は、そのループの別の繰り返しを処理しているプロセッサからでも見ることができます。

readonly 変数

`#pragma MP taskloop readonly` *(読み取り専用変数リスト)*

`readonly` 変数は、ループの繰り返しで変更されない共有変数の特殊なクラスです。変数を読み取り専用として指定すると、ループの繰り返しを処理しているそれぞれのプロセッサに対して、個々にコピーされた変数値が使用されます。

storeback 変数

`#pragma MP taskloop storeback` *(ストアバック変数リスト)*

このプラグマは、現在のループで `storeback` 変数として扱われる必要のあるすべての変数を指定するために使用します。

`storeback` 変数とは、ループの中で変数値が計算され、その値がループの終了後に使用される変数のことです。ループの最後の繰り返しにおける `storeback` 変数の値が、ループの終了後に利用可能になります。このような変数は、その変数が明示的な宣言やデフォルトのスコープ規則によってスレッド固有変数となっている場合には、この指令を使用して明示的に `storeback` 変数として宣言するとよいでしょう。

なお、`storeback` 変数に対する最終的な戻し操作 (ストアバック操作) は、明示的に並列化されたループの最後の繰り返しにおいて、その中で実際に `storeback` の値が変更されたかどうかには関係なく実行される点に注意してください。すなわち、ループの最後の繰り返しを処理するプロセッサと、`storeback` 変数の最終的な値を保持　ているプロセッサとは、異なる可能性があります。次の例を考えてみましょう。

```
#pragma MP taskloop private(x)
#pragma MP taskloop storeback(x)
   for (i=1; i <= n; i++) {
      if (...) {
          x=...
      }
}
   printf (“%d”, x);
```

前述の例では、`printf()` 呼び出しによって出力される `storeback` 変数 `x` の値は、`i` ループを直列に実行した場合の出力値とは異なる可能性があります。なぜならば、明示的に並列化されたループでは、ループの最後の繰り返し (すなわち `i==n` のとき) を処理し、`x` に対してストアバック操作を行うプロセッサは、現在最後に更新された `x` の値を保持するプロセッサとは同じでないことがあるからです。このような潜在的な問題に対し、コンパイラは警告メッセージを出力します。

明示的に並列化されたループでは、配列として参照される変数を `storeback` 変数としては扱いません。したがって、このような変数にストアバック処理が必要な場合 (たとえば、配列として参照される変数がスレッド固有変数として宣言されている場合) には、その変数を ストアバック変数リストに含める必要があります。

savelast

`#pragma MP taskloop savelast`

このプラグマは、ループ内のすべてのスレッド固有変数をストアバック変数として扱うために使用します。このプラグマの構文を次に示します。

`#pragma MP taskloop savelast`

各変数をストアバック変数として宣言するときには、それぞれのスレッド固有変数をリストするよりも、この形式が便利であることがよくあります。

reduction 変数

`#pragma MP taskloop reduction` *(list_of_reduction_variables)* このプラグマは、縮約変数リストにあるすべての変数が、そのループに対して `reduction` 変数として扱われるために使用します。`reduction` 変数とは、ループのある繰り返しを処理している個々のプロセッサによって、その値が部分的に計算され、最終値がすべての部分値から計算される変数のことをいいます。`reduction` 変数リストにより、そのループが縮約ループであることをコンパイラに指示し、適切な並列縮約用のコードを生成できるようにします。次の例を考えてみましょう。

```
#pragma MP taskloop reduction(x)
    for (i=0; i<n; i++) {
         x = x + a[i];
}
```

ここでは変数 *x* が `(sum)` 縮約変数であり、`i` ループが `(sum)` 縮約ループになっています。

スケジューリングの制御

Solaris Studio ISO C コンパイラには、指定されたループのスケジューリングを戦略的に制御するために、`taskloop` プラグマと同時に使用するいくつかのプラグマが用意されています。このプラグマの構文を次に示します。

`#pragma MP taskloop schedtype` *(スケジューリング型)*

このプラグマによって、並列化されたループをスケジュールするためのスケジューリング型を指定することができます。スケジューリング型には、次のいずれかを指定できます。

- `static`

  `static` スケジューリングでは、ループのすべての繰り返しが、そのループを処理するすべてのプロセッサに均等に配分されます。次の例を考えてみましょう。

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(static)
```

```
    for (i=0; i<1000; i++) {
...
}
```

前述の例では、4 個のプロセッサが、ループの繰り返しを 250 ずつ処理します。

- `self` [*(chunk_size)*]

  `self` スケジューリングでは、ループのすべての繰り返しが処理されるまで、固定された回数の繰り返しチャンクサイズを、そのループを処理するそれぞれのプロセッサで処理します。オプションの *chunk_size* には、使用するチャンクサイズを指定します。*chunk_size* は、正の整定数か、もしくは整数型の変数でなければいけません。変数の *chunk_size* が指定された場合は、そのループを開始する前に、その変数が正の整数値であるかどうかが評価されます。最小チャンクサイズが指定されていない場合、もしくは、この値が正でない場合、チャンクサイズはコンパイラによって決められます。次の例を考えてみましょう。

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(self(120))
for (i=0; i<1000; i++) {
...
}
```

前述の例では、ループを処理するそれぞれのプロセッサに割り当てられる繰り返し数は、割り当て順に解釈すると次のようになります。

120、120、120、120、120、120、120、120、40。

- `gss` [(*min_chunk_size*)]

  `guided self` スケジューリングでは、ループのすべての繰り返しが処理されるまで、可変数の繰り返し (「最小チャンクサイズ」) を、そのループを処理するそれぞれのプロセッサで処理します。オプションの *min_chunk_size* を指定すると、可変なチャンクサイズが最低でも *min_chunk_size* になるように設定されます。*min_chunk_size* は、正の整定数か、もしくは整数型の変数でなければいけません。変数の *min_chunk_size* が指定された場合は、そのループを開始する前に、その変数が正の整数値であるかどうかが評価されます。最小チャンクサイズが指定されていない場合、もしくは、この値が正でない場合、チャンクサイズはコンパイラによって決められます。次の例を考えてみましょう。

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(gss(10))
for (i=0; i<1000; i++) {
...
}
```

前述の例では、ループを処理するそれぞれのプロセッサに割り当てられる繰り返し数は、割り当て順に解釈すると次のようになります。

250、188、141、106、79、59、45、33、25、19、14、11、10、10、10。

- `factoring` [(*min_chunk_size*)]

factoring スケジューリングでは、ループのすべての繰り返しが処理されるまで、可変数の繰り返し (「最小チャンクサイズ」) を、そのループを処理するそれぞれのプロセッサで処理します。オプションの *min_chunk_size* を指定すると、可変なチャンクサイズが最低でも *min_chunk_size* になるように設定されます。*min_chunk_size* は、正の整定数か、もしくは整数型の変数でなければいけません。変数の *min_chunk_size* が指定された場合は、そのループを開始する前に、その変数が正の整数値であるかどうかが評価されます。最小チャンクサイズが指定されていない場合、もしくは、この値が正でない場合、チャンクサイズはコンパイラによって決められます。次の例を考えてみましょう。

```
#pragma MP taskloop maxcpus(4)
#pragma MP taskloop schedtype(factoring(10))
for (i=0; i<1000; i++) {
...
}
```

前述の例では、ループを処理するそれぞれのプロセッサに割り当てられる繰り返し数は、割り当て順に解釈すると次のようになります。

125、125、125、125、62、62、62、62、32、32、32、32、16、16、16、16、10、10、10、10、

3.9 メモリーバリアー組み込み関数

コンパイラには、SPARC プロセッサと x86 プロセッサ用のさまざまなメモリーバリアー組み込み関数を定義するヘッダーファイル mbarrier.h が用意されています。これらの組み込み関数は、開発者が独自の同期プリミティブを使用してマルチスレッドコードを記述するために使用できます。これらの組み込み関数が特定の状況で必要かどうか、また、いつ必要かを判断するには、ご使用のプロセッサのドキュメントを参照することをお勧めします。

mbarrier.h によりサポートされるメモリーオーダリング組み込み関数

- __machine_r_barrier() — これは、*read* バリアーです。これにより、バリアー前のすべてのロード操作が、バリアー後のすべてのロード操作の前に完了します。
- __machine_w_barrier() — これは、*write* バリアーです。これにより、バリアー前のすべての格納操作が、バリアー後のすべての格納操作の前に完了します。
- __machine_rw_barrier() — これは、*read—write* バリアーです。これにより、バリアー前のすべてのロードおよび格納操作が、バリアー後のすべてのロードおよび格納操作の前に完了します。
- __machine_acq_barrier() — これは、*acquire* セマンティクスを持つバリアーです。これにより、バリアー前のすべてのロード操作が、バリアー後のすべてのロードおよび格納操作の前に完了します。
- __machine_rel_barrier() — これは、*release* セマンティクスを持つバリアーです。これにより、バリアー前のすべてのロードおよび格納操作が、バリアー後のすべての格納操作の前に完了します。

- `__compiler_barrier()` — コンパイラが、バリアーを越えてメモリーアクセスを移動しないようにします。

`__compiler_barrier()` 組み込み関数を除くすべてのバリアー組み込み関数は、メモリーオーダリング組み込み関数を生成します。x86 では、`mfence`、`sfence`、または `lfence` 組み込み関数で、SPARC プラットフォームでは `membar` 組み込み関数です。

`__compiler_barrier()` 組み込み関数は、命令を生成せず、代わりに今後メモリー操作を開始する前にそれまでのメモリー操作をすべて完了する必要があることをコンパイラに通知します。この実際の結果として、ローカルでないすべての変数、および `static` 記憶クラス指定子を持つローカル変数が、バリアー前のメモリーに再度格納されてバリアー後に再ロードされるため、コンパイラではバリアー前のメモリー操作とバリアー後のメモリー操作が混在することはありません。ほかのすべてのバリアーには、`__compiler_barrier()` 組み込み関数の動作が暗黙的に含まれています。

たとえば、次のコードでは、`__compiler_barrier()` 組み込み関数が存在しているためコンパイラによる 2 つのループのマージが止まります。

```
#include "mbarrier.h"
int thread_start[16];
void start_work()
{
/* Start all threads */
   for (int i=0; i<8; i++)
   {
     thread_start[i]=1;
   }
   __compiler_barrier();
/* Wait for all threads to complete */
   for (int i=0; i<8; i++)
   {
      while (thread_start[i]==1){}
   }
}
```

◆ ◆ ◆ 第 4 章

4 lint ソースコード検査プログラム

この章では、lint プログラムを使用して C のコードを検査し、コンパイルの失敗や、実行時に予期しない結果を招く可能性のあるエラーを見つける方法を説明します。多くの場合 lint は、コンパイラが必ずしも検出しない誤ったコード、エラーを起こしやすいコード、あるいは標準外コードについて警告を出します。

lint プログラムは C コンパイラにより生成されるすべてのエラーと警告のメッセージを表示します。さらに潜在的バグと移植上の問題に関する警告も表示します。多くの場合、lint から表示されたメッセージは、プログラムのサイズと必要な記憶領域を縮小し、全体の効率を改善する手助けとなります。

lint プログラムはコンパイラと同じロケールを使用し、lint の出力は stderr に送られます。型に基づく別名の明確化を実行する前に、lint を使用してコードをチェックする詳細と例については、128 ページの「4.6.3 lint フィルタ」を参照してください。

4.1 基本 lint と拡張 lint

lint プログラムは次の 2 つのモードで動作します。

- 基本モード - デフォルトの lint プログラムです。
- 拡張モード - 基本 lint で実行される処理に加えて、さらに詳しい別のコード解析を行います。

基本 lint でも拡張 lint でも、ファイル全域 (ライブラリを含む) で矛盾した定義や使用を検出し、ファイルを個別に独立して処理する C コンパイラの不足を補います。特に大きなプロジェクト環境において 1 つの関数が何百ものモジュールで使用される場合、lint は、ほかの方法で探し出すことが困難なバグを発見するのに役立ちます。たとえば、期待しているよりも 1 つ少ない引数で呼び出された関数は、呼び出し時にプッシュされなかった値をスタックから取り出し、そのスタック位置のメモリーの状態によって正しい結果や間違った結果を返します。このような依存性

やマシンアーキテクチャーへの依存性を検出することにより、lint はユーザー自身のマシンや別のマシンで実行されるコードを確かなものにすることができます。

拡張モードでは、lint は基本モードの場合よりさらに詳しい報告を出します。基本モードの lint には次の機能が含まれています。

- ソースプログラムの構造およびフロー解析
- 定数の伝播と定数式の評価
- 制御フローとデータフローの解析
- データ型使用状況の解析

拡張モードでは、lint は次の問題を検出することができます。

- 使用されていない #include 指令、変数、手続き
- 解放後のメモリー使用
- 使用されていない割り当て
- 初期化前の変数値の使用
- 割り当てられていないメモリーの解放
- 定数データセグメントへの書き込み時のポインタの使用
- 等しくないマクロの再定義
- 到達しないコード
- 共用体での値の型利用の適合性
- 実際の引数の暗黙の型変換

4.2 lint 使用方法

lint プログラムは、コマンド行から起動します。基本モードで lint を起動するには、次のコマンドを使用します。

```
% lint file1.c file2.c
```

拡張 lint は -Nlevel または -Ncheck オプションを使用して呼び出します。たとえば、次のようにして拡張 lint を起動できます。

```
% lint -Nlevel=3 file1.c file2.c
```

lint は、2 つのパスでコードの検査をします。lint は、最初のパスでは C ソースファイルに個別のエラー条件を、第 2 のパスでは C ソースファイル間の不整合を検査します。このプロセスは、lint が -c を指定して呼び出されていなければユーザーには見えません。

```
% lint -c file1.c file2.c
```

この場合の lint は、最初のパスのみを実行し、第 2 のパスに関連する情報、つまり *file1*.c と *file2*.c 間の定義および仕様の不一致に関する情報を *file1*.ln および *file2*.ln と名づけられた中間ファイルに収集します。

```
% ls
file1.c
file1.ln
file2.c
file2.ln
```

このように、lint の -c オプションは cc の -c オプションがコンパイラのリンク編集段階を抑制するのと同じように動作します。一般に、lint のコマンド行構文は cc コマンド行構文に従っています。

次のように .ln ファイルに lint を実行します。

```
% lint file1.ln file2.ln
```

この場合、第 2 のパスは実行されます。lint は、そのコマンド行の順番で .c または .ln ファイルをいくつでも処理します。次のようなコマンド行があります。

```
% lint file1.ln file2.ln file3.c
```

このコマンド行は、*file3*.c の内部のエラーと 3 つのファイルすべての整合性を検査するように lint に指令します。

lint は cc と同じ順序でインクルードヘッダーファイルのディレクトリを検索します。cc の -I オプションを使用するように、lint の -I オプションを使用できます。62 ページの「2.16 インクルードファイルを指定する方法」を参照してください。

lint コマンド行には、複数のオプションを指定することができます。どのオプションも引数を取らず、複数の文字から成るオプションがない場合は、オプション文字を連結して指定することができます。

```
% lint -cp -Idir1 -Idir2 file1.c file2.c
```

このコマンドは lint に次のことを指示します。

- 第 1 のパスのみを実行する
- 移植性検査も実行する
- 指定されたディレクトリでインクルードするヘッダーファイルを検索する

lint にはオプションが数多くあります。これらのオプションを使うと、lint で特定の処理を実行し、特定の条件について報告することができます。

4.3 lint のオプション

lint プログラムは、静的なアナライザです。そのため、検出した依存性に関する実行時の結果を評価できません。たとえば、あまり重要ではない何百もの到達不可能な break 文を持ち、これについてユーザーがほとんど何もすることができないプログラムがあるとすると、lint はそれに忠実にフラグを立ててしまいます。この場合、lint のコマンド行オプションと指令 (ソーステキストに埋め込まれた特別のコメント) が役に立ちます。次にその例を示します。

- -b オプションを指定して lint を実行し、到達不可能な break 文に対するすべての警告を抑制することができます。
- 注釈 /*NOTREACHED*/ を到達不可能な文の前に付けて、その文に対する診断を抑制することができます。

lint のオプションを次にアルファベット順に説明します。いくつかの lint オプションは、lint 診断メッセージの抑制に関連しています。アルファベット順の説明のあと、表 4–8 にこれらのオプションとそれが抑制するメッセージの一覧を示します。拡張 lint を呼び出すオプションは -N で始まります。

lint は、-A、-D、-E、-g、-H、-O、-P、-U、-Xa、-Xc、-Xs、-Xt、-Y を含む多くの cc コマンド行オプションを認識しますが、-g と -O は無視します。認識されないオプションがあると警告が出され、そのオプションは無視されます。

4.3.1 -#

冗長モードをオンにし、呼び出すごとに各構成要素を表示します。

4.3.2 -###

呼び出すごとに各構成要素を表示しますが、実際には実行しません。

4.3.3 -a

一定のメッセージを抑制します。表 4–8 を参照してください。

4.3.4 -b

一定のメッセージを抑制します。表 4–8 を参照してください。

4.3.5 -C *filename;*

指定されたファイル名を持つ .ln ファイルを作成します。これらの .ln ファイルは lint の最初のパスだけで作成されます。*filename;* は絶対パス名でもかまいません。

4.3.6 -c

コマンド行で指定された .c ファイルごとに、lint の第 2 パスに関連する情報からなる .ln ファイルを作成します。第 2 パスは実行されません。

4.3.7 -dirout=*dir*

lint 出力ファイル (.ln ファイル) を入れるディレクトリを指定します。このオプションは -c オプションに影響を与えます。

4.3.8 -err=warn

-err=warn は -errwarn=%all のマクロです。108 ページの「4.3.15 -errwarn=*t*」を参照してください。

4.3.9 -errchk=*l*(, *l*)

l で指定した検査を実行します。デフォルトは、-errchk=%none です。-errchk を指定すると、-errchk=%all を指定する場合と同様に機能します。*l* には、次に示す 1 つまたは複数の項目をコンマで区切って指定します。たとえば、-errchk=longptr64,structarg のように指定します。

表 4–1 -errchk のフラグ

値	意味
%all	-errchk による検査をすべて実行します。
%none	-errchk による検査を行いません。これはデフォルト値です。
[no%]locfmtchk	printf のような書式文字列を lint の初回受け渡しで検査します。-errchk=locfmtchk を使用するかどうかに関係なく、lint は常に 2 回目の受け渡しで printf のような書式文字列を検査します。
[no%]longptr64	ロング整数、およびポインタのサイズが 64 ビットと標準整数のサイズが 32 ビットの環境への移植性を検査します。明示的なキャストが使用されている場合でも、ポインタ式とロング整数式の標準整数への代入を検査します。
[no%]structarg	値渡しされた構造体引数を検査します。仮引数の型が不明の場合は、その旨が報告されます。
[no%]parentheses	コード内の優先順位を明確に検査します。このオプションは、コードの保守性を高めるために使用します。-errchk=parentheses で警告が返された場合は、さらに括弧を使用して、コード内の演算の優先順位を明確に指示することを検討してください。
[no%]signext	符号なし整数型の式における符号付き整数値の符号拡張を、ISO C の通常の値保持規則が認める状態について検査します。このオプションは、-errchk=longptr64 が一緒に指定された場合にはエラーメッセージを出力するだけです。

表 4–1 -errchk のフラグ　(続き)

値	意味
[no%]sizematch	小さな整数に大きな整数が代入される場合について検査し、警告します。この警告は、サイズが同じであっても、符号が異なる整数間の代入 (unsigned int を signed int になど) についても出力されます。

4.3.10 -errfmt=*f*

lint 出力の書式を指定します。*f* には、macro、simple、src、tab のいずれか 1 つを指定できます。

表 4–2 -errfmt のフラグ

値	意味
macro	マクロを展開して、エラーのあるソースコード、行番号、場所を表示します。
simple	エラーのある行番号と場所番号 (大括弧内) を表示し、1 行の (簡単な) 診断メッセージを示します。-s オプションと同様ですが、エラー位置に関する情報が入っています。
src	エラーのあるソースコード、行番号、場所を表示します。マクロは展開しません。
tab	表形式で表示します。これはデフォルト値です。

デフォルトは -errfmt=tab です。-errfmt だけを指定すると、-errfmt=tab を指定するのと同じことになります。

複数の書式を指定すると最後に指定した書式が使用され、lint は使用されない書式について警告を出します。

4.3.11 -errhdr=*h*

-Ncheck も指定すると、lint でヘッダーファイルの一定のメッセージレポート作成ができます。*h* には、次の 1 つまたは複数の項目をコンマで区切って指定します。*dir*、no%*dir*、%all、%none、%user

表 4–3 -errhdr のフラグ

値	意味
dir	ディレクトリ *dir* からインクルードされたヘッダーファイル用の -Ncheck のメッセージを報告します。

表 4–3 -errhdr のフラグ (続き)

値	意味
no%*dir*	ディレクトリ *dir* からインクルードされたヘッダーファイル用の -Ncheck のメッセージを報告しません。
%all	使用されているすべてのヘッダーファイルを検査します。
%none	ヘッダーファイルを検査しません。これはデフォルト値です。
%user	使用されているすべてのユーザー定義のヘッダーファイルを検査します。すなわち、/usr/include およびそのサブディレクトリに入っているヘッダーファイルとコンパイラが提供しているヘッダーファイルを除く、すべてのヘッダーファイルを検査します。

デフォルトは -errhdr=%none です。-errhdr だけを指定すると、-errhdr=%user を指定するのと同じことになります。

次に例を示します。

```
% lint -errhdr=inc1 -errhdr=../inc2
```

この例は、ディレクトリ inc1 と ../inc2 内で使用されているヘッダーファイルを検査します。

```
% lint -errhdr=%all,no%../inc
```

この例は、ディレクトリ ../inc に入っているものを除く、使用されているすべてのヘッダーファイルを検査します。

4.3.12 -erroff=*tag*(, *tag*)

lintエラーメッセージ を抑制または使用可能にします。

t には、次の 1 つまたは複数の項目をコンマで区切って指定します。*tag*、no%*tag*、%all、%none。

表 4–4 -erroff のフラグ

値	意味
tag	*tag* で指定したメッセージを抑制します。-errtags=yes オプションで、メッセージのタグを表示することができます。
no%*tag*	*tag* で指定したメッセージを使用可能にします。
%all	すべてのメッセージを抑制します。
%none	すべてのメッセージを使用可能にします。これはデフォルト値です。

デフォルトは -erroff=%none です。-erroff と指定すると、-erroff=%all を指定した場合と同じ結果が得られます。

次に例を示します。

```
% lint -erroff=%all,no%E_ENUM_NEVER_DEF,no%E_STATIC_UNUSED
```

この例は、「列挙型が定義されていません」と「静的シンボルが使用されていません」のメッセージだけを表示し、その他のメッセージは抑制します。

```
% lint -erroff=E_ENUM_NEVER_DEF,E_STATIC_UNUSED
```

この例は、「列挙型が定義されていません」と「静的シンボルが使用されていません」のメッセージだけを抑制します。

4.3.13 -errsecurity=*v*

-errsecurity オプションを使用して、コードのセキュリティーに問題がないか検査することができます。

v には、次のいずれかを指定します。

表 4–5 -errsecurity のフラグ

値	意味
core	このレベルでは、たいていの場合で安全でない、または検査することの難しいソースコードの構文がないかどうかを検査します。このレベルで行われる検査には次のものがあります。 ■ printf() および scanf() 系の関数での変数書式文字列の使用 ■ scanf() 関数における非結合文字列 (%s) 形式の使用 ■ 安全な使用法のない関数の使用: gets()、cftime()、ascftime()、creat() ■ O_CREAT と組み合わせた open() の不正使用 このレベルで警告が生成されるソースコードはバグと考えてください。問題のコードを変更することを推奨します。どんな場合でも、単純明快でより安全な別の方法があります。

表 4–5 -errsecurity のフラグ (続き)

値	意味
standard	このレベルの検査には、core レベルの検査に加えて、安全かもしれないが、より良い別の方法がある構文の検査があります。新しく作成したコードの検査には、このレベルを推奨します。このレベルで追加される検査には、次のものがあります。 ■ strlcpy() 以外の文字列コピー関数の使用 ■ 脆弱な乱数関数の使用 ■ 安全でない関数を使った一時ファイルの生成 ■ fopen() を使ったファイルの作成 ■ シェルを呼び出す関数の使用 このレベルで警告を生成するソースコードは、新しいコードまたは大幅に修正したコードに書き換えてください。従来のコードに含まれるこうした警告に対処することと、アプリケーションを不安定にするリスクとのバランスを検討してください。
extended	このレベルでは、core および standard レベルの検査を含む完全な検査が行われます。また、状況によっては安全でない可能性がある構文について、多数の警告が生成されます。このレベルの検査は、コードを見直す際の一助になりますが、許容しうるソースコードが守る必要のある基準と考える必要はありません。このレベルで追加される検査には、次のものがあります。 ■ ループ内での getc() または fgetc() の呼び出し ■ パス名競合になりがちな関数の使用 ■ exec() 系の関数の使用 ■ stat() とほかの関数との間の競合 このレベルで警告が生成されるコードを見直して、安全上の潜在的な問題があるかどうかを判定することができます。
%none	-errsecurity 検査を無効にします。

-errsecurity の値が指定されていない場合は、-errsecurity=%none に設定されます。-errsecurity は指定されているが、引数が指定されていない場合は、-errsecurity=standard に設定されます。

4.3.14 -errtags=*a*

各エラーメッセージのメッセージタグを表示します。*a* には yes または no のいずれかを指定します。デフォルトは -errtags=no です。-errtags だけを指定すると、-errtags=yes を指定するのと同じことになります。

すべての -errfmt オプションに使用できます。

4.3.15 -errwarn=*t*

指定された警告メッセージが表示された場合、lint はエラーステータスを返して終了します。*t* には、次の 1 つまたは複数の項目をコンマで区切って指定します。*tag*、no%*tag*、 %all、%none。指定する順序は重要です。たとえば、%all,no%*tag* と指定した場合、*tag* 以外の警告が発行されると、lint は致命的なエラーステータスで終了します。-errwarn の値を次に示します。

表 4–6 -errwarn のフラグ

tag	*tag* に指定されたメッセージが警告メッセージとして発行された場合、lint は致命的なエラーステータスで終了します。*tag* に指定されたメッセージが発行されない場合は無効です。
no%*tag*	タグに指定されたメッセージが警告メッセージとしてだけ発行された場合に、lint が致命的なエラーステータスで終了することがないようにします。*tag* に指定されたメッセージが発行されない場合は無効です。このオプションは、*tag* または %all を使用して以前に指定されたメッセージが警告メッセージとして発行されても lint が致命的なエラーステータスで終了しないようにする場合に使用してください。
%all	警告メッセージが何か発行される場合に lint が致命的なエラーステータスで終了するようにします。%all に続いて no%*tag* を使用して、特定の警告メッセージを対象から除外することもできます。
%none	どの警告メッセージが発行されても lint が致命的なエラーステータスで終了することがないようにします。

デフォルトは -errwarn=%none です。-errwarn だけを指定した場合、-errwarn=%all を指定したことと同じになります。

4.3.16 -F

コマンド行で指定された .c ファイルを参照するとき、そのベース名ではなくコマンド行に与えられたパス名を出力します。

4.3.17 -fd

古い形式の関数定義または宣言について報告します。

4.3.18 -flagsrc=*file*

ファイル中に格納されたオプションを用いて *lint* を実行します。ファイルには、1 行に 1 つずつ、複数のオプションを指定できます。

4.3.19 -h

一定のメッセージを抑制します。表 4–8 を参照してください。

4.3.20 -I*dir*

インクルード用ヘッダーファイルをディレクトリから検索します。

4.3.21 -k

/* LINTED [メッセージ] */ 指令または注釈 NOTE(LINTED(*message*)) の動作を変更します。通常 lint は、前述のような指令のあとにコードが続く場合、警告メッセージを抑制します。lint は、メッセージを抑制する代わりに、指令または注釈の中のコメントを含むメッセージを出力します。

4.3.22 -L*dir*

-l とともに使用し、ディレクトリの lint ライブラリを検索します。

4.3.23 -l*x*

lint ライブラリ llib-l*x*.ln にアクセスします。

4.3.24 -m

一定のメッセージを抑制します。表 4–8 を参照してください。

4.3.25 -m32|-m64

分析するプログラムのメモリーモデルを指定します。また、選択したメモリーモデル (32 ビットまたは 64 ビット) に対応する lint ライブラリを検索します。

32 ビット C プログラムの確認には -m32 を使用し、64 ビット C プログラムの確認には -m64 を使用します。

ILP32 メモリーモデル (32 ビット int、long、ポインタデータ型) は 64 ビット対応ではないすべての Solaris プラットフォームおよび Linux プラットフォームのデフォルトです。LP64 メモリーモデル (64 ビット long、ポインタデータ型) は 64 ビット対応の Linux プラットフォームのデフォルトです。-m64 は LP64 モデル対応のプラットフォームでのみ使用できます。

以前のリリースのコンパイラでは、メモリーモデル、ILP32 または LP64 は、-Xarch オプションを選択して指定されていました。Solaris Studio 12 以降のコンパイラでは、このようなことはありません。ほとんどのプラットフォームでは、コマンド行に -m64 を追加するだけで 64 ビットプログラムで lint を実行することができます。

事前定義のマクロについては、この lint オプションの一覧の次の節を参照してください。

4.3.26 -Ncheck=*c*

ヘッダーファイル中の宣言の対応とマクロの検査を行います。*c* には、検査項目である macro、extern、%all、%none、no%macro、no%extern の 1 つまたは複数をコンマで区切って指定します。

表 4–7 -Ncheck のフラグ

値	意味
macro	ファイル間でのマクロ定義の一貫性を検査します。
extern	ソースファイルとそれに関連するヘッダーファイルとの間の宣言の 1 対 1 対応を検査します (たとえば file1.c と file1.h)。ヘッダーファイルの extern 宣言に余分も不足もないことを確認します。
%all	-Ncheck のすべての検査を実行します。
%none	-Ncheck の検査を実行しません。これはデフォルト値です。
no%macro	-Ncheck のマクロ検査を実行しません。
no%extern	-Ncheck の extern 検査を実行しません。

デフォルトは -Ncheck=%none です。-Ncheck だけを指定すると、-Ncheck=%all を指定するのと同じことになります。

値はコンマを用いて組み合わせることができます (例: -Ncheck=extern,macro)。

次に例を示します。

```
% lint -Ncheck=%all,no%macro
```

この例はマクロ以外のすべての検査項目を実行します。

4.3.27 -Nlevel=*n*

拡張 lint 解析のレベルを指定することによって、問題報告の拡張 lint モードを有効にします。このオプションによって、検出するエラーの量を制御することができます。レベルが高いほど検証にかかる時間は長くなります。*n* は数値で、1、2、3、4 のいずれかです。デフォルトはありません。-Nlevel が指定されなかった場合は、lint の基本解析モードが使用されます。引数なしで -Nlevel が指定された場合は、-Nlevel=4 に設定されます。

基本および拡張 lint モードについては、100 ページの「4.2 lint 使用方法」を参照してください。

4.3.27.1 -Nlevel=1

個々の手続きを解析します。いくつかのプログラムの実行パスで発生する無条件エラーを報告します。大域的なデータおよび制御のフロー解析は行いません。

4.3.27.2 -Nlevel=2

大域的なデータおよびフローを含め、プログラム全体を解析します。いくつかのプログラムの実行パスで発生する無条件エラーを報告します。

4.3.27.3 -Nlevel=3

-Nlevel=2 で実行される解析に加えて、定数の伝播、定数が実際の引数として使用されている場合を含め、プログラム全体を解析します。

この解析レベルでの C プログラムの検査は、直前のレベルより 2 倍から 4 倍長い時間がかかります。これは、lint がプログラムの変数に対して取り得る値の集合を作成し、プログラムの部分解釈を行うためです。これらの変数値の集合は、定数と、プログラムで使用可能な定数オペランドを含む条件文に基づいて作成され、ほかの集合 (定数伝播の形式) を作成するときの基準になります。そのあと、解析の結果として受け取った集合は、次のアルゴリズムに従って誤りがないか評価されます。

オブジェクトが取り得る値の集合の中に正しい値が存在する場合は、その値が次の伝搬の基準として使用されます。正しい値が存在しない場合は、エラーと診断されます。

4.3.27.4 -Nlevel=4

-Nlevel=3 で実行される解析に加えて、プログラム全体を解析して一定のプログラム実行パスが使用された場合に発生する条件付きエラーも報告します。

この解析レベルでは、さらに多くの診断メッセージが出力されます。一般的に、この解析アルゴリズムは、不正な値に対してエラーメッセージが生成されることを除けば、-Nlevel=3 の解析アルゴリズムと同じです。このレベルでの解析に要する時間

は、2 桁 (約 20 倍から 100 倍) ほど増加する可能性があります。余計にかかる時間は、再帰、条件文などの面でのプログラムの複雑さに比例して長くなります。このため、100,000 行を超えるプログラムに対してこのレベルの解析を行うことはあまり現実的ではありません。

4.3.28 -n

デフォルトの lint 標準ライブラリとの互換性検査を抑制します。

4.3.29 -o*x*

lint は llib-lx.*ln* という名前の lint ライブラリを作成します。このライブラリは、lint が第 2 パスで使用する .ln ファイルから作成されます。-c オプションを使用すると、すべての -o オプションが無効になります。不要なメッセージを表示しないで llib-l*x*.ln を作成するには、-x オプションを使用します。lint ライブラリのソースファイルが外部からの参照専用である場合は、-v オプションが便利です。作成された lint ライブラリは、あとで lint が -l*x* で呼び出された場合に使用することができます。

デフォルトでは、ライブラリは lint の基本形式で作成されます。拡張 lint モードを使用した場合は、ライブラリは拡張モードで作成されるため、それ以外のモードでは使用できなくなります。

4.3.30 -p

移植性に関連する一定のメッセージを使用可能にします。

4.3.31 -Rファイル

cxref(1) で使用する *.ln* ファイルをファイルに書き込みます。lint が拡張モードで起動されている場合、このオプションは拡張モードを取り消します。

4.3.32 -s

「警告:」または「エラー:」で始まる単一の診断メッセージを生成します。デフォルトでは、lint は複合的な出力を生成するためにいくつかのメッセージをバッファリングします。

4.3.33 -u

一定のメッセージを抑制します。表 4–8 を参照してください。このオプションは、大型プログラムのファイルの一部分に対して lint を実行する場合に適しています。

4.3.34 -V

製品名とリリース時期を標準エラーに書き込みます。

4.3.35 -v

一定のメッセージを抑制します。表 4–8 を参照してください。

4.3.36 -W*file*

cflow(1) で使用する *.ln* ファイルをファイルに書き込みます。lint が拡張モードで起動されている場合、このオプションは拡張モードを取り消します。

4.3.37 -XCC=*a*

C++ 形式のコメントを受け入れます。このオプションを使用すると、// を使用してコメントの始まりを示すことができます。*a* には yes または no のいずれかを指定します。デフォルトは -XCC=no です。-XCC だけを指定すると、-XCC=yes を指定するのと同じことになります。

注 – -xc99=none を使用する場合のみ、このオプションを指定する必要があります。デフォルトの -xc99=all では、lint は // で指定したコメントを受け入れます。

4.3.38 -Xalias_level[=*l*]

l には、 any、basic、weak、layout、strict、std、strong のいずれか 1 つが入ります。各レベルの明確化の詳細については、表 B–13 を参照してください。

-Xalias_level を指定しない場合、フラグのデフォルトは -Xalias_level=any になります。このことは、型に基づく別名解析が行われないことを意味します。-Xalias_level を指定してもレベルを設定しない場合、デフォルトは -Xalias_level=layout になります。

lint を実行する際に、明確化のレベルをコンパイラの実行レベルよりも緩やかに設定してください。明確化のレベルをコンパイルより厳密に設定して lint を実行すると、解釈の困難な結果が生成され、誤解を招く恐れがあります。

明確化の詳細と、明確化を支援するために作成されたプラグマのリストについては、128 ページの「4.6.3 lint フィルタ」を参照してください。

4.3.39 -Xarch=amd64

(Solaris オペレーティングシステム) 推奨されていません。使用しないでください。109 ページの「4.3.25 -m32|-m64」を参照してください。

4.3.40 -Xarch=v9

(Solaris オペレーティングシステム) 推奨されていません。使用しないでください。109 ページの「4.3.25 -m32|-m64」を参照してください。

4.3.41 -Xc99[= *o*]

-Xc99 フラグは、C99 規格 (『Programming Language - C (ISO/IEC 9899:1999)』) からの実装機能に対するコンパイラの認識状況を制御します。

o には、次のいずれかを指定します。all、none。

-Xc99=none を指定すると、C99 機能に対する認識がオフになります。-Xc99=all を指定すると、サポートされている C99 機能に対する認識がオンになります。

引数を付けずに -Xc99 を発行すると、-Xc99=all と同じ結果になります。

注 – コンパイラのサポートレベルは、デフォルトでは表 C–6 で説明している C99 の機能になりますが、Solaris が提供する /usr/include の標準ヘッダーファイルは、1999 ISO/IEC C 規格にまだ準拠していません。エラーメッセージが生成される場合は、-Xc99=none を指定して、前述のヘッダー用に 1990 ISO/IEC C 規格を使用してみてください。

4.3.42 -Xkeeptmp=*a*

lint の実行中、一時ファイルを自動的に削除せず、作成した状態のままにします。*a* には yes または no のいずれかを指定します。デフォルトは -Xkeeptmp=no です。-Xkeeptmp だけを指定すると、-Xkeeptmp=yes を指定するのと同じことになります。

4.3.43 -Xtemp=*dir*

一時ファイルのディレクトリをディレクトリに設定します。このオプションを指定しないと、一時ファイルは /tmp に格納されます。

4.3.44 -Xtime=*a*

各 lint パスの実行時間を報告します。*a* には yes または no のいずれかを指定します。デフォルトは -Xtime=no です。-Xtime だけを指定すると、-Xtime=yes を指定するのと同じことになります。

4.3.45 -Xtransition=*a*

K&R C と Solaris Studio ISO C の相違を検出した場合に警告を出します。*a* には yes または no を指定します。デフォルトは -Xtransition=no です。-Xtransition だけを指定すると、-Xtransition=yes を指定するのと同じことになります。

4.3.46 -Xustr={ascii_utf16_ushort| no}

このオプションは、U"*ASCII_*文字列" という書式の文字列リテラルについて unsigned short int の配列としての認識を有効にします。デフォルトは -Xustr=no です。このオプションは、コンパイラによる U"*ASCII_*文字列" という文字列リテラルの認識を無効にします。-Xustr=ascii_utf16_ushort は、コンパイラによる U"*ASCII_*文字列" の文字列リテラルの認識を有効にします。

4.3.47 -x

一定のメッセージを抑制します。表 4–8 を参照してください。

4.3.48 -y

コマンド行で指定されたすべての .c ファイルを、/* LINTLIBRARY */ 指令で開始した場合または注釈 NOTE(LINTLIBRARY) が付いている場合と同じように扱います。lint ライブラリは、通常、/* LINTLIBRARY */ 指令または注釈 NOTE(LINTLIBRARY) を使用して作成します。

4.4 lint のメッセージ

大部分の lint のメッセージは簡単な 1 行の文で、問題が起こって診断されるたびに出力されます。インクルードファイルで検出されたエラーはコンパイラでは複数回報告されますが、lint ではそのファイルがほかのソースファイルに何度インクルードされようとも一度報告されるだけです。複合メッセージは、ファイル全域の矛盾に対して、また時にはファイル内の問題に対しても表示されます。単一メッセージは、検査しているファイルで問題が発生するごとに知らせます。lint フィルタ (127 ページの「4.6.2 lint ライブラリ」を参照) を使用して各現象ごとに表示されるメッセージを要求する時に、-s オプションを使用して lint を実行することにより、複雑なメッセージを簡単なものに変換することができます。

lint のメッセージは stderr に書き込まれます。

4.4.1 メッセージを抑制するオプション

いくつかの lint オプションを使用して、lint の診断メッセージを抑制することができます。メッセージを抑制するには、-erroff オプションのあとに 1 つ以上の タグ を指定して実行してください。これらのニーモニックタグは、-errtags=yes オプションで表示することができます。

次の表に lint のメッセージを抑制するオプションを示します。

表 4–8 メッセージを抑制する lint オプション

オプション	抑制されるメッセージ
-a	代入によって暗黙的により小さい型に変換されます より大きな整数型への変換は符号拡張が不正確になる可能性があります
-b	到達できない文です
-h	等価演算子 "==" の使用が想定される場所に代入演算子 "=" が使用されています 演算子 "!" のオペランドが定数です case 文を通り抜けます ポインタのキャストによって境界整列が不正確になる可能性があります 優先度が混乱する可能性があります; 括弧 文が帰結していません: if 文が帰結していません: else
-m	大域的に宣言されていますが静的 (static) にすることができます
-erroff=*tag*	タグで指定した 1 つまたは複数の lint メッセージ

表 4–8 メッセージを抑制する lint オプション (続き)

オプション	抑制されるメッセージ
-u	名前が定義されていますが使用されていません 未定義の名前が使用されています
-v	引数が関数中で使用されていません
-x	名前が宣言されていますが使用も定義もされていません

4.4.2 lint メッセージの形式

lint プログラムに特定のオプションを付けると、エラーが発生した行へのポインタを付けて、ソースファイルの正確な行が示されます。この機能を使用可能にするオプションは -errfmt=*f* です。このオプションを指定しておくと、lint は次の情報を出力します。

- ソースの行と位置
- マクロの展開
- エラーを起こしやすいスタック

たとえば、次に示すプログラム Test1.c にはエラーがあります。

```
1 #include <string.h>
2 static void cpv(char *s, char* v, unsigned n)
3 { int i;
4   for (i=0; i<=n; i++){
5        *v++ = *s++;}
6 }
7 void main(int argc, char* argv[])
8 {
9     if (argc != 0){
10        cpv(argv[0], argc, strlen(argv[0]));}
11}
```

そこで、次のようなオプションを使用して Test1.c に lint を実行します。

```
% lint -errfmt=src -Nlevel=2 Test1.c
```

結果として、次のような出力が得られます。

```
      |static void cpv(char *s, char* v, unsigned n)
      |            ^  line 2, Test1.c
      |
      |         cpv(argv[0], argc, strlen(argv[0]));
      |                     ^  line 10, Test1.c
warning: improper pointer/integer combination: arg #2
      |
      |static void cpv(char *s, char* v, unsigned n)
      |                               ^  line 2, Test1.c
      |
```

```
      |cpv(argv[0], argc, strlen(argv[0]));
      |                       ^ line 10, Test1.c
      |
      |        *v++ = *s++;
      |         ^  line 5, Test1.c
warning:use of a pointer produced in a questionable way
    v defined at Test1.c(2)    ::Test1.c(5)
      call stack:
          main()                ,    Test1.c(10)
          cpv()                 ,    Test1.c(5)
```

1 つめの警告は、2 つのコード行の間で矛盾があることを示しています。2 つめの警告には、その時のコールスタックとエラーに到るまでの制御フローが表示されます。

次に示すプログラム Test2.c には、前述のものとは異なる種類のエラーがあります。

```
1 #define AA(b) AR[b+l]
2 #define B(c,d) c+AA(d)
3
4 int x=0;
5
6 int AR[10]={1,2,3,4,5,6,77,88,99,0};
7
8 main()
9  {
10  int y=-5, z=5;
11  return B(y,z);
12 }
```

そこで、次のようなオプションを使用して Test2.c に lint を実行します。

```
% lint -errfmt=macro Test2.c
```

結果として、次のような出力が得られます。

```
      | return B(y,z);
      |        ^  line 11, Test2.c
      |
      |#define B(c,d) c+AA(d)
      |                 ^  line 2, Test2.c
      |
      |#define AA(b) AR[b+l]
      |                   ^  line 1, Test2.c
error: undefined symbol: l
|
      |    return B(y,z);
      |           ^  line 11, Test2.c
      |
      |#define B(c,d) c+AA(d)
      |                 ^  line 2, Test2.c
      |
      |#define AA(b) AR[b+l]
      |                   ^  line 1, Test2.c
variable may be used before set: l
lint: errors in Test2.c; no output created
lint: pass2 not run - errors in Test2.c
```

4.5 lint の指令

4.5.1 事前定義された値

lint を実行すると、lint トークンが事前定義されます。事前定義されたトークンのリストについては、cc(1) のマニュアルページも参照してください。

4.5.2 指令

lint 指令を /*...*/ の形式で注釈として表記する方法は、現在サポートされていますが、将来はサポートされなくなる予定です。指令を注釈として挿入する際は、ソースコードの注釈 NOTE(...) として表記することをお勧めします。

次のようにファイル note.h をインクルードして、lint 指令をソースコードの注釈として指定してください。

```
#include <note.h>
```

lint は、ソースコードの注釈を別のツールと共有します。Solaris Studio C コンパイラをインストールすると、/usr/lib/note/SUNW_SPRO-lint ファイルが自動的にインストールされます。このファイルには、LockLint が認識する注釈の名前がすべて記述されています。ただし、Solaris Studio C のソースコードを検査する lint は、/usr/lib/note と Solaris Studio のデフォルトの場所である <*install-directory*>/prod/lib/note の全ファイルを検索して、該当する注釈を探します。

次のように、環境変数 NOTEPATH を設定することにより、/usr/lib/note 以外の位置を指定することもできます。

```
setenv NOTEPATH $NOTEPATH:other_location
```

次の表に、lint 指令と動作を示します。

表 4–9 lint 指令

指令	処理
NOTE(ALIGNMENT(*fname*,*n*)) *n*=1, 2, 4, 8, 16, 32, 64, 128	lint に関数結果を *n* バイトで整列させます。たとえば、malloc() は、char* または void* を返すように定義されていますが、実際にはワードで、または場合によってはダブルワードで整列したポインタを返します。 不正な境界整列に関するメッセージが抑制されます。 ■ 不正確な境界整列

表 4–9 lint 指令 (続き)

指令	処理
NOTE(ARGSUSED(n)) /*ARGSUSEDn*/	指令の次に来る関数に対して、-v オプションのような動作を行います。 次のメッセージが抑制されます。指令のあとに来る関数定義の最初の *n* 個以降のすべての引数を対象します。デフォルトは 0 です。NOTE 形式の場合は、必ず *n* を指定します。 ■ 引数が関数中で使用されていません
NOTE(ARGUNUSED (引数[,引数...]))	lint が、指定した引数の使用状況を検査しないようにします (このオプションは、指令の次に来る関数に対してのみ有効です)。 次のメッセージが抑制されます。NOTE または指令で指定された引数すべてを対象とします。 ■ 引数が関数中で使用されていません
NOTE(CONSTCOND) /*CONSTCOND*/	条件式中の定数オペランドに関する警告を抑制します。次のメッセージが抑制されます。NOTE(CONSTANTCONDITION) または /* CONSTANTCONDITION */ も使用できます。 条件のコンテキストに定数があります 演算子 "!" のオペランドが定数です 論理式が常に偽です:演算子 "&&" 論理式が常に真です:演算子 "||"
NOTE(EMPTY) /*EMPTY*/	if 文に続く null 文の内容に関する警告を抑制します。この指令は、条件式とセミコロンの間で指定します。この指令は、有効な else 文を持つ空の if 文をサポートするためにあります。また、空の else 文に対するメッセージも抑制します。 次のメッセージが抑制されます (if の条件式とセミコロンの間に挿入された場合)。 ■ 文が帰結していません: else (else 文とセミコロンの間に挿入された場合) ■ 文が帰結していません: if

表 4–9 lint 指令 (続き)

指令	処理
NOTE(FALLTHRU) /*FALLTHRU*/	case 文または default ラベルの文までの通り抜けに関する警告を抑制します。この指令は、ラベルの直前で指定します。 次のメッセージが抑制されます。指令のあとに来る case 文が対象となります。NOTE(FALLTHROUGH) または /* FALLTHROUGH */ も使用できます。 ■ case 文を通り抜けます
NOTE(LINTED (メッセージ)) /*LINTED [メッセージ]*/	使用されない変数または関数に関する警告を除く、ファイル内の警告をすべて抑制します。この指令は、lint の警告が表示された行の直前で指定します。-k オプションは、lint がこの指令を扱う方法を変更します。lint は、メッセージを抑制する代わりに、コメントに含まれているメッセージがある場合は、そのメッセージを表示します。この指令は、lint 実行後にフィルタを行うための -s オプションと組み合わせて使用すると便利です。 -k が指定されない場合、指令のあとに来るコード行の次のもの以外のファイル内問題に属するすべての警告を抑制します。 ■ 引数が関数中で使用されていません ■ 宣言がブロック中で使用されていません ■ 変数が関数中で設定されていますが使用されていません ■ 静的シンボルが使用されていません ■ 変数が関数中で使用されていません 先行するコード行では、メッセージは無視されます。
NOTE(LINTLIBRARY) /*LINTLIBRARY*/	-o が指定された場合、この指令が先頭に付く .c ファイル中の定義だけをライブラリ .ln ファイルに書き込みます。ファイル内で使用されない関数および関数の引数に関する内容を抑制します。

表 4–9 lint 指令 (続き)

指令	処理
NOTE(NOTREACHED) /*NOTREACHED*/	到達不可コードに関するコメントを適切な時点で停止します。このコメントは、通常、exit(2) などの、関数に対するコールの直後に位置します。 次のメッセージが抑制されます。 ■ 到達できない文です 指令のあとに来る到達されない文が対象の場合。 ■ case 文を通り抜けます 指令のあとの case 文で、指令の前の case 文から到達されないものが対象の場合。 ■ 関数が値を返さずに終了しています
NOTE(PRINTFLIKE(*n*)) NOTE(PRINTFLIKE(*fun_name,n*)) /*PRINTFLIKE*n*/	指令のあとに来る関数定義の第 *n* 番目の引数を [fs]printf() の書式文字列として扱い、後述のメッセージを有効にします。残りの引数と変換指示子の間の不整合も対象にします。lint はデフォルトで、標準 C ライブラリで提供される [fs]printf() 関数を呼び出すときのエラーに対してこれらの警告を出します。 NOTE 形式の場合は、必ず *n* を指定します。 ■ 書式文字列が正しくありません 書式から参照される引数が足りません。書式から参照されていない引数があります ■ 書式から参照される引数が足りません ■ 書式から参照される引数が多すぎます
NOTE(PROTOLIB(*n*)) /*PROTOLIB*n*/	*n* が 1 で NOTE(LINTLIBRARY) または /* LINTLIBRARY */ が使用される場合、この指令が先頭に付く .c ファイルの関数プロトタイプ宣言だけをライブラリ .ln に書き込みます。デフォルトは処理を取り消す 0 です。 NOTE 形式の場合は、必ず *n* を指定します。
NOTE(SCANFLIKE(*n*)) NOTE(SCANLIKE*(fun_name,n*)) /*SCANFLIKE*n*/	関数定義の第 n 番目の引数が *[fs]scanf()* の書式文字列として扱われること以外は NOTE(PRINTFLIKE(n)) または /* PRINTFLIKE*n* */ と同じです。デフォルトでは、lint は標準 C ライブラリで提供される [fs]scanf() 関数を呼び出すときのエラーに対し警告を出します。 NOTE 形式の場合は、必ず *n* を指定します。

表 4–9 lint 指令 (続き)

指令	処理
NOTE(VARARGS(*n*)) NOTE(VARARGS(*fun_name,n*)) /*VARARGS*n*/	指令のあとに来る関数宣言の中の可変数の引数を検査する通常の処理を抑制します。最初の *n* 個の引数のデータ型を検査します。*n* が指定されていない場合は、n= 0 とみなします。新規のコードを書く場合やコードを更新する場合、定義の中で末尾に省略記号 (...) を使用することを推奨します。 この指令の直後で定義されている関数に関しては、次のメッセージが抑制されます。*n* 以上の引数を持つ関数に対する呼び出しを対象にします。NOTE 形式の場合は、必ず *n* を指定します。 ■ functions called with variable number of arguments

4.6 lint の参考情報と例

lint が行う検査、lint ライブラリ、および lint フィルタなどに関する lint の参考情報について説明します。

4.6.1 lint が行う診断

lint 固有 の診断は、矛盾した使い方、移植不能のコード、疑わしい言語構造の 3 つの広い条件カテゴリ に対して表示されます。この節では、各カテゴリにおける lint の動作の例を示し、どのような対応が可能かを説明します。

4.6.1.1 整合性の検査

ファイル全域とファイル内部における変数、引数、関数の矛盾した使用を検査します。概して lint が古いスタイルの関数に対して検査していたのと同様に、プロトタイプの使用、宣言、引数を検査します。プログラムが関数プロトタイプを使用していない場合、lint は関数の呼び出しごとにコンパイラより厳しく引数の数と型を検査します。lint は、[fs]printf() と [fs]scanf() の制御文字列の変換指示子と引数の不一致も識別します。

次に例を示します。

- lint はファイル内で呼び出した関数に値を返すことなくそのまま終了してしまうような非 void 型関数にフラグを立てます。以前、プログラマは fun() {} のように戻り型を省略することによって「関数は値を返さない」ということを示しました。しかし、fun() が戻り型 int を持っているとみなすコンパイラには何の意味もありません。この問題を解決するには、戻り型 void の関数として宣言します。

- lint はファイル全域で非 void 型関数が値を返さず、しかも式の中でその値が使用されている場合や、これとは反対に、関数が返す値があとの呼び出しで時々または常に無視されるという場合を検出します。値が常に無視されるのは、関数定義が不十分だと考えられます。時々無視されるのは、間違ったプログラミングスタイルをとっていることが考えられます (エラー状態のテストが行われていないなど)。strcat()、strcpy()、および sprintf() のような文字列関数や、printf() と putchar() のような出力関数の戻り値を検査する必要がない場合、その問題となる呼び出しは void 型にキャストしてください。
- lint は次の場合に変数や関数を識別します。宣言されたが定義または使用されていない。使用されたが定義されていない。定義されたが使用されていない。したがって、一緒に読み込まれるファイルのすべてにではなくその一部に lint が適用されると、lint は次の場合に警告を出します。
  - そのファイルで宣言された関数と変数がほかの場所で定義または使用された。
  - そのファイルで使用された関数と変数がほかの場所で定義されていた。
  - そのファイルで定義された関数と変数がほかの場所で使用された。

    1 つめの場合を抑制するには -x オプションを、あとの 2 つの場合を抑制するには -u オプションを使用してください。

4.6.1.2 移植性の検査

一部の移植不可能なコードは、lint のデフォルトの動作によってフラグを付けられます。また、ほかにも少数の状況で、 -p または -Xc を指定して lint を起動すると、診断されることがあります。lint は ISO C 規格に一致しない言語構造を検査します。-p および -Xc のもとで発行されるメッセージに関しては、127 ページの「4.6.2 lint ライブラリ」を参照してください。

次に例を示します。

- 一部の C 言語の実装では、signed または unsigned のどちらも明示的に宣言されていない文字変数は、符号付き (signed) の量として扱われ、通常は -128 〜 127 の範囲になります。ほかの実装では、これらは負にならない量として扱われ、通常は 0 〜 255 の範囲になります。そのため次のテストは

```
char c;
c = getchar();
if (c == EOF) ...
```

そこで EOF が値 -1 を持つテストは、文字変数が負でない値を取るマシンでは常に失敗します。-p オプションで呼び出した lint は、普通の char が負の値を取る可能性があるような比較をすべて検査します。しかし前述の例では、c を signed char で宣言しても、問題が除去されるのではなく診断が除去されるだけです。これは、getchar() が入力可能な文字と明確な EOF 値を返さなければならず、char がその値を格納することができないためです。これは、処理系ごとに定義される符号拡張

から生ずるもっとも一般的な例です。これにより、lint の移植性オプションを注意深く使用すると移植性に関係しないバグを発見するのに役立つということがわかります。ここでは c を int で宣言します。

- 同様の問題がビットフィールドにもあります。定数値がビットフィールドに代入される場合、その値を保持するにはフィールドが小さすぎる場合があります。int 型のビットフィールドを符号なし (unsigned) の量として取り扱うマシンでは、int x:3 の範囲で許可される値が 0 ～ 7 であるのに対し、符号付き (signed) の量として取り扱うマシンでは -4 ～ 3 になります。ただし、int 型として宣言された 3 ビットのフィールドは、後者のマシンでは値 4 を保持できません。-p を指定して呼び出された lint は、unsigned int または signed int を除き、すべてのビットフィールドの型にフラグを付けます。これらのみが、移植可能なビットフィールド型です。コンパイラは、ビットフィールドの型 int、char、short、および long をサポートしますが、これらは unsigned、signed またはそのどちらでもない場合があります。さらにコンパイラは enum のビットフィールドの型もサポートします。
- 大きなサイズの型が小さなサイズの型に代入されると、バグが発生することがあります。有効なビットが切り捨てられると正確な値を保持できなくなります。

```
short s;
long l;
s = l;
```

lint は、デフォルトでこのような代入すべてを知らせます。診断は、-a オプションを指定して呼び出すことにより抑制することができます。どのオプションを指定して lint を呼び出しても、ほかの診断をも抑制する可能性があることに注意してください。2 つ以上の診断を抑制するオプションについては、127 ページの「4.6.2 lint ライブラリ」にあるリストを参照してください。

- あるオブジェクト型へのポインタをより厳密な境界整列要求を持つオブジェクト型のポインタにキャストすると、移植性がなくなることがあります。lint のフラグは次のようになります。

```
int *fun(y)
char *y;
{
    return(int *)y;
}
```

大部分のマシンでは、int は char とは異なり任意のバイト境界から開始することができないため、lint はフラグを立てます。-h を指定して lint を実行することによってこの診断を抑制することができます。この場合もまた、ほかのメッセージを抑制する可能性があります。汎用ポインタ void * を使用すればほかの影響を回避することができます。

- ISO C は、複雑な式の評価順序を定義していません。この意味は、関数呼び出し、入れ子になった代入文、またはインクリメントとデクリメント演算子から副作用が生じる場合 (すなわち、式評価の副作用として変数が変更される時)、副作

用の生じる順序はマシンへの依存度が高いということです。デフォルトでは、lint は副作用で変更されたり同一式内でほかの場所に使用される変数を知らせます。

```
int a[10];
main()
{
    int i = 1;
    a[i++] = i;
}
```

この例での a[1] の値は、あるコンパイラでは 1、別のコンパイラでは 2 という可能性もあります。ビット単位の論理演算子 & がこのような診断をもたらすことがあるのは、誤って 論理演算子 && の代わりに使用される場合です。

```
if ((c = getchar()) != EOF & c != '0')
```

4.6.1.3 疑わしい言語構造

lint は、プログラマの意図には反するが、言語構造上は正しい箇所についても報告します。次に例を示します。

- unsigned 変数は常に負ではない値を持ちます。そのため次のテストは

```
unsigned x;
if (x < 0) ...
```

常に失敗します。一方、

```
unsigned x;
if (x > 0) ...
```

これは次のように指定するのと同じことです。

```
if (x != 0) ...
```

最初の例は意図したものではない可能性があります。lint は、負の定数または 0 と unsigned 変数との疑わしい比較を知らせます。unsigned 変数を負数のビットパターンと比較するには、その負数を unsigned にキャストします。

```
if (u == (unsigned) -1) ...
```

または、接尾辞 U を使用します。

```
if (u == -1U) ...
```

- lint は、副作用が予想される状況で使用される副作用のない式、すなわちプログラマの意図に反した式を知らせます。代入演算子が予想されるところ、つまり副作用が予想されたところで等価演算子が存在する場合は追加の警告が発行されます。

```
int fun()
{
    int a, b, x, y;
    (a = x) && (b == y);
}
```

- lint は、論理演算子とビット単位の演算子 (具体的には、&、|、^、<<、>>) の両方が混在する式に括弧を入れるように注意を与えます。これは演算子の優先度を間違って解釈することにより、不正確な結果になる可能性があります。ビット単位の演算子 & の優先度は論理演算子 == より低いため、式はユーザーの意図とは異なる次のような式として評価されます。

```
if (x & a == 0) ...
```

この式は、次のように評価されます。

```
if (x & (a == 0)) ...
```

これは、ユーザーの意図とは異なる可能性が高いものです。-h を指定して lint を起動すると、診断のものは無効になります。

4.6.2 lint ライブラリ

lint ライブラリを使用して、呼び出したライブラリ関数とユーザープログラムとの互換性を検査することができます。関数戻り型の宣言、関数が期待する引数の数と型などを検査します。標準 lint ライブラリは、C 言語処理系で供給されるライブラリに対応し、一般にはシステムの標準位置であるディレクトリに格納されています。慣例では、lint ライブラリは llib-lx.ln という形の名前を持ちます。

lint 標準 C ライブラリの llib-lc.ln は、デフォルトで lint コマンド行に追加されます。ライブラリ関数との互換性の検査は、-n オプションを指定して呼び出すことにより抑制することができます。そのほかの lint ライブラリは、-l に対して引数として指定することでアクセスされます。次に例を示します。

```
% lint -lx file1.c file2.c
```

この例では、lint ライブラリ llib-l*x*.ln との互換性について、file1.c と file2.c の関数と変数の使用方法を調べるよう lint に指示します。定義だけからなるライブラリファイルは、厳密に通常のソースファイルと .ln ファイルとして処理されます。ただしライブラリファイルで関数と変数が矛盾したまま使用されるか、またはライブラリファイルで定義されてもソースファイルでは使用されない関数と変数に対しては警告を出しません。

自分の lint ライブラリを作成するには、C ソースファイルの先頭に NOTE(LINTLIBRARY) 指令を挿入し、次いで -o オプションとそのライブラリ名を与える -l オプションとともにそのファイルに対して lint を実行してください。

```
% lint -ox file1.c file2.c
```

前述のコマンド行により、NOTE(LINTLIBRARY) が先頭に付いたソースファイル中の定義だけがファイル llib-l*x*.ln に書き込まれます。lint -o と cc -o の類似に注意してください。ライブラリは、同様に関数プロトタイプ宣言のファイルから作成されます。ただし、NOTE(LINTLIBRARY) と NOTE(PROTOLIB(*n*)) の両方が宣言ファイルの先頭に挿入されている場合は別です。*n* が 1 の場合、プロトタイプ宣言は古いスタイルの定義と同様にライブラリ .ln ファイルに書き込まれます。*n* がデフォルトの 0 の場合、処理はキャンセルされます。-y を指定して lint を呼び出しても、lint ライブラリを作成することができます。次のようなコマンド行があるとします。

```
% lint -y -ox file1.c file2.c
```

前述のコマンド行で指定された各ソースファイルは NOTE(LINTLIBRARY) で開始したかのように扱われ、その定義だけが llib-l*x*.ln に書き込まれます。

デフォルトでは、lint は標準位置で lint ライブラリを検索します。標準位置以外のディレクトリで lint ライブラリを検索するように lint に指示するには、-L オプションを使用してディレクトリのパスを指定します。

```
% lint -Ldir -lx file1.c file2.c
```

拡張モードでは、lint は基本モードで生成される .ln ファイルより多くの情報が格納された .ln ファイルを生成します。拡張モードの lint は、基本モードまたは拡張モードのどちらの lint で生成された .ln ファイルでもすべて読み取って理解することができます。基本モードの lint は、基本モードの lint を用いて生成された .ln ファイルだけを読み取って理解することができます。

デフォルトでは、lint は /usr/lib ディレクトリのライブラリを使用します。これらのライブラリは基本 lint 形式です。makefile を一度実行して新しい形式の拡張 lint ライブラリを作成すれば、拡張 lint をより効率的に利用することができます。makefile を実行して新しいライブラリを作成するには、次のコマンドを入力してください。

```
% cd <install-directory>/prod/src/lintlib; make
```

ここで、<*install-directory*> はインストールディレクトリです。makefile の実行後、lint は /usr/lib ディレクトリ内のライブラリの代わりに拡張モードの新ライブラリを使うようになります。

指定されたディレクトリは標準位置の前に検索されます。

4.6.3 lint フィルタ

lint フィルタは、プロジェクト固有のポストプロセッサ (後処理) です。典型的な例では awk スクリプトや類似のプログラムを使用して lint の出力を読み取り、ユーザーのプロジェクトが特に問題ないと判断したメッセージを捨てます。たとえば、時々または常に無視される値を返す文字列関数などです。lint オプション

と指令だけでは出力に対して十分な制御が与えられない時は、lint フィルタを使用するとカスタマイズされた診断レポートを作成することができます。

lint の 2 つのオプションはフィルタを開発する際に特に役立ちます。

- -s を指定して lint を呼び出すと、複合診断が問題の発生ごとに表示される単純な一行メッセージに変換されます。この解析されたメッセージ書式は awk スクリプトによる分析に適しています。
- -k を指定して lint を呼び出すと、ソースファイルに書き込まれたコメントが出力されるので、プロジェクトの決定を文書化したり後処理の動作を指定するのに便利です。コメントが予想される lint メッセージを示していて、報告されたメッセージがそれと同一であった場合、メッセージは除かれます。-k を使用するときは NOTE(LINTED(メッセージ)) 指令をコメントしたいコードの前の行に挿入してください。ここでのメッセージは、lint が -k を指定して呼び出されたときに出力されるコメントです。

  (メッセージ) のあるファイルに対して -k が使用されない場合の lint の動作については、表 4–9 を参照してください。 *NOTE(LINTED (*メッセージ*))*。

◆ ◆ ◆ 第 5 章

5 型に基づく別名解析

この章では、xalias_level オプションといくつかのプラグマを使用して、型に基づく別名解析および最適化を実行する方法について説明します。これらの拡張機能を使用すると、ユーザーの C 言語プログラムでのポインタの使用方法について、型に基づく情報を示すことができます。C コンパイラはそれらの情報を利用し、ユーザーのプログラムにおけるポインタベースのメモリー参照の別名明確化について、非常に効率性の高いジョブを実行します。

このコマンドの構文の詳細については、256 ページの「B.2.72 -xalias_level[=*l*]」を参照してください。また、lint プログラムの型に基づく別名解析機能については、113 ページの「4.3.38 -Xalias_level[=*l*]」 を参照してください。

5.1 型に基づく解析の概要

-xalias_level オプションを使用すると、7 つの別名レベルのいずれか 1 つを指定できます。各レベルは、ユーザーの C 言語プログラムでのポインタの使用方法について、特定のプロパティーセットを指定します。

コンパイル時に -xalias_level オプションを上位に設定していくと、コンパイラは、コードのポインタに関する仮定を徐々に拡張していきます。コンパイラの作成する仮定が少ないと、それだけプログラミングの自由度が向上します。ただし、狭い仮定で実行された最適化により、実行時のパフォーマンスが向上しないことがあります。より上位レベルの -xalias_level オプションから得られる仮定に従ってコードを作成すると、最終的な最適化で実行時のパフォーマンスが向上する可能性が高くなります。

-xalias_level オプションは、各翻訳単位に適用される別名レベルを指定します。より詳細に設定したほうがよい場合、新しいプラグマを使用すると、適用されている別名レベルを無効にし、個々の型または翻訳単位のポインタ変数間の別名設定の関係性を明示的に指定できます。これらのプラグマは、翻訳単位におけるポインタの

用法がいずれかの別名レベルで扱われていても、少数の特定ポインタ変数がいずれかのレベルで許可されていない不規則な方法で使用される場合にもっとも役立ちます。

5.2 微調整におけるプラグマの使用

より詳細に設定したほうが型に基づいた解析に望ましい場合、次のプラグマを使用すると、適用されている別名レベルを無効にし、個々の型または翻訳単位のポインタ変数間で別名設定の関係性を指定できます。これらのプラグマは、翻訳単位におけるポインタの用法がいずれかの別名レベルと一貫していても、少数の特定ポインタ変数がいずれかのレベルで許可されていない不規則な方法で使用される場合にもっとも役立ちます。

注 – プラグマより先に命名済みの型または変数を宣言する必要があります。この作業を怠ると、警告メッセージが発行され、プラグマが無視されます。プラグマの意味の適用される最初のメモリー参照のあとにプラグマを配置した場合、プログラムは未定義の結果を生成します。

プラグマの定義において、次の用語を使用します。

用語	意味
level	256 ページの「B.2.72 -xalias_level[=*l*]」に一覧表示されている任意の別名レベル
type	次のいずれかです。 ■ `char`、`short`、`int`、`long`、`long long`、`float`、`double`、`long double` ■ `void`。すべてのポインタの型を示します。 ■ `typedef` *name*。`typedef` 宣言で定義される型の名前。 ■ `struct` *name*。`struct` *tag* 名が後続するキーワード *struct* のことです。 ■ `union`。`union` *tag* 名が後続するキーワード *union* のことです。
pointer_name	翻訳単位におけるポインタ型の変数の名前

5.2.1 #pragma alias_level *level* (*list*)

level は、次の別名レベルのいずれかと置き換えます。`any`、`basic`、`weak`、`layout`、`strict`、`std`、または `strong`。*list* は、単一の型またはコンマで区切った型のリストと置き換えることも、単一のポインタまたはコンマで区切ったポインタのリストで置き換えることもできます。たとえば、`#pragma alias_level` を次のように発行できます。

- `#pragma alias_level` *level* `(type [, type] )`

- `#pragma alias_level` *level* `(pointer [, pointer] )`

このプラグマは、指定の別名レベルがリストの型に対応する翻訳単位のすべてのメモリー参照、または命名済みのポインタ変数が参照解除されている翻訳単位のすべての参照解除に適用されることを指定します。

特定の参照解除に対し複数の別名レベルを指定した場合、ポインタ名によって適用されたレベルがほかのすべてのレベルに優先します。型名によって適用されたレベルは、オプションによって適用されたレベルに優先します。次の例では、`#pragma alias_level` を `any` より上位に設定してプログラムをコンパイルした場合に、`std` レベルが *p* に適用されます。

```
typedef int * int_ptr;
int_ptr p;
#pragma alias_level strong (int_ptr)
#pragma alias_level std (p)
```

5.2.1.1 #pragma alias (type, type [, type]…)

このプラグマは、リストされているすべての型が相互に別名設定することを指定します。次の例では、コンパイラは、間接アクセス `*pt` が間接アクセス `*pf` を別名設定することを仮定します。

```
#pragma alias (int, float)
int *pt;
float *pf;
```

5.2.1.2 #pragma alias (pointer, pointer [, pointer] …)

このプラグマは、命名済みのポインタ変数の参照解除の地点で、参照解除されているポインタ値がそのほかの命名済みポインタ変数と同じオブジェクトをポイントできることを指定します。ただし、ポインタは、命名済みの変数に含まれるオブジェクトだけに制限されず、リストに含まれていないオブジェクトをポイントできます。このプラグマは、適用される別名レベルの別名設定仮定を無効にします。次の例では、プラグマに続く間接アクセス `p` と `q` が (2 つのポインタの型に関係なく) 別名設定すると見なされます。

```
#pragma alias(p, q)
```

5.2.1.3 #pragma may_point_to (pointer, variable [, variable] …)

このプラグマは、命名済みのポインタ変数の参照解除の地点で、参照解除されているポインタ値が命名済みの変数に含まれているオブジェクトにポイントできることを指定します。ただし、ポインタは、命名済みの変数に含まれるオブジェクトだけに制限されず、リストに含まれていないオブジェクトをポイントできます。このプラグマは、適用される別名レベルの別名設定仮定を無効にします。次の例では、コンパイラは、間接アクセス `*p` が直接アクセス `a`、`b`、および `c` を別名設定すると仮定します。

```
#pragma alias may_point_to(p, a, b, c)
```

5.2.1.4 #pragma noalias (type, type [, type]…)

このプラグマは、リストされている型が相互に別名設定しないことを指定します。次の例では、コンパイラは、間接アクセス *p が間接アクセス *ps を別名設定しないと仮定します。

```
struct S {
   float f;
   ...} *ps;

#pragma noalias(int, struct S)
int *p;
```

5.2.1.5 #pragma noalias (pointer, pointer [, pointer] …)

このプラグマは、命名済みのポインタ変数の参照解除の地点で、参照解除されているポインタがそのほかの命名済みポインタ変数と同じオブジェクトをポイントしないことを指定します。このプラグマは、適用されているそのほかすべての別名レベルを無効にします。次の例では、コンパイラは、間接アクセス *p が間接アクセス *q を (2 つのポインタの型に関係なく) 別名設定しないことを仮定します。

```
#pragma noalias(p, q)
```

5.2.1.6 #pragma may_not_point_to (pointer, variable [, variable] …)

このプラグマは、命名済みのポインタ変数の参照解除の地点で、参照解除されているポインタ値が命名済みの変数に含まれているオブジェクトをポイントしないことを指定します。このプラグマは、適用されているそのほかすべての別名レベルを無効にします。次の例では、コンパイラは、間接アクセス *p が直接アクセス a、b、または c を別名設定しないことを仮定します。

```
#pragma may_not_point_to(p, a, b, c)
```

5.3 lint によるチェック

lint プログラムは、同一レベルの型に基づく別名の明確化を、コンパイラの -xalias_level コマンドとして認識します。また、lint プログラムは、この章で説明されている型に基づく別名の明確化に関連するプラグマも認識します。-Xalias_level コマンドの詳細は、113 ページの「4.3.38 -Xalias_level[=*l*]」を参照してください。

lint が検出を行い、警告メッセージを生成する状況は 4 つあります。

- 構造体ポインタへスカラーポインタをキャストする
- 構造体ポインタへ void ポインタをキャストする

- スカラーポインタへ構造体フィールドをキャストする
- 明示的な別名設定を行わずに、Xalias_level=strict レベルの構造体ポインタへ構造体ポインタをキャストする

5.3.1 構造体ポインタへのスカラーポインタのキャスト

次の例では、integer 型のポインタ p が struct foo 型のポインタとしてキャストされます。lint -Xalias_level=weak (またはそれ以上) を指定すると、エラーが生成されます。

```
struct foo {
    int a;
    int b;
  };

struct foo *f;
int *p;

void main()
{
    f = (struct foo *)p; /* struct pointer cast of scalar pointer error */
}
```

5.3.2 構造体ポインタへの **void** ポインタのキャスト

次の例では、void 型のポインタ vp が構造体のポインタとしてキャストされます。lint -Xalias_level=weak (またはそれ以上) を指定すると、警告メッセージが生成されます。

```
struct foo {
    int a;
    int b;
  };

struct foo *f;
void *vp;

void main()
{
    f = (struct foo *)vp; /* struct pointer cast of void pointer warning */
}
```

5.3.3 構造体ポインタへの構造体フィールドのキャスト

次の例では、構造体メンバーのアドレス **foo.b** は構造体ポインタとしてキャストされ、**f2** に割り当てられます。lint -Xalias_level=weak (またはそれ以上) を指定すると、エラーが生成されます。

```
struct foo{
    int a;
    int b;
  };

struct foo *f1;
struct foo *f2;

void main()
{
    f2 = (struct foo *)&f1->b; /* cast of a scalar pointer to struct pointer error*/
}
```

5.3.4 明示的な別名設定が必要

次の例では、struct fooa 型のポインタ f1 が型 struct foob 型のポインタとしてキャストされています。lint で -Xalias_level=strict (またはそれ以上) を指定する場合、構造体の型がまったく同じ (同じ型で同数の構造体フィールド) でないかぎり、このようなキャストは明示的な別名設定を必要とします。また、別名レベルが standard と strong の場合、別名設定を実行するにはタグの一致が必要であると仮定されます。f1 への割り当て前に #pragma alias (struct fooa、struct foob) を実行すると、lint は警告メッセージの生成を停止します。

```
struct fooa {
    int a;
};

struct foob {
    int b;
};

struct fooa *f1;
struct foob *f2;

void main()
{
    f1 = (struct fooa *)f2; /* explicit aliasing required warning */
}
```

5.4 メモリー参照の制限の例

ここでは、実際のソースファイルに登場する可能性の高いコードの例を説明します。それぞれの例のあとに、コンパイラの仮定について説明します。これらの仮定は、適用レベルの型に基づいた解析によって作成されます。

5.4.1 例 1

次のコードを考えてみましょう。さまざまなレベルの別名設定でコンパイルすることにより、それぞれの型の別名設定の関係性を理解できます。

```
struct foo {
    int f1;
    short f2;
    short f3;
    int f4;
} *fp;

struct bar {
    int b1;
    int b2;
    int b3;
} *bp;

int *ip;
short *sp;
```

この例が -xalias_level=any オプションでコンパイルされる場合、コンパイラは次の間接アクセスを相互の別名とみなします。

*ip、*sp、*fp、*bp、fp->f1、fp->f2、fp->f3、fp->f4、bp->b1、bp->b2、bp->b3

この例が -xalias_level=basic オプションでコンパイルされる場合、コンパイラは次の間接アクセスを相互の別名とみなします。

*ip、*bp、fp->f1、 fp->f4、bp->b1、bp->b2、 bp->b3

また、*sp、fp->f2、および fp->f3 は相互に別名設定でき、*sp および *fp も相互に別名設定できます。

しかし、-xalias_level=basic を指定した場合、コンパイラは次のように仮定します。

- *ip は *sp を別名設定しません。
- *ip は、fp->f2 および fp->f3 を別名設定しません。
- *sp は、fp->f1、fp->f4、bp->b1、bp->b2、および bp->b3 を別名設定しません。

2 つの間接アクセスの基本型が異なるため、コンパイラはこれらの仮定を作成します。

この例が -xalias_level=weak オプションでコンパイルされる場合、コンパイラは、次の別名情報を仮定します。

- *ip は、*fp、 fp->f1、fp->f4、*bp、bp->b1、bp->b2、および bp->b3 を別名設定できます。
- *sp は、*fp、fp->f2、および fp->f3 を別名設定できます。
- fp->f1 は、bp->b1 を別名設定できます。
- fp->f4 は、bp->b3 を別名設定できます。

コンパイラは、fp->fp1 が bp->b2 を別名設定しないと仮定します。これは、f1 が構造体に 0 のオフセットを保持するフィールドである一方で、b2 が構造体に 4 バイト

のオフセットを保持するフィールドであるからです。同様に、コンパイラは、fp->f1 が bp->b3 を別名設定せず、fp->f4 が bp->b1 または bp->b2 を別名設定しないと仮定します。

この例が -xalias_level=layout オプションでコンパイルされる場合、コンパイラは、次の情報を仮定します。

- *ip は、*fp、 *bp、 fp->f1、 fp->f4、 bp->b1、 bp->b2、および bp->b3 を別名設定できます。
- *sp は、*fp、 fp->f2、および fp->f3 を別名設定できます。
- fp->f1 は、bp->b1 および *bp を別名設定できます。
- *fp および *bp は相互に別名設定できます。

fp->f4 は、bp->b3 を別名設定しません。これは、f4 と b3 が foo および bar の共通初期シーケンスで対応するフィールドではないからです。

この例が -xalias_level=strict オプションでコンパイルされる場合、コンパイラは、次の別名情報を仮定します。

- *ip は、*fp、 fp->f1、 fp->f4、 *bp、 bp->b1、 bp->b2、および bp->b3 を別名設定できます。
- *sp は、*fp、 fp->f2、および fp->f3 を別名設定できます。

-xalias_level=strict を指定すると、コンパイラは、*fp、*bp、fp->f1、fp->f2、fp->f3、fp->f4、bp->b1、bp->b2、および bp->b3 が相互に別名設定しないと仮定します。これは、フィールド名が無視されるときに、foo および bar が同じではないからです。ただし、fp は fp->f1 を別名設定し、bp は bp->b1 を別名設定します。

この例が -xalias_level=std オプションでコンパイルされる場合、コンパイラは、次の別名情報を仮定します。

- *ip は、*fp、 fp->f1、 fp->f4、 *bp、 bp->b1、 bp->b2、および bp->b3 を別名設定できます。
- *sp は、*fp、 fp->f2、および fp->f3 を別名設定できます。

ただし、fp->f1 は、bp->b1、bp->b2、または bp->b3 を別名設定しません。これは、フィールド名が注視されるときに、foo および bar が同じではないからです。

この例が -xalias_level=strong オプションでコンパイルされる場合、コンパイラは、次の別名情報を仮定します。

- *ip は、fp->f1、 fp->f4、 bp->b1、 bp->b2、および bp->b3 を別名設定しません。ポインタ (*ip など) が構造体内部をポイントしないはずだからです。
- 同様に、*sp は、fp->f1 または fp->f3 を別名設定しません。
- 型が異なるため、*ip は、*fp、*bp、および *sp を別名設定しません。

- 型が異なるため、*sp は、*fp、*bp、および *ip を別名設定しません。

5.4.2 例 2

次の例のソースコードを考えてみましょう。さまざまなレベルの別名設定でコンパイルすることにより、それぞれの型の別名設定の関係性を理解できます。

```
struct foo {
    int f1;
    int f2;
    int f3;
} *fp;

struct bar {
    int b1;
    int b2;
    int b3;
} *bp;
```

この例が -xalias_level=any オプションでコンパイルされる場合、コンパイラでは、次の別名情報を仮定します。

*fp、*bp、fp->f1、fp->f2、fp->f3、bp->b1、bp->b2、および bp->b3 はすべて相互に別名設定できます。2 つのメモリーアクセスが -xalias_level=any レベルで相互に別名設定するからです。

この例が -xalias_level=basic オプションでコンパイルされる場合、コンパイラでは、次の別名情報を仮定します。

*fp、*bp、fp->f1、fp->f2、fp->f3、bp->b1、bp->b2、および bp->b3 はすべて相互に別名設定できます。すべての構造体フィールドが同じ基本型であるため、ポインタ *fp および *bp を使用する 2 つのフィールドアクセスは、この例において相互に別名設定できます。

この例が -xalias_level=weak オプションでコンパイルされる場合、コンパイラは、次の別名情報を仮定します。

- *fp および *fp は相互に別名設定できます。
- fp->f1 は、bp->b1、 *bp、および *fp を別名設定できます。
- fp->f2 は、bp->b2、 *bp、および *fp を別名設定できます。
- fp->f3 は、bp->b3、 *bp、および *fp を別名設定できます。

ただし、-xalias_level=weak を指定すると、次の制限が課されます。

- fp->f1 は、bp->b2 または bp->b3 を別名設定しません。f1 のオフセットが 0 である一方で、b2 が 4 バイトのオフセットを保持し、b3 が 8 バイトのオフセットを保持するからです。

- fp->f2 は、bp->b1 または bp->b3 を別名設定しません。f2 が 4 バイトのオフセットを保持する一方で、b1 が 0 バイトのオフセットを保持し、b3 が 8 バイトのオフセットを保持するからです。
- fp->f3 は、bp->b1 または bp->b2 を別名設定しません。f3 が 8 バイトのオフセットを保持する一方で、b1 のオフセットが 0 バイトであり、b2 が 4 バイトのオフセットを保持するからです。

この例が -xalias_level=layout オプションでコンパイルされる場合、コンパイラでは、次の別名情報を仮定します。

- *fp および *bp は相互に別名設定できます。
- fp->f1 は、bp->b1、 *bp、および *fp を別名設定できます。
- fp->f2 は、bp->b2、 *bp、および *fp を別名設定できます。
- fp->f3 は、bp->b3、 *bp、および *fp を別名設定できます。

ただし、-xalias_level=layout を指定すると、次の制限が課されます。

- fp->f1 は、bp->b2 または bp->b3 を別名設定しません。foo および bar の共通初期シーケンスにおいて、フィールド f1 がフィールド b1 に対応するからです。
- fp->f2 は、bp->b1 または bp->b3 を別名設定しません。foo および bar の共通初期シーケンスにおいて、フィールド f2 がフィールド b2 に対応するからです。
- fp->f3 は、bp->b1 または bp->b2 を別名設定しません。foo および bar の共通初期シーケンスにおいて、フィールド f3 がフィールド b3 に対応するからです。

この例が -xalias_level=strict オプションでコンパイルされる場合、コンパイラは、次の別名情報を仮定します。

- *fp および *bp は相互に別名設定できます。
- fp->f1 は、bp->b1、 *bp、および *fp を別名設定できます。
- fp->f2 は、bp->b2、 *bp、および *fp を別名設定できます。
- fp->f3 は、bp->b3、 *bp、および *fp を別名設定できます。

ただし、-xalias_level=strict を指定すると、次の制限が課されます。

- fp->f1 は、bp->b2 または bp->b3 を別名設定しません。foo および bar の共通初期シーケンスにおいて、フィールド f1 がフィールド b1 に対応するからです。
- fp->f2 は、bp->b1 または bp->b3 を別名設定しません。foo および bar の共通初期シーケンスにおいて、フィールド f2 がフィールド b2 に対応するからです。
- fp->f3 は、bp->b1 または bp->b2 を別名設定しません。foo および bar の共通初期シーケンスにおいて、フィールド f3 がフィールド b3 に対応するからです。

この例が -xalias_level=std オプションでコンパイルされる場合、コンパイラは、次の別名情報を仮定します。

fp->f1、 fp->f2、 fp->f3、 bp->b1、 bp->b2、 および bp->b3 は相互に別名設定しません。

この例が -xalias_level=strong オプションでコンパイルされる場合、コンパイラは、次の別名情報を仮定します。

fp->f1、fp->f2、fp->f3、 bp->b1、bp->b2、および bp->b3 は相互に別名設定しません。

5.4.3 例 3

次の例のソースコードは、特定レベルの別名設定が内部ポインタを処理できないことを示します。内部ポインタの定義については、表 B–13 を参照してください。

```
struct foo {
        int f1;
        struct bar *f2;
        struct bar *f3;
        int f4;
        int f5;
        struct bar fb[10];
} *fp;

struct bar
        struct bar *b2;
        struct bar *b3;
        int b4;
} *bp;

bp=(struct bar*)(&fp->f2);
```

この例の参照解除は、weak、layout、strict、または std でサポートされません。ポインタ割り当て bp=(struct bar*)(&fp->f2) の実行後、対になった 次のメモリーアクセスは同じメモリー位置に接触します。

- fp->f2 および bp->b2 は、同じメモリー位置にアクセスします。
- fp->f3 および bp->b3 は、同じメモリー位置にアクセスします。
- fp->f4 および bp->b4 は、同じメモリー位置にアクセスします。

ただし、オプション weak、layout、strict、および std を指定する場合、コンパイラは、fp->f2 および bp->b2 が別名設定しないことを仮定します。コンパイラがこのように仮定する理由は、b2 のオフセットが 0 である一方で f2 が 4 バイトのオフセットを保持することと、foo および bar が共通の初期シーケンスを保持しないことにあります。同様に、コンパイラは、bp->b3 が fp->f3 を別名設定しないこと、および bp->b4 が fp->f4 を別名設定しないことも仮定します。

そのため、ポインタ割り当てbp=(struct bar*)(&fp->f2) により、別名情報に関するコンパイラの仮定が正しくない状況が作成されます。これにより、最適化が正しく行われない可能性があります。

次の例に示されている変更を行なったあとで、コンパイルを実行してください。

```
struct foo {
        int f1;
        struct bar fb;   /* Modified line */
#define f2 fb.b2         /* Modified line */
#define f3 fb.b3         /* Modified line */
#define f4 fb.b4         /* Modified line */
        int f5;
        struct bar fb[10];
} *fp;

struct bar
        struct bar *b2;
        struct bar *b3;
        int b4;
} *bp;

bp=(struct bar*)(&fp->f2);
```

ポインタ割り当て `bp=(struct bar*)(&fp->f2)` の実行後、対になった 次のメモリーアクセスは同じメモリー位置に接触します。

- `fp->f2` および `bp->b2`
- `fp->f3` および `bp->b3`
- `fp->f4` および `bp->b4`

前述のコード例に示された変更内容を調べることにより、式 `fp->f2` が式 `fp->fb.b2` のもう 1 つの書式であることが理解できます。`fp->fb` の型が `bar` であるため、`fp->f2` は `bar` の `b2` フィールドにアクセスします。さらに、`bp->b2` も `bar` の `b2` フィールドにアクセスします。そのため、コンパイラは、`fp->f2` が `bp->b2` を別名設定することを仮定します。同様に、コンパイラは、`fp->f3` が `bp->b3` を別名設定し、`fp->f4` が `bp->b4` を別名設定することも仮定します。その結果、コンパイラの仮定する別名設定は、ポインタ割り当てで設定された実際の別名と一致します。

5.4.4 例 4

次の例のソースコードを考えてみましょう。

```
struct foo {
        int f1;
        int f2;
} *fp;

struct bar {
        int b1;
        int b2;
} *bp;

struct cat {
        int c1;
        struct foo cf;
        int c2;
        int c3;
```

```
} *cp;

struct dog {
        int d1;
        int d2;
        struct bar db;
        int d3;
} *dp;
```

この例が -xalias_level=weak オプションでコンパイルされる場合、コンパイラは、次の別名情報を仮定します。

- fp->f1 は、bp->b1、cp->c1、dp->d1、cp->cf.f1、および df->db.b1 を別名設定できます。
- fp->f2 は、bp->b2、cp->cf.f1、dp->d2、cp->cf.f2、 df->db.b2、および cp->c2 を別名設定できます。
- bp->b1 は、fp->f1、cp->c1、dp->d1、cp->cf.f1、および df->db.b1 を別名設定できます。
- bp->b2 は、fp->f2、cp->cf.f1、dp->d2、cp->cf.f1、および df->db.b2 を別名設定できます。

fp->f2 は、cp->c2 を別名設定できます。*dp が *cp を別名設定でき、*fp が dp->db を別名設定できるからです。

- cp->c1 は、fp->f1、bp->b1、dp->d1、および dp->db.b1 を別名設定できます。
- cp->cf.f1 は、fp->f1、fp->f2、bp->b1、bp->b2、 dp->d2、および dp->d1 を別名設定できます。

cp->cf.f1 は、dp->db.b1 を別名設定しません。

- cp->cf.f2 は fp->f2、bp->b2、dp->db.b1、および dp->d2 を別名設定できます。
- cp->c2 は、dp->db.b2 を別名設定できます。

cp->c2 は dp->db.b1 を別名設定せず、 cp->c2 は dp->d3 を別名設定しません。

オフセットに関連して、*dp が cp->cf を別名設定する場合にかぎり、cp->c2 は db->db.b1 を別名設定できます。ただし、*dp が cp->cf を別名設定する場合、dp->db.b1 は foo cf の末尾を超えて別名設定する必要がありますが、これはオブジェクトの制限事項で禁じられています。そのため、コンパイラは、cp->c2 が db->db.b1 を別名設定できないと仮定します。

cp->c3 は、dp->d3 を別名設定できます。

cp->c3 は、dp->db.b2 を別名設定しません。参照解除に関連する型のフィールドのオフセットが異なり、重複することがないため、これらのメモリー参照は別名設定を行いません。この事実に基づき、コンパイラは、それらのメモリー参照が別名設定できないと仮定します。

- dp->d1 は、fp->f1、bp->b1、および cp->c1 を別名設定できます。

- dp->d2 は、fp->f2、bp->b2、および cp->cf.f1 を別名設定できます。
- dp->db.b1 は、fp->f1、bp->b1、および cp->c1 を別名設定できます。
- dp->db.b2 は、fp->f2、bp->b2、cp->c2、および cp->cf.f1 を別名設定できます。
- dp->d3 は、cp->c3 を別名設定できます。

dp->d3 は、cp->cf.f2 を別名設定しません。参照解除に関連する型のフィールドのオフセットが異なり、重複することがないため、これらのメモリー参照は別名設定を行いません。この事実に基づき、コンパイラは、それらのメモリー参照が別名設定できないと仮定します。

この例が -xalias_level=layout オプションでコンパイルされる場合、コンパイラでは、次の別名情報だけを想定します。

- fp->f1、bp->b1、cp->c1、および dp->d1 はすべて相互に別名設定できます。
- fp->f2、bp->b2、および dp->d2 はすべて相互に別名設定できます。
- fp->f1 は、cp->cf.f1 および dp->db.b1 を別名設定できます。
- bp->b1 は、cp->cf.f1 および dp->db.b1 を別名設定できます。
- fp->f2 は、cp->cf.f2 および dp->db.b2 を別名設定できます。
- bp->b2 は、cp->cf.f2 および dp->db.b2 を別名設定できます。

この例が -xalias_level=strict オプションでコンパイルされる場合、コンパイラでは、次の別名情報だけを想定します。

- fp->f1 および bp->b1 は相互に別名設定できます。
- fp->f2 および bp->b2 は相互に別名設定できます。
- fp->f1 は、cp->cf.f1 および dp->db.b1 を別名設定できます。
- bp->b1 は、cp->cf.f1 および dp->db.b1 を別名設定できます。
- fp->f2 は、cp->cf.f2 および dp->db.b2 を別名設定できます。
- bp->b2 は、cp->cf.f2 および dp->db.b2 を別名設定できます。

この例が -xalias_level=std オプションでコンパイルされる場合、コンパイラでは、次の別名情報だけを想定します。

- fp->f1 は、cp->cf.f1 を別名設定できます。
- bp->b1 は、dp->db.b1 を別名設定できます。
- fp->f2 は、cp->cf.f2 を別名設定できます。
- bp->b2 は、dp->db.b2 を別名設定できます。

5.4.5 例 5

次の例のソースコードを考えてみましょう。

```
struct foo {
        short f1;
        short f2;
        int   f3;
} *fp;
```

```
struct bar {
        int b1;
        int b2;
} *bp;

union moo {
        struct foo u_f;
        struct bar u_b;
} u;
```

それぞれの別名レベルに基づいて、コンパイラは次のように仮定します。

- この例が `-xalias_level=weak` オプションでコンパイルされる場合、`fp->f3` および `bp->b2` は相互に別名設定できます。
- この例が `-xalias_level=layout` オプションでコンパイルされる場合、フィールドは相互に別名設定できません。
- この例が `-xalias_level=strict` オプションでコンパイルされる場合、`fp->f3` および `bp->b2` は相互に別名設定できます。
- この例が `-xalias_level=std` オプションでコンパイルされる場合、フィールドは相互に別名設定できません。

5.4.6 例 6

次の例のソースコードを考えてみましょう。

```
struct bar;

struct foo {
        struct foo *ffp;
        struct bar *fbp;
} *fp;

struct bar {
        struct bar *bbp;
        long        b2;
} *bp;
```

それぞれの別名レベルに基づいて、コンパイラは次のように仮定します。

- この例が `-xalias_level=weak` オプションでコンパイルされる場合、`fp->ffp` および `bp->bbp` だけが相互に別名設定できます。
- この例が `-xalias_level=layout` オプションでコンパイルされる場合、`fp->ffp` および `bp->bbp` だけが相互に別名設定できます。
- この例が `-xalias_level=strict` オプションでコンパイルされる場合、フィールドは別名設定できません。タグが削除されたあとも、2 つの構造体の型が異なるからです。
- この例が `-xalias_level=std` オプションでコンパイルされる場合、フィールドは別名設定できません。2 つの型とタグが同じではないからです。

5.4.7 例 7

次の例のソースコードを考えてみましょう。

```
struct foo;
struct bar;
#pragma alias (struct foo, struct bar)

struct foo {
        int f1;
        int f2;
} *fp;

struct bar {
        short b1;
        short b2;
        int   b3;
} *bp;
```

この例のプラグマにより、foo および bar が相互に別名設定できることがコンパイラに伝えられます。コンパイラは、別名情報について次のように仮定します。

- fp->f1 は、bp->b1、bp->b2、および bp->b3 を別名設定できます。
- fp->f2 は、 bp->b1、bp->b2、および bp->b3 を別名設定できます。

◆ ◆ ◆ 第 6 章

ISO C への移行

この章では、K&R C のアプリケーションを移植し、9899:1990 ISO/IEC C 規格に適合させるために役立つ情報について説明します。この章の内容は、最新の 9899:1999ISO/IEC C 規格に適合させる必要がないため、`-xc99=none` を使用する場合を想定しています。C コンパイラのデフォルトは、9899:1999 ISO/IEC C 規格をサポートする `-xc99=all` です。

6.1 基本モード

ISO C コンパイラでは、古い形式と新しい形式の両方の C コードを使用できます。次の `-X` (大文字の X であることに注意) オプションを `-xc99=none` と併せて使用すると、コンパイラに ISO C 規格への準拠の度合いを指定できます。`-Xa` はデフォルトのモードです。コンパイラのデフォルトのモードは `-xc99=all` であるため、各 `-X` オプションを指定した場合の動作は `-xc99` の設定に依存します。

6.1.1 -Xc

(c = conformance) ISO C に最大限に準拠します。K&R C との拡張互換性はありません。ISO C にない構文を使用しているプログラムに対して、エラーと警告を発行します。

6.1.2 -Xa

ISO C に K&R C との拡張互換性を持たせます。ISO C に従って意味解釈を変更します。同じ言語構造に対して K&R C と ISO C の意味解釈が異なる場合は、相違に関する警告を発行した上で、ISO C に準拠した解釈を行います。これは、デフォルトのモードです。

6.1.3 -Xt

(t = transition) ISO C に K&R C との拡張互換性を持たせます。ISO C に従った意味解釈の変更は行いません。同じ構文に対して K&R C と ISO C の意味解釈が異なる場合は、相違に関する警告を発行した上で、K&R C に準拠した解釈を行います。

6.1.4 -Xs

(s = K&R C) コンパイルした言語には、ISO K&R C と互換性を持つすべての機能が含まれます。ISO C と K&R C の間で動作が異なるすべての言語構文に対して、警告を発行します。

6.2 古い形式の関数と新しい形式の関数の併用

1990 ISO C 規格での最大の変更点は、C++ 言語の機能である関数プロトタイプを使用できることです。各関数にパラメータの数と型を指定することにより、すべての通常のコンパイルにおいて、関数呼び出しごとに (`lint` のように) 引数とパラメータが検査されるだけではなく、引数が (代入だけで) 自動的に関数が期待する型に変換されます。プロトタイプを使用するように変更できる (また、変更すべき) 既存の C コードが非常に多く存在するため、1990 ISO C 規格には、古い形式と新しい形式の関数宣言を併用する規則が含まれています。

1999 ISO C 規格によって、古い形式の関数宣言は廃止されました。

6.2.1 新しいコードを書く

まったく新しいプログラムを書くとき、ヘッダーでは、新しい形式の関数宣言 (関数プロトタイプ) を使用し、それ以外の C ソースファイルでは、新しい形式の関数宣言と関数定義を使用します。しかし、ISO C 以前のコンパイラを持つマシンにコードを移植する可能性がある場合は、ヘッダーとソースファイルの両方で、マクロ `__STDC__` (ISO C コンパイルシステム専用に定義されている) を使用することをお勧めします。例については、149 ページの「6.2.3 併用に関する考慮点」を参照してください。

同じオブジェクトまたは関数に対して 2 つの互換性のない宣言が同じスコープの中にある場合、ISO C 準拠のコンパイラは診断メッセージを発行しなければいけません。すべての関数がプロトタイプで宣言および定義され、適切なヘッダーが正しいソースファイルにインクルードされている場合、すべての呼び出しは関数の定義に従うはずです。この取り決めによって、起こりがちな C プログラミングの誤りを防ぐことができます。

6.2.2 既存のコードを更新する

既存のアプリケーションで関数プロトタイプを利用したい場合、どれくらいのコードを変更するかによって、更新方法が異なります。

1. 変更せずに再コンパイルする

   コードを変更しなくても、-v オプションでコンパイラを実行すると、パラメータの型と数の不一致について警告が発行されます。

2. ヘッダーだけに関数プロトタイプを追加する

   大域的な関数へのすべての呼び出しが診断の対象になります。

3. ヘッダーには関数プロトタイプを追加し、各ソースファイルの先頭には局所 (静的な) 関数に対する関数プロトタイプを追加する

   関数へのすべての呼び出しが診断の対象になります。ただしこの方法では、ソースファイル内で局所関数ごとに 2 回インタフェースを入力する必要があります。

4. すべての関数宣言と関数定義を、関数プロトタイプを使用するように変更する

結果として受ける恩恵とそのための負荷を考えると、ほとんどの場合、前述の 2 か 3 が適切な選択だと言えるでしょう。ただしこれらを選択する場合、古い形式と新しい形式を併用するための規則を詳細に知っておく必要があります。

6.2.3 併用に関する考慮点

関数プロトタイプ宣言と古い形式の関数定義がともに機能するためには、両方が機能的に同じインタフェースを指定しなければいけません。つまり、ANSI/ISO C の用語を使用する「互換形式」を持っていなければいけません。

可変引数を持つ関数の場合は、ANSI/ISO C の省略記号と古い形式の `varargs()` 関数定義を併用することはできません。固定数のパラメータを持つ関数の場合、以前の実装で渡したとおりのパラメータの型を指定するだけです。

K&R C では、各引数は、呼び出された関数に渡される直前に、デフォルトの引数拡張に従って変換されました。 このような拡張では、`int` より狭いすべての整数型を `int` サイズに拡張し、また、任意の `float` 引数を `double` に拡張するように指定されていたため、コンパイラとライブラリの両方が単純化されていました。関数プロトタイプを使用すると、よりわかりやすく表現できます。つまり、指定したパラメータの型が、そのまま、関数に渡されるパラメータの型となります。

したがって、既存の (古い形式の) 関数定義用に関数プロトタイプを書く場合、関数プロトタイプに次の型のパラメータは使用できません。

char	signed char	unsigned char	float
short	signed short	unsigned short	

プロトタイプを書く際には、さらに 2 つの問題があります。typedef 名と、狭い unsigned 型の拡張規則です。

古い形式の関数内のパラメータが typedef 名で宣言されている場合 (off_t や ino_t など)、typedef 名がデフォルトの引数拡張によって影響を受ける型を指しているかどうかを確認することが重要です。前述の 2 つの typedef 名を例にすると、off_t は long です。したがって、関数プロトタイプで使用することは適切な使用方法です。しかし、ino_t は unsigned short であったため、プロトタイプで使用すると、古い形式の定義とプロトタイプが異なる互換性のないインタフェースを指定するため、診断メッセージが発行されます。

最後の問題は、unsigned short の代わりに何を使用するかです。K&R C と 1990 ANSI/ISO C コンパイラ間の最大の非互換性の 1 つは、unsigned char と unsigned short を int 値に広げるための拡張規則です (153 ページの「6.4 拡張: 符号なし保存と値の保持」を参照)。このような古い形式のパラメータにあたる型は、コンパイル時に使用するコンパイルモードによって異なります。

- -Xs と -Xt では unsigned int を使用する
- -Xa と -Xc では int を使用する

最良の方法は、int または unsigned int のどちらかを指定するように古い形式の定義を変更して、一致する型を関数プロトタイプで使用することです。必要であれば、関数を入力したあとでも、より狭い型の値を局所変数に代入できます。

前処理によって影響を受ける可能性のあるプロトタイプでは、ID の使用に気を付けてください。次の例を考えてみましょう。

```
#define status 23
void my_exit(int status);   /* Normally, scope begins */
                            /* and ends with prototype */
```

関数プロトタイプは、狭い型を持つ古い形式の関数定義と併用できません。

```
void foo(unsigned char, unsigned short);
void foo(i, j) unsigned char i; unsigned short j; {...}
```

__STDC__ を適切に使用すれば、古いコンパイラと新しいコンパイラの両方で使用できるヘッダーファイルを作成できます。

```
header.h:
    struct s { /* .  .  .  */ };
    #ifdef __STDC__
       void errmsg(int, ...);
       struct s *f(const char *);
```

```
      int g(void);
  #else
    void errmsg();
    struct s *f();
    int g();
  #endif
```

次の関数はプロトタイプを使用していますが、古いシステムでもコンパイルできます。

```
struct s *
#ifdef __STDC__
    f(const char *p)
#else
    f(p) char *p;
#endif
{
    /* .  .  .  */
}
```

次の例は、更新したソースファイルです (選択肢 3 を使用したもの)。局所関数は古い形式の定義を使用していますが、新しいコンパイラ用にプロトタイプも含まれています。

```
source.c:
   #include “header.h”
      typedef /* .  .  .  */ MyType;
   #ifdef __STDC__
      static void del(MyType *);
      /* .  .  .  */
      static void
      del(p)
      MyType *p;
      {
      /* .  .  .  */
      }
      /* .  .  .  */
```

6.3 可変引数を持つ関数

以前の実装では、関数が期待するパラメータの型を指定できませんでした。しかし、ISO C でプロトタイプを使用すれば、これを指定できます。`printf()` などの関数をサポートするために、プロトタイプの構文では特別な省略記号 (…) が終了を示す記号として使用されます。実装によっては可変引数を処理するために特別なことを行う必要があるため、ISO C では、すべての宣言とこのような関数などの定義が末尾に省略記号を含むべきであると規定しています。

パラメータの「…」部分には名前がないため、`stdarg.h` に含まれている特別なマクロにはこれらの引数へアクセスするための関数が含まれています。初期のバージョンではこのような関数は `varargs.h` に含まれている同様なマクロを使用しなければいけませんでした。

次に、これから書こうとする関数が errmsg() というエラーハンドラであると仮定します。この関数は void を返し、固定パラメータとして、エラーメッセージの詳細を指定する int だけを持つと仮定します。このパラメータのあとには、ファイル名または行番号 (あるいは、その両方) を指定できます。そして、ファイル名または行番号のあとには、(printf() のような) エラーメッセージのテキストを指定する書式と引数を指定できます。

初期のコンパイラで前述の例をコンパイルするには、ISO C コンパイルシステム専用に定義されたマクロ __STDC__ を多く使用する必要があります。したがって、適切なヘッダーファイルにおける関数の宣言は次のようになります。

```
#ifdef __STDC__
    void errmsg(int code, ...);
#else
    void errmsg();
#endif
```

errmsg() の定義を持つファイルは、古い形式と新しい形式を併用できます。まず、インクルードするヘッダーはコンパイルシステムによって異なります。

```
#ifdef __STDC__
#include <stdarg.h>
#else
#include <varargs.h>
#endif
#include <stdio.h>
```

そのあとで fprintf() と vfprintf() を呼び出すため、stdio.h をインクルードしています。

次は関数の定義です。識別子 va_alist と va_dcl は古い形式の varargs.h インタフェースの一部です。

```
void
#ifdef __STDC__
errmsg(int code, ...)
#else
errmsg(va_alist) va_dcl /* Note: no semicolon! */
#endif
{
   /* more detail below */
}
```

古い形式の変数引数メカニズムでは固定パラメータを指定できないため、固定パラメータは、可変部分の前でアクセスされるように配置しなければいけません。また、パラメータの「…」部分に名前がないため、新しい va_start() マクロは 2 番目の引数 (「…」の直前にあるパラメータの名前) を持ちます。

拡張として、Solaris Studio ISO C では、固定パラメータなしで関数を宣言および定義できます。

```
int f(...);
```

このような関数の場合、va_start() は 2 番目の引数を空にして呼び出す必要があります。

```
va_start(ap,)
```

次は関数の本体です。

```
{
    va_list ap;
    char *fmt;
#ifdef __STDC__
    va_start(ap, code);
#else
    int code;
    va_start(ap);
    /* extract the fixed argument */
    code = va_arg(ap, int);
#endif
    if (code & FILENAME)
        (void)fprintf(stderr, "\"%s\": ", va_arg(ap, char *));
    if (code & LINENUMBER)
        (void)fprintf(stderr, "%d: ", va_arg(ap, int));
    if (code & WARNING)
        (void)fputs("warning: ", stderr);
    fmt = va_arg(ap, char *);
    (void)vfprintf(stderr, fmt, ap);
    va_end(ap);
}
```

va_arg() と va_end() マクロは両方とも古い形式と ISO C バージョンで同様に動作します。va_arg() は ap の値を変更するため、vfprintf() への呼び出しを次のようにすることはできません。

```
(void)vfprintf(stderr, va_arg(ap, char *), ap);
```

マクロ FILENAME、LINENUMBER、および WARNING の定義は、おそらく、errmsg() の宣言と同じヘッダーに含まれています。

errmsg() への呼び出しの例は次のようになります。

```
errmsg(FILENAME, "<command line>", "cannot open: %s\n",
argv[optind]);
```

6.4 拡張: 符号なし保存と値の保持

1990 ISO C 規格の「Rationale」(論理的根拠) 節に、次のような情報があります。「QUIET CHANGE」(メッセージなしの変更)。符号なし保存演算変換に依存するプログラムは、おそらくはメッセージを発行せずに、異なる動作を行います。これは、現在広く行われている慣習に対して委員会が行なったもっとも重大な変更であると考えられます。

この節では、この変更がコーディングにどのように影響するかを説明します。

6.4.1 背景

K&R の『プログラミング言語 C』によると、unsigned は 1 つだけの型を指定していました。つまり、unsigned char、unsigned short、unsigned long はありませんでした。しかし、ほとんどの C コンパイラにはすぐにこれらの型が追加されました。unsigned long を実装せず、残りの 2 つだけを実装するコンパイラもあります。当然、式の中でこれらの新しい型がほかの型と併用されている場合、実装によって異なる型拡張規則が適用されました。

ほとんどの C コンパイラでは、より簡単な規則「符号なし保存」が使用されています。 つまり、unsigned 型を拡張する必要があるときは unsigned 型に拡張します。そして、unsigned 型が signed 型と混合されているときも、unsigned 型に拡張されます。

ISO C では、「値の保持」という規則も指定されています。この規則では、拡張結果の型は、オペランドの型の相対的なサイズによって異なります。unsigned char または unsigned short を拡張するとき、int がより小さい型の値をすべて表現できる大きさである場合は、拡張結果の型は int になります。それ以外の場合、unsigned int になります。この「値の保持」規則に従えば、ほとんどの式が無難な演算結果になります。

6.4.2 コンパイルの動作

ISO C コンパイラは、移行モード (-Xt) または ISO 以前のモード (-Xs) では、符号なし保存拡張規則を適用します。準拠モード (-Xc) および ISO モード (-Xa) では、値保持拡張規則を使用します。

6.4.3 例 1: キャストの使用

次のコードでは、unsigned char が int より小さいと仮定します。

```
int f(void)
{
    int i = -2;
    unsigned char uc = 1;

   return (i + uc) < 17;
}
```

前述のコードを使用すると、-xtransition オプションを使用したときに、次の警告が発行されます。

```
line 6: warning: semantics of "<" change in ISO C; use explicit cast
```

加算の結果の型は int (値保持) または unsigned int (符号なし保存) です。しかし、どちらの場合でもビットパターンは同じです。2 の補数を使用するマシンでは、次のようになります。

```
    i:       111...110 (-2)
+   uc:      000...001 ( 1)
==================
        111...111 (-1 or UINT_MAX)
```

このビット表現は、int では -1 に対応し、unsigned int では UINT_MAX に対応します。したがって、結果の型が int の場合、符号付き比較が使用され、「より小さいか」の答えは真になります。結果の型が unsigned int の場合、符号なしの比較が行われ、「より小さいか」の答えは偽になります。

キャストの加算を使用すると、2 つの動作のうち、どちらを希望するかを指定できます。

```
value preserving:
    (i + (int)uc) < 17
unsigned preserving:
    (i + (unsigned int)uc) < 17
```

コンパイラが異なれば同じコードに対する解釈も異なるため、この式は曖昧になる可能性があります。キャストの加算を使用することにより、コードが読みやすくなると同時に、警告メッセージも発行されなくなります。

6.4.4 ビットフィールド

同じ動作が、ビットフィールド値の拡張にも適用されます。ISO C では、int または unsigned int ビットフィールド内のビットの数が int 中のビットの数よりも少ない場合、拡張される型は int です。それ以外の場合、拡張される型は unsigned int です。ほとんどの古い C コンパイラでは、明示的な符号なしビットフィールドの場合、拡張される型は unsigned int です。それ以外の場合は int です。

この場合も、キャストを使用することにより、曖昧になることを防ぐことができます。

6.4.5 例 2 : 同じ結果

次のコードでは、unsigned short と unsigned char の両方が int よりも狭いと仮定します。

```
int f(void)
{
    unsigned short us;
    unsigned char uc;
    return uc < us;
}
```

この例では、2 つの自動変数は int または unsigned int のどちらかに拡張されます。したがって、比較対象は符号なしになることも、符号付きになることもあります。しかし、どちらを選んでも結果は同じなので、警告は発行されません。

6.4.6 整数定数

式と同様に、ある整数定数の型の規則も変更されました。K&R C では、接尾辞なしの 10 進定数の型は、その値が int に収まる場合だけ int でした。接尾辞なしの 8 進定数または 16 進定数の型は、その値が unsigned int に収まる場合だけ int でした。それ以外の場合、整数定数の型は long でした。したがって、値が結果の型に収まらないことがありました。1990 ISO/IEC C 規格では、定数の型は、次のリストのうち、値を格納できる最初の型となります。

- 接尾辞なし 10 進数: int、 long、 unsigned long
- 接尾辞なし 8 進数または 16 進数: int、 unsigned int、 long、 unsigned long
- 接尾辞 U 付き: unsigned int、 unsigned long
- 接尾辞 L 付き: long、 unsigned long
- 接尾辞 UL 付き : unsigned long

ISO C コンパイラで -xtransition オプションを使用するとき、定数の型規則によって式の動作が異なる場合は警告が発行されます。古い整数定数の型規則は、移行モード (-Xt) だけで適用されます。ISO モード (-Xa) と準拠モード (-Xc) では、新しい規則が適用されます。

注 – 接尾辞なしの 10 進定数の型規則は、1999 ISO C 規格に従って変更されています。35 ページの「2.1.1 整数定数」を参照してください。

6.4.7 例 3 : 整数定数

次のコードでは、int が 16 ビットであると仮定します。

```
int f(void)
{
    int i = 0;

    return i > 0xffff;
}
```

16 進定数の型は int (2 の補数を使用するマシン上で - 1 の値を持つ) または unsigned int (65535 の値を持つ) のどちらかです。比較結果は、ANSI 以前モード (-Xs) と移行モード (-Xt) では真で、ANSI モード (-Xa) と準拠モード (-Xc) では偽です。

この場合も、キャストを適切に使用することにより、コードが読みやすくなり、警告も発行されなくなります。

```
-Xt, -Xs modes:
    i > (int)0xffff

-Xa, -Xc modes:
    i > (unsigned int)0xffff
       or
    i > 0xffffU
```

接尾辞 u 文字は ISO C の新しい機能であるため、古いコンパイラではおそらくエラーメッセージが生成されます。

6.5 トークン化と前処理

以前のバージョンの C でもっとも不明確な仕様は、各ソースファイルを文字の集合から一連のトークンに変換して構文解析できるようにするまでの操作でしょう。具体的には、空白 (コメントも含む) の認識、連続した文字のトークン化、前処理指令行の処理、およびマクロの置換などがあります。しかし、これら操作の優先順位は保証されていませんでした。

6.5.1 ISO C の翻訳段階

ISO C では、このような翻訳段階の順番が指定されています。

ソースファイル内のすべての 3 文字表記シーケンスが置換されます。ANSI/ISO C は、9 つの 3 文字表記シーケンスを持っています。これらのシーケンスはもともと文字セットの不完全な点を補うために導入されました。しかし、現在では、この 3 文字シーケンスは ISO 646-1983 文字セットに含まれない文字を指定するために使用されています。

表 6–1 3 文字シーケンス

3 文字シーケンス	変換後の文字
??=	#
??-	~
??(	[
??)	]
??!	|
??<	{
??>	}
??/	\

表 6–1　3 文字シーケンス　　(続き)

3 文字シーケンス	変換後の文字
??’	^

ISO C コンパイラは前述のシーケンスを理解するはずです。しかし、これらのシーケンスを使用することはお勧めしません。-xtransition オプションを使用したとき、移行モード (-Xt) では、ISO C コンパイラは 3 文字シーケンスを置換するたびに警告を発行します (コメント内でも)。たとえば、次の例を考えてください。

```
/* comment *??/
/* still comment? */
```

??/ はバックスラッシュになります。この文字とそれに続く改行は削除されます。結果として、次のようになります。

```
/* comment */* still comment? */
```

2 行目の最初の / は、コメントの終わりです。次のトークンは * です。

1. バックスラッシュと改行文字の組み合わせがすべて削除されます。
2. ソースファイルが前処理トークンと空白文字のシーケンスに変換されます。各コメントは必要最低限の空白文字で置換されます。
3. すべての前処理指令が処理され、すべてのマクロ呼び出しが置換されます。#include でインクルードされた各ソースファイルは、内容が指令行に置換される前の初期段階で実行されます。
4. すべてのエスケープシーケンス (文字定数と文字列リテラル) が解釈されます。
5. 隣接する文字列リテラルが連結されます。
6. すべての前処理トークンが通常のトークンに変換されます。コンパイラはこれらのトークンを適切に構文解析して、コードを生成します。
7. すべての外部オブジェクトと関数参照が解釈処理され、最終的なプログラムになります。

6.5.2 古い C の翻訳段階

以前の C コンパイラは、このような単純な順番に従いませんでした。また、これらの段階がいつ適用されるかも保証されていませんでした。コンパイラとは別のプリプロセッサが、マクロを置換して指令行を処理するときに、トークンと空白を認識していました。そして、コンパイラがプリプロセッサの出力を適切に再トークン化し、言語を構文解析し、コードを生成していました。

プリプロセッサ内のトークン化処理は必要に応じて行われる操作で、マクロ置換は (トークンベースではなく) 文字ベースの操作として行われます。したがって、前処理中にトークンと空白は大きく変動する可能性がありました。

2 つの方法の間には、いくつか異なる点があります。この節の後半では、マクロ置換中に発生する行の連結、マクロ置換、文字列化、およびトークンの連結によって、コードの動作がどのように変化するかを説明します。

6.5.3 論理的なソース行

K&R C では、バックスラッシュと改行を組み合わせた次の行には、指令、文字列リテラル、文字定数しか指定できませんでした。ANSI/ISO C ではこの概念が拡張され、バックスラッシュと改行の組みの次の行に、あらゆるものを指定できるようになりました。K&R では 1 行は 1 行でしたが ANSIC では複数行組み合わせて 1 行とでき、これが論理行です。したがって、バックスラッシュと改行の組み合わせのどちら側にあるかによってトークンの認識が異なるコードは、期待どおりに動作しません。

6.5.4 マクロ置換

ISO C 以前には、マクロ置換処理については詳細に定義されていません。この曖昧さのために、処理系に多くの差が生まれました。したがって、明白な定数置換や簡単な関数のようなマクロよりも複雑なものを持つコードは、おそらく完全には移植できません。このマニュアルでは、古い C と ISO C 間のマクロ置換実装の違いをすべて説明することはできません。ほとんどすべてのマクロ置換の結果は、前とまったく同じトークンの連続になります。ただし、ISO C マクロ置換アルゴリズムは、古い C ではできなかったことができます。たとえば、次を見てください。

```
#define name (*name)
```

この例は、すべての `name` を `name` 経由の間接参照で置換します。古い C プリプロセッサは数多くの括弧とアスタリスクを生成し、ときには、マクロの再帰についてエラーを生成する場合もあります。

ANSI/ISO C によるマクロ置換方法の主な変更は、マクロ置換演算子 `#` と `##` のオペランド以外のマクロ引数が要求であること、置換トークンリストでの置換前に再帰的に展開することです。ただし、この変更によって、実際に生成されるトークンに差が生じることは滅多にありません。

6.5.5 文字列の使用

注 – ISO C では、-xtransition オプションを使用するとき、次の例 (?) を使用すると、古い機能を使用しているという警告が生成されます。移行モード (-Xt と -Xs) の場合のみ、結果は以前のバージョンの C と同じになります。

K&R C では、次のコードは文字列リテラル「"x y!"」を生成しました:

```
#define str(a) "a!"   ?
str(x y)
```

プリプロセッサは、文字列リテラルと文字定数の内部で、マクロパラメータのように見える文字を検索していました。ISO C はこの機能の重要性を認識していましたが、トークンの部分にこの操作を行うことはできませんでした。ISO C では、前述のマクロの呼び出しはすべて文字列リテラル「a!」を生成します。ISO C で以前の効果を得るためには、# マクロ置換演算子と文字列リテラルの連結を使用してください。

```
#define str(a) #a "!"
str(x y)
```

前述のコードは、2 つの文字列リテラル "x y" と "!" を生成し、 連結したあと、同じ文字列 "x y!" を生成します。

文字定数用の操作を完全に代用するものはありません。この機能の主な使用方法は次のようなものでした。

```
#define CNTL(ch) (037 & 'ch')    ?
CNTL(L)
```

これは、次を生成します。

```
(037 & 'L')
```

これは、ASCII の Control-L 文字と同じです。最良の解決策は、このマクロを次のように変更することです。

```
#define CNTL(ch) (037 & (ch))
CNTL('L')
```

このコードの方が読みやすく式にも適用できるため、より使いやすくなっています。

6.5.6 トークンの連結

K&R C では、2 つのトークンを連結するために、少なくとも 2 つの方法がありました。次の 2 つの呼び出しは、2 つのトークン x と 1 から 1 つの識別子 x1 を生成します。

```
#define self(a) a
#define glue(a,b) a/**/b ?
self(x)1
glue(x,1)
```

ISO C では、どちらの方法も使用できません。ISO C では、前述の呼び出しは、両方とも 2 つの別々のトークン x と 1 を生成します。しかし、前述の呼び出しの内 2 番目の方法については、## マクロ置換演算子を使用すれば、ISO C 用に書き換えることができます。

```
#define glue(a,b) a ## b
glue(x, 1)
```

と ## は、__STDC__ が定義されているときだけ、マクロ置換演算子として使用しなければいけません。## は実際の演算子のため、定義と呼び出しの両方で空白をより自由に使うことができます。

コンパイラは、未定義の ## 演算に対して警告の診断を発行するようになりました (C 規格、3.4.3 節)。未定義とは、## を前処理したときの結果に、単一のトークンではなく、複数のトークンが含まれていることを意味します (C 規格、6.10.3.3(3) 節)。未定義の ## 演算の結果は、現在では、## のオペランドを連結することによって作成された文字列をプリプロセスすることによって生成された個別のトークンのうち最初のものと定義されます。

前述の古い形式の連結方法のうち、最初の方法を直接代用できる方法はありません。しかし、この方法では呼び出し時に連結の処理が必要なため、ほかの方法に比べてあまり使用されることはありませんでした。

6.6 const と volatile

キーワード const は、ISO C に組み込まれた C++ 機能の 1 つでした。アナログキーワード volatile が ISO C 委員会により考案されたとき、「型修飾子」カテゴリが作成されました。

6.6.1 右辺値 (lvalue) 専用の型

const と volatile は識別子の型の一部であり、記憶クラスの一部ではありません。ただし、この部分は多くの場合、式の評価中にオブジェクトの値が取り出されるとき (正確には、lvalue が rvalue になるとき) に、型の一番上の部分から削除されます。これらの用語はプロトタイプ代入式「L=R」から来ています。この意味は、左側がオブジェクト (lvalue) を直接参照しなければならず、右側が値 (rvalue) であるだけでよいということです。したがって、lvalues である式だけが const または volatile (あるいは、その両方) で修飾できます。

6.6.2 派生型の型修飾子

型修飾子は型名と派生型を変更します。派生型は C の宣言の一部であり、何度も適用することによって、より複雑な型 (ポインタ、配列、関数、構造体、共用体) を構築できます。関数を除き、1 つまたは両方の型修飾子を使用すると、派生型の動作を変更できます。

たとえば、次を見てください。

```
const int five = 5;
```

これは、型が const int であり、値が正しいプログラムによって変更されないオブジェクトを宣言し、初期化します。キーワードの順番は C にとって重要ではありません。たとえば、次を見てください。

```
int const five = 5;
```

および

```
const five = 5;
```

この 2 つの宣言の効果は前述の宣言と同じです。

次を見てください。

```
const int *pci = &five;
```

この宣言は、型が const int へのポインタである (つまり、以前宣言されたオブジェクトを指している) オブジェクトを宣言します。ポインタ自身は修飾型を持ちません。つまり、ポインタは修飾型を指すため、プログラムの実行中に任意の int を指すように変更できます。pci を使用して、pci が指すオブジェクトを変更することはできません。このためには、次のようにキャストを使用します。

```
*(int *)pci = 17;
```

pci が実際に const オブジェクトを指す場合、このコードの動作は未定義です。

次を見てください。

```
extern int *const cpi;
```

この宣言は、プログラム内のどこかに、型が int への const ポインタである大域オブジェクトの定義があることを意味します。この場合、正しいプログラムでは cpi の値は変更されません。しかし、cpi を使用して、cpi が指すオブジェクトを変更することはできます。前述の宣言において、const が * のあとにあることに注意してください。次の 2 つの宣言の効果は同じです。

```
typedef int *INT_PTR;
extern const INT_PTR cpi;
```

前述の宣言は、次の宣言のように連結できます。この場合、オブジェクトの型は const int への const ポインタであると宣言されます。

```
const int *const cpci;
```

6.6.3 const は readonly を意味する

なお、キーワードとしては通常 const よりも readonly を選択するほうが便利です。このように const を解釈すると、次のような宣言は簡単に理解できます。

```
char *strcpy(char *, const char *);
```

この宣言では、2 番目のパラメータは文字値を読み取るためだけに使用され、最初のパラメータはその値が指す文字を上書きすることを意味しています。さらに、前述の例の事実に関わらず、cpi の型は const int へのポインタです。したがって、実際に型が const int で宣言されたオブジェクトを指していないかぎり、その値が指すオブジェクトの値は別の方法で変更できます。

6.6.4 const の使用例

const の 2 つの主な使用法は、コンパイル時に初期化された大きな情報テーブルが未変更であると宣言することと、ポインタパラメータが指しているオブジェクトを変更しないことを指定することです。

最初の使用法では、同じプログラムのほかの並行呼び出しが、プログラムのデータ部分を共有可能にします。つまり、データはメモリーの読み取り専用部分にあるため、この不変データを変更しようとする試みを、ある種類のメモリー保護障害で即座に検出できます。

2 番目の使用法では、実行中にメモリー障害が発生する前に、潜在的なエラーを見つけることができます。たとえば、ヌル文字を挿入できない文字列に対して、ある関数が一時的にヌル文字を挿入しようとした場合、その関数は、コンパイル時、ヌル文字を挿入できない文字列へのポインタが渡されたときに検出されます。

6.6.5 volatile は文字どおりの解釈を意味する

ここまでは、例の中でいずれも const を使用してきました。概念としてはこの方が簡単だからです。しかし、volatile はどのような意味でしょうか。volatile という言葉は「揮発性の」、つまりすぐに変わってしまうという意味を持ちます。そのためコンパイラでは、コード生成時にこのようなオブジェクトにアクセスするためのショートカットは行われません。ANSI/ISO C では、オブジェクトを volatile 修飾型として宣言するかどうかはプログラマの責任であると規定しています。

6.6.6 volatile の使用例

volatile は、通常、次の 4 つのオブジェクトに使用します。

- メモリーにマップされた入出力ポートであるオブジェクト
- 複数の並行プロセス間で共有されるオブジェクト
- 非同期シグナルハンドラによって変更されるオブジェクト
- setjmp を呼び出す関数中で宣言され、その値が setjmp への呼び出しとそれに対応する longjmp への呼び出し間で変更される自動記憶オブジェクト

最初の 3 つの例はすべて、特定の動作を行うオブジェクトのインスタンスです。つまり、その値は、プログラムの実行中の任意の時点で変更できます。したがって、外見上は無限ループに見える次のコードは、

```
flag = 1;
while (flag);
```

flag が volatile 修飾型を持つ間は有効な文となります。おそらく、ある非同期イベントが将来 flag をゼロに設定することもあります。volatile 修飾型を持たない場合、flag の値はループ本体内では変更されないため、コンパイルシステムによって前述のループは、完全に flag の値を無視する本当の無限ループに変更されることもあり得ます。

4 番目の例は、setjmp を呼び出す関数に対して局所的な変数を含んでいるため、より複雑です。setjmp と longjmp の動作についての細字部分には、4 番目の例に一致するオブジェクトの値は保証されないという注記があります。もっとも望ましい動作を行うためには、setjmp を呼び出す関数と longjmp を呼び出す関数の間で、longjmp がすべてのスタックフレームを検査して、保存されたレジスタ値と比較することが必要です。スタックフレームは非同期的に作成される可能性があるため、この作業はより難しくなります。

自動オブジェクトを volatile 修飾型で宣言したとき、コンパイルシステムは、プログラマが書いたものと完全に一致するコードを生成します。したがって、このような自動オブジェクトに対する最新の値は常に、レジスタではなく、メモリー内に存在します。そして、longjmp が呼び出されたときに最新であることが保証されます

6.7 複数バイト文字とワイド文字

最初に、ISO C の国際化はライブラリ関数だけに影響がありました。しかし、国際化の最終段階 (複数バイト文字とワイド文字) は言語属性にも影響します。

6.7.1 アジア言語は複数バイト文字を必要とする

アジア言語を使用するコンピュータ環境における基本的な難しさは、膨大な数の表意文字を入出力しなければならないことです。通常のコンピュータアーキテクチャーの制限内で動作するためには、このような表意文字はバイトシーケンスとして符号化します。関連するオペレーティングシステム、アプリケーションプログラム、および端末は、このようなバイトシーケンスを個々の表意文字として認識します。さらに、すべてのこのような符号化によって、通常の 1 バイト文字を表意文字のバイトシーケンスと混合できます。個々の表意文字を認識することがどのくらい困難であるかは、使用する符号化方式によって異なります。

「複数バイト文字」は、ISO C の定義では、使用する符号化方式の種類に関係なく、表意文字を符号化するバイトシーケンスを示します。すべての複数バイト文字は「拡張文字セット」に属します。通常の 1 バイト文字は、単に複数バイト文字の特別なケースです。符号化に必要な唯一の条件は、どの複数バイト文字もヌル文字を符号化の一部として使用できないということです。

ISO C では、プログラムのコメント、文字列リテラル、文字定数、およびヘッダー名がすべて複数バイト文字のシーケンスであると規定されています。

6.7.2 符号化の種類

符号化方式は 2 つの種類に分けることができます。1 つは、各複数バイト文字が自己識別性を持つ方式です。つまり、どの複数バイト文字も簡単に 2 つの複数バイト文字の間に挿入できます。

もう 1 つは、特別なシフトバイトの存在が後続のバイトの解釈を変更する方式です。たとえば、あるキャラクタ端末で行描画モードを切り替えるために使用する方式がそうです。このシフト状態依存符号化による複数バイト文字で書かれたプログラムの場合、ISO C では、コメント、文字列リテラル、文字定数、およびヘッダー名の始まりと終わりがすべてシフトなし状態でなければならないと規定しています。

6.7.3 ワイド文字

複数バイト文字の処理で不都合が発生した場合は、すべての文字を一定のバイト数またはビット数にすることで解決できることがあります。このような文字セットには何千または何万もの表意文字があるため、これらすべてを保持するには、大きさが 16 ビットまたは 32 ビットの整数値を使用しなければいけません (完全な中国語には 65,000 以上もの表意文字がある)。ISO C には、拡張文字セットのすべてを保持するために十分な大きさを持つ実装定義の整数型として、`typedef` 名 `wchar_t` があります。

各ワイド文字には、それに対応する複数バイト文字があります (その逆もある)。つまり、通常の 1 バイト文字に対応するワイド文字は、その 1 バイト値と同じ値を持つ必要があります (ヌル文字も含む)。しかし、マクロ EOF が char として表現できないように、マクロ EOF の値が wchar_t に格納できるかどうかは保証されていません。

6.7.4 変換関数

1990 ISO/IEC C 規格では、複数バイト文字とワイド文字を管理するために、5 つのライブラリ関数を規定しています。1999 ISO/IEC C 規格では、さらに多くのこうした関数を規定しています。

6.7.5 C 言語の機能

アジア言語環境においてプログラマがより柔軟にプログラムを組むために、ISO C では、ワイド文字定数とワイド文字列リテラルを提供しています。この 2 つの形式は、直前に文字「L」の接頭辞が付くことを除き、通常の (ワイドでない) バージョンと同じです。

- 「x」 通常の文字定数
- 「¥」 通常の文字定数
- L'x' ワイド文字定数
- L'¥' ワイド文字定数
- "abc¥xyz" 通常の文字列リテラル
- L"abcxyz" ワイド文字列リテラル

複数バイト文字は、通常とワイドの両方のバージョンで有効です。表意文字 ¥ を生成するために必要なバイトシーケンスは符号化によって異なります。しかし、文字定数「¥」が複数のバイトから構成される場合、「ab」が実装により定義されるのと同様に、その値は実装により定義されます。エスケープシーケンスを除き、通常の文字列リテラルには、引用符の間に指定されたものと同じバイト数 (指定したすべての複数バイト文字のバイト数も含む) が含まれます。

コンパイルシステムがワイド文字定数またはワイド文字列リテラルを検出したとき、各複数バイト文字は (mbtowc() 関数を呼び出したように) ワイド文字に変換されます。したがって、L'¥' の型は wchar_t です。abc¥xyz の型は長さが 8 の wchar_t の配列です。通常の文字列リテラルと同様に、各ワイド文字列リテラルは、値がゼロの余分な要素が追加されます。しかし、この要素は、ゼロの値を持つ wchar_t です。

通常の文字列リテラルが文字配列初期化の簡単な方法として使用できるのと同様に、ワイド文字列リテラルも wchar_t 配列を初期化するために使用できます。

```
wchar_t *wp = L"a¥z";
wchar_t x[] = L"a¥z";
wchar_t y[] = {L'a', L'¥', L'z', 0};
wchar_t z[] = {'a', L'¥', 'z', '\0'};
```

前述の例では、3 つの配列 x、y、z と、wp が指す配列の長さは同じです。すべての配列は同じ値で初期化されます。

最後に、通常の文字列リテラルと同様に、隣接するワイド文字列リテラルは連結されます。しかし、1990 ISO/IEC C 規格では、通常の文字列リテラルとワイド文字列リテラルが隣接する場合、その動作は定義されていません。また、1990 ISO/IEC C 規格では、このような連結が受け付けられない場合、コンパイラはエラーを発行する必要はありません。

6.8 標準ヘッダーと予約名

標準化作業の初期の段階において ISO 規格委員会は、ライブラリ関数、マクロ、およびヘッダーファイルを ISO C の一部として含むことを選択しました。

この節では、さまざまな予約名のカテゴリとその予約名が必要な基本的な理由を示します。最後には、プログラムで予約名を使用しないようにするための規則を示します。

6.8.1 標準ヘッダー

標準ヘッダーは次のとおりです。

表 6–2 標準ヘッダー

assert.h	locale.h	stddef.h
ctype.h	math.h	stdio.h
errno.h	setjmp.h	stdlib.h
float.h	signal.h	string.h
limits.h	stdarg.h	time.h

ほとんどの実装では、さらに多くのヘッダーが用意されています。しかし、1990 ISO/IEC C に厳密に準拠するプログラムが使用できるのは、前述のヘッダーだけです。

これらヘッダーの一部の内容については、ほかの規格ではわずかに異なります。たとえば、POSIX (IEEE 1003.1) は、fdopen を stdio.h で宣言するように指定しています。これら 2 つの規格が共存するために、POSIX では、このような追加の名前が存在することを保証するためには任意のヘッダーをインクルードする前にマクロ _POSIX_SOURCE を #defined で定義しなければならないと規定しています。X/Open の『Portability Guide』によると、X/Open もこのマクロ方式を使用して拡張しています。X/Open のマクロは _XOPEN_SOURCE です。

ISO C は、標準ヘッダーがそれ自身だけで完結し、べき等 (何度指定しても同じ) であることを要求しています。どの標準ヘッダーも、その前後でほかのヘッダーを `#included` でインクルードする必要はありません。標準ヘッダーは何度 `#included` でインクルードしても、問題は発生しません。ISO C 規格では、安全なコンテキストにおいてのみ、標準ヘッダーを `#included` でインクルードすることを要求します。したがって、ヘッダーで使用される名前は変更されないことが保証されます。

6.8.2 実装で使用される予約名

ISO C 規格は、標準ライブラリについて、より多くの制限を実装に課しています。過去において、ほとんどのプログラマは UNIX システムでは独自の関数に `read` や `write` などの名前を使用しないように学びました。ISO C では、予約されている名前だけを実装内の参照で使用するように規定しています。

したがって ISO C 規格では、実装で使用する可能性があるすべての名前のサブセットが予約されています。この名前のクラスは下線で始まり、もう 1 つの下線または大文字の英字が続く識別子から構成されます。この名前のクラスは、次の正規表現に一致するすべての名前を含みます。

```
_[_A-Z][0-9_a-zA-Z]*
```

厳密には、プログラムがこのような識別子を使用する場合、その動作は未定義です。したがって、`_POSIX_SOURCE` (または、`_XOPEN_SOURCE`) を使用するプログラムの動作は未定義です。

ただし、動作がどれぐらい未定義なのかは異なります。POSIX 準拠の実装で `_POSIX_SOURCE` を使用する場合、ユーザーのプログラムの未定義の動作が特定のヘッダー内に追加された特定の名前から構成されていることと、受け入れられる標準にユーザーのプログラムが準拠していることは予測できます。ISO C 規格におけるこの故意の抜け道により、実装は外見上互換性のない仕様に準拠できます。一方、POSIX 規格に準拠しない実装は、`_POSIX_SOURCE` などの名前に遭遇したとき、任意の方法で動作できます。

ISO C 規格では、下線で始まるほかのすべての名前が (局所的なスコープではなく) ヘッダーファイルにおける通常のファイルのスコープの識別子として、および構造体と共用体のタグとして使用するために予約されています。従来通り、`_filbuf` と `_doprnt` という名前の関数によりライブラリの隠れた部分を実装することはできます。

6.8.3 拡張用の予約名

明示的に予約されたすべての名前に加えて、ISO C 規格は、次の特定のパターンに一致する名前を (実装と将来の規格用に) 予約しています。

表 6–3 拡張用の予約名

ファイル	予約名のパターン
errno.h	E[0-9A-Z].*
ctype.h	(to|is)[a-z].*
locale.h	LC_[A-Z].*
math.h	現在の関数名[fl]
signal.h	(SIG|SIG_)[A-Z].*
stdlib.h	str[a-z].*
string.h	(str|mem|wcs)[a-z].*

前述のリストにおいて、大文字の英字で始まる名前はマクロで、関連するヘッダーがインクルードされるときだけ予約されます。残りの名前は関数を示し、大域的なオブジェクトや関数を指定する場合には使用できません。

6.8.4 安全に使用できる名前

ANSI/ISO C の予約名と衝突しないようにするためには、次の 4 つの簡単な規則に従う必要があります。

- すべてのシステムヘッダーは、ユーザーのソースファイルの最初に #include でインクルードする。_POSIX_SOURCE または _XOPEN_SOURCE (あるいは、その両方) の #define 行がある場合は、そのあとでインクルードする。
- 下線で始まる名前は定義または宣言しない。
- すべてのファイルスコープのタグと通常名の最初の数文字では、下線または大文字の英字を使用する。stdarg.h または varargs.h 内の va_ 接頭辞に注意する。
- すべてのマクロ名の最初の数文字では、数字または小文字の英字を使用する。errno.h を #included でインクルードする場合、E で始まるほとんどすべての名前は予約されています。

ほとんどの実装はデフォルトで標準ヘッダーに名前を追加しているため、これらの規則は一般的なガイドラインに過ぎません。

6.9 国際化

164 ページの「6.7 複数バイト文字とワイド文字」 では、標準ライブラリの国際化を紹介しました。この節では、国際化の影響を受けるライブラリ関数について説明し、これらの機能を利用するにはどのようにプログラムを書けばいいかのヒントを提供します。この節では、1990 ISO/IEC C 規格に関する国際化についてのみ説明します。1999 ISO/IEC C 規格には、この節で説明する国際化のサポートについて大幅な拡張機能はありません。

6.9.1 ロケール

C プログラムは常に、現在のロケール (国、文化、および言語に適切な規約を記述した情報の集まり) を持っています。ロケールは文字列の名前を持っています。標準化されたロケール名は、"C" と "" の 2 つだけです。どのプログラムも "C" ロケールから始まります。つまり、すべてのライブラリ関数は従来どおりに動作します。"" ロケールは、各処理系がプログラムの呼び出しに最適であると推測する規約セットです。"C" と "" の動作は同じになることもあります。ほかのロケールは各処理系によって提供されます。

実用性と便宜上の目的により、ロケールはカテゴリに分類されます。プログラムは、ロケール全体を変更することも、1 つまたは複数のカテゴリを変更することもできます。一般的に各カテゴリは、ほかのカテゴリが影響を与える関数とは関係なく、複数の関数に影響を与えます。したがって、一時的に 1 つのカテゴリを変更することにも意味があります。

6.9.2 `setlocale()` 関数

`setlocale()` 関数は、プログラムのロケールとのインタフェースです。一般的に、国の規約を呼び出して使用するプログラムは、プログラムの実行パスの前のほうで、次のような呼び出しを行わなければいけません。

```
#include <locale.h>
/*...*/
setlocale(LC_ALL, "");
```

これは、プログラムの実行パスの前のほうで行います。プログラムの実行パスの最初の部分 `LC_ALL` は 1 つのカテゴリではなく、ロケール全体を指定するマクロであるため、この呼び出しによって、プログラムの現在のロケールが適切なローカルバージョンに変更されます。次に、標準的なカテゴリを示します。

`LC_COLLATE`	ソート情報
`LC_CTYPE`	文字分類情報

LC_MONETARY	通貨の出力情報
LC_NUMERIC	数値の出力情報
LC_TIME	日付と時刻の出力情報

前述の任意のマクロを setlocale() への最初の引数として渡すことによって、そのカテゴリを指定できます。

setlocale() 関数は、特定のカテゴリ (または、LC_ALL) に対する現在のロケールの名前を返します。2 番目の引数がヌルポインタの場合は、照会専用として機能します。したがって次のようなコードを使用すると、制限された期間だけロケール (または、その一部) を変更できます。

```
#include <locale.h>
/*...*/
char *oloc;
/*...*/
oloc = setlocale(LC_category, NULL);
if (setlocale(LC_category, "new") != 0)
{
        /* use temporarily changed locale */
    (void)setlocale(LC_category, oloc);
}
```

ほとんどのプログラムではこの機能は必要ありません。

6.9.3 変更された関数

変更が適切で可能である場合、既存のライブラリ関数はロケールに依存する動作を含むように拡張されました。これらの関数は、次の 2 つのグループに分類できます。

- ctype.h ヘッダーで宣言される関数 (文字の分類と変換)
- 数値を出力可能な形式から内部的な形式に (または、その逆に) 変換する関数 (printf() や strtod() など)

すべての ctype.h 述語関数 (isdigit() と isxdigit() を除く) は、現在のロケールの LC_CTYPE カテゴリが "C" 以外の場合に、追加の文字に対してゼロでない (真の) 値を返すことができます。スペイン語ロケールでは isalpha('ñ') は真になります。同様に、文字変換関数 tolower() と toupper() は、isalpha() 関数で識別される特別な英字を適切に処理できます。ctype.h 関数は、ほとんどの場合、文字引数による索引付きテーブル検索を使用して実装されるマクロです。これらの関数の動作を変更するには、テーブルを新しいロケールの値に再設定します。したがって、パフォーマンスに影響はありません。

出力可能な浮動小数点値を書き込んだり解釈したりする前述の関数は、現在のロケールの LC_NUMERIC カテゴリが "C" 以外の場合に、ピリオド (.) 以外の小数点文字を使用するように変更できます。千単位区切り型文字で数値を出力可能な形式に変換するための規定はありません。出力刷可能な形式から内部的な形式に変換するときにも、実装では、"C" 以外のロケールの場合に、このような追加の形式を受け入れることが許可されています。小数点文字を使用する関数は、printf() と scanf() のグループ、atof()、および strtod() です。実装での定義を拡張できる関数は、atof()、atoi()、atol()、strtod()、strtol()、strtoul()、および scanf() のグループです。

6.9.4 新しい関数

新しい標準関数として、特定のロケールに依存する機能が追加されました。ロケール自身を制御する setlocale() 以外にも、ANSI/ISO C 規格には次の新しい関数が導入されました。

localeconv()	数値/通貨の規約
strcoll()	2 つの文字列の照合順序
strxfrm()	照合のために文字列を変換する
strftime()	日付と時刻の形式を決定する

さらに、複数バイト関数 mblen()、mbtowc()、mbstowcs()、wctomb()、および wcstombs() があります。

localeconv() 関数は、現在のロケールの LC_NUMERIC と LC_MONETARY カテゴリに適切な、書式化された数値および通貨の情報に便利な情報を含む構造体へのポインタを返します。この関数は、動作が複数のカテゴリに依存する唯一の関数です。数値の場合、構造体は、小数点文字、千単位区切り文字、および区切り文字を置く場所を記述します。通貨値を書式化する方法を記述する構造体のメンバーは、ほかにも 15 個あります。

strcoll() 関数は、strcmp() 関数と似ていますが、現在のロケールの LC_COLLATE カテゴリに従って、2 つの文字列を比較するところが異なります。strxfrm() 関数は、変換後の 2 つの文字列を strcmp() に渡すと、変換前の 2 つの文字列を strcoll() に渡した場合に返される順番と似た順番が返されるように、文字列を別の文字列に変換します。

strftime() 関数は、struct tm に値を持つ sprintf() で使用される書式化と似た書式化と、さらに、現在のロケールの LC_TIME カテゴリに依存する日付と時刻の書式を提供します。この関数は、UNIX System V Release 3.2 の一部としてリリースされた ascftime() 関数に基づいています。。

6.10 式のグループ化と評価

C の設計において Dennis Ritchie が行なった選択の 1 つとして、式の中で数学的に交換可能で結合可能な演算子が隣接する場合、括弧が存在する場合でも、その式を再配置する権利をコンパイラに与えました。このことは、Kernighan と Ritchie 著の『プログラミング言語 C』の付録に明示的に記載されています。しかし、ISO C は、この権利をコンパイラに与えませんでした。

この節では、前述の 2 つの C の定義間の違いを説明します。また、次のコードにおける式文を考えることによって、式の副作用、グループ化、および評価の間の区別を明らかにします。

```
int i, *p, f(void), g(void);
/*...*/
i = *++p + f() + g();
```

6.10.1 定義

式の副作用とは、メモリーへの変更と、`volatile` 修飾オブジェクトへのアクセスのことです。前述の式の副作用とは、`i` と `p` の更新と、関数 `f()` と `g()` 内に含まれる任意の副作用です。

式のグループ化とは、値をほかの値や演算子と結合させる方法です。前述の式のグループ化は、主に加算を実行する順番です。

式の評価には、その結果の値を生成するために必要なすべてが含まれます。式を評価するためには、指定したすべての副作用が以前のシーケンスポイントから次のシーケンスポイントまでの間で発生しなければならず、指定した演算が特定のグループ化で実行されなければいけません。前述の式の場合、`i` と `p` の更新は、以前の文からこの式文の `;` までの間に発生しなければいけません。関数への呼び出しは、以前の文からその戻り値が使用されるまでの間に、任意の順番で発生できます。特に、メモリーを更新する演算子には、演算の値が使用される前に新しい値を代入しなければならないという制約はありません。

6.10.2 K&R C の再配置の権利

前述の式では加算が数学的に交換可能で、また結合可能であるため、K&R C の再配置の権利が前述の式に適用されます。通常の括弧と実際の式のグループ化を区別するために、左右の中括弧でグループ化を示します。この式の場合、次の 3 つのグループ化が考えられます。

```
i = { {*++p + f()} + g() };
i = { *++p + {f() + g()} };
i = { {*++p + g()} + f() };
```

前述のすべてのグループ化は、K&R C の規則であれば有効です。さらに、たとえば、次のように式を書き換えた場合でも、前述のすべてのグループ化は有効です。

```
i = *++p + (f() + g());
i = (g() + *++p) + f();
```

オーバーフローによって例外が発生するか、あるいは、オーバーフローで加算と減算が逆にならないアーキテクチャー上でこの式が評価される場合、加算の 1 つがオーバーフローしたとき、前述の 3 つのグループ化の動作は異なります。

このようなアーキテクチャー上では、K&R C では、式を分割することによって強制的にグループ化するしか方法がありません。次に、前述の 3 つのグループ化を強制的に行うために式を分割した例を示します。

```
i = *++p; i += f(); i += g()
i = f(); i += g(); i += *++p;
i = *++p; i += g(); i += f();
```

6.10.3 ISO C の規則

ISO C では、数学的に交換可能で結合可能であるが、対象となるアーキテクチャー上では実際にそうではない演算を再配置することは許可されていません。したがって、ANSI/ISO C の文法の優先度と結合規則では、すべての式のグループ化が完全に記述されています。つまり、すべての式は、構文解析されるとおりにグループ化されなければいけません。前述の式は、次の方法でグループ化されます。

```
i = { {*++p + f()} + g() };
```

このコードでもなお「`f()` が `g()` よりも前に呼び出されなければならない」、あるいは、「`g()` が呼び出されるよりも前に `p` が増分されなければならない」ということはありません。

ISO C では、予想外のオーバーフローが発生しないように式を分割する必要があります。

6.10.4 括弧

ISO C では、不十分な理解と不正確な表現のために、括弧の信頼性と括弧に従った評価について、間違って記述されることがしばしばあります。

ISO C の式は構文解析で指定されるグループ化を持つため、括弧は、どのように式が構文解析されるかを制御する方法としてだけ機能します。つまり、式の自然な優先度と結合規則が括弧とまったく同じ重要さを持ちます。

前述の式は、次のように書くこともできます。

```
i = (((*(++p)) + f()) + g());
```

グループ化と評価に与える影響は、括弧を使用しない場合と同じです。

6.10.5 as if 規則

K&R C の再配置規則には、いくつかの理由がありました。

- 再配置によって、より多くの最適化の機会が生まれること (たとえば、コンパイル時の定数折り畳み)
- ほとんどのマシンにおいて、再配置によって整数型の式の結果が変わらないこと
- すべてのマシンにおいて、いくつかの演算が数学的にも演算的にも交換可能で結合可能であること

ISO C 委員会は、記述される対象アーキテクチャーに適用されるときに、再配置規則は「as if」規則のインスタンスになるものであると、最終的に確信しました。ISO C の「as if」規則は、有効な C プログラムの動作を変更しないかぎり、実装が必要に応じて抽象マシン記述から離れることを一般的に許可しています。

したがって、すべてのビット単位の 2 項演算子 (シフトを除く) は任意のマシンで再配置できます。これは、このような再グループ化を確認できる方法がないためです。2 の補数を使用するマシンでオーバーフローが発生しない場合は、いくつかの理由のため、乗算または加算を含む整数式は再配置できます。

したがって、C におけるこの変更は、ほとんどの C プログラマには重要な影響を与えません。。

6.11 不完全な型

C の当初から内在し、C の基本的な部分であるがまだ真価を認められていない部分を正式なものとするために、ISO C 規格は「不完全な型」を導入しました。この節では、不完全な型がどこで許可されるかと、なぜ便利であるかを説明します。

6.11.1 型

ISO は C の型を、関数、オブジェクト、および不完全の 3 つに区分しました。関数型の定義は明白です。オブジェクト型は、サイズが不明なオブジェクトを除く、そのほかすべてのものを示します。ANSI/ISO C 規格は、明示されるオブジェクトのサイズが既知でなければならないことを指定するために、「オブジェクト型」を使用します。しかし、`void` 以外の不完全な型もオブジェクトを指すことは十分に理解してください。

不完全な型には、void、不特定長の配列、および不特定内容の構造体と共用体の 3 つの種類しかありません。型 void は、完成させることができない不完全な型であるという点でほかの 2 つとは異なります。そして、特別な関数の戻り型とパラメータ型として機能します。

6.11.2 不完全な型を完全にする

不完全な配列型を完全なものにするには、同じオブジェクトを示す同じスコープ内にある後続の宣言で、配列のサイズを指定します。同じ宣言でサイズが不明な配列 (不特定長の配列) が宣言および初期化されるとき、その配列は、宣言の終わりから初期化の終わりまでの間だけ、不完全な型になります。

不完全な構造体型または共用体型を完成させるには、同じタグの同じスコープ内にある後続の宣言で、構造体型または共用体型の内容を指定します。

6.11.3 宣言

不完全な型を使用できる宣言もありますが、完全なオブジェクト型が必要な宣言もあります。オブジェクト型を必要とする宣言は、配列要素、構造体または共用体のメンバー、および関数に局所的なオブジェクトです。ほかのすべての宣言は、不完全な型を許可します。特に、次の構造が許可されています。

- 不完全な型へのポインタ
- 不完全な型を返す関数
- 不完全な関数パラメータ型
- 不完全な型の typedef 名

関数の戻り型とパラメータ型は特別です。このような方法で使用される不完全な型 (void を除く) は、関数が宣言または呼び出されるときまでに完全にならなければいけません。void の戻り型は、値を返さない関数を指定します。また、void の単一のパラメータ型は、引数を受け入れない関数を指定します。

配列と関数のパラメータ型はポインタ型に書き換えられるため、配列のパラメータ型は外見上不完全ですが、実際には不完全ではありません。典型的な main の argv (つまり、char *argv[]) の宣言は、不特定長の文字ポインタの配列として、文字ポインタへのポインタとして書き換えられます。

6.11.4 式

ほとんどの式演算子では完全なオブジェクト型が必要ですが、例外が 3 つあります。単項 & 演算子、コンマ演算子の最初のオペランド、および ?: 演算子の 2 番目と 3 番目のオペランドです。ポインタのオペランドを受け入れるほとんどの演算子は、ポインタ演算が要求されないかぎり、不完全な型へのポインタも許可します。この中には、単項 * 演算子も含まれます。たとえば、次の例を見てください。

```
void *p
```

&*p は、この例を使用する有効な式の一部です。

6.11.5 正当性

なぜ不完全な型が必要なのでしょうか。void を除いて、C ではほかの方法で扱えない不完全な型の唯一の機能は、構造体と共用体の前方参照です。たとえば、2 つの構造体がお互いを指すポインタを必要とする場合、これを実現するためには、不完全な型を使用しなければいけません。

```
struct a { struct b *bp; };
struct b { struct a *ap; };
```

異なる形式のポインタや異なる種類のデータ型を持つ、強力な型依存プログラミング言語には、すべて前述のようなケースを処理するための方法が用意されています。

6.11.6 例

不完全な構造体型や共用体型には typedef 名の定義が役立ちます。データ構造が複雑な (お互いへのポインタを多数持つような) 場合は、構造体への typedef のリストを前方に (中心となるヘッダーに) 指定することによって、宣言が簡単になります。

```
typedef struct item_tag Item;
typedef union note_tag Note;
typedef struct list_tag List;
.  .  .
struct item_tag { .  .  .  };
.  .  .
struct list_tag {
    struct list_tag {
};
```

さらに、内容がプログラムの残りで使用できてはいけない構造体や共用体に対しては、内容なしのタグをヘッダーに宣言できます。プログラムのほかの部分は、何の問題もなく不完全な構造体や共用体へのポインタを使用できます。ただし、そのメンバーは使用できません。

不特定長の外部配列は不完全な型として頻繁に使用されます。一般的に、配列の内容を使用するために、配列の大きさを知る必要はありません。

6.12 互換型と複合型

K&R C では (ISO C の場合はさらに顕著ですが)、同じ要素を参照する 2 つの宣言を別のものとして扱うことができます。ISO C は、このような「ある程度似ている」型を示すために、「互換型」という用語を使用します。この節では、この互換型と、2 つの互換型を結合した「複合型」を説明します。

6.12.1 複数の宣言

C プログラムにおいて各オブジェクトまたは関数の宣言が 1 度しか許されていないのであれば、互換型は必要ないはずです。しかし、同じ要素を参照する複数の宣言を許可するリンク、関数のプロトタイプ、および分割コンパイルには、このような機能が必要です。複数の翻訳単位 (ソースファイル) 間では、型の互換性の規則は 1 つの翻訳単位内のものとは異なります。

6.12.2 分割コンパイル間の互換性

各コンパイルでは別々のソースファイルを参照するため、分割コンパイル間の互換型に対して、ほとんどの規則の内容は次のように構造化されています。

- 一致するスカラー (整数、浮動小数点、およびポインタ) 型は、同じソースファイル内にある場合のように、互換性を持たなければならない。
- 一致する構造体、共用体、および列挙型のメンバー数は同じでなければならない。一致する各メンバーは (分割コンパイルという意味で) 互換型を持たなければならない (ビットフィールド幅も含む)。
- 一致する構造体のメンバーの順番は、同じでなければならない。共用体と列挙型のメンバーの順番は問題にならない。
- 一致する列挙型のメンバーの値は、同じでなければならない。

  さらに、構造体、共用体、および列挙型のメンバーの名前 (名前なしメンバーに名前がないということ) も一致しなければいけません。しかし、それぞれのタグは必ずしも一致する必要はありません。

6.12.3 単一のコンパイルでの互換性

同じスコープ内の 2 つの宣言が同じオブジェクトまたは関数を記述するとき、この 2 つの宣言は互換型を指定しなければいけません。これら 2 つの型は次に、最初の 2 つと互換性を持つ、1 つの複合型に結合されます。複合型についてはあとで説明します。

互換型は再帰的に定義されます。一番下は型指定子のキーワードです。これらの規則は、unsigned short は unsigned short int と同じであり、型指定子なしの型は int を持つ型と同じであることを示します。ほかのすべての型は、派生元の型が互換性を持つときだけ、互換性を持ちます。たとえば、修飾子 const と volatile が同じであり、修飾されていない基底型が互換性を持つ場合、2 つの修飾型は互換性を持ちます。

6.12.4 互換ポインタ型

2 つのポインタ型が互換性を持つためには、この 2 つのポインタが指す型が互換性を持ち、2 つのポインタが同じように修飾されていなければいけません。ポインタの修飾子は * のあとに指定されることを念頭に置いて、次の例を見てください。

```
int *const cpi;
int *volatile vpi;
```

前述の 2 つの宣言は、同じ型 int を指すが修飾が異なる 2 つのポインタを宣言しています。

6.12.5 互換配列型

2 つの配列型が互換性を持つためには、この 2 つの配列の要素の型が互換性を持たなければいけません。両方の配列の型のサイズが指定されている場合は、両方のサイズも一致しなければいけません。つまり、不完全な配列型 (175 ページの「6.11 不完全な型」を参照) は、ほかの不完全な配列型とも、サイズが指定されている配列型とも互換性を持ちます。

6.12.6 互換関数型

関数が互換性を持つためには、次の規則に従わなければいけません。

- 2 つの関数型が互換性を持つためには、その戻り型が互換性を持たなければいけません。どちらか、あるいは両方の関数型がプロトタイプを持つ場合、規則はより複雑になります。
- プロトタイプを持つ 2 つの関数型が互換性を持つためには、(省略記号 (...) も含む) パラメータの数が同じで、対応するパラメータもパラメータ互換でなければいけません。
- 古い形式の関数定義がプロトタイプを持つ関数型と互換性を持つためには、プロトタイプの最後のパラメータが省略記号 (...) であってはいけません。プロトタイプの各パラメータは、デフォルトの引数拡張の適用後、対応する古い形式のパラメータとパラメータ互換でなければいけません。

- 古い形式の関数宣言 (定義ではない) が、プロトタイプを持つ関数型と互換性を持つためには、プロトタイプの最後のパラメータが省略記号 (...) であってはいけません。プロトタイプのすべてのパラメータは、デフォルトの引数拡張で影響を受けない型でなければいけません。
- 2 つの型がパラメータ互換になるためには、これら 2 つの型は、1 番上に修飾子があればそれが削除されたあと、そして、関数型または配列型が適切なポインタ型に変換されたあとに、互換性を持たなければいけません。

6.12.7 特別な場合

`signed int` は `int` と同じように動作します。ただし、ビットフィールドで通常の `int` が unsigned 動作を示す数になる場合を除きます。

また、列挙型は同じ整数型と互換性を持たなければいけません。移植可能なプログラムの場合、これは、列挙型が別の型であることを意味します。一般的に、ISO C 規格はこのように列挙型を扱います。

6.12.8 複合型

2 つの互換型から 1 つの複合型への作成も再帰的に定義されます。不完全な配列型や古い形式の関数型を使用することにより、互換型をお互いに異なるようにできます。同様に、複合型をもっとも簡単に記述するには、元の両方の型 (元の型のすべての使用可能な配列サイズとパラメータリストも含む) と型の互換性を持たせればよいでしょう。

◆ ◆ ◆ 第 7 章

7 64 ビット環境に対応するアプリケーションへの変換

この章では、32 ビットまたは 64 ビットのコンパイル環境用のコードを作成するために必要なことについて説明します。

32 ビット、64 ビット両方のコンパイル環境で動作するコードを作成または変更する場合、次の 2 つの基本的な問題に直面します。

- 異なるデータ型モデル間でのデータ型の統一
- 異なるデータ型モデルを使用するアプリケーション間の相互動作

通常、複数のソースツリーを保守するより、`#ifdef` をできるだけ少なくした 1 つのソースコードを保守するほうが便利です。このため、この章では、32 ビットと 64 ビット両方のコンパイラ環境で正しく機能するコードを作成する際のガイドラインを示します。場合によっては、現在のコードを再コンパイルして、64 ビットライブラリに再リンクすればよいだけのこともあります。しかし、コードの修正が必要になる場合もあり得るため、この章では、こうした変換をより簡単に行うためのツールと参考情報について説明します。

7.1 データ型モデルの相違点

32 ビットと 64 ビットコンパイル環境の最大の違いは、データ型モデルにあります。

32 ビットアプリケーション用の C のデータ型モデルは ILP32 モデルです。この名前は、integer、long、pointer が 32 ビットデータ型であることから名付けられています。long と pointer が 64 ビットの大きさになったことから名付けられた LP64 データ型モデルは、業界の関連企業から構成されるコンソーシアムが作成したものです。残りの C のデータ型の `int`、`long long`、`short`、`char` はどちらのデータ型モデルでも同じです。

C の整数型間の標準の関係は、次に示すようにデータ型モデルに関係なく有効です。

```
sizeof (char) <= sizeof (short) <= sizeof (int) <= sizeof (long)
```

ILP32 と LP64 データ型モデルの基本的な C のデータ型と対応するサイズ (単位: ビット) は、次の表に示すとおりです。

表 7–1 ILP32 と LP64 のデータ型のサイズ

C データ型	LP32	LP64
char	8	8
short	16	16
int	32	32
long	32	64
long long	64	64
pointer	32	64
enum	32	32
float	32	32
double	64	64
long double	128	128

現在の 32 ビットアプリケーションでは integer、pointer、long が同じサイズであるとみなされることが多くあります。LP64 データ型モデルでは、long と pointer のサイズが変更されているため、この変更だけでも、ILP32 から LP64 への変換で多くの問題が発生する可能性があります。

また、宣言と型変換を調べることも非常に重要です。データ型が変わると、式の評価方法が影響を受ける可能性があります。データ型のサイズが変わると、標準 C 変換規則の処理が影響を受けます。意図したことを正しく示すには、定数の型を明示的に宣言してください。式で型変換を使用して、意図したとおりに式が評価されるようにすることもできます。このことは、意図したことを指示する上で明示的な型変換が欠かせない符号拡張部で特に必要になります。

7.2 単一ソースコードの実現

この節では、32 ビットと 64 ビットの両方でコンパイル可能な単一ソースコードの作成に使用できる資源をいくつか紹介します。

7.2.1 派生型

32 ビットと 64 ビットのどちらのコンパイル環境でも安全なコードにするには、システム派生型を使用します。一般的に、変更の可能性がある場合には派生型を使用す

ることをお勧めします。派生データ型を使用すると、データ型モデルの変更あるいは移植に際して、システム派生型を変更すればよいだけになります。

システムインクルードファイルの <sys/types.h> および <inttypes.h> には、32 ビットと 64 ビットのどちらにも安全なアプリケーションの作成に役立つ定数、マクロ、派生型が含まれています。

7.2.1.1 <sys/types.h>

アプリケーションのソースファイルに <sys/types.h> をインクルードして、_LP64 および _ILP32 の定義を使用できるようにしてください。このヘッダーには、必要に応じて使用される基本派生型もいくつか含まれています。特に次は大切です。

- clock_t クロックの刻み数でシステム時間を表します。
- dev_t デバイス番号に使用されます。
- off_t ファイルのサイズとオフセットに使用されます。
- ptrdiff_t 2 つのポインタの減算結果用の符号付き整数型です。
- size_t メモリー上のオブジェクトのサイズをバイト数で表します。
- ssize_t バイト数あるいはエラー発生通知を返す関数によって使用されます。
- time_t 秒数で時間をカウントします。

これらの派生型はすべて、ILP32 コンパイル環境では 32 ビット量のままですが、LP64 コンパイル環境では、64 ビット量になります。

7.2.1.2 <inttypes.h>

<inttypes.h> インクルードファイルには、コンパイル環境に関係なく、明示的にサイズ指定されたデータ項目との互換性を持たせるのに役立つ定数、マクロ、派生型が含まれています。このファイルには、8、16、32、64 ビットオブジェクトを操作するための仕組みも含まれています。inttypes.h は、新しい 1999 ISO/IEC C 規格の一部であり、このファイルが 1999 ISO/IEC C 規格に含まれるように、ファイルの内容は従っています。近い将来、このファイルは更新され、1999 ISO/IEC C 規格に完全に適合する予定です。<inttypes.h> に含まれることが議論されている基本機能としては、次のものがあります。

- 固定幅の整数型
- uintptr_t などの便利な型
- 定数マクロ
- 制限
- 書式文字列マクロ

次に <inttypes.h> のこれらの基本機能について詳しく説明します。

固定幅の整数型

<inttypes.h> が提供する固定幅の整数型には、int8_t、int16_t、int32_t、int64_t などの符号付整数型と、uint8_t、uint16_t、uint32_t、uint64_t などの符号なし整数型があります。

指定数のビットを保持できる最小サイズの整数型として定義されている派生型としては、int_least8_t、...、int_least64_t、uint_least8_t、...、uint_least64_t などがあります。

ループカウンタやファイル記述子などの演算に int または unsigned int を使用することは問題ありません。配列インデックスに long を使用することも問題ありません。しかし、これらの固定幅型はむやみに使用しないでください。固定幅の型は、次の明示的なバイナリ表現に使用してください。

- ディスク上のデータ
- データ回線上のデータ
- ハードウェアレジスタ
- バイナリのインタフェース仕様
- バイナリのデータ構造体

unintptr_t などの便利な型

<inttypes.h> ファイルには、ポインタを保持するのに十分な大きさの符号付き整数型と符号なし整数型 intptr_t と uintptr_t が含まれます。また、<inttypes.h> は符号付きと符号なし整数型の中で最長 (ビット) の整数型である intmax_t と uintmax_t も提供します。

uintptr_t 型は、符号なし long などの基本型ではなく、ポインタ用の整数型として使用してください。ILP32 と LP64 コンパイル環境で符号なし long とポインタが同じサイズであるとしても、uintptr_t を使用するということは、データ型モデルが変わった場合に、その影響を受けるのは uintptr_t の定義だけになることを意味します。このため、ほかの多くのシステムにコードを移植できるようになります。また、これは、C で自分の意図していることをより明確に表現する手段になります。

intptr_t および uintptr_t 型は、アドレス演算でポインタの型変換を行うときに大変役立ちます。この目的には、long や符号なし long ではなく、intptr_t と uintptr_t 型を使用してください。

定数マクロ

定数のサイズと符号の指定には、INT8_C(c) ... INT64_C(c)、UINT8_C(c) ... UINT64_C(c) マクロを使用してください。基本的に、これらのマクロは、必要に応じて定数の末尾に l、ul、ll、ull という文字列を追加します。たとえば、INT64_C(1) は、ILP32 では、定数 1 に ll、LP 64 では l を付加します。

定数を最大型にするときは、INTMAX_C(c) と UINTMAX_C(c) を使用してください。これらのマクロは、186 ページの「7.3 LP64 データ型モデルへの変換」で説明している定数型を指定する際に大変役立ちます。

制限

<inttypes.h> で定義されている上下制限は、いろいろな整数型の最小値と最大値を指示する定数です。これには、INT8_MIN ... INT64_MIN、INT8_MAX ... INT64_MAX などの固定幅型をそれぞれの符合なし型に対する最小値と最大値が含まれます。

<inttypes.h> ファイルには、最小サイズのそれぞれの型に対する最小値と最大値も含まれます。これには、INT_LEAST8_MIN ... INT_LEAST64_MIN、INT_LEAST8_MAX ... INT_LEAST64_MAX 型やこれらに対応する符号なし型があります。

また、<inttypes.h> には、サポートされる最大整数型の最小値と最大値も定義されています。これには、INTMAX_MIN、INTMAX_MAX、これらに対応する符号なし型があります。

書式文字列マクロ

<inttypes.h> ファイルには printf(3S) および scanf(3S) の書式指示子を指定するマクロも含まれています。基本的にこれらのマクロは、引数のビット数がマクロ名に組み込まれていることを条件に、書式指示子の前に l または ll を付加して、引数が long または long long のどちらであるかを示します。

次の例に示すように、最小および最大整数型を 10 進、8 進、符号なし、16 進の形式で表示する、printf(3S) 用のマクロがあります。

```
int64_t i;
printf("i =%" PRIx64 "\n", i);
```

同様に、最小および最大整数型を 10 進、8 進、符号なし、16 進の形式で読み取る、scanf(3S) 用のマクロがあります。

```
uint64_t u;
scanf("%" SCNu64 "\n", &u);
```

これらのマクロはむやみに使用しないでください。183 ページの「固定幅の整数型」で説明しているように、固定幅型に対して使用するのがもっとも適しています。

7.2.2 ツール

lint プログラムの -errchk オプションは、64 ビットへの移植でエラーになる可能性のある問題を検出します。cc -v を指定して、-v を付けずにコンパイルするよりも、より厳密な意味検査を行うようコンパイラに指示することもできます。-v オプションは、指定されたファイルに対して lint に似た検査もいくつか行います。

64 ビット環境で安全なコードにするには、Solaris オペレーティングシステムに含まれているヘッダーファイルを使用してください。このヘッダーファイルには、64 ビットコンパイル環境用の派生型とデータ構造体の正しい定義が含まれています。

7.2.2.1 lint

32 ビットおよび 64 ビットの両方のコンパイル環境用に作成したコードの検査には、lint を使用してください。LP64 の警告を生成するには、-errchk=longptr64 オプションを使用します。また、ロング整数とポインタのサイズが 64 ビットで普通の整数のサイズが 32 ビットの環境への移植性を検査する場合も -errchk=longptr64 フラグを使用してください。-errchk=longptr64 フラグは、明示的な型変換が使用されているときにも、ポインタ式とロング整数式の普通の整数への代入を検査します。

符号なし整数型の式における符号付き整数値の符号拡張を ISO C の通常の値保持規則が認めるコードを検索するには、-errchk=longptr64,signext オプションを使用してください。

Solaris の 64 ビットコンパイル環境でだけ実行するコードを検査する場合は、lint の -m64 オプションを使用してください。

警告する場合、lint は問題のコードの行番号とその問題の内容や、ポインタが関連するかどうかを示すメッセージを表示します。また、関係するデータ型のサイズも示します。ポインタが関係していること、データ型のサイズがわかれば、64 ビットの問題を特定し、32 ビットとそれより小さい型の間に以前から存在している問題を避けることできます。

ただし、64 ビット環境でエラーになる可能性のある問題について警告を出すといっても、lint によってすべての問題が検出できるわけではありません。多くの場合、意図したとおりであり、正しいコードであっても、警告は出されます。

行の前に “NOTE(LINTED(“<optional message”>))” の形式のコメントを挿入すると、特定の行に対する警告を抑止できます。この機能は、lint に型変換や代入などの行を無視させる場合に役立ちます。ただし、現実には存在する問題が隠される可能性があるため、“NOTE(LINTED(“<optional message”>))” コメントを使用するときは、細心の注意を払ってください。NOTE を使用する場合、#include<note.h> のようにして note.h をインクルードしてください。詳細は、lint のマニュアルページを参照してください。

7.3 LP64 データ型モデルへの変換

この節では実際の例を使用して、コードを変換したときに発生する可能性のある一般的な問題をいくつか紹介します。対応する lint の警告がある場合は、その警告も示します。

7.3.1 整数とポインタのサイズの変更

ILP32 コンパイル環境では整数とポインタは同じサイズであるため、コードには、この前提に立って作成されているものがあります。アドレス演算では、ポインタはしばしば int または unsigned int に型変換されます。LP64 コンパイル環境への変換で

は、ポインタは long に型変換してください。これは、ILP32 と LP64 データ型モデルで、long とポインタが同じサイズであるためです。明示的に unsigned long を使用するのではなく、uintptr_t を使用してください。uintptr_t の方が目的の用途により近く、コードの移植性を高めるため、将来的に変更しなくてもよいようにします。次の例を考えてみましょう。

```
char *p;
p = (char *) ((int)p & PAGEOFFSET);
%
warning: conversion of pointer loses bits
```

修正版は次のようになります。

```
char *p;
p = (char *) ((uintptr_t)p & PAGEOFFSET);
```

7.3.2 整数とロング整数のサイズの変更

ILP32 データ型モデルでは実際には整数とロング整数が区別されないため、ほとんどの場合、既存のコードでは区別なしに整数とロング整数が使用されています。整数とロング整数が区別なしに使用されているコードは、修正して ILP32 と LP64 両方のデータ型モデルの条件に準拠するようにしてください。ILP32 データ型モデルでは整数とロング整数はともに 32 ビットですが、LP64 データ型モデルではロング整数は 64 ビットです。

次の例を考えてみましょう。

```
int waiting;
long w_io;
long w_swap;
...
waiting = w_io + w_swap;

%
warning: assignment of 64-bit integer to 32-bit integer
```

LP64 データ型モデルでは、int や unsigned int 型配列に比べて、long や unsigned long 型などのの大きな配列が原因で、重大なパフォーマンスの低下を招くことがあります。long や unsigned long 型の大きな配列はまた、キャッシュミスの大幅な増加や使用メモリーの増加の原因になることもあります。

このため、アプリケーションの用途上 long 型と同程度に int 型で問題がないならば、long 型ではなく int 型を使用する方が安全です。

ポインタ型配列の代わりに int 型配列を使用すべきという意見もあります。一部 C 言語で開発されたアプリケーションは、LP64 データ型モデルに変換すると、深刻なパフォーマンスの低下を招きます。これは、そうした C 言語で開発されたアプリケーションが多数の大きなポインタ型配列に依存しているためです。

7.3.3 符号拡張

型の変換と拡張規則はいくぶん曖昧ですから、64 ビットコンパイル環境への移行で、符号拡張はよく問題になります。符号拡張の問題を避けるには、明示的な型変換を使用して、意図した結果を得られるようにしてください。

符号拡張が発生する理由を理解するには、ISO C の変換規則の知識が役立ちます。32 ビットと 64 ビットコンパイル環境間で最大の符号拡張問題を引き起こすと思われる変換規則は、次の処理で適用されます。

- 整数の拡張

  整数を必要とする式では、符号の有無に関係なく、`char`、`short`、`enumerated type`、ビットフィールドを使用することができます。

  整数が元の型が取り得る値をすべて保持できる場合、値は整数に変換され、それ以外の場合は、符号なし整数に変換されます。
- 符号付きと符号なし整数間の変換

  負符号付きの整数を同じまたは大きい型の符号なし整数に拡張する場合は、最初に大きな型符号付き整数に拡張され、次に符号なし値に変換されます。

次のコードを 64 ビットプログラムとしてコンパイルすると、`addr` と `a.base` の両方が符号なしの型であっても、`addr` 変数は符号拡張されます。

```
%cat test.c
struct foo {
unsigned int base:19, rehash:13;
};

main(int argc, char *argv[])
{
  struct foo a;
  unsigned long addr;

  a.base = 0x40000;
  addr = a.base << 13;  /* Sign extension here! */
  printf("addr 0x%lx\n", addr);

 addr = (unsigned int)(a.base << 13); /* No sign extension here! */
 printf("addr 0x%lx\n", addr);
}
```

ここで符号拡張が起きるのは、次のように変換規則が適用されるためです。

- `a.base` は、整数拡張規則により符号なし `int` から `int` に変換されます。つまり、式の `a.base << 13` は `int` 型ですが、符号拡張はまだ発生していません。
- 式の `a.base << 13` は `int` 型ですが、符号付きと符号なし整数拡張規則により、`addr` に代入する前に `long`、次に符号なし `long` へと変換されます。符号拡張は、`int` から `long` に変換したときに発生します。

```
% cc -o test64 -xarch=v9 test.c
% ./test64
addr 0xffffffff80000000
addr 0x80000000
%
```

同じ例を 32 ビットプログラムとしてコンパイルすると、符号拡張はまったく表示されません。

```
cc -o test test.c
%test

addr 0x80000000
addr 0x80000000
```

変換規則の詳細については、ANSI/ISO C 規格の仕様書を参照してください。この規格には通常の演算変換や整数定数に関する有用な規則も規定されています。

7.3.4 整数の代わりのポインタ演算

ポインタ演算が常にデータ型モデルから独立しているのに対し、整数は独立していないことがあるため、一般的には整数を使用するより、ポインタ演算を使用する方がよいでしょう。また、通常、ポインタ演算を使用することによって、コードを簡単にすることもできます。次の例を考えてみましょう。

```
int *end;
int *p;
p = malloc(4 * NUM_ELEMENTS);
end = (int *)((unsigned int)p + 4 * NUM_ELEMENTS);

%
warning: conversion of pointer loses bits
```

修正版は次のようになります。

```
int *end;
int *p;
p = malloc(sizeof (*p) * NUM_ELEMENTS);
end = p + NUM_ELEMENTS;
```

7.3.5 構造体

アプリケーションの内部データ構造体に穴がないか検査してください。境界整列条件を満たすには、構造体のフィールドとフィールドの間にパディングをします。このパディングは、ロング整数またはポインタフィールドが LP64 データ型モデル用に 64 ビットになったときに適用します。SPARC プラットフォームの 64 ビットコンパイル環境では、あらゆる種類の構造体が、その中の最大量のサイズに合わせて整列されます。構造体を整列し直すときは、ロング整数およびポインタフィールドを構造体の先頭に移動するという簡単な規則に従ってください。次の例を考えてみましょう。

```
struct bar {
   int i;
   long j;
   int k;
   char *p;
};   /* sizeof (struct bar) = 32 */
```

次は、同じ構造体の例です。ロング整数およびポインタデータ型を構造体の先頭で定義しています。

```
struct bar {
  char *p;
  long j;
  int i;
  int k;
};   /* sizeof (struct bar) = 24 */
```

7.3.6 共用体

ILP32 と LP64 データ型モデルの間では、共用体のフィールドのサイズが変わる可能性があるため、共用体は必ず検査してください。

```
typedef union {
   double _d;
   long _l[2];
} llx_t;
```

修正版は次のようになります。

```
typedef union {
   double _d;
   int _l[2];
} llx_t;
```

7.3.7 型定数

精度が足りないと、一部の定数式でデータが失われることがあります。定数式でデータ型を指定するときは明示的に行なってください。u、U、l、L のいくつかを組み合わせて、すべての整定数の型を指定してください。型変換を使用して、定数式の型を指定することもできます。次の例を考えてみましょう。

```
int i = 32;
long j = 1 << i; /* j will get 0 because RHS is integer */
                                /* expression */
```

修正版は次のようになります。

```
int i = 32;
long j = 1L << i;
```

7.3.8 暗黙の宣言に対する注意

`-xc99=none` を使用する場合、C コンパイラは、モジュールで使用されていて、外部定義または宣言されてない関数や変数をすべて整数とみなします。このようにして使用されるロング整数やポインタは、コンパイラの暗黙の整数宣言によって切り捨てられます。この問題を避けるには、C モジュールではなく、ヘッダーに関数または変数に対する適切な extern 宣言を挿入してください。そして、その関数または変数を使用する C モジュールにヘッダーをインクルードしてください。システムヘッダーによって定義されている関数あるいは変数であっても、コードに正しいヘッダーをインクルードする必要があります。次の例を考えてみましょう。

```
int
main(int argc, char *argv[])
{
  char *name = getlogin();
  printf("login = %s\n", name);
  return (0);
}

%
warning: improper pointer/integer combination: op "="
warning: cast to pointer from 32-bit integer
implicitly declared to return int
getlogin        printf
```

次の修正版には正しいヘッダーが含まれています。

```
#include <unistd.h>
#include <stdio.h>

int
main(int argc, char *argv[])
{
  char *name = getlogin();
  (void) printf("login = %s\n", name);
  return (0);
}
```

7.3.9 `sizeof( )` は符号なし `long`

LP64 データ型モデルでは、`sizeof()` の有効な型は unsigned long です。ときには `sizeof()` は、`int` 型の引数を待つ関数に渡されたり、整数に代入あるいは型変換されたりします。そうした場合は、切り捨てによってデータが失われることがあります。

```
long a[50];
unsigned char size = sizeof (a);

%
warning: 64-bit constant truncated to 8 bits by assignment
warning: initializer does not fit or is out of range: 0x190
```

7.3.10 型変換で意図を明確にする

変換規則により、関係式は扱いにくいことがあります。必要に応じて型変換を追加することによって、式の評価方法を明示するようにしてください。

7.3.11 書式文字列の変換操作を検査する

printf(3S)、sprintf(3S)、scanf(3S)、sscanf(3S) に対する書式文字列が long あるいは pointer 引数を受け付けられるようになっていることを確認してください。pointer 引数については、書式文字列中の変換操作を %p で指定して、32 ビットおよび 64 ビット両方のコンパイル環境で機能するようにします。

```
char *buf;
struct dev_info *devi;
...
(void) sprintf(buf, "di%x", (void *)devi);

%
warning: function argument (number) type inconsistent with format
sprintf (arg 3)     void *: (format) int
```

修正版は次のようになります。

```
char *buf;
struct dev_info *devi;
...
(void) sprintf(buf, ”di%p", (void *)devi);
```

long 引数については、書式文字列中の変換操作文字の前に long サイズ指定の l を付加します。また、buf の指し示す記憶場所が 16 桁を保持できる大きさであるか確認してください。

```
size_t nbytes;
u_long align, addr, raddr, alloc;
printf("kalloca:%d%%%d from heap got%x.%x returns%x\n",
nbytes, align, (int)raddr, (int)(raddr + alloc), (int)addr);

%
warning: cast of 64-bit integer to 32-bit integer
warning: cast of 64-bit integer to 32-bit integer
warning: cast of 64-bit integer to 32-bit integer
```

修正版は次のようになります。

```
size_t nbytes;
u_long align, addr, raddr, alloc;
printf("kalloca:%lu%%%lu from heap got%lx.%lx returns%lx\n",
nbytes, align, raddr, raddr + alloc, addr);
```

7.4 そのほかの注意事項

この節では、アプリケーションを完全な 64 ビットプログラムに変換するときに発生する問題を取り上げます。

7.4.1 サイズが大きくなった派生型

いくつかの派生型が変更されており、64 ビットコンパイル環境で 64 ビット量を表すようになっています。32 ビットアプリケーションがこの変更の影響を受けることはありませんが、これらの型で表されるデータを消費またはエクスポートする 64 ビットアプリケーションは、評価し直す必要があります。たとえば utmp(4) あるいは utmpx(4) ファイルを直接操作するアプリケーションがこれにあたります。64 ビットアプリケーション環境で正しく動作させるには、utmp または utmpx ファイルに直接にアクセスしないようにしてください。代わりに、getutxent(3C) および関連する系列の関数を使用します。

7.4.2 変更の副作用の検査

ある場所で型を変更したために、別のコード部分で予想外の 64 ビット変換が発生することがあります。たとえば、それまで int を返していて、現在は ssize_t を返すようになった関数のすべての呼び出し元を検査してください。

7.4.3 long のリテラル使用の合理性の確認

long と定義された変数は、ILP32 データ型モデルでは 32 ビット、LP64 データ型モデルでは 64 ビットです。可能な場合は、こうした変数を定義し直し、移植性に優れた派生型を使用することによって問題の発生を回避してください。

これに関連して、LP64 データ型モデルでは、いくつかの派生型が変更されています。たとえば、pid_t は 32 ビット環境では long のままですが、64 ビット環境では int になります。

7.4.4 明示的な 32 ビットと 64 ビットプロトタイプに対する #ifdef の使用

場合によっては、32 ビットや 64 ビット専用のインタフェースを使用しなければならないことがあります。そうしたインタフェースには、ヘッダー中で _LP64 または _ILP32 の機能テストマクロを指定して区別できます。同様に、32 ビットまた 64 ビット環境で動作するコードでは、コンパイルモードに従って適切な #ifdefs を使用する必要があります。

7.4.5 呼び出し規則の変更

構造体を値によって渡し、64 ビット環境用にコードをコンパイルした場合、その構造体は、コピーへのポインタとしてではなく、レジスタ中で渡されます (構造体がそのようにできるほどの大きさの場合)。その場合、C コードと手書きのアセンブリコード間で構造体を渡そうとすると、問題が起きることがあります。

浮動小数点パラメータも同様に機能します。値で渡される浮動小数点値は浮動小数点レジスタ中で渡されます。

7.4.6 アルゴリズムの変更

64 ビット環境で安全なコードを作成したら、コードを見直して、アルゴリズムとデータ構造体が正しく機能することを確認してください。データ構造体のデータ型が大きいほど、使用する空間が増えることがあります。コードのパフォーマンスも影響を受けるかもしれません。こうしたことに注意し、必要に応じてコードを修正してください。

7.5 変換前の確認事項

コードを 64 ビットに変換するにあたっては次の事項を確認してください。

- すべてのデータ構造体とインタフェースを見直して、64 ビット環境でも問題がないことを確認します。
- コードに `<inttypes.h>` をインクルードして、多数の基本派生方とともに `_ILP32` または `_LP64` の定義を取り込みます。システムプログラムは `_ILP32` または `_LP64` の定義を取得するために `<sys/types.h>` (または少なくとも `<sys/isa_defs.h>`) をインクルードする可能性があります。
- スコープが局所ではない関数プロトタイプと外部宣言はヘッダーに移動し、コード中にヘッダーをインクルードします。
- lint は、`-m64` を使用し、`-errchk=longptr64` および `signext` オプションを付けて実行します。1 つ 1 つすべての警告に目を通してください。必ずしもすべての警告について、コードの変更が必要になるわけではありません。変更によっては、32 ビットと 64 ビットモードの両方で lint を再度実行してください。
- アプリケーションの 64 ビット版だけ提供するのでないかぎり、32 ビットと 64 ビットの両方でコードをコンパイルしてください。
- アプリケーションのテストは、32 ビット版は 32 ビットオペレーティングシステム上で、64 ビット版は 64 ビットオペレーティングシステム上で行なってください。32 ビット版は、64 ビットオペレーティングシステム上でテストすることもできます。

◆ ◆ ◆ 第 8 章

8 cscope: 対話的なCプログラムの検査

cscope は、プログラムC、lex、または yacc ソースファイル内のコードの特定の要素を探し出す対話型プログラムです。cscope ブラウザを使用すると、従来のエディタよりも効率的にソースファイルを検索、編集できます。これは、cscope が関数呼び出し (関数がいつ呼び出され、いつその関数を実行するか) についてと、C 言語の識別子とキーワードを理解しているためです。

本章は cscope ブラウザについて説明します。この章は、このリリースに付属している cscope ブラウザの使い方を学ぶための資料として利用できます。

注 – cscope プログラムは、まだ 1999 ISO/IEC C 規格用に作成されたコードを認識できるように更新されていません。たとえば、1999 ISO/IEC C 規格で導入された新しいキーワードを認識しません。

8.1 cscope プロセス

cscope は、C、lex、yacc のソースファイルを読み取り、ファイル内の関数、関数呼び出し、マクロ、変数、前処理シンボルのシンボル相互参照表を作成します。次に作成した表を検索して、ユーザーが指定したシンボルの位置を探し出します。cscope は、最初にメニューを表示し、実行したい検索のタイプを聞いてきます。たとえば、cscope で特定の関数を呼び出しているすべての関数を検索することがあります。

検索が終了すると、cscope は結果を表示します。リストの各エントリ行には、cscope によって指定したコードが見つかったファイル名、行番号、その行のテキストが含まれます。この例では、指定された関数を呼び出している関数名も表示されます。リストを表示したあとは、新しく検索するか、あるいはリストに表示された行をエディタで調べるかを選択することができます。後者の場合、cscope はその行があるファイルをエディタで読み込んで、その行にカーソルを移動します。こ

こで、その行の前後関係を調べることができます。さらにほかのファイルと同じように編集することもできます。エディタを終了したら、メニューに戻って新しい検索を始めます。

作業内容によって手順も変わってくるので、cscope の使用方法は 1 通りではありません。cscope の詳しい使用方法や、コード全体を調べることなくプログラム内のバグを探し出す方法については、次の節で説明します。。

8.2 基本的な使用方法

たとえば、プログラム prog の開始直後に 「out of storage」というエラーメッセージが表示されることがあると想定します。これを解決するには、まず cscope を使用してコード内のメッセージを発行している場所を探し出さなければいけません。この場合、次の手順で実行します。

8.2.1 ステップ 1: 環境設定

cscope は、画面指向ツールです。使用できる端末は、端末情報ユーティリティー (terminfo) データベースに書かれているものに限られます。TERM 環境変数を自分の端末タイプに設定してあることを確認してください。 cscope は TERM 環境変数の値を見て、それが terminfo データベースに存在するか確認します。まだ設定していない場合は、次のようにして TERM に値を設定し、それをシェルに伝えます。

B シェルの場合は次のように入力します。

```
$ TERM=term_name; export TERM
```

C シェルの場合は次のように入力します。

```
% setenv TERM term_name
```

次に、EDITOR 環境変数に値を設定します。 デフォルトでは、cscope は vi エディタを起動します (本章の例も vi を使用して説明しています)。vi を使用しない場合は、EDITOR 環境変数を任意のエディタ名に変更して、EDITOR をエクスポートします。

B シェルの場合は次のように入力します。

```
$ EDITOR=emacs; export EDITOR
```

C シェルの場合は次のように入力します。

```
% setenv EDITOR emacs
```

cscope とエディタ間のインタフェースを設定しなければいけません。詳細は、210 ページの「8.2.9 エディタのコマンド行構文」を参照してください。

cscope を表示するためだけに使用する (編集は使用しない) 場合は、VIEWER 環境編集を pg に設定して VIEWER をエクスポートします。cscope は vi の代わりに pg を起動します。

環境変数 VPATH には、ソースファイルの検索対象ディレクトリを指定します。206 ページの「8.2.6 ビューパス (Viewpath)」を参照してください。。

8.2.2 ステップ 2: cscope プログラムの起動

デフォルトでは、cscope は現ディレクトリ内にあるすべての C、lex、および yacc のソースファイルのシンボル相互参照表、および現ディレクトリまたは標準位置内にあるすべてのインクルードヘッダーファイルのシンボル相互参照表を作成します。したがって、表示するプログラムのすべてのソースファイルが現ディレクトリにあり、かつそのヘッダーファイルが現ディレクトリまたは標準位置にある場合は、cscope を引数なしで起動します。

```
% cscope
```

特定のソースファイルを表示する場合は、そのファイルの名前を引数にして cscope を起動します。

```
% cscope file1.c file2.c file3.h
```

cscope のほかの起動方法については、204 ページの「8.2.5 コマンド行オプション」を参照してください。

プログラムを表示するため、最初に cscope が使用されるときにシンボル相互参照表が作成されます。デフォルトでは、作成されたシンボル相互参照表は現ディレクトリ内の cscope.out ファイルに格納されます。そのあと cscope を再び起動すると、前回と比較してソースファイルが修正されていたとき、またはソースファイルのリストが異なるときだけ相互参照表が作成し直されます。相互参照表を再び作成する時には、変更されていないファイルのデータは前回の相互参照表からコピーされます。これによって、最初の作成時より作成速度が速くなり、起動時のスタートアップ時間も短くなります。。

8.2.3 ステップ 3: コード位置の確定

本節の最初で述べた本来の作業に戻り、「out of storage」のエラーメッセージの原因となっている場所を確定します。cscope が起動され、相互参照表が作成されました。画面には、cscope の作業メニューが表示されます。

cscope の作業メニュー

```
% cscope

cscope     Press the ? key for help
```

```
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

Return キーを押すと、カーソルが下に移動し (画面の一番下まで移動すると、先頭に戻る)、^p (Ctrl+p キー) を押すと、上に移動します。また、上矢印 (ua) と下矢印 (da) キーも使用できます。次の単一キーコマンドを使用すれば、メニュー操作とそのほかの作業が行えます。

表 8–1 cscope メニュー操作コマンド

Tab	次の入力フィールドへ移動する
Return	次の入力フィールドへ移動する
^n	次の入力フィールドへ移動する
^p	前の入力フィールドへ移動する
^y	最後に入力したテキストを検索する
^b	逆方向にパターンを検索する
^f	順方向にパターンを検索する
^c	検索時に大文字と小文字を区別するか否かのトグルスイッチ (大文字と小文字を区別しない場合、たとえば FILE 文字列は file と File の両方と一致)。
^r	相互参照表を再作成する
!	対話型シェルを起動する (**^d** で cscope に復帰)
^l	画面を描き直す
?	コマンドのリストを表示する
^d	cscope を終了する

検索文字列の最初の文字が前述のいずれかのコマンドと一致する場合は、検索文字列の前にバックスラッシュ (\) を加えてコマンドと区別します。

たとえば、カーソルを 5 番目のメニュー項目「Find this text string」に移動して文字列「out of storage」を入力し、Return キーを押します。

cscope 関数: 文字列検索の要求

```
$ cscope

cscope     Press the ? key for help


Find this C symbol
Find this global definition
Find functions called by this function
Find functions calling this function
Find this text string:  out of storage
Change this text string
Find this egrep pattern
Find this file
Find files #including this file
```

注 – 6 番目の「Change this text string」項目以外のメニュー項目についても同じ手順に従ってください。6 番目の項目はほかの項目よりも多少複雑なので手順が異なります。文字列の変更方法については、207 ページの「8.2.8 例」を参照してください。

cscope は指定された文字列を検索し、それを含む行を見つけ出して次のように検索結果を表示します。

cscope 関数: 文字列を含む cscope 行のリスト表示

```
Text string: out of storage

  File Line
1 alloc.c 63 (void) fprintf(stderr, "\n%s:  out of storage\n", argv0);


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

cscope によって検索結果が正常に表示されたら、次の操作を選択します。行を変更したり、またはその行の前後をエディタで調べることができます。あるいは、cscope の検索結果のリストが一画面に収まらない場合は、リストの次の部分を見ることもできます。cscope が指定した文字列を検索したあとに使用可能なコマンドを次に示します。

表 8–2 最初の検索後に使用するコマンド

1 -9	この行を含むファイルを編集する (入力した番号は cscope が表示したリストの行番号に対応する)
スペース	次画面のリストを表示する

表 8–2 最初の検索後に使用するコマンド (続き)

+	次画面のリストを表示する
^v	次画面のリストを表示する
–	前画面のリストを表示する
^e	表示されたファイル順に編集する
>	表示されているリストをファイルへ追加する
|	全行をパイプでシェルコマンドに渡す

ここでも、検索文字列の最初の文字が前述のいずれかのコマンドと一致する場合は、検索文字列の前にバックスラッシュ (\) を加えてコマンドと区別します。

次に、新しく検索した行の前後を調べます。「**1**」 (リスト内の行番号) を入力してください。エディタが起動され、`alloc.c` ファイルが読み込まれます。カーソルは、`alloc.c` の 63 行目の先頭に移動します。

cscope 関数: コード行の検査

```
{
   return(alloctest(realloc(p, (unsigned) size)));
}

/* check for memory allocation failure */

static char *
alloctest(p)
char *p;
{
    if (p == NULL) {
        (void) fprintf(stderr, "\n%s:  out of storage\n", argv0);
        exit(1);
    }
    return(p);
}
~
~
~
~
~
~
~
"alloc.c" 67 lines, 1283 characters
```

変数 p が NULL のときに、エラーメッセージが出力されることがわかります。`alloctest()` に渡される引数がなぜ NULL になったのかを調べるには、まず `alloctest()` を呼び出している関数を確定する必要があります。

通常の終了方法でエディタを終了し、作業メニューに戻ります。ここで、4 番目の項目「`Find functions calling this function`」のあとに **`alloctest`** と入力します。

cscope 関数: alloctest() を呼び出す関数のリストの要求

```
Text string: out of storage

  File Line
1 alloc.c 63(void)fprintf(stderr,"\n%s:  out of storage\n",argv0);


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:  alloctest
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

cscope は検索を実行し、次の 3 つの関数のリストを表示します。

cscope 関数: alloctest() を呼び出す Listing 関数

```
Functions calling this function: alloctest
File Function Line
1 alloc.c mymalloc 33 return(alloctest(malloc((unsigned) size)));
2 alloc.c mycalloc 43 return(alloctest(calloc((unsigned) nelem, (unsigned) size)));
3 alloc.c myrealloc 53 return(alloctest(realloc(p, (unsigned) size)));


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

今度は、mymalloc() を呼び出す関数を調べます。cscope は、次のような 10 個の関数を見つけ出します。そのうち 9 個を画面に表示し、残りの 1 個を見るにはスペースバーを押すように指示しています。

cscope 関数: mymalloc() を呼び出す Listing 関数

```
Functions calling this function: mymalloc

File         Function       Line
1 alloc.c    stralloc       24 return(strcpy(mymalloc
                            (strlen(s) + 1), s));
2 crossref.c crossref       47 symbol = (struct symbol *)mymalloc
                            (msymbols * sizeof(struct symbol));
3 dir.c      makevpsrcdirs  63 srcdirs = (char **) mymalloc
                            (nsrcdirs * sizeof(char*));
4 dir.c      addincdir      167 incdirs = (char **)mymalloc
```

```
                            (sizeof(char *));
5 dir.c      addincdir      168 incnames = (char **)
                            mymalloc(sizeof(char *));
6 dir.c      addsrcfile     439 p = (struct listitem *) mymalloc
                            (sizeof(struct listitem));
7 display.c  dispinit       87 displine = (int *) mymalloc
                            (mdisprefs * sizeof(int));
8 history.c  addcmd         19  h = (struct cmd *) mymalloc
                            (sizeof(struct cmd));
9 main.c     main           212 s = mymalloc((unsigned )
                            (strlen(reffile) +strlen(home) + 2));

* 9 more lines - press the space bar to display more *
Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

「out of storage」のエラーメッセージはプログラムの開始直後に出力されるので、関数 `dispinit()` (表示の初期化) 内で問題が発生していることが推測できます。

`dispinit()` はリストの 7 番目の関数なので、これを参照するには「**7**」と入力します。

cscope 関数: `dispinit()` をエディタで参照

```
void
dispinit()
{
        /* calculate the maximum displayed reference lines */
    lastdispline = FLDLINE -  4;
    mdisprefs = lastdispline -  REFLINE + 1;
    if (mdisprefs > 9) {
       mdisprefs = 9;
    }
       /* allocate the displayed line array */
   displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
{
    char file[PATHLEN + 1]; /* file name */
    char function[PATLEN + 1];/* function name */
    char linenum[NUMLEN + 1]; /* line number */
    int screenline; /* screen line number */
    int width; /* source line display width */
    register int i, j;
"display.c" 622 lines, 14326 characters
```

mymalloc() が失敗したのは、非常に大きな数または負数を引数にして呼び出されたためです。FLDLINE と REFLINE が取り得る値を調べてみると、mdisprefs の値が負になる場合があることがわかります。この場合、mymalloc() は負数を引数にして呼び出されます。

8.2.4 ステップ 4: コードの編集

ウィンドウ形式の端末では、任意のサイズのウィンドウを複数使用できます。「out of storage」のエラーメッセージが出力されたのは、prog を実行するウィンドウ内の行数が少なすぎたためと考えられます。つまり、mymalloc() が負数を引数にして呼び出された場合にこのような状況が発生する可能性があるということです。今後このような状況が発生した場合に、もっとわかりやすいエラーメッセージ、たとえば「Screen too small」を出力してプログラムを中止するように設定しておくとよいでしょう。それには dispinit() 関数を次のように編集します。

cscope 関数: 問題箇所の修正

```
void
dispinit()
{
/* calculate the maximum displayed reference lines */
    lastdispline = FLDLINE - 4;
    mdisprefs = lastdispline - REFLINE + 1;
    if (mdisprefs > 9) {
       mdisprefs = 9;
    }
/* allocate the displayed line array */
    displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* display a page of the references */

void
display()
{
    char file[PATHLEN + 1]; /* file name */
    char function[PATLEN + 1];/* function name */
    char linenum[NUMLEN + 1]; /* line number */
    int screenline; /* screen line number */
    int width; /* source line display width */
    register int i, j;
"display.c" 622 lines, 14326 characters
```

以上で、本節の最初で調査を開始した問題箇所は修正されました。これで、行数が少なすぎるウィンドウ内で prog を実行したときに、単に意味不明のエラーメッセージ「out of storage」を出力して中止するのではなく、ウィンドウサイズを検査してわかりやすいエラーメッセージを出力したあとに終了するようになります。

8.2.5 コマンド行オプション

すでに述べたとおり、cscope はデフォルトでは現ディレクトリ内にある C、lex、および yacc ソースファイルのシンボル相互参照表を作成します。次に例を示します。

```
% cscope
```

これは次のように指定するのと同じことです。

```
% cscope *.[chly]
```

指定したソースファイルを表示するには、ソースファイル名を引数に指定して cscope を起動します。

```
% cscope file1.c file2.c file3.h
```

cscope のコマンド行オプションを使用して、相互参照表に含まれるソースファイルをさらに自由に指定することもできます。それには、次のように -s オプションのあとにコンマで区切られた任意の数のディレクトリ名を指定して cscope を起動します。

```
% cscope– s dir1,dir2,dir3
```

cscope は現ディレクトリ内だけでなく、指定されたディレクトリ内にあるすべてのソースファイルを対象に相互参照表を作成します。ファイル中にリストされているソースファイル (ファイル名をスペースやタブまたは復帰改行で区切ったもの) のすべてを表示するには、– i オプションとリストを持つファイル名を指定して cscope を起動します。

```
% cscope– i file
```

ソースファイルがディレクトリツリーの中にある場合は、次のコマンドでディレクトリツリー内のすべてのソースファイルを簡単に表示できます。

```
% find .–  name '*.[chly]'– print | sort > file
% cscope– i file
```

このオプションを使用しても、コマンド行でファイルが指定されている場合、cscope によって指定されたファイル以外については無視されます。

-I オプションは、cc に対する -I オプションと同じような形式で cscope にも指定できます。62 ページの「2.16 インクルードファイルを指定する方法」を参照してください。

-f オプションを使用すると、デフォルトの cscope.out 以外のファイルを相互参照ファイルとして指定できます。このオプションは、同じディレクトリ内に異なるシンボル相互参照ファイルを保管するのに役立ちます。たとえば、2 つのプログラムが同じディレクトリ内にあるが、すべてのファイルを共有しているとは限らない場合に使用します。

```
% cscope– f admin.ref admin.c common.c aux.c libs.c
% cscope– f delta.ref delta.c common.c aux.c libs.c
```

この例では、2 つのプログラム admin と delta のソースファイルは同じディレクトリ内にありますが、プログラムを構成するファイルは異なっています。cscope 起動時に、別のシンボル相互参照ファイルを指定しておくことによって、2 つのプログラムの相互参照情報を別々に保管できます。

– p*n* オプションを使用すると、cscope は検索結果でリストされたファイルのあるパス名やそのパス名の一部を表示することができます。– p のあとの *n* には、パス名の中で最後から何番目までの要素を表示させたいかを指定します。デフォルト値は 1 で、これはファイル名そのものを意味します。したがって現ディレクトリが home/common の場合、次のコマンドによって

```
% cscope– p2
```

cscope によって検索結果のリストに表示されるパス名が、common/file1.c や common/file2.c のように表示されます。

表示するプログラムが大量のソースファイルを含む場合、– b オプションを使用して、相互参照表を作成したあとで cscope を終了することができます。このとき、cscope は作業メニューを表示しません。パイプラインで、cscope– b を batch(1) コマンドとともに使用する場合、cscope は相互参照表をバックグラウンドで作成します。

```
% echo ’cscope -b’ | batch
```

相互参照表がいったん作成されると、その後、ソースファイルまたはソースファイルのリストを変更しないかぎり、次のように指定するだけで

```
% cscope
```

相互参照表がコピーされ、通常どおり作業メニューが表示されます。このコマンドシーケンスを使用すると cscope の初期処理の終了を待たずに作業を続けることができます。

– d オプションは、cscope にシンボル相互参照表を更新させません。このオプションを指定すると、cscope はソースファイルの変更を検査しないため時間の節約になります。変更されていないと確信できる場合にのみ使用してください。

注 – -d オプションの使用には注意が必要です。ソースファイルが変更されていることに気付かずに – d オプションを使用すると、cscope は古いシンボル相互参照表を使用して照会に応じてしまいます。

ほかのコマンド行オプションについては、cscope(1) のマニュアルページを参照してください。。

8.2.6 ビューパス (Viewpath)

前述のように cscope は、デフォルトでは現ディレクトリ内のソースファイルを検索します。環境変数 VPATH が設定されているときは、cscope は VPATH に指定されたディレクトリ内のソースファイルを検索します。ビューパスとは、順序付けされたディレクトリのリストで、リスト内の各ディレクトリの下は同じディレクトリ構造になっています。

たとえば、ユーザーがあるソフトウェアプロジェクトのメンバーであるとします。/fs1/ofc 下のディレクトリには、正式バージョンのソースファイルがあります。メンバーはホームディレクトリ (/usr/you) を持っており、ソフトウェアシステムを変更する場合は、変更するファイルだけを /usr/you/src/cmd/prog1 にコピーします。全プログラムの正式バージョンは、/fs1/ofc/src/cmd/prog1 にあります。

cscope を使用して、prog1 を構成する 3 つのファイル (f1.c、f2.c、f3.c) を表示します。まず VPATH を /usr/you と /fs1/ofc に設定してエクスポートします。

B シェルの場合は次のように入力します。

```
$ VPATH=/usr/you:/fs1/ofc; export VPATH
```

C シェルの場合は次のように入力します。

```
% setenv VPATH /usr/you:/fs1/ofc
```

次に、現ディレクトリを /usr/you/src/cmd/prog1 に移動して cscope を起動します。

```
% cscope
```

cscope はビューパスにあるすべてのファイルの位置を調べます。同じファイルが複数のディレクトリにある場合は、cscope はVPATH 内で先に現れたディレクトリの下にあるファイルを使用します。したがって、f2.c がユーザーのディレクトリにあり (3 つのファイルはすべて正式バージョン用ディレクトリの下にもある場合)、cscope は f2.c はユーザーディレクトリのものを、f1.c および f3.c は正式バージョン用のディレクトリのものを検査します。

VPATH 内の最初のディレクトリは、作業用ディレクトリの接頭辞 (通常は $HOME) でなければいけません。VPATH 内のコロンで区切られたそれぞれのディレクトリは、/ から始まる絶対パス名でなければいけません。。

8.2.7 cscope とエディタ呼び出しのスタック

cscope とエディタの呼び出しはスタックできます。たとえば、cscope がエディタを起動してシンボルへの参照を調べているときに、ほかにも参照関係を調べたいシンボルがある場合、エディタ内部から再び cscope を起動して 2 番目の参照関係を調べ

ることができます。現在起動中の cscope やエディタを終了する必要はありません。一番最後に起動した cscope またはエディタコマンドを正常に終了すれば、1 つ前の状態に戻ることができます。

8.2.8 例

cscope が次の 3 つの作業を行うのにどのように使用されるかを見ていきます。対象とする作業は、定数をプリプロセッサシンボルに変更する、関数に引数を追加する、変数の値を変更するの 3 つです。最初の例では、文字列の変更手順を示します。この作業は、cscope メニューのほかの作業項目とは少し異なっています。変更したい文字列を入力すると、cscope はそれを置き換える新しい文字列を聞いてきます。画面には古い文字列を含む行が表示されます。ここで、どの行に含まれる文字列を変更するかを指定します。

8.2.8.1 例 1: 定数をプリプロセッサシンボルに変更する

たとえば、定数 100 をプリプロセッサシンボル MAXSIZE に変更するとします。6 番目のメニュー項目「Change this text string」を選択して **\100** と入力します。1 の前にはバックスラッシュを加えて、cscope のメニュー項目番号を意味する 1 と区別します。Return キーを押すと cscope は新しい文字列を聞いてくるので、**MAXSIZE** と入力します。

cscope 関数: 文字列の変更

```
cscope            Press the ? key for help


Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string: \100
Find this egrep pattern:
Find this file:
Find files #including this file:
To:  MAXSIZE
```

cscope は、指定された文字列を含む行を表示します。どの行の文字列を変更するかが選択されるまで入力待ちになります。

cscope 関数: 変更行に対するプロンプト

```
cscope            Press the ? key for help


Find this C symbol:
Find this global definition:
Find functions called by this function:
```

```
Find functions calling this function:
Find this text string:
Change this text string: \100
Find this egrep pattern:
Find this file:
Find files #including this file:
To:  MAXSIZE
```

リストの 1、2、3 行目 (ソースファイル内の行番号はそれぞれ 4、26、8 行目) に含まれる定数 `100` は、`MAXSIZE` に変更すべきだとわかります。さらに、`read.c` の `0100` (4 行目) と `err.c` の `100.0` (5 行目) は変更すべきではないこともわかります。次の単一キーコマンドを使用して、変更したい行を選択します。

表 8–3 変更する行を選択するコマンド

1-9	変更対象の行をマークしたり、マークを削除する
*	すべての表示行を変更対象としてマークしたり、マークを削除する
スペース	次画面のリストを表示する
+	次画面のリストを表示する
–	前画面のリストを表示する
a	すべての行を変更対象としてマークする
^d	マークされた行を変更して終了する
Esc	マークされた行を変更しないで終了する

この場合、**1**、**2**、および **3** を入力します。入力した番号は画面上には表示されません。代わりに `cscope` は各行の行番号のあとに `>` (右不等号) を表示することによって、変更箇所を示します。

`cscope` 関数: 変更行のマーキング

```
Change "100" to "MAXSIZE"

  File Line
1>init.c 4 char s[100];
2>init.c 26 for (i = 0; i < 100; i++)
3>find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0; /* get percentage */

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
```

```
Find files #including this file:
Select lines to change (press the ? key for help):
```

ここで、**^d** を入力して選択行を変更します。cscope は変更後の各行を表示し、作業の継続を促します。

cscope 関数: 変更後のテキスト行表示

```
Changed lines:

    char s[MAXSIZE];
    for (i = 0; i < MAXSIZE; i++)
    if (c < MAXSIZE) {

Press the RETURN key to continue:
```

このプロンプトに対して Return キーを押すと、cscope は画面を書き換えて変更行を指定する前の画面に戻ります。

次に新しいシンボル MAXSIZE の #define 文を追加します。#define 文を追加するヘッダーファイルは、現在表示されている行の参照元ファイルの中にはありません。したがって、**!** と入力してシェルに入る必要があります。シェルプロンプトが画面の一番下に現れます。あとは、エディタを起動して #define 文を追加します。

cscope 関数: シェルへの一時移行

```
Text string: 100

  File Line
1 init.c 4 char s[100];
2 init.c 26 for (i = 0; i < 100; i++)
3 find.c 8 if (c < 100) {
4 read.c 12 f = (bb & 0100);
5 err.c 19 p = total/100.0;                                    /* get percentage */

Find this C symbol:
Find this global definition:
Find functions called by this function:
Find functions calling this function:
Find this text string:
Change this text string:
Find this egrep pattern:
Find this file:
Find files #including this file:
$ vi defs.h
```

cscope セッションへ戻るには、エディタを終了し、**^d** を入力してシェルを終了させます。

8.2.8.2 例 2: 関数に引数を追加する

関数に引数を追加するには、関数そのものを編集することとその関数が呼び出されているすべての箇所に新しい引数を追加することの 2 つのステップがあります。cscope を使用して簡単にこのステップを実行できます。

まず、2 番目のメニュー項目「Find this global definition」を使用して、関数を編集します。次に、その関数がどこで呼び出されているかを探します。4 番目のメニュー項目「Find functions calling this function」を使用すると、ある関数を呼び出しているすべての関数のリストを表示することができます。このリストを使用して、リストの各行番号を個々に入力してエディタを起動するか、または **^e** を入力して、各行のすべての参照元ファイルを対象にエディタを自動的に起動することができます。このような修正処理に cscope を使用すると、修正を必要とする関数はすべて修正され、見落とすことがありません。

8.2.8.3 例 3: 変数の値を変更する

変更内容がコードにどのように影響するかを見たいときに、表示手段として cscope が力を発揮します。

変数の値またはプリプロセッサシンボルを変更する場合を考えてみます。実際に変更する前に、最初のメニュー項目「Find this C symbol」を使用して、変更によって影響を受ける参照箇所のリストを表示します。それから、エディタを起動して各参照箇所を調べます。これによって、変更によるすべての影響を予測できます。同様に cscope を使用して、間違いなく変更されたことも確認できます。

8.2.9 エディタのコマンド行構文

デフォルトでは、cscope は vi エディタを起動します。EDITOR 環境変数に任意のエディタ名を設定して EDITOR をエクスポートすると、デフォルトを変更することができます。この手順については、196 ページの「8.2.1 ステップ 1: 環境設定」で述べたとおりです。ただし、cscope は、使用するエディタのコマンド行構文が次の形式であるとみなします。

```
% editor +linenum filename
```

これは vi と同じです。使用したいエディタがこのようなコマンド行構文を持っていない場合は、cscope とエディタ間のインタフェースを定義する必要があります。

ed を使用する場合を考えてみます。ed では、コマンド行内に行番号を指定することができないので、そのままでは cscope のエディタとして使用できません。そこで、次のような行を含むシェルスクリプトを作成します。

```
/usr/bin/ed $2
```

ここでは、シェルスクリプトを myedit とします。環境変数 EDITOR の値をこのシェルスクリプトに設定して EDITOR をエクスポートします。

B シェルの場合は次のように入力します。

```
$ EDITOR=myedit; export EDITOR
```

C シェルの場合は次のように入力します。

```
% setenv EDITOR myedit
```

cscope は、指定されたリスト項目 (たとえば、main.c の 17 行目) を読み込んでエディタを起動するとき、次のようなコマンド行を使用してシェルスクリプトを起動します。

```
% myedit +17 main.c
```

myedit は第一引数の行番号 ($1) を無視して、第二引数のファイル名 ($2) だけを使用して ed を正しく呼び出します。希望する行を表示および編集するには、適切な ed コマンドを実行する必要があります。すなわち、17 行目に自動的に移動することはありません。

8.3 不明な端末タイプのエラー

次のエラーメッセージが出力されることがあります。

```
Sorry, I don't know how to deal with your "term" terminal
```

このメッセージは、現在ロードされている端末情報ユーティリティー (terminfo) データベース内に使用端末が含まれていないことを意味します。TERM に正しい値が設定されていることを確認してください。それでもメッセージが出力される場合は、端末情報ユーティリティーを再ロードしてください。

次のようなメッセージも表示されることがあります。

```
Sorry, I need to know a more specific terminal type than "unknown"
```

196 ページの「8.2.1 ステップ 1: 環境設定」で説明した手順に従って、TERM を設定してエクスポートしてください。

◆ ◆ ◆ 付 録 A

機能別コンパイラオプション

この付録では、機能別に C コンパイラのオプションをまとめています。各オプションおよびコンパイラコマンド行構文の詳細は、表 A–15 を参照してください。

A.1 機能別に見たオプションの要約

この節には、参照しやすいように、コンパイラオプションが機能別に分類されています。各オプションの詳細は、表 A–15 および付録 B 「C コンパイラオプションリファレンス」 を参照してください。フラグによっては、複数の使用目的があるため、複数の個所に記載されているものがあります。

これらのオプションは、特に記載がないかぎりすべてのプラットフォームに適用されます。Solaris SPARC システム版のオペレーティングシステムに特有の機能は (SPARC) として表記され、x64 システム版のオペレーティングシステムに特有の機能は *(x86)* として表記されます。Solaris プラットフォームのみに適用されるオプションには、(Solaris) というマークが付きます。Linux プラットフォームのみを対象とするオプションには、(Linux) というマークが付きます。

A.1.1 最適化とパフォーマンスのオプション

表 A–1 最適化とパフォーマンスのオプション表

オプション	処理
`-fast`	実行可能コードの速度を向上させるコンパイルオプションの組み合わせを選択します。
`-fma`	(SPARC) 浮動小数点の積和演算 (FMA) 命令の自動生成を有 効にします。
`-p`	プロファイルデータ収集用のオブジェクトコードを用意します。

表 A–1 最適化とパフォーマンスのオプション表 (続き)

オプション	処理
-xalias_level	コンパイラが型ベースの別名の分析と最適化を実行します。
-xannotate	(Solaris) binopt(1) のようなバイナリ変更ツールを使用してあとで変換できるバイナリを作成するようコンパイラに指示します。
-xbinopt	あとで最適化、変換、分析を行うために、バイナリを準備します。
-xbuiltin	標準ライブラリ関数を呼び出すコードの最適化率を上げます。
-xdepend	ループの繰り返し内部でのデータ依存性の解析およびループ再構成を実行します。
-xF	リンカーによるデータと関数の順序の並べ替えを有効にします。
-xhwcprof	(SPARC) コンパイラのハードウェアカウンタによるプロファイリングのサポートを有効にします。
-xinline	指定された関数だけをインライン化します。
-xinstrument	スレッドアナライザで分析するために、プログラムをコンパイルして計測します。
-xipo	内部手続き解析パスを呼び出すことにより、プログラム全体の最適化を実行します。
-xipo_archive	ファイル間の最適化にアーカイブ (.a) ライブラリを含める
-xjobs	コンパイラが生成するプロセスの数を設定します。
-xlibmil	実行速度を上げるため、一部のライブラリルーチンをインライン化します。
-xlic_lib=sunperf	Sun のパフォーマンスライブラリにリンクします。
-xlinkopt	再配置可能なオブジェクトファイルのリンク時の最適化を実行します。
-xlibmopt	最適化された数学ルーチンのライブラリを使用します。
-xmaxopt	このコマンドは、pragma opt のレベルを指定されたレベルに限定します。
-xnolibmil	数学ライブラリのルーチンをインライン化しません。
-xnolibmopt	最適化された数学ルーチンのライブラリを使用しません。
-x0	オブジェクトコードを最適化します。
-xnorunpath	実行可能ファイル内の共有ライブラリの実行時検索パスのインクルードを抑止します。
-xpagesize	スタックとヒープ用の優先ページサイズを設定します。

表 A–1 最適化とパフォーマンスのオプション表 (続き)

オプション	処理
-xpagesize_stack	スタック用の優先ページサイズを設定します。
-xpagesize_heap	ヒープ用の優先ページサイズを設定します。
-xpch	ソースファイルが共通インクルードファイルセットを共有しているようなアプリケーションのコンパイル時間を短縮します。
-xpec	自動チューニングシステム (Automatic Tuning System、ATS) と組み合わせて使用できる、移植可能な実行可能コード (Portable Executable Code、PEC) バイナリを生成します。詳細は、http://cooltools.sunsource.net を参照してください。
-xpchstop	-xpch とともに使用して、活性文字列の最後のインクルードファイルを指定します。
-xpentium	*(x86)* Pentium プロセッサ用に最適化を行います。
-xprefetch	先読み命令を有効にします。
-xprefetch_level	-xprefetch=auto で設定される先読み命令の自動挿入の優先度を制御します。
-xprefetch_auto_type	間接先読み命令の生成方法を制御します。
-xprofile	プロファイルのデータを収集、または最適化のためにプロファイルを使用します。
-xprofile_ircache	-xprofile=collect 段階で保存されたコンパイルデータを再利用して、-xprofile=use 段階のコンパイル時間を向上します。
-xprofile_pathmap	単一のプロファイルディレクトリで複数のプログラムまたは共有ライブラリをサポートします。
-xrestrict	ポインタ値の関数引数を制限付きポインタとして扱います。
-xsafe	(SPARC) メモリーに関するトラップが発生しないことを前提にします。
-xspace	コードサイズを増やすループの最適化や並列化を行いません。
-xunroll	ループを *n* 回展開するようオプティマイザに指示します。

A.1.2 コンパイル時とリンク時のオプション

次の表は、リンク時とコンパイル時の両方に指定する必要があるオプションをまとめています。

表 A–2 コンパイル時とリンク時のオプション表

オプション	処理
-fast	実行可能コードの速度を向上させるコンパイルオプションの組み合わせを選択します。
-m32|-m64	コンパイルされたバイナリオブジェクトのメモリーモデルを指定します。
-mt	-D_REENTRANT -lthread に展開されるマクロオプションです。
-p	prof(1) がプロファイルデータを収集するためのオブジェクトコードを作成します。
-xarch	命令セットアーキテクチャーを指定します。
-xautopar	複数プロセッサ用の自動並列化を有効にします。
-xhwcprof	(SPARC) コンパイラのハードウェアカウンタによるプロファイリングのサポートを有効にします。
-xipo	内部手続き解析パスを呼び出すことにより、プログラム全体の最適化を実行します。
-xlinkopt	再配置可能なオブジェクトファイルのリンク時の最適化を実行します。
-xmemalign	(SPARC) 想定するメモリー境界整列の最大値と、境界整列に失敗したデータがアクセスされた際の動作を指定します。
-xopenmp	明示的な並列化のための OpenMP インタフェースをサポートします。このインタフェースには、ソースコード指令セット、実行時ライブラリルーチン、環境変数などが含まれます。
-xpagesize	スタックとヒープ用の優先ページサイズを設定します。
-xpagesize_stack	スタック用の優先ページサイズを設定します。
-xpagesize_heap	ヒープ用の優先ページサイズを設定します。
-xpg	gprof(1) によるプロファイルの準備として、データを収集するためのオブジェクトコードを生成します。
-xprofile	プロファイルのデータを収集、または最適化のためにプロファイルを使用します。
-xvector=lib	ベクトルライブラリ関数を自動的に呼び出すようにします。

A.1.3 データ境界整列のオプション

表 A–3 データ境界整列のオプション表

オプション	処理
-xchar_byte_order	複数文字からなる定数である文字を指定されたバイト順序で配置することにより、整定数を生成します。
-xdepend	ループの繰り返し内部でのデータ依存性の解析およびループ再構成を実行します。
-xmemalign	(SPARC) 想定するメモリー境界整列の最大値と、境界整列に失敗したデータがアクセスされた際の動作を指定します。
-xopenmp	明示的な並列化のための OpenMP インタフェースをサポートします。このインタフェースには、ソースコード指令セット、実行時ライブラリルーチン、環境変数などが含まれます。

A.1.4 数値と浮動小数点のオプション

表 A–4 数値と浮動小数点のオプション表

オプション	処理
-flteval	*(x86)* 浮動小数点の評価を制御します。
-fma	(SPARC) 浮動小数点の積和演算 (FMA) 命令の自動生成を有 効にします。
-fnonstd	浮動小数点演算ハードウェアの非標準の初期化を行います。
-fns	非標準の浮動小数点モードに切り替えます。
-fprecision	*(x86)* 浮動小数点制御ワードの丸め精度モードのビットを初期化します。
-fround	プログラム初期化中に、実行時に確立される IEEE 754 丸めモードを設定します。
-fsimple	オプティマイザが浮動小数点演算に関する前提事項を単純化できるようにします。
-fsingle	float 式を倍精度ではなく単精度で評価します。
-fstore	*(x86)* 浮動小数点式または関数の値を、代入式の左辺値の型に変換します。
-ftrap	起動時に有効になる IEEE 754 トラップモードを設定します。
-nofstore	*(x86)* 浮動小数点式または関数の値を、代入式の左辺値の型に変換しません。

表 A–4 数値と浮動小数点のオプション表 (続き)

オプション	処理
-xdepend	ループの繰り返し内部でのデータ依存性の解析およびループ再構成を実行します。
-xlibmieee	例外が起きた場合の数学ルーチンの戻り値を強制的に IEEE 754 形式にします。
-xsfpconst	接尾辞のない浮動小数点定数を単精度で表します。
-xvector	ベクトルライブラリ関数を自動的に呼び出すようにします。

A.1.5 並列化のオプション

表 A–5 並列化のオプション表

オプション	処理
-mt	-D_REENTRANT -lthread に展開されるマクロオプションです。
-xautopar	複数プロセッサ用の自動並列化を有効にします。
-xcheck	スタックオーバーフローに関する実行時検査を追加し、ローカル変数を初期化します。
-xdepend	ループの繰り返し内部でのデータ依存性の解析およびループ再構成を実行します。
-xloopinfo	並列化されているループとされていないループを示します。
-xopenmp	明示的な並列化のための OpenMP インタフェースをサポートします。このインタフェースには、ソースコード指令セット、実行時ライブラリルーチン、環境変数などが含まれます。
-xreduction	自動並列化中の縮約の認識を有効にします。
-xrestrict	ポインタ値の関数パラメータを制限付きのポインタとして扱います。
-xvpara	#pragma MP 指令が指定されているが、正しく並列化指定されていないループについて警告を出します。
-xthreadvar	スレッドローカルな変数の実装を制御します。
-Zll	lock_lint 用にプログラムデータベースだけ作成して、実行可能なコードの生成は行いません。

A.1.6 ソースコードのオプション

表 A–6 ソースコードのオプション表

オプション	処理
-A	#assert 前処理指令に似せて、指定の *tokens* を使用し *name* を述語として関連付けます。
-C	C プリプロセッサがコメントを削除しないようにします。ただし前処理指令の行にあるコメントは削除されます。
-D	#define 前処理指令に似せて、指定の *tokens* を使用し *name* を関連付けます。
-E	プリプロセッサのみでソースファイルを処理し、出力を stdout に送ります。
-fd	K&R 形式の関数の宣言や定義を報告します。
-H	現在のコンパイルでインクルードされたファイルのパス名を 1 行に 1 つずつ標準エラーに表示します。
-I	ディレクトリをリストに追加します。このリストは相対ファイル名で指定される #include ファイルを検索するときのリストです。
-include	コンパイラは引数 *filename* を、主要なソースファイルの 1 行目に記述されているかのように #include プリプロセッサ指令として処理します。
-P	ソースファイルのプリプロセッサ処理のみを行います。
-U	初期定義されているプリプロセッサシンボル *name;* をすべて削除します。
-X	ANSI/ISO C に準拠する度合いを指定します。
-xCC	C++ 形式のコメントを受け入れます。
-xc99	サポートされる C99 機能をコンパイラで認識させるかどうかを制御します。
-xchar	文字が符号なしと定義されるシステムからの移行を支援します。
-xcsi	C コンパイラが、ISO C ソース文字コードの要件に準拠していないロケールで記述されたソースコードを受け付けることを可能にします。
-xM	指定した C プログラムに対してプリプロセッサだけを実行します。その際、メイクファイルの依存関係を生成してその結果を標準出力に出力します。
-xM1	-xM と同様に依存関係を収集しますが、/usr/include ファイルは除きます。

表 A–6 ソースコードのオプション表　(続き)

オプション	処理
-xMD	-xM と同様にメイクファイルの依存関係を生成しますが、コンパイルを含みます。
-xMF	メイクファイルの依存関係情報を保存するファイル名を指定します。
-xMMD	メイクファイルの依存関係を生成しますが、システムヘッダーを除外します。
-xP	このモジュールで定義されたすべての K&R C 関数に対するプロトタイプを出力します。
-xpg	*gprof*(1) によるプロファイルの準備として、データを収集するためのオブジェクトコードを生成します。
-xtrigraphs	3 文字表記シーケンスの認識状況を判定します。
-xustr	16 ビット文字から成る文字列リテラルの認識を有効にします。

A.1.7 コンパイル済みコードのオプション

表 A–7 コンパイル済みコードのオプション表

オプション	処理
-c	*ld*(1) によるリンクを行わず、現在の作業用ディレクトリ内にソースファイルごとに .o ファイルを作成します。
-o	出力ファイルに名前を付けます。
-S	アセンブリソースファイルを作成しますが、アセンブルは行いません。

A.1.8 コンパイルモードのオプション

表 A–8 コンパイルモードのオプション表

オプション	処理
-#	冗長モードを有効にします。コマンドオプションの展開内容と呼び出されたすべての構成要素が表示されます。
-###	呼び出された各構成要素が表示されますが、実行はされません。また、コマンドオプションの展開内容も表示されます。
-features	C 言語の各種機能を有効または無効にします。
-keeptmp	コンパイル中に作成される一時ファイルを自動的に削除しないで保持します。

表 A–8 コンパイルモードのオプション表 *(続き)*

オプション	処理
-V	cc コンパイラの実行時に各構成要素の名前とバージョン番号を表示します。
-W	引数を C コンパイルシステムの構成要素に渡します。
-X	ANSI/ISO C に準拠する度合いを指定します。
-xc99	サポートされる C99 機能をコンパイラで認識させるかどうかを制御します。
-xchar	char の符号を保持します。
-xhelp	オンラインヘルプ情報を表示します。
-xjobs	コンパイラが生成するプロセスの数を設定します。
-xpch	ソースファイルが共通インクルードファイルセットを共有しているようなアプリケーションのコンパイル時間を短縮します。
-xpchstop	-xpch とともに使用して、活性文字列の最後のインクルードファイルを指定します。
-xtemp	cc が使用する一時ファイルの *dir* を設定します。
-xtime	コンパイルの各構成要素が占有した実行時間と資源を報告します。
-Y	コンパイルシステムの構成要素を配置する新しいディレクトリを指定します。
-YA	コンパイラの構成要素を検索するデフォルトのディレクトリを変更します。
-YI	インクルードファイル検索時のデフォルトディレクトリを変更します。
-YP	ライブラリファイルを検索するデフォルトのディレクトリを変更します。
-YS	起動用のオブジェクトファイルを検索するデフォルトのディレクトリを変更します。

A.1.9 診断のオプション

表 A–9 診断のオプション表

オプション	処理
-errfmt	警告メッセージからの型用に「error:」という接頭辞をメッセージに付けます。
-errhdr	ヘッダーファイルから指定したグループへの警告を制限します。

表 A–9 診断のオプション表 (続き)

オプション	処理
-erroff	コンパイラからの警告メッセージを出力しません。
-errshort	コンパイラが型の不一致を検出をする際に出力されるエラーメッセージの詳細度を制御します。
-errtags	各警告メッセージのメッセージタグを表示します。
-errwarn	指定された警告メッセージが表示される場合、cc はエラーステータスを返して終了します。
-v	より厳しい意味検査およびほかの lint に似た検査を行います。
-w	コンパイラからの警告メッセージを出力しません。
-xe	ソースファイル上で構文および意味検査のみを行います。オブジェクトコードや実行可能コードは生成しません。
-xtransition	K&R C と Solaris Studio ISO C との間の相違に対して警告を出します。
-xvpara	#pragma MP 指令が指定されているが、正しく並列化指定されていないループについて警告を出します。

A.1.10 デバッグオプション

表 A–10 デバッグのオプション表

オプション	処理
-xcheck	スタックオーバーフローに関する実行時検査を追加し、ローカル変数を初期化します。
-g	デバッガ用に追加のシンボルテーブル情報を作成します。
-s	出力されるオブジェクトファイルからシンボリックデバッグのための情報をすべて削除します。
-xdebugformat	stab 形式ではなく dwarf 形式でデバッグ情報を生成します。
-xpagesize	スタックとヒープ用の優先ページサイズを設定します。
-xpagesize_stack	スタック用の優先ページサイズを設定します。
-xpagesize_heap	ヒープ用の優先ページサイズを設定します。
-xs	dbx のためのオブジェクトファイルの自動読み取りを無効にします。
-xvis	(SPARC) コンパイラによる VIS 命令セットに定義されているアセンブリ言語のテンプレートの認識を有効にします。

A.1.11 リンクとライブラリのオプション

表 A–11 リンクとライブラリのオプション表

オプション	処理
-B	ライブラリのリンクを静的と dynamic のどちらにするかを指定します。
-d	リンクエディタに動的なリンクまたは静的なリンクを指定します。
-G	動的にリンクされる実行可能プログラムではなく、共有オブジェクトを作成することをリンクエディタに指示します。
-h	共有動的ライブラリに name; を付けます。これによってバージョンの異なるライブラリを作成できます。
-i	LD_LIBRARY_PATH の設定を無視するオプションをリンカーへ渡します。
-L	リンカーがライブラリを検索するリストに dir を付け加えます。
-l	オブジェクトライブラリ lib*name*.so、または lib*name*.a をリンクの対象とします。
-mc	オブジェクトファイルの .comment セクションから重複している文字列を削除します。
-mr	.comment セクションからすべての文字列を削除し、オブジェクトファイルの *string* を挿入します。
-Q	出力ファイルに識別情報を入れるかどうかを設定します。
-R	コロンで区切られたディレクトリのリストを、ライブラリ検索ディレクトリとして、実行時リンカーに渡します。
-xMerge	データセグメントをテキストセグメントにマージします。
-xcode	コードアドレス空間を指定します。
-xldscope	変数定義と関数定義のデフォルトのスコープを制御して、より高速で安全な共有ライブラリを作成します。
-xnolib	デフォルトのライブラリをリンクしません。
-xnolibmil	数学ライブラリのルーチンをインライン化しません。
-xstrconst	このオプションは、将来のリリースでは推奨されません。代わりに、-features=[no%]conststrings を使用します。 デフォルトのデータセグメントではなくテキストセグメントの読み出し専用データセクションに、文字列リテラルを挿入します。

A.1.12 対象プラットフォームのオプション

表 A–12 対象プラットフォームのオプション表

オプション	処理
-m32|-m64	コンパイルされたバイナリオブジェクトのメモリーモデルを指定します。
-xarch	命令セットアーキテクチャーを指定します。
-xcache	オプティマイザ用のキャッシュ特性を定義します。
-xchip	オプティマイザ用の対象プロセッサを指定します。
-xregs	生成コード用のレジスタの使用法を指定します。
-xtarget	最適化と命令セットの対象となるシステムを指定します。

A.1.13 x86 固有のオプション

表 A–13 x86 固有のオプション表

オプション	処理
-flteval	浮動小数点の評価を制御します。
-fprecision	浮動小数点制御ワードの丸め精度モードのビットを初期化します。
-fstore	浮動小数点式または関数の値を、代入式の左辺値の型に変換します。
-nofstore	浮動小数点式または関数の値を、代入式の左辺値の型に変換しません。
-xmodel	64 ビットオブジェクトの形式を Solaris x86 プラットフォーム用に変更します。
-xpentium	Pentium プロセッサ用に最適化を行います。

A.1.14 ライセンスオプション

表 A–14 ライセンスのオプション表

オプション	処理
-xlicinfo	ライセンスシステムについての情報を返します。

A.1.15 廃止オプション

次の表に、非推奨になったオプションを示します。引き続きコンパイラはこれらのオプションを受け付けますが、将来は使用できなくなる可能性があります。できるだけ速やかに推奨代替オプションを使うようにしてください。

表 A–15 廃止のオプション表

オプション	処理
-dalign	代わりに -xmemalign=8s を使用してください。
-KPIC (SPARC)	代わりに -xcode=pic32 を使用してください。
-Kpic (SPARC)	代わりに -xcode=pic13 を使用してください。
-misalign	代わりに -xmemalign=1i を使用してください。
-misalign2	代わりに -xmemalign=2i を使用してください。
-x386	代わりに、-xchip=generic を使用してください。
-x486	代わりに、-xchip=generic を使用してください。
-xa	代わりに、-xprofile=tcov を使用してください。
-xarch=v7,v8,v8a	廃止。
-xcg	-xarch、-xchip、-xcache のデフォルト値を活かすために、代わりに -O を使用してください。
-xcrossfile	代わりに -xipo を使用します。
-xnativeconnect	廃止。これに代わるオプションはありません。
-xprefetch=yes	代わりに - xprefetch=auto,explicit を使用します。
-xprefetch=no	代わりに -xprefetch=no%auto,no%explicit を使用します。
-xsb	廃止。これに代わるオプションはありません。
-xsbfast	廃止。これに代わるオプションはありません。
-xtarget=386	代わりに -xtarget=generic を使用してください。
-xtarget=486	代わりに -xtarget=generic を使用してください。
-xvector=yes	代わりに、--xvector=lib を使用します。
-xvector=no	代わりに、-xvector=none を使用します。

◆ ◆ ◆ 付 録 B

C コンパイラオプションリファレンス

この章では、C コンパイラオプションについてアルファベット順に説明します。機能別のオプションは、付録 A「機能別コンパイラオプション」を参照してください。たとえば、表 A–1 には、最適化とパフォーマンスのすべてのオプションがまとめられています。

C コンパイラは、デフォルトでは 1999 ISO/IEC C 規格の構文の一部を認識します。特に、サポートされている機能については、付録 D「C99 でサポートされている機能」を参照してください。コンパイラで 1990 ISO/IEC C 規格の機能だけを使用する場合は、`-xc99=none` コマンドを使用します。

K&R (Kernighan と Ritchie) C プログラムを ISO C に移植する場合、互換性フラグに関する節、254 ページの「B.2.68 -X[c|a|t|s]」をよく参照してください。それらのオプションを使用することで、ISO C への移行が容易になります。また、136 ページの「5.4 メモリー参照の制限の例」の移行に関する説明も参照してください。

B.1 オプションの構文

`cc` コマンドの構文を次に示します。

```
% cc [options] filenames [libraries]...
```

ここで

- *options* は、表 A–15 で説明している各種のオプションで、複数指定可能です。
- *filename;* は、実行可能プログラムの作成に使用するファイル名で、複数指定可能です。

  C コンパイラは *filename;* で指定されたファイルリストに含まれている C ソースファイルとオブジェクトファイルのリストを受け取ります。生成された実行可能コードは、`-o` オプションを使用した場合を除いて `a.out` に出力されます。`-o` オプションを使用した場合には、コードは `-o` オプションで指定したファイルに出力されます。

C コンパイラを使用すると、次のファイルのどのような組み合わせに対しても、コンパイルとリンクを行うことができます。

- 接尾辞 .c の C ソースファイル
- 接尾辞 .il のインラインテンプレートファイル (.c ファイルで指定される場合のみ)
- 接尾辞 .i の前処理済みソースファイル
- 接尾辞 .o のオブジェクトコードファイル
- 接尾辞 .s のアセンブラソースファイル

リンク後、C コンパイラは実行可能コードの形式になったリンク済みファイルを、a.out ファイルまたは -o オプションで指定したファイルに出力します。コンパイラが .i または .c の各入力ファイルに対応するオブジェクトコードを生成する場合は、現在の作業ディレクトリにオブジェクト (.o) ファイルを作成します。

ライブラリは複数の標準ライブラリやユーザー提供のライブラリです。ライブラリには関数、マクロ、そして定数の定義が含まれます。

オプションライブラリの検索に使用するデフォルトのディレクトリを変更する場合は、-Y*P*,*dir* を参照してください。*dir* には、複数のパスをコロンで区切って指定します。デフォルトのライブラリ検索順序を表示するには、**-###** または **-xdryrun** オプションを使用し、**ld** 起動時の **-Y** オプションを検査します。

cc は getopt を使用してコマンド行オプションの構文を解析します。オプションは単一文字、または後ろに引数を 1 つとる単一文字によって指定します。getopt(3c) を参照してください。

B.2 cc オプション

この節では、cc オプションについてアルファベット順に説明します。これらの説明は cc(1) のマニュアルページでも見ることができます。1 行に要約した説明が必要な場合は、cc -flags オプションを使用してください。

特定のプラットフォームに固有と表記されたオプションを別のプラットフォームで使用してもエラーは起きません。単に無視されます。

B.2.1 -#

冗長モードをオンに設定し、コマンドオプションの展開を表示します。要素が呼び出されるごとにその要素を表示します。

B.2.2 -###

呼び出された各構成要素が表示されますが、実行はされません。また、コマンドオプションの展開内容を表示します。

B.2.3 --A*name*[(*tokens*)]

#assert 前処理指令に似せて、指定の *tokens* を使用し *name* を述語として関連付けます。事前表明 (preassertion) は次のとおりです。

- system(unix)
- machine(sparc) (SPARC)
- machine(i386) *(x86)*
- cpu(sparc) (SPARC)
- cpu(i386) *(x86)*

-Xc モードの場合、これらの事前表明は有効になりません。

-A のあとに続く文字がハイフン (-) だけの場合は、事前定義のマクロ (__ から始まる以外のマクロ) および事前定義の表明はすべて失われます。

B.2.4 -B[static| dynamic]

リンク時に結合するライブラリを静的と動的のどちらにするかを指定します。static (静的) と指定するとライブラリが非共有ライブラリであることを示し、dynamic (動的) と指定すると共有ライブラリであることを示します。

-Bdynamic を指定すると、-l*x* オプションが指定されていれば、リンカーは lib*x*.so というファイルを探し、次に lib*x*.a というファイルを探します。

-Bstatic を指定すると、リンカーは lib*x*.a というファイルだけを探します。このオプションは、コマンド行中で何度も指定して、切り替えることができます。このオプションと引数は ld(1) に渡されます。

注 – Solaris の 64 ビットコンパイル環境では、多くのシステムライブラリ (libc など) は、動的ライブラリのみ使用することができます。このため、コマンド行の最後に -Bstatic を使用しないでください。

このオプションと引数はリンカーに渡されます。

B.2.5 -C

C プリプロセッサがコメントを削除しないようにします。ただし前処理指令の行にあるコメントは削除されます。

B.2.6 -c

C コンパイラ ld(1) によるリンクを行わず、ソースファイルごとに o ファイルを作成します。-o オプションを使用すると、1 つのオブジェクトファイルを明示的に指定することができます。コンパイラが .i または .c の各入力ファイルに対応するオブジェクトコードを生成する場合は、現在の作業ディレクトリにオブジェクト (.o) ファイルを作成します。リンクを行わないと、オブジェクトファイルの削除も行われません。

B.2.7 -D*name*[(*arg*[,*arg*])][=*expasion*]

#define 前処理マクロが指令によって定義されるのと同様に、オプションの引数を使用してマクロを定義します。=*expansion* が指定されていない場合は、コンパイラは 1 であると仮定します。

コンパイラの定義済みマクロのリストについては、**cc**(1) のマニュアルページを参照してください。

B.2.8 -d[y|n]

-dy はリンクエディタに動的なリンクを指定します (デフォルト)。

-dn はリンクエディタに静的なリンクを指定します。

このオプションとその引数は ld(1) に渡されます。

注 – このオプションを動的ライブラリと組み合わせて使用すると、重大なエラーが発生します。ほとんどのシステムライブラリは、動的ライブラリでのみ使用できます。

B.2.9 -dalign

(SPARC) 廃止。このオプションは使わないでください。代わりに -xmemalign=8s を使用してください。詳細は、293 ページの「B.2.116 -xmemalign=*ab*」 を参照してください。廃止オプションの全一覧は、225 ページの「A.1.15 廃止オプション」 を参照してください。x86 プラットフォームでは、このオプションはメッセージを表示せずに無視されます。

B.2.10 -E

プリプロセッサのみでソースファイルを処理し、出力を stdout に送ります。プリプロセッサはコンパイラ内部に直接組み込まれます。/usr/ccs/lib/cpp が直接呼び出される -Xs モードの場合は除きます。プリプロセッサの行番号付け情報も含みます。-P オプションも参照してください。

B.2.11 -errfmt[=[no%]error]

このオプションは、エラーメッセージの最初に「error:」という接頭辞を追加して、警告メッセージと区別しやすくする場合に使用します。接頭辞は、-errwarn によってエラーに変換された警告にも追加されます。

表 B–1 -errfmt のフラグ

フラグ	意味
error	すべてのエラーメッセージに接頭辞「error: 」を追加します。
no%error	エラーメッセージに接頭辞「error: 」を追加しません。

このオプションを指定しない場合は、-errfmt=no%errorに設定されます。-errfmt を値なしで指定した場合は、コンパイラでは -errfmt=error が指定されます。

B.2.12 -errhdr=h]

ヘッダーファイルからの警告を、次のフラグによって示されたヘッダーファイルグループに限定します。

表 B–2 –errhdr オプション

値	意味
%all	使用しているすべてのヘッダーファイルを検査します。
%none	ヘッダーファイルを検査しません。
%user	/usr/include とそのサブディレクトリにあるものを除き、すべてのユーザーヘッダーファイルを検査します。また、コンパイラによって提供されたすべてのヘッダーファイルを検査します。これはデフォルト値です。

B.2.13 -erroff[= *t*]

このコマンドは、C コンパイラの警告メッセージを無効にします。エラーメッセージには影響しません。このオプションは、-errwarn でゼロ以外の終了状態を発生させるように指定されているかどうかにかかわらず、すべての警告メッセージに適用されます。

t には、次の 1 つまたは複数の項目をコンマで区切って指定します。*tag*、no%*tag*、%all、%none。指定順序によって実行内容が異なります。たとえば、「%all,no%*tag*」と指定すると、*tag* 以外のすべての警告メッセージを抑制します。次の表は、-erroff の値を示しています。

表 B–3 -erroff のフラグ

フラグ	意味
tag	*tag* のリストに指定されているメッセージを抑制します。-errtags=yes オプションで、メッセージのタグを表示することができます。
no%*tag*	*tag* 以外のすべての警告メッセージを抑制します。
%all	すべての警告メッセージを抑制します。
%none	すべてのメッセージの抑制を解除します (デフォルト)。

デフォルトは -erroff=%none です。-erroff と指定すると、-erroff=%all を指定した場合と同じ結果が得られます。

-erroff オプションで無効にできるのは、C コンパイラのフロントエンドで -errtags オプションを指定したときにタグを表示する警告メッセージだけです。無効にするエラーメッセージをさらに詳細に設定することができます。47 ページの「2.11.6 error_messages」を参照してください。

B.2.14 -errshort[= *i*]

このオプションは、コンパイラで型の不一致が検出されたときに生成されるエラーメッセージの詳細さを設定する場合に使用します。大きな集合体が関係する型の不一致がコンパイラで検出される場合にこのオプションを使用すると特に便利です。

i には、次のいずれかを指定します。

表 B–4 -errshort のフラグ

フラグ	意味
short	エラーメッセージは、型の展開なしの簡易形式で出力されます。集合体のメンバー、関数の引数、戻り値の型は展開されません。
full	エラーメッセージは、完全な冗長形式で出力されます。不一致の型が完全に展開されます。
tags	エラーメッセージは、タグ名がある型の場合はそのタグ名付きで出力されます。タグ名がない場合は、型は展開された形式で出力されます。

-errshort を指定しない場合は、コンパイラでは -errshort=full が指定されます。-errshort を値なしで指定した場合は、コンパイラでは -errshort=tags が指定されます。

このオプションは累積されず、コマンド行で最後に指定した値が有効になります。

B.2.15 -errtags[= *a*]

C コンパイラのフロントエンドで出力される警告メッセージのうち、-erroff オプションで無効にできる、または -errwarn オプションで重大なエラーに変換できるメッセージのメッセージタグを表示します。C コンパイラのドライバおよび C のコンパイルシステムのほかのコンポーネントから出力されるメッセージにはエラータグが含まれないため、-erroff で無効にしたり、-errwarn で重大なエラーに変換したりすることはできません。

a には yes または no のいずれかを指定します。デフォルトは -errtags=no です。-errtags だけを指定すると、-errtags=yes を指定するのと同じことになります。

B.2.16 -errwarn[= *t*]

指定した警告メッセージが生成された場合に、重大なエラーを出力して C コンパイラを終了する場合は、-errwarn オプションを使用します。

t には、次の 1 つまたは複数の項目をコンマで区切って指定します。*tag*、no%*tag*、%all、%none。このとき、順序が重要になります。たとえば、%all,no%*tag* と指定すると、*tag* 以外のすべての警告メッセージが生成された場合に、重大なエラーを出力して cc を終了します。

C コンパイラで生成される警告メッセージは、コンパイラのエラーチェックの改善や機能追加に応じて、リリースごとに変更されます。-errwarn=%all を指定してエラーなしでコンパイルされるコードでも、コンパイラの次期リリースではエラーを出力してコンパイルされる可能性があります。

-errwarn オプションを使用して、障害状態で C コンパイラを終了するように指定できるのは、C コンパイラのフロントエンドで -errtags オプションを指定したときにタグを表示する警告メッセージだけです。

-errwarn の値を次の表に示します。

表 B–5 -errwarn のフラグ

フラグ	意味
tag	*tag* に指定されたメッセージが警告メッセージとして発行されると、cc は致命的エラーステータスを返して終了します。*tag* に指定されたメッセージが発行されない場合は無効です。
no%*tag*	*tag* に指定されたメッセージが警告メッセージとしてのみ発行された場合に、cc が致命的なエラーステータスを返して終了しないようにします。*tag* に指定されたメッセージが発行されない場合は無効です。このオプションは、*tag* または %all を使用して以前に指定したメッセージが警告メッセージとして発行されても cc が致命的エラーステータスで終了しないようにする場合に使用してください。
%all	警告メッセージが何か発行される場合にコンパイラが致命的なエラーステータスを返して終了します。%all に続いて no%*tag* を使用して、特定の警告メッセージを対象から除外することもできます。
%none	どの警告メッセージが発行されてもコンパイラが致命的なエラーステータスを返して終了することがないようにします。

デフォルトは -errwarn=%none です。-errwarn だけを指定した場合、-errwarn=%all を指定したことと同じになります。

B.2.17 -fast

このオプションは、実行ファイルの実行時のパフォーマンスのチューニングで効果的に使用することができるマクロです。-fast は、コンパイラのリリースによって変更される可能性があるマクロで、ターゲットのプラットフォーム固有のオプションに展開されます。-# オプションまたは -xdryrun を使用して -fast の展開を調べ、-fast の該当するオプションを使用して実行可能ファイルのチューニングを行なってください。

-fast の展開には、コンパイラが最適化済み数学ルーチンのライブラリを使用できるようにする -xlibmopt オプションが含まれます。詳細は、288 ページの「B.2.104 -xlibmopt」を参照してください。

-fast オプションは、errno の値に影響します。詳細は、58 ページの「2.13 errno の値の保持」を参照してください。

-fast を指定してコンパイルしたモジュールは、-fast を指定してリンクする必要があります。215 ページの「A.1.2 コンパイル時とリンク時のオプション」に、コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの全一覧をまとめています。

-fast オプションは、特にコンパイルするマシンとは異なるターゲットで実行するプログラムでは使用できません。そのような場合は、-fast のあとに適切な -xtarget オプションを指定します。次に例を示します。

```
cc -fast -xtarget=generic ...
```

SUID によって規定された例外処理に依存する C モジュールに対しては、-fast のあとに -xnolibmil を指定します。

```
% cc -fast -xnolibmil
```

-xlibmil を使用すると、例外発生時でも errno が設定されず、また、matherr(3m) が呼び出されません。

-fast オプションは、厳密な IEEE 754 規格準拠を必要とするプログラムには適していません。

次に、-fast により指定されるオプションをプラットフォームごとに示します。

表 B–6 -fast 展開時のフラグ

オプション	SPARC	x86
-fns	X	X
-fsimple=2	X	X
-fsingle	X	X
-nofstore	-	X
-xalias_level=basic	X	X
-xbuiltin=%all	X	X
-xlibmil	X	X
-xlibmopt	X	X
-xmemalign=8s	X	-
-xO5	X	X
-xprefetch=auto,explicit	X	-
-xregs=frameptr	-	X
-xtarget=native	X	X

注 – 一部の最適化では、プログラムの動作が特定の動作になることを想定しています。プログラムの動作がその想定に適合していない場合は、アプリケーションがクラッシュする、または誤った結果が生成されることがあります。プログラムが -fast を指定したコンパイルに適しているかどうかを特定するには、各オプションの説明を参照してください。

これらのオプションにより最適化を実行した場合は、プログラムの動作が ISO C および IEEE の規格での定義と異なることがあります。詳細については、各オプションの説明を参照してください。

-fast はコマンド行でマクロ展開のように動作します。したがって、最適化レベルとコード生成の内容を -fast のあとに指定したオプションで指定した場合は、-fast での指定は無視されます。「-fast -xO4」でコンパイルすることは「-xO2 -xO4」の組み合わせでコンパイルすることと同じで、後ろの指定が優先されます。

x86 では、–fast オプションに –xregs=frameptr が含まれます。特に C、Fortran、および C++ の混合ソースコードをコンパイルする場合は、その詳細について、このオプションの説明を参照してください。

このオプションは、IEEE 規格例外処理に依存するプログラムには使用しないでください。数値結果が異なったり、プログラムが途中で終了したり、予想外の SIGFPE シグナルが発生する可能性があります。

実行中のプラットフォームで -fast の実際の展開を表示するには、次のものを使用します。

```
% cc -fast -xdryrun
```

B.2.18 -fd

K&R 形式の関数の宣言や定義を報告します。

B.2.19 -features=[*v*]

次の表に、*v* の代わりに使用できる値を示します。

表 B–7 -features のフラグ

値	意味
[no%]conststrings	読み取り専用メモリー内で文字列リテラルの配置を有効または無効にします。デフォルトは -features=conststrings であり、文字列リテラルを読み取り専用データセクションに配置します。文字列リテラルのあるメモリー上の位置に書き込みを行おうとするプログラムを、このオプションを指定してコンパイルすると、セグメント例外が発生することに注意してください。
extensions	サイズがゼロの構造体または共用体の宣言、および有効な値を返す return 文を持つ void 関数を使用できます。
extinl	extern インライン関数を大域関数として生成します。これがデフォルトで、1999 C 規格に準拠しています。-features=no%extinl を指定して新しいコードをコンパイルすると、extern インライン関数は、C および C++ コンパイラの古いバージョンで提供されていたのと同じ処理を受けます。
no%extinl	extern インライン関数を静的関数として生成します。
%none	このオプションを無効にします。

古い C および C++ オブジェクト (このリリースより前の Solaris Studio コンパイラで作成されたオブジェクト) は、そのオブジェクトの動作変更なしに、新しい C および C++ オブジェクトとリンクできます。規格に適合した動作を実現するには、最新のコンパイラを使って古いコードをコンパイルする必要があります。

-features に値を指定しない場合は、-features=extinl に設定されます。

B.2.20 -flags

使用できる各コンパイラオプションの要約を出力します。

B.2.21 -flteval[={any|2}]

(x86) このオプションは、浮動小数点式の評価方法の制御に使用します。

表 B–8 -flteval のフラグ

フラグ	意味
2	浮動小数点式を long dboule 型で評価します。
any	式を構成している変数および定数の型の組み合わせに基づいて浮動小数点式を評価します。

-flteval が指定されない場合は、-flteval=any に設定されます。値を付けずに -flteval が指定された場合は、-flteval=2 に設定されます。

-flteval=2 を指定する場合、次のオプションは指定してはいけません。

- -fprecision
- -nofstore
- -xarch=amd64
- -xarch=sse2

356 ページの「D.1.1 浮動小数点評価における精度」も参照してください。

B.2.22 -fma[={none| fused}]

(SPARC) 浮動小数点の積和演算 (FMA) 命令の自動生成を有効にします。-fma=none を指定すると、これらの命令の生成を無効にします。-fma=fused を指定すると、コンパイラは浮動小数点の積和演算 (FMA) 命令を使用して、コードのパフォーマンスを改善する機会を検出しようとします。

デフォルトは -fma=none です。

コンパイラが積和演算 (FMA) 命令を生成するための最小要件は、-xarch=sparcfmaf と、最適化レベルが -xO2 以上であることです。積和演算 (FMA) 命令をサポートしていないプラットフォームでプログラムが実行されないようにするため、コンパイラは積和演算 (FMA) 命令を生成する場合、バイナリプログラムにマーク付けをします。

積和演算 (FMA) により、積と和 (乗算と加算) の間で中間の丸め手順が排除されます。その結果、**-fma=fused** を指定してコンパイルしたプログラムは、精度は減少ではなく増加する傾向にありますが、異なる結果になることがあります。

B.2.23 -fnonstd

(SPARC) このオプションは、-fns および -ftrap=common 用のマクロです。

B.2.24 -fns[={no|yes}]

- (SPARC) SPARC の非標準の浮動小数点モードに切り替えます。

  デフォルトは -fns=no であり、SPARC 標準の符号小数点モードと同じです。 -fns は、-fns=yes と同じことを意味します。

  オプションの =yes または =no を使用すると、-fast のように、-fns を含むほかのマクロフラグに続く -fns フラグを切り替えることができます。

一部の SPARC システムでは、非標準の浮動小数点モードは「段階的アンダーフロー」を無効にします。つまり、小さな結果は、非正規数にはならず、ゼロに切り捨てられます。また、非正規オペランドはメッセージなしにゼロに変更されます。このような SPARC システムでは、ハードウェアの段階的アンダーフローや非正規数がサポートされておらず、このオプションを使用するとプログラムのパフォーマンスを著しく改善することができます。

非標準モードを有効にすると、浮動小数点演算は IEEE 754 規格に準拠しない結果を生成する場合があります。詳細は、『数値計算ガイド』を参照してください。

このオプションは、SPARC システム上だけで有効であり、さらに、メインプログラムをコンパイルするときに使用する場合だけ有効です。x86 システムでは、このオプションは無視されます。

- (*x86*) SSE flush-to-zero モードを選択します。利用可能な場合には、denormals-are-zero モードが選択されます。

  このオプションは、非正規数の結果をゼロにフラッシュします。また利用可能な場合には、非正規数オペランドもゼロとして扱われます。

  このオプションは、SSE や SSE2 命令セットを利用しない従来の x86 浮動小数点演算には影響しません。

B.2.25 -fPIC

-KPIC と同義です。

B.2.26 -fpic

-Kpic と同義です。

B.2.27 -fprecision=*p*

(*x86*) `-fprecision={single, double, extended}`

浮動小数点制御ワードの丸め精度モードのビットを、単精度 (24 ビット)、倍精度 (53 ビット) または拡張精度 (64 ビット) に設定します。デフォルトの浮動小数点丸め精度モードは拡張モードです。

x86 では、浮動小数点丸め精度モードの設定は精度に対してのみ影響します。指数の有効範囲に対しては影響しません。

このオプションは、x86 システムでメインプログラムのコンパイル時に使用する場合にのみ有効で、64 ビット (`-m64`) または SSE2 対応 (`-xarch=sse2`) プロセッサでコンパイルする場合は無視されます。SPARC システムでも無視されます。

B.2.28 -fround=*r*

プログラム初期化中に、実行時に確立される IEEE 754 丸めモードを設定します。

r は、nearest、 tozero、negative、positive のいずれかです。

デフォルトは、-fround=nearest です。

ieee_flags サブルーチンと同等です。

r を tozero、negative、positive のいずれかにすると、プログラムが実行を開始するときに、丸め方向モードがそれぞれ、ゼロの方向に丸める、負の無限の方向に丸める、正の無限の方向に丸めるに設定されます。*r* が nearest のとき、あるいは -fround フラグを使用しないとき、丸め方向モードは初期値から変更されません (デフォルトは nearest)。

このオプションは、メインプログラムをコンパイルするときにだけ有効です。

B.2.29 -fsimple[= *n*]

オプティマイザが浮動小数点演算に関する前提事項を単純化できるようにします。

デフォルトは -fsimple=0 です。-fsimple は -fsimple=1 と同義です。

n を指定する場合、0、1、2 のいずれかにしなければいけません。

表 B–9 -fsimple のフラグ

値	意味
-fsimple=0	前提事項の単純化を行えないようにします。IEEE 754 に厳密に準拠します。

表 B–9 -fsimple のフラグ (続き)

値	意味
-fsimple=1	適度の単純化を行えるようにします。生成されるコードは IEEE 754 に厳密には準拠していませんが、大半のプログラムの数値結果は変わりありません。 -fsimple=1 の場合、次に示す内容を前提とした最適化が行われます。 ■ IEEE 754 のデフォルトの丸めとトラップモードが、プロセスの初期化以後も変わらない。 ■ 浮動小数点例外以外には、目に見える結果が生じない演算は削除できる。 ■ オペランドに無限または NaN を持つ演算は、その結果に NaN を伝達する必要はない。たとえば、x*0 は 0 で置き換えられる。 ■ 演算はゼロの符号を区別しない。 -fsimple=1 を指定すると、オプティマイザは必ず丸めまたは例外に応じた、完全な最適化を行います。特に、浮動小数点演算を、実行時に一定に保たれる丸めモードにおいて異なる結果を生成する浮動小数点演算と置き換えることはできません。
-fsimple=2	-fsimple=1 のすべての機能が含まれ、-xvector=simd が有効な場合に、SIMD 命令を使用して縮約を計算できるようにします。 コンパイラは積極的な浮動小数点演算の最適化を試み、この結果、丸めの変化によって、多くのプログラムが異なる数値結果を生じる可能性があります。たとえば、-fsimple2 を指定し、あるループ内に x/y の演算があった場合、x/y がループ内で少なくとも 1 回は必ず評価され、z=1/y で、ループの実行中に y と z が一定の値をとることが明らかである場合、オプティマイザは x/y の演算をすべて x*z で置き換えます。

本来浮動小数点例外を発生しないプログラムならば、-fsimple=2 を指定してもオプティマイザが浮動小数点例外を発生させることはありません。

最適化が精度に与える影響の詳細は、『Techniques for Optimizing Applications: High Performance Computing』(Rajat Garg と Ilya Sharapov 著) をお読みください。

B.2.30 -fsingle

-Xt モードまたは -Xs モードの場合にかぎり、float 式を倍精度ではなく単精度で評価します。-Xa モードまたは -Xc モードを使用している場合、 float 式はすでに単精度として評価されているのでこのオプションは無効です。

B.2.31 -fstore

(x86) 浮動小数点式または関数が、ある変数に代入されるか、より小さい型の浮動小数点にキャストされる場合に、コンパイラがその値をレジスタに残さないで、代入値の左側に表記される型に変換するようにします。小数点の丸めおよび切り上げを行うため、結果はレジスタの値から生成される数値と異なる可能性があります。これは、デフォルトのモードです。

このオプションを解除するには、オプション -nofstore を使用してください。

B.2.32 -ftrap=*t*[,*t*...]

SIGFPE ハンドラを組み込まずに、起動時に有効にする IEEE トラップモードの設定のみ行います。トラップの設定と SIGFPE ハンドラの組み込みを同時に行うには、ieee_handler(3M) か fex_set_handling(3M) を使用します。複数の値を指定すると、それらの値は左から右に処理されます。

t には次の値のいずれかを指定できます。

表 B–10 -ftrap フラグ

フラグ	意味
[no%]division	ゼロによる除算をトラップします [しません]。
[no%]inexact	正確でない結果をトラップします [しません]。
[no%]invalid	無効な操作をトラップします [しません]。
[no%]overflow	オーバーフローをトラップします [しません]。
[no%]underflow	アンダーフローをトラップします [しません]。
%all	前述のすべてをトラップします。
%none	前述のどれもトラップしません。
common	無効、ゼロ除算、オーバーフローをトラップします。

[no%] 形式のオプションは、下の例に示すように、%all や commonフラグの意味を変更するときだけ使用します。[no%] 形式のオプション自体は、特定のトラップを明示的に無効にするものではありません。

-ftrap を指定しない場合、コンパイラは -ftrap=%none とみなします。

たとえば、-ftrap=%all,no%inexact は、inexact 以外のすべてのトラップを設定することを意味します。

1 つのルーチンを -ftrap=*t* オプションでコンパイルした場合は、そのプログラムのルーチンすべてを、-ftrap=*t* オプションを使用してコンパイルしてください。途中から異なるオプションを使用すると、予想に反した結果が生じることがあります。

-ftrap=inexact のトラップは慎重に使用してください。-ftrap=inexact では、浮動小数点の値が正確でないとトラップが発生します。たとえば、次の文ではこの条件が発生します。

```
x = 1.0 / 3.0;
```

このオプションは、メインプログラムをコンパイルするときにだけ有効です。このオプションを使用する際には注意してください。IEEE トラップを有効にするには -ftrap=common を使用してください。

B.2.33 -G

動的にリンクされた実行可能ファイルではなく、共有オブジェクトを生成します。このオプションは ld(1) に渡されます。このオプションは -dn オプションと併用することはできません。

-G オプションを使用すると、コンパイラはデフォルトの -l オプションを ld に渡しません。共有ライブラリを別の共有ライブラリに依存させる場合は、必要な -l オプションをコマンド行に渡す必要があります。

コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションと -G オプションを組み合わせて共有ライブラリを作成した場合は、生成された共有オブジェクトとのリンクでも、必ず同じオプションを指定してください。

274 ページの「B.2.86 -xcode[= *v*]」で説明しているように、共有オブジェクトの作成では、-xarch=v9 を付けてコンパイルしたすべてのオブジェクトファイルもまた、明示的な -xcode 値を付けてコンパイルする必要があります。

B.2.34 -g

dbx(1) およびパフォーマンスアナライザ analyzer(1) によるデバッグ用のシンボルテーブル情報を生成します。

-xO3 以下の最適化レベルで -g を指定すると、ほとんど完全な最適化と可能なかぎりのシンボル情報を取得することができます。末尾呼び出しの最適化とバックエンドのインライン化は無効です。

-xO4 以下の最適化レベルで -g を指定すると、完全な最適化と可能なかぎりのシンボル情報が得られます。

-g オプションでコンパイルすると、パフォーマンスアナライザの機能をフルに利用できます。一部のパフォーマンス分析機能は -g を必要としませんが、注釈付きのソースコード、一部の関数レベルの情報、およびコンパイラの注釈メッセージを確認するには、-g でコンパイルする必要があります。詳細は、analyzer(1) のマニュアルページおよび『プログラムのパフォーマンス解析』のマニュアルを参照してください。

-g オプションで生成される注釈メッセージは、プログラムのコンパイル時にコンパイラが実行した最適化と変換について説明します。メッセージを表示するには、er_src(1) コマンドを使用します。これらのメッセージはソースコードでインタリーブされます。

注 – 以前のリリースでは、このオプションは、コンパイラのリンク専用の呼び出しにおいて、デフォルトで強制的にリンカー (ld) ではなく、インクリメンタルリンカー (ild) を使用するようにしていました。すなわち、-g が指定されたときのコンパイラは、そのデフォルトの動作として、コマンド行に -G またはソースファイルの指定がなくてもオブジェクトファイルのリンクで必ず、ld の代わりに ild を自動的に呼び出していました。現在、このようなことはありません。インクリメンタルリンカーは利用できなくなりました。

デバッグの詳細については、『dbx コマンドによるデバッグ』を参照してください。

B.2.35 -H

現在のコンパイルでインクルードされたファイルのパス名を 1 行に 1 つずつ標準エラーに出力します。字下げして表示されるので、ファイルがさらにファイルをインクルードする様子を見ることができます。

次で、sample.c は stdio.h ファイルと math.h ファイルをインクルードします。math.h は floatingpoint.h ファイルをインクルードし、floatingpoint.h はさらに、sys/ieeefp.h を使用する関数をインクルードします。

```
% cc -H sample.c
    /usr/include/stdio.h
    /usr/include/math.h
        /usr/include/floatingpoint.h
            /usr/include/sys/ieeefp.h
```

B.2.36 -h *name;*

共有動的ライブラリに名前を付けます。これによって、異なったバージョンのライブラリを作成できます。一般に -h のあとの *name;* は、-o オプションのあとに指定するファイル名と同じです。-h と *name;* の間の空白は任意です。

リンカーは指定された *name;* をライブラリに割り当て、この名前をライブラリのイントリンシック名としてライブラリファイルに記録します。-h*name;* オプションがない場合、イントリンシック名はライブラリファイルに記録されません。

実行時リンカーはライブラリを実行可能ファイルにロードするとき、イントリンシック名をライブラリファイルから実行可能ファイル中の、必要とする共有ライブラリファイルのリストにコピーします。実行可能ファイルはこのリストを持っています。共有ライブラリにイントリンシック名がないと、リンカーは代わりにその共有ライブラリファイルのパス名を使用します。

B.2.37 -I[- | *dir*]

-I *dir* は、相対ファイル名、つまり、/(スラッシュ) から始まらないディレクトリパスを持つ #include ファイルを検索するディレクトリのリスト内の /usr/include の前に dir を追加します。

複数の -I オプションが指定された場合は、指定された順序でディレクトリが調べられます。

コンパイラの検索パターンの詳細については、63 ページの「2.16.1 -I- オプションによる検索アルゴリズムの変更」を参照してください。

B.2.38 -i

オプションをリンカーへ渡して、LD_LIBRARY_PATH または LD_LIBRARY_PATH_64 の設定を無視します。

B.2.39 -include *filename;*

このオプションを指定すると、コンパイラは *filename* を、主要なソースファイルの 1 行目に記述されているかのように #include プリプロセッサ指令として処理します。ソースファイル t.c の考慮:

```
main()
{
   ...
}
```

t.c を **cc -include t.h t.c** コマンドを使用してコンパイルする場合は、ソースファイルに次の内容が含まれているかのようにコンパイルが進行します。

```
#include "t.h"
main()
{
   ...
}
```

コンパイラが *filename* を検索する最初のディレクトリは現在の作業ディレクトリであり、ファイルが明示的にインクルードされている場合のようにメインのソースファイルが存在するディレクトリになるわけではありません。たとえば、次のディレクトリ構造では、同じ名前を持つ 2 つのヘッダーファイルが異なる場所に存在しています。

```
foo/
   t.c
   t.h
   bar/
     u.c
     t.h
```

作業ディレクトリが foo/bar であり、**cc ../t.c -include t.h** コマンドを使用してコンパイルする場合は、コンパイラによって foo/bar ディレクトリから取得された t.h がインクルードされますが、ソースファイル t.c 内で #include 指令を使用した場合の foo/ ディレクトリとは異なります。

-include で指定されたファイルをコンパイラが現在の作業ディレクトリ内で見つけることができない場合は、コンパイラは通常のディレクトリパスでこのファイルを検索します。複数の -include オプションを指定する場合は、コマンド行で指定された順にファイルはインクルードされます。

B.2.40 -KPIC

(SPARC) 廃止。このオプションは使わないでください。代わりに -xcode=pic32 を使用してください。

詳細は、274 ページの「B.2.86 -xcode[= *v*]」 を参照してください。225 ページの「A.1.15 廃止オプション」に、廃止のオプションの全一覧をまとめています。

(x86) -KPIC は -Kpic と同じです。

B.2.41 -Kpic

(SPARC) 廃止。このオプションは使わないでください。代わりに -xcode=pic13 を使用してください。詳細は、274 ページの「B.2.86 -xcode[= *v*]」 を参照してください。225 ページの「A.1.15 廃止オプション」に、廃止のオプションの全一覧をまとめています。

(x86) 共用ライブラリ (小型モデル) で使用するための位置に依存しないコードを生成します。最大 2**11 個の独自の外部シンボルを参照できます。

B.2.42 -keeptmp

コンパイル中に作成される一時ファイルを自動的に削除しないで保持します。

B.2.43 -L*dir*

ld(1) がライブラリを検索するディレクトリのリストに *dir* を付け加えます。このオプションとその引数は ld(1) に渡されます。

注 – コンパイラがインストールされている位置の /usr/include、 /lib、/usr/lib を検索ディレクトリに指定しないでください。

B.2.44 -l*name;*

オブジェクトライブラリ lib *name*.so または lib*name* .a をリンクの対象とします。シンボルは左から右へ解決されるため、コマンド行でのライブラリの指定順が重要になります。

このオプションはソースファイル引数のあとに指定してください。

B.2.45 -m32|-m64

コンパイルされたバイナリオブジェクトのメモリーモデルを指定します。

-m32 を使用すると、32 ビット実行可能ファイルと共有ライブラリが作成されます。-m64 を使用すると、64 ビット実行可能ファイルと共有ライブラリが作成されます。

ILP32 メモリーモデル (32 ビット int、long、ポインタデータ型) は 64 ビット対応ではないすべての Solaris プラットフォームおよび Linux プラットフォームのデフォルトです。LP64 メモリーモデル (64 ビット long、ポインタデータ型) は 64 ビット対応の Linux プラットフォームのデフォルトです。-m64 は LP64 モデル対応のプラットフォームでのみ使用できます。

-m32 を使用してコンパイルされたオブジェクトファイルまたはライブラリを、-m64 を使用してコンパイルされたオブジェクトファイルまたはライブラリにリンクすることはできません。

-m32|-m64 を指定してコンパイルしたモジュールは、-m32 |-m64 を指定してリンクする必要があります。215 ページの「A.1.2 コンパイル時とリンク時のオプション」に、コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの一覧をまとめています。

x86/x64 プラットフォームで大量の静的データを持つアプリケーションを -m64 を使用してコンパイルするときは、-xmodel=medium も必要になることがあります。一部の Linux プラットフォームは、ミディアムモデルをサポートしていません。

以前のコンパイラリリースでは、-xarch で命令セットを選択すると、メモリーモデル ILP32 または LP64 が使用されていました。Solaris Studio 12 以降のコンパイラでは、このようなことはありません。ほとんどのプラットフォームでは、-m64 をコマンド行に追加するだけで、64 ビットオブジェクトが作成されます。

Solaris では、-m32 がデフォルトです。64 ビットプログラムをサポートする Linux システムでは、-m64 -xarch=sse2 がデフォルトです。

-xarch も参照してください。

B.2.46 -mc

オブジェクトファイルの .comment セクションから重複している文字列を削除します。-mc フラグを使用すると、mcs -c が起動されます。

B.2.47 -misalign

(SPARC) 廃止。このオプションは使わないでください。代わりに -xmemalign=1i オプションを使ってください。詳細は、293 ページの「B.2.116 -xmemalign=*ab*」を参照してください。225 ページの「A.1.15 廃止オプション」に、廃止のオプションの全一覧をまとめています。

B.2.48 -misalign2

(SPARC) 廃止。このオプションは使わないでください。代わりに -xmemalign=2i オプションを使ってください。詳細は、293 ページの「B.2.116 -xmemalign=*ab*」を参照してください。225 ページの「A.1.15 廃止オプション」に、廃止のオプションの全一覧をまとめています。

B.2.49 -mr[, *string*]

-mr は、.comment セクションからすべての文字を削除します。このフラグを使用すると、mcs -d -a が呼び出されます。

-mr, *string* はオブジェクトファイルの .comment セクションからすべての文字列を削除して、*string* を挿入します。*string* に空白が含まれている場合は二重引用筆囲みます。*string* がなければ .comment セクションは空になります。このオプションは -d -*string* として mcs に渡されます。

B.2.50 -mt[={ yes|no}]

このオプションを使用して、Solaris スレッドまたは POSIX スレッドの API を使用しているマルチスレッド化コードをコンパイルおよびリンクします。-mt=yes オプションにより、ライブラリが適切な順序でリンクされることが保証されます。

このオプションは -D_REENTRANT をプリプロセッサに渡します。

Solaris スレッドを使用するには、thread.h ヘッダーファイルをインクルードし、–mt=yes オプションを使用してコンパイルします。Solaris プラットフォームで POSIX スレッドを使用するには、pthread.h ヘッダーファイルをインクルードし、–mt=yes オプションを使用してコンパイルします。

Linux プラットフォーム上では、POSIX スレッドの API のみが使用できます (Linux プラットフォームには libthread はありません)。したがって、Linux プラットフォームで –mt=yes を使用すると、–lthread の代わりに –lpthread が追加されます。Linux プラットフォームで POSIX スレッドを使用するには、–mt を使用してコンパイルします。

–G を使用してコンパイルする場合は、–mt=yes を指定しても、–lthread と –lpthread のどちらも自動的には含められません。共有ライブラリを構築する場合は、これらのライブラリを明示的にリストする必要があります。

(OpenMP 共有メモリー並列化 API を使用するための) –xopenmp オプションには、–mt=yes が自動的に含まれます。

-mt=yes を指定してコンパイルを実行し、リンクを個別の手順でリンクする場合は、コンパイル手順と同様にリンク手順でも -mt=yes オプションを使用する必要があります。-mt=yes を使用して 1 つの変換ユニットをコンパイルおよびリンクする場合は、-mt=yes を指定してプログラムのすべてのユニットをコンパイルおよびリンクする必要があります。

-mt=yes は、コンパイラのデフォルトの動作です。この動作が望ましくない場合は、-mt=no でコンパイルします。

オプション –mt は –mt=yes と同じです。

B.2.50.1 関連項目

–xnolib、Solaris 『Multithreaded Programming Guide』、および『Linker and Libraries Guide』

B.2.51 -native

このオプションは、-xtarget=native と同義です。

B.2.52 -nofstore

*(x86)*浮動小数点式または関数がある変数に代入されるか、より小さい型の浮動小数点にキャストされる場合に、代入値の左側に表記される型に変換せずに、コンパイラがその値をレジスタに残すようにします。242 ページの「B.2.31 -fstore」も参照してください。

B.2.53 -O

デフォルトの最適化レベルの -xO3 を使ってください。このリリースでは、-O は、-xO2 ではなく、-xO3 に展開されます。

このデフォルトの変更によって、実行時のパフォーマンスが向上します。ただし、あらゆる変数を自動的に volatile と見なすことを前提にするプログラムの場合、-x03 は不適切なことがあります。このことを前提とする代表的なプログラムとしては、専用の同期方式を実装するデバイスドライバや古いマルチスレッドアプリケーションがあります。回避策は、-O ではなく、-xO2 を使ってコンパイルすることです。

B.2.54 -o *filename*

出力ファイルに、デフォルトの *a.out* の代わりに filename という名前を付けます。cc はソースファイルに上書きしないので、*filename* にはソースファイルと同じ名前は使用できません。このオプションとその引数は ld(1) に渡されます。

B.2.55 -P

ソースファイルのプリプロセッサ処理のみを行います。.i 接尾辞の付いたファイルに結果を出力します。-E オプションと異なり、出力ファイルに C のプリプロセッサ行番号付け情報は含まれません。-E オプションも参照してください。

B.2.56 -p

廃止。309 ページの「B.2.132 -xpg」を参照してください。

B.2.57 -Q[y|n]

出力ファイルに識別情報を入れるかどうかを設定します。デフォルトは -Qy です。

-Qy を指定すると、起動した各コンパイラツールの識別情報が出力ファイルの .comment 部分に追加され、mcs でのアクセスが可能になります。これはソフトウェア管理に役立ちます。

-Qn を指定すると、この情報が抑制されます。

B.2.58 -qp

-pと同じです。

B.2.59 -R*dir*[:*dir*]

コロンで区切られたディレクトリのリストを、ライブラリ検索ディレクトリとして、実行時リンカーに渡します。リストが空でなければ、出力オブジェクトファイルに記録され、実行時リンカーに渡されます。

LD_RUN_PATH と -R オプションの両方が指定されたときは、この -R オプションが優先されます。

B.2.60 -S

cc to アセンブリソースファイルを作成しますが、アセンブルは行いません。

B.2.61 -s

出力されるオブジェクトファイルからシンボリックデバッグのための情報をすべて削除します。このオプションは、-g とともに指定することはできません。

ld(1) へ引き渡します。

B.2.62 -traceback[={ %none|common|*signals_list*}]

実行時に重大エラーが発生した場合にスタックトレースを発行します。

-traceback オプションを指定すると、プログラムによって特定のシグナルが生成された場合に、実行可能ファイルで stderr へのスタックトレースが発行されて、コアダンプが実行され、終了します。複数のスレッドが 1 つのシグナルを生成すると、スタックトレースは最初のスレッドに対してのみ生成されます。

追跡表示を使用するには、リンク時に -traceback オプションをコンパイラコマンド行に追加します。このオプションはコンパイル時にも使用できますが、実行可能バイナリが生成されない場合無視されます。-traceback を -G とともに使用して共有ライブラリを作成すると、エラーが発生します。

表 B–11 -traceback オプション

オプション	意味
common	sigill、sigfpe、sigbus、sigsegv、または sigabrt の共通シグナルのいずれかのセットが発生した場合にスタックトレースを発行することを指定します。
signals_list	スタックトレースを生成するシグナルの名前を小文字で入力してコンマで区切ったリストを指定します。sigquit、sigill、sigtrap、sigabrt、sigemt、sigfpe、sigbus、sigsegv、sigsys、sigxcpu、sigxfsz のシグナル (コアファイルが生成されるシグナル) をキャッチできます。 これらのシグナルの前に no% を付けると、シグナルのキャッチは無効になります。 たとえば、**-traceback=sigsegv,sigfpe** と指定すると、sigsegv または sigfpe が発生した場合にスタックトレースとコアダンプが生成されます。
%none または none	追跡表示を無効にします。

このオプションを指定しない場合、デフォルトは -traceback=%none になります。

= 記号を指定せずに、-traceback だけを指定すると、-traceback=common と同義になります。

注: コアダンプが不要な場合は、次を使用して coredumpsize 制限を 0 に設定できます。

```
% limit coredumpsize 0
```

-traceback オプションは、実行時のパフォーマンスに影響しません。

B.2.63 -U*name;*

プリプロセッサシンボル *name;* の定義を削除します。このオプションは、コマンド行ドライバで配置された要素も含む同一コマンド行において -D で作成されたプリプロセッサシンボル *name;* の初期定義を削除します。

-U は、ソースファイルのプリプロセッサ指令に影響しません。コマンド行に複数の -U オプションを配置できます。

コマンド行の -D と -U に同じ *name* が指定された場合、オプションの配置順に関係なく、*name* の定義が削除されます。次の図で、-U は __sun の定義のを削除します。

```
cc -U__sun text.c
```

test.c の次の書式のプリプロセッサ文は、__sun の定義が削除されているために有効になりません。

```
#ifdef(__sun)
```

定義済みシンボルのリストについては、230 ページの「B.2.7 -D*name*[(*arg*[,*arg*])][=*expasion*]」 を参照してください。

B.2.64 -V

コンパイラの実行時に cc によって各構成要素の名前とバージョン番号を表示します。

B.2.65 -v

より厳しい意味検査およびほかの lint に似た検査を行います。次は

```
#include <stdio.h>
main(void)
{
     printf("Hello World.\n");
}
```

支障なくコンパイルと実行ができるコードです。-v を使用すると、コンパイルは行われますが次の警告が表示されます。

```
"hello.c", line 5: warning: function has no return statement:
 main
```

-v は lint(1) が発する警告をすべて表示するわけではありません。lint で前述の例を実行すると確認することができます。

B.2.66 -W*c*,*arg*

指定されたコンパイラ構成要素 *c* に、*arg* を渡します。コンポーネントのリストについては、表 1–1 を参照してください。

各引数はコンマで区切ります。すべての -W arg は、通常のコマンド行引数のあとに渡されます。コンマを引数に含めるには、コンマの直前にエスケープ文字 \ (バックスラッシュ) を使用します。すべての -W arg は、通常のコマンド行引数のあとに渡されます。

たとえば、`-Wa,-o,objfile` は、`-o` と `objfile` を、この順番でアセンブラに渡します。また、`-Wl,-I,`*name*; を指定すると、リンク段階で動的リンカー `/usr/lib/ld.so.1` のデフォルト名が無効になります。

そのほかのコマンド行オプションで引数がツールに渡される順番は、変更されることがあります。

c には、次のいずれかを指定します。

表 B–12　`-W` のフラグ

フラグ	意味
`a`	アセンブラ : (`fbe`); (`gas`)
`c`	C コードジェネレータ : (`cg`) *(SPARC)*;
`d`	cc ドライバ
`h`	中間コード翻訳 (`ir2hf`)(*x86*)
`l`	リンクエディタ (`ld`)
`m`	mcs
`O` (大文字の o)	手続き間オプティマイザ
o (小文字の o)	ポストオプティマイザ
`p`	プリプロセッサ (`cpp`)
`u`	C コードジェネレータ (`ube`) (*x86*)
`0` (ゼロ)	コンパイラ (`acomp`) (ssbd, *SPARC*)
`2`	オプティマイザ: (`iropt`)

B.2.67 -w

コンパイラからの警告メッセージを出力しません。

このオプションは `error_messages` プラグマを無効にします。

B.2.68 -X[c|a|t|s]

`-X` (大文字の `X` である点に注意) オプションでは ISO C に準拠する度合いを指定します。`-xc99` の値により、`-X` オプションが適用される ISO C 規格のバージョンが異なります。`-xc99` オプションは、デフォルトでは `-xc99=all` になっています。この場合は、1999 ISO/IEC C 規格のサブセットをサポートします。`-xc99=none` を指定した場合

は、1990 ISO/IEC C 規格をサポートします。サポートされている 1999 ISO/IEC の機能については、表 C–6 を参照してください。ISO/IEC C と K&R C の相違点については、399 ページの「G.1 libfast.a Library (SPARC)」 を参照してください。

デフォルトのモードは -Xa です。

-Xc

(c = conformance) ISO C にない言語構造を使用しているプログラムに対してエラーや警告を発行します。このオプションは ISO C に最大限に準拠するもので、K&R C との拡張互換性はありません。--Xc オプションを指定すると事前定義されたマクロ __STDC__ の値は 1 になります。

-Xa

これは、コンパイラのデフォルトのモードです。ISO C に K&R C との拡張互換性を持たせます。ISO C に従うように意味処理を変更します。同じ言語構造に対して ISO C と K&R C の意味処理が異なる場合は ISO C に準拠した解釈を行います。-Xa オプションを -xtransition オプションと併せて使用すると、異なる意味論に関する警告が出力されます。-Xa オプションを指定すると事前定義されたマクロ __STDC__ の値は -0 になります。

-Xt

(t = 遷移) このオプションでは、ISO C で求められる意味処理の変更を行わずに ISO C と K&R C の拡張互換性が使用されます。K&R C と ISO C で同じ構文に異なる意味処理が指定される場合、コンパイラは K&R C の解釈を使用します。-Xt オプションを -xtransition オプションと併せて使用すると、異なる意味論に関する警告が出力されます。-Xt オプションを指定すると事前定義されたマクロ __STDC__ の値は 0 になります。

-Xs

(s = K&R C) ISO C と K&R C の間で動作が異なるすべての言語構造に対して警告を発行します。K&R C と互換性のあるすべての機能がコンパイルされます。このオプションでは、前処理用に cpp が呼び出され、__STDC__ は定義されません。

B.2.69 -x386

(x86) 廃止。このオプションは使わないでください。代わりに、-xchip=generic を使用してください。225 ページの「A.1.15 廃止オプション」 に、廃止のオプションの全一覧をまとめています。

B.2.70 -x486

(x86) 廃止。このオプションは使わないでください。代わりに、`-xchip=generic` を使用してください。225 ページの「A.1.15 廃止オプション」に、廃止のオプションの全一覧をまとめています。

B.2.71 -xaddr32[=yes|no]

(*Solaris x86/x64* のみ) コンパイラフラグ `-xaddr32=yes` は、結果として生成される実行可能ファイルまたは共有オブジェクトを 32 ビットアドレス空間に制限します。

この方法でコンパイルする実行可能ファイルは、32 ビットアドレス空間に制限されるプロセスを作成する結果になります。

`-xaddr32=no` を指定する場合は、通常の 64 ビットバイナリが作成されます。

`-xaddr32` オプションを指定しないと、**`-xaddr32=no`** が使用されます。

`-xaddr32` だけを指定すると、**`-xaddr32=yes`** が使用されます。

このオプションは、**`-m64`** のコンパイルのみ、および `SF1_SUNW_ADDR32` ソフトウェア機能をサポートしている Solaris プラットフォームのみに適用できます。Linux カーネルはアドレス空間の制限をサポートしていないので、Linux では、このオプションは使用できません。

単一のオブジェクトファイルが **`-xaddr32=yes`** を指定してコンパイルされた場合は、出力ファイル全体が **`-xaddr32=yes`** を指定してコンパイルされたものと、リンク時に想定されます。

32 ビットアドレス空間に制限される共有オブジェクトは、制限された 32 ビットモードのアドレス空間内で実行されるプロセスから読み込む必要があります。

詳細は、『Linker and Libraries Guide』で説明されている `SF1_SUNW_ADDR32` ソフトウェア機能の定義を参照してください。

B.2.72 -xalias_level[=*l*]

コンパイラで `-xalias_level` オプションを使用して、型に基づく別名の解析による最適化でのレベルを指定します。このオプションは、コンパイル対象の変換ユニットで、指定した別名レベルを有効にします。

`-xalias_level` コマンドを発行しない場合、コンパイラは `-xalias_level=any` を仮定します。値を設定しないで `-xalias_level` を指定する場合、デフォルトは `-xalias_level=layout` になります。

-xalias_level オプションを使用するには、-xO3 以上の最適化レベルが必要です。最適化がこれよりも低く設定されている場合は、警告が表示され、-xalias_level オプションは無視されます。

-xalias_level オプションを発行しても、別名レベルごとに記述されている規則と制限を遵守しない場合、プログラムが未定義の動作をします。

l を次の表のいずれかの用語で置き換えます。

表 B–13 別名明確化のレベル

フラグ	意味
any	コンパイラは、すべてのメモリー参照がこのレベルで別名設定できると仮定します。-xalias_level=any レベルで型に基づく別名分析は行われません。
basic	-xalias_level=basic オプションを使用する場合、コンパイラは、さまざまな C 言語基本型を呼び出すメモリー参照が相互に別名設定しないと仮定します。また、コンパイラは、ほかのすべての型への参照が C 言語基本型と同様に相互に別名設定できると仮定します。コンパイラは、char * を使用する参照がそのほかの型を別名設定できると仮定します。 たとえば、-xalias_level=basic レベルにおいて、コンパイラは、int * 型のポインタ変数が float オブジェクトにアクセスしないことを仮定します。そのため、コンパイラは、float * 型のポインタが int * 型のポインタで参照される同一メモリーを別名設定しないと仮定する最適化を実行すると安全です。
weak	-xalias_level=weak オプションを使用する場合、コンパイラは、任意の構造体ポインタが構造体の型にポイントできると仮定します。 コンパイルされるソースの式で参照されるか、コンパイルされるソースの外側から参照される型への参照を保持する構造体または共用体は、コンパイルされるソースの式より先に宣言しなければいけません。 この制限事項を遵守するには、コンパイルされるソースの式で参照されるオブジェクトの型を参照する型を含むプログラムの全ヘッダーファイルを取り込みます。 -xalias_level=weak レベルで、コンパイラは、さまざまな C 言語基本型に関連するメモリー参照が相互に別名設定しないと仮定します。コンパイラは、char * を使用する参照がそのほかの型に関連するメモリー参照を別名設定すると仮定します。

表 B–13 別名明確化のレベル (続き)

フラグ	意味
layout	-xalias_level=layout オプションを使用すると、コンパイラは、メモリー内に同一の型のシーケンスを保持する型に関連するメモリー参照が相互に別名設定できると仮定できます。 コンパイラは、メモリーで同一に見えない型を保持する 2 つの参照が相互に別名設定しないと仮定します。コンパイラは、構造体の初期メンバーがメモリーで同じに見える場合、さまざまな構造体の型の別名を介して 2 つのメモリーがアクセスを実行すると仮定します。ただし、このレベルで、構造体へのポインタを使用して、2 つの構造体の間にあるメモリーで同じに見えるメンバーの一般的な初期シーケンスの外側にある類似しない構造体オブジェクトのフィールドにアクセスすべきではありません。そのような参照が相互に別名設定しないとコンパイラが仮定しているからです。 -xalias_level=layout オプションを使用すると、コンパイラは、メモリー内に同一の型のシーケンスを保持する型に関連するメモリー参照が相互に別名設定できると仮定できます。コンパイラは、char * を使用する参照がそのほかの型に関連するメモリー参照を別名設定できると仮定します。
strict	-xalias_level=strict オプションを使用すると、コンパイラは、タグを削除したときに同一となるメモリー参照 (構造体や共用体などの型に関連するもの) が相互に別名設定できると仮定します。逆に言えば、コンパイラは、タグを削除したあとも同一にならない型に関連するメモリー参照は相互に別名設定しないと仮定します。 ただし、コンパイルされるソースの式で参照されるか、コンパイルされるソースの外側から参照されるオブジェクトの一部となる型への参照を含む構造体または共用体の型は、コンパイルされるソースの式より先に宣言しなければいけません。 この制限事項を遵守するには、コンパイルされるソースの式で参照されるオブジェクトの型を参照する型を含むプログラムの全ヘッダーファイルを取り込みます。-xalias_level=strict レベルで、コンパイラは、さまざまな C 言語基本型に関連するメモリー参照が相互に別名設定しないと仮定します。コンパイラは、char * を使用する参照がそのほかの型を別名設定できると仮定します。
std	-xalias_level=std オプションを使用する場合、コンパイラは、型とタグが別名に対し同一でなくてはならないが、char * を使用する参照がそのほかの型を別名設定できると仮定します。この規則は、1999 ISO C 規格に記載されているポインタの参照解除についての制限事項と同じです。この規則を正しく使用するプログラムは移植性が非常に高く、最適化によって良好なパフォーマンスが得られるはずです。
strong	-xalias_level=strong オプションを使用する場合、std レベルと同じ規則が適用されますが、それに加えて、コンパイラは、型 char * のポインタを使用する場合にかぎり、型 char のオブジェクトにアクセスできると仮定します。また、コンパイラは、内部ポインタが存在しないと仮定します。内部ポインタは、構造体のメンバーをポイントするポインタとして定義されます。

B.2.73 - xannotate[=yes|no]

(Solaris) binopt(1) のようなバイナリ変更ツールを使用してあとで変換できるバイナリを作成するようコンパイラに指示します。これらのツールによるバイナリファイルの変更を防止するには、-xannotate=no オプションを使用します。-xannotate=yes オプションを有効にするには、最適化レベル -xO1 以上で使用する必要があります。

Linux プラットフォームでは、このオプションはありません。

B.2.74 -xarch=*isa*

対象となる命令セットアーキテクチャー (*ISA*) を指定します。

このオプションは、コンパイラが生成するコードを、指定した命令セットアーキテクチャーの命令だけに制限します。このオプションは、すべてのターゲットを対象とするような命令としての使用は保証しません。ただし、このオプションを使用するとバイナリプログラムの移植性に影響を与える可能性があります。

注 – 意図するメモリーモデルとして LP64 (64 ビット) または ILP32 (32 ビット) を指定するには、それぞれ -m64 または -m32 オプションを使用してください。次に示すように以前のリリースとの互換性を保つ場合を除いて、-xarch オプションでメモリーモデルを指定できなくなりました。

別々の手順でコンパイルしてリンクする場合は、両方の手順に同じ -xarch の値を指定してください。

B.2.74.1 SPARC での -xarch のフラグ

次の表に、SPARC プラットフォームでの各 -xarch キーワードの詳細を示します。

表 B–14 SPARC プラットフォームでの -xarch のフラグ

フラグ	意味
generic	ほとんどのプロセッサに共通の命令セットを使用します。これはデフォルト値です。
generic64	多くのシステムで良好な 64 ビットパフォーマンスを得るためのコンパイルをします(Solaris のみ)。 このオプションは -m64 -xarch=generic に相当し、以前のリリースとの互換性のために用意されています。64 ビットでのコンパイルを指定するには、次のものではなく -m64 を使用してください。 - xarch=generic64

表 B–14 SPARC プラットフォームでの -xarch のフラグ (続き)

フラグ	意味
native	このシステムで良好なパフォーマンスを得られるようにコンパイルします。現在コンパイルしているシステムプロセッサにもっとも適した設定を選択します。
native64	このシステムで良好なパフォーマンスを得られるようにコンパイルします (Solaris のみ)。このオプションは -m64 -xarch=native に相当し、以前のリリースとの互換性のために用意されています。
sparc	SPARC-V9 ISA 用のコンパイルを実行しますが、VIS (Visual Instruction Set) は使用せず、その他の実装に固有の ISA 拡張機能も使用しません。このオプションを使用して、コンパイラは、V9 ISA で良好なパフォーマンスが得られるようにコードを生成できます。
sparcvis	SPARC-V9 + VIS (Visual Instruction Set) version 1.0 + UltraSPARC 拡張機能用のコンパイルを実行します。このオプションを使用すると、コンパイラは、UltraSPARC アーキテクチャー上で良好なパフォーマンスが得られるようにコードを生成することができます。
sparcvis2	UltraSPARC アーキテクチャー + VIS (Visual Instruction Set) version 2.0 + UltraSPARC-III 拡張機能用のオブジェクトコードを生成します。
sparcvis3	SPARC VIS version 3 の SPARC-V9 ISA 用にコンパイルします。SPARC-V9 命令セット、VIS (Visual Instruction Set) version 1.0 を含む UltraSPARC 拡張機能、VIS (Visual Instruction Set) version 2.0、積和演算 (FMA) 命令、および VIS (Visual Instruction Set) version 3.0 を含む UltraSPARC-III 拡張機能の命令をコンパイラが使用できるようになります。
sparcfmaf	SPARC-V9 命令セット、VIS (Visual Instruction Set) version 1.0 を含む UltraSPARC 拡張機能、VIS (Visual Instruction Set) version 2.0 を含む UltraSPARC-III 拡張機能、および浮動小数点積和演算用の SPARC64 VI 拡張機能の命令をコンパイラが使用できるようになります。 -xarch=sparcfmaf は fma=fused と組み合わせて使用し、ある程度の最適化レベルを指定することで、コンパイラが自動的に積和命令の使用を試みるようにする必要があります。
sparcima	sparcima 版の SPARC-V9 ISA 用にコンパイルします。コンパイラが、SPARC-V9 命令セットに加えて、VIS (Visual Instruction Set) Version 1.0 を含む UltraSPARC 拡張機能、VIS (Visual Instruction Set) Version 2.0 を含む UltraSPARC-III 拡張機能、浮動小数点の積和演算用の SPARC64 VI 拡張機能、整数の積和演算用の SPARC64 VII 拡張機能からの命令を使用できるようにします。
v9	-m64 -xarch=sparc に相当します。64 ビットメモリーモデルを得るために -xarch=v9 を使用する古いメイクファイルとスクリプトでは、-m64 だけを使用すれば十分です。
v9a	-m64 -xarch=sparcvis に相当し、以前のリリースとの互換性のために用意されています。

表 B–14 SPARC プラットフォームでの -xarch のフラグ (続き)

フラグ	意味
v9b	-m64 -xarch=sparcvis2 に相当し、以前のリリースとの互換性のために用意されています。

また、次のことにも注意してください。

- generic、sparc、sparcvis2、sparcvis3、sparcfmaf、sparcima でコンパイルされたオブジェクトライブラリファイル (.o) をリンクして、一度に実行できます。ただし、実行できるのは、リンクされているすべての命令セットをサポートしているプロセッサのみです。
- 特定の設定で、生成された実行可能ファイルが実行されなかったり、従来のアーキテクチャーよりも実行速度が遅くなったりする場合があります。また、4 倍精度 (REAL*16 および long double) 浮動小数点命令は、これらの命令セットアーキテクチャーのいずれにも実装されないため、コンパイラは、それらの命令を生成したコードで使用しません。

B.2.74.2 x86 での -xarch のフラグ

次の表に、x86 プラットフォームでの -xarch フラグを示します。

表 B–15 x86 での -xarch のフラグ

フラグ	意味
amd64	-m64 -xarch=sse2 に相当します (Solaris のみ)。64 ビットメモリーモデルを得るために -xarch=amd64 を使用する古いメイクファイルとスクリプトでは、 -m64 だけを使用すれば十分です。
amd64a	-m64 -xarch=sse2a に相当します (Solaris のみ)。
generic	ほとんどのプロセッサに共通の命令セットを使用します。これはデフォルトであり、–m32 でコンパイルする場合の pentium_pro、および –m64 でコンパイルする場合の sse2 に相当します。
generic64	多くのシステムで良好な 64 ビットパフォーマンスを得るためのコンパイルをします(Solaris のみ)。このオプションは -m64 -xarch=generic に相当し、以前のリリースとの互換性のために用意されています。64 ビットでのコンパイルを指定するには、次のものではなく -m64 を使用してください。 - xarch=generic64
native	このシステムで良好なパフォーマンスを得られるようにコンパイルします。現在コンパイルしているシステムプロセッサにもっとも適した設定を選択します。

表 B–15 x86 での -xarch のフラグ (続き)

フラグ	意味
native64	このシステムで良好なパフォーマンスを得られるようにコンパイルします (Solaris のみ)。このオプションは -m64 -xarch=native に相当し、以前のリリースとの互換性のために用意されています。
pentium_pro	命令セットを 32 ビット pentium_pro アーキテクチャーに限定します。
pentium_proa	AMD 拡張機能 (3DNow!、3DNow! 拡張機能、および MMX 拡張機能) を 32 ビット pentium_pro アーキテクチャーに追加します。
sse	SSE 命令セットを pentium_pro アーキテクチャーに追加します。
ssea	AMD 拡張機能 (3DNow!、3DNow! 拡張機能、および MMX 拡張機能) を 32 ビット SSE アーキテクチャーに追加します。
sse2	SSE2 命令セットを pentium_pro アーキテクチャーに追加します。
sse2a	AMD 拡張機能 (3DNow!、3DNow! 拡張機能、および MMX 拡張機能) を 32 ビット SSE2 アーキテクチャーに追加します。
sse3	SSE3 命令セットを SSE2 命令セットに追加します。
ssse3	SSSE3 命令セットで、pentium_pro、SSE、SSE2、および SSE3 の各命令セットを補足します。
sse4_1	SSSE4.1 命令セットで、pentium_pro、SSE、SSE2、SSE3、および SSSE3 の各命令セットを補足します。
sse4_2	SSSE4.2 命令セットで、pentium_pro、SSE、SSE2、SSE3、SSSE3、および SSSE4.1 の各命令セットを補足します。
amdsse4a	AMD SSE4a 命令セットを使用します。

B.2.74.3 x86 の特記事項

x86 Solaris プラットフォーム用にコンパイルを行う場合に注意が必要な、重要な事項がいくつかあります。

-xarch を sse、sse2、sse2a、または sse3 以降に設定してコンパイルしたプログラムは、必ずこれらの拡張子と機能を提供するプラットフォームでのみ実行してください。

Pentium 4 互換プラットフォームの場合、Solaris OS リリースは SSE/SSE2 に対応しています。これより前のバージョンの Solaris OS は SSE/SSE2 に対応していません。-xarch で選択した命令セットが、実行中の Solaris OS で有効ではない場合、コンパイラはその命令セットのコードを生成またはリンクできません。

コンパイルとリンクを別々に行う場合は、必ずコンパイラを使ってリンクし、-xarch 設定で適切な起動ルーチンがリンクされるようにしてください。

x86 の 80 バイト浮動小数点レジスタが原因で、x86 での演算結果が SPARC の結果と異なる数値になる場合があります。この差を最小にするには、 `-fstore` オプションを使用するか、ハードウェアが SSE2 をサポートしている場合は `-xarch=sse2` でコンパイルします。

Solaris と Linux でも、固有の数学ライブラリ (たとえば、`sin(x)`) が同じではないため、演算結果が異なることがあります。

B.2.74.4 バイナリの互換性の妥当性検査

Solaris Studio 11 と Solaris 10 OS から、これらの特殊化された `-xarch` ハードウェアフラグを使用してコンパイルし、構築されたプログラムバイナリは、適切なプラットフォームで実行されることが確認されます。

Solaris 10 以前のシステムでは妥当性検査が行われないため、これらのフラグを使用して構築したオブジェクトが適切なハードウェアに配備されることをユーザが確認する必要があります。

これらの `-xarch` オプションでコンパイルしたプログラムを、適切な機能または命令セット拡張に対応していないプラットフォームで実行すると、セグメント例外や明示的な警告メッセージなしの不正な結果が発生することがあります。

このことは、.il インラインアセンブリ言語関数を使用しているプログラムや、SSE、SSE2、SSE2a、および SSE3 の命令と拡張機能を利用している `__asm()` アセンブラコードにも当てはまります。

B.2.74.5 相互の関連性

このオプションは単体でも使用できますが、`-xtarget` オプションの展開の一部でもあります。したがって、特定の `-xtarget` オプションで設定される `-xarch` のオーバーライドにも使用できます。`-xtarget=ultra2` は `-xarch=sparcvis -xchip=ultra2 -xcache=16/32/1:512/64/1` に展開されます。次のコマンドでは、`-xarch=generic` は、`-xtarget=ultra2` の展開で設定された `-xarch=sparcvis` より優先されます。

```
example% cc -xtarget=ultra2 -xarch=generic foo.c
```

B.2.74.6 警告

このオプションを最適化と併せて使用する場合、適切なアーキテクチャーを選択すると、そのアーキテクチャー上での実行パフォーマンスを向上させることができます。ただし、適切な選択をしなかった場合、パフォーマンスが著しく低下するか、あるいは、作成されたバイナリプログラムが目的のターゲットプラットフォーム上で実行できない可能性があります。

別々の手順でコンパイルしてリンクする場合は、両方の手順に同じ `-xarch` の値を指定してください。

B.2.75 -xautopar

注 – このオプションは OpenMP の並列化命令を有効にしません。

複数プロセッサ用の自動並列化を有効にします。依存性の解析 (ループの繰り返し内部でのデータ依存性の解析) およびループ再構成を実行します。最適化が -xO3 以上でない場合は -xO3 に上げられ、警告が出されます。

独自のスレッド管理を行なっている場合には、-xautopar を使用しないでください。

実行速度を上げるには、マルチプロセッサシステムが必要です。シングルプロセッサシステムでは、通常、生成されたバイナリの実行速度は低下します。

並列化されたプログラムをマルチスレッド環境で実行するには、実行前に OMP_NUM_THREADS 環境変数を設定しておく必要があります。詳細は、『Solaris Studio OpenMP API ユーザーズガイド』を参照してください。

-xautopar を使用してコンパイルとリンクを 1 度に実行する場合、リンクには自動的にマイクロタスキングライブラリおよびスレッドに対して安全な C 実行時ライブラリが含まれます。-xautopar を使用してコンパイルとリンクを別々に実行する場合、リンクにも -xautopar を指定しなければいけません。表 A–2 に、コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの全一覧をまとめています。

B.2.76 -xbinopt={prepare| off}

(SPARC) あとで最適化、変換、分析を行うために、バイナリを準備します。binopt(1) を参照してください。このオプションは、実行可能ファイルまたは共有オブジェクトの構築に使用できます。このオプションを有効にするには、最適化レベル -xO1 以上で使用する必要があります。このオプションを使用すると、構築したバイナリのサイズが少し増えます。

コンパイルとリンクを別々に行う場合は、両方の手順に -xbinopt を指定する必要があります。

```
example% cc -c -xO1 -xbinopt=prepare a.c b.c
example% cc -o myprog -xbinopt=prepare a.o
```

一部のソースコードがコンパイルに使用できない場合も、このオプションを使用してそのほかのコードがコンパイルされます。このとき、最終的なバイナリを作成するリンク手順で、このオプションを使用する必要があります。この場合、このオプションでコンパイルされたコードだけが最適化、変換、分析できます。

-xbinopt=prepare と -g を指定してコンパイルすると、デバッグ情報が取り込まれるので、実行可能ファイルのサイズが増えます。デフォルトは -xbinopt=off です。

B.2.77 -xbuiltin[=(%all|%none)]

標準ライブラリ関数を呼び出すコードをさらに最適化する場合、-xbuiltin[=(%all|%none)] コマンドを使用します。多くの標準ライブラリ関数 (math.h、stdio.h で定義された関数など) は、さまざまなプログラムで一般的に使用されます。このコマンドを使用すると、コンパイラは、パフォーマンス向上の見込める場所で組み込み関数またはインラインシステム関数を代用します。オブジェクトファイル内のコンパイラのコメントからコンパイラが実際に置換を行う関数を特定する方法については、er_src(1) のマニュアルページを参照してください。

ただし、こうした置換によって errno の値の信頼性が失われることがあります。プログラムが errno の値に依存している場合、このオプションの使用は避けてください。58 ページの「2.13 errno の値の保持」も参照してください。

-xbuiltin を指定しない場合は、デフォルトは -xbuiltin=%none になります。この場合は、標準ライブラリの関数の代替やインライン化は行われません。-xbuiltin を引数なしで指定した場合は、デフォルトは -xbuiltin%all になります。この場合は、最適化に有効と判断されるときには、組込型関数の代替や標準ライブラリの関数のインライン化が実行されます。

-fast を指定してコンパイルする場合は、-xbuiltin は %all になります。

注 – -xbuiltin は、システムのヘッダーファイルで定義された大域関数だけをインライン化します。ユーザー定義の静的関数はインライン化しません。

B.2.78 -xCC

-xc99=none および -xCC を指定した場合は、コンパイラで C++ 形式のコメントが認識されます。このオプションを使用すると、// を使用してコメントの始まりを示すことができます。

B.2.79 -xc99[= *o*]

-xc99 オプションは、C99 規格 (『Programming Language - C (ISO/IEC 9899:1999)』) からの実装機能に対するコンパイラの認識状況を制御します。

o には、次の項目をコンマで区切って指定します。

表 B–16 -xc99 のフラグ

フラグ	意味
[no]_lib	1990 C 規格と 1999 C 規格の両方で使用されるルーチンの 1999 C 標準ライブラリ意味処理を有効 [無効] にします。

表 B–16 -xc99 のフラグ (続き)

フラグ	意味
all	サポートされている C99 言語機能に認識可能にして、1990 C 規格と 1999 C 規格の両方にあるルーチンの 1999 C 標準ライブラリ意味処理を有効にします。
none	サポートされている C99 言語機能に認識不可にして、1990 C 規格と 1999 C 規格の両方にあるルーチンの 1999 C 標準ライブラリ意味処理を無効にします。

-xc99 を指定しない場合は、コンパイラではデフォルトで -xc99=all,no_lib が設定されます。値を指定しないで -xc99 を指定すると、オプションは -xc99=all に設定されます。

注 – コンパイラのサポートレベルは、デフォルトでは C99 規格の言語機能になりますが、Solaris 8 および Solaris 9 オペレーティングシステムが提供している標準のヘッダーファイル (/usr/include にある) は、1999 ISO/IEC C 規格に準拠していません。エラーメッセージが生成される場合は、-xc99=none を指定して、前述のヘッダー用に 1990 ISO/IEC C 規格を使用してみてください。

1990 C 規格と 1999 C 規格の両方にあるルーチンの 1999 C 標準ライブラリ意味処理は、Solaris 8 および Solaris 9 ソフトウェアで使用できず、このため有効にすることはできません。Solaris 8 または 9 で直接または間接に -xc99=lib が指定された場合は、エラーメッセージが発行されます。

B.2.80 -xcache[= *c*]

オプティマイザ用のキャッシュ特性を定義します。この定義によって、特定のキャッシュが使用されるわけではありません。

注 – このオプションは単独で指定できますが、-xtarget オプションのマクロ展開の一部です。-xtarget オプションによって指定された値を変更するのが主な目的です。

このリリースで、キャッシュを共有できるスレッド数を指定するオプションの特性 [/t*i*] が導入されました。t の値を指定しない場合のデフォルトは 1 です。

c には次のいずれかを指定します。

- generic
- native
- *s*1/ *l*1/*a* 1[/*t*1]

- *s*1/*l*1/*a* 1[/*t*1]:*s*2/*l* 2/*a*2[/*t*2]
- *s*1/*l*1/*a* 1[/*t*1]:*s*2/*l* 2/*a*2[/*t*2]:*s*3/*l* 3/*a*3[/*t*3]

s、*l*、*a*、*t* の各特性の定義は次のとおりです。

si	レベル *i* のデータキャッシュのサイズ (キロバイト単位)
li	レベル *i* のデータキャッシュのラインサイズ (バイト単位)
ai	レベル *i* のデータキャッシュの結合特性
ti	レベル *i* でキャッシュを共有するハードウェアスレッドの数

次に、-xcache の値を示します。

表 B–17 -xcache のフラグ

フラグ	意味
generic	これはデフォルトです。ほとんどの SPARC プロセッサに良好なパフォーマンスを提供し、どの x86、SPARC の各プロセッサでも著しいパフォーマンス低下が生じないようなキャッシュ特性を使用するように、コンパイラに指示します。 これらの最高のタイミング特性は、新しいリリースごとに、必要に応じて調整されます。
native	ホスト環境に対して最適化されたパラメータを設定します。
s1/*l1*/*a 1*[/*t*1]	レベル 1 のキャッシュ特性を定義します。
s1/*l1*/*a 1*[/*t*1]:*s2*/*l 2*/*a2*[/*t*2]	レベル 1 と 2 のキャッシュ特性を定義します。
s1/*l1*/*a 1*[/*t*1]:*s2*/*l 2*/*a2*[/*t*2]:*s3*/*l 3*/*a3*[/*t*3]	レベル 1、2、3 のキャッシュ特性を定義します。

例: -xcache=16/32/4:1024/32/1 では、次の内容を指定します。

レベル 1 のキャッシュ	レベル 2 のキャッシュ
16K バイト	1024K バイト
ラインサイズ 32 バイト	ラインサイズ 32 バイト
4 ウェイアソシアティブ	直接マップ結合

B.2.81 –xcg[89|92]

(SPARC) 廃止。このオプションは使わないでください。最新の Solaris オペレーティングシステムは、SPARC V7 アーキテクチャーをサポートしません。このオプションでコンパイルすると、現在の SPARC プラットフォームでの実行速度が遅いコードが生成されます。-O を使用して、-xarch、-xchip、および -xcache のコンパイラのデフォルトを利用します。

B.2.82 -xchar[= *o*]

オプションこのオプションは、char 型が符号なしで定義されているシステムからのコード移植を簡単にするためのものです。そのようなシステムからの移植以外では、このオプションは使用しないでください。符号付きまたは符号なしであると明示的に示すように書き直す必要があるのは、符号に依存するコードだけです。

o には、次のいずれかを指定します。

表 B–18 -xchar のフラグ

フラグ	意味
signed	char 型で定義された文字定数および変数を符号付きとして処理します。コンパイル済みコードの動作に影響しますが、ライブラリルーチンの動作には影響しません。
s	signed と同義です。
unsigned	char 型で定義された文字定数および変数を符号なしとして処理します。コンパイル済みコードの動作に影響しますが、ライブラリルーチンの動作には影響しません。
u	unsigned と同義です。

-xchar を指定しない場合は、コンパイラでは -xchar=s が指定されます。

-xchar を値なしで指定した場合は、コンパイラでは -xchar=s が指定されます。

-xchar オプションは、-xchar を指定してコンパイルしたコードでだけ、char 型の値の範囲を変更します。システムルーチンまたはヘッダーファイル内の char 型の値の範囲は変更しません。特に、CHAR_MAX および CHAR_MIN の値 (limits.h で定義される) は、このオプションを指定しても変更されません。したがって、CHAR_MAX および CHAR_MIN は、通常の char で符号化可能な値の範囲を示さなくなります。

-xchar を使用するときは、マクロでの値が符号付きの場合があるため、char を定義済みのシステムマクロと比較する際には特に注意してください。これは、マクロを使用してエラーコードを戻すルーチンでもっとも一般的です。エラーコードは、一般的には負の値になっています。したがって、char をそのようなマクロによる値と比較するときは、結果は常に false になります。負の数値が符号なしの型の値と等しくなることはありません。

ライブラリを使用してエクスポートしているインタフェース用のルーチンは、-xchar を使用してコンパイルしないようにお勧めします。Solaris ABI では char 型を符号付きとして指定すると、システムライブラリが指定に応じた動作をします。char を符号なしにする影響は、システムライブラリで十分にテストされたわけではありません。このオプションを使用しないで、char 型の符号の有無に依存しないようにコードを変更してください。char 型の符号は、コンパイラやオペレーティングシステムによって異なります。

B.2.83 -xchar_byte_order[= *o*]

複数文字からなる定数である文字を指定されたバイト順序で配置することにより、整定数を生成します。*o* には、次のいずれかを指定します。

- low: 複数文字定数の文字を低いバイトから順に配置する
- high: 複数文字定数の文字を高いバイトから順に配置する
- default: 複数文字定数の文字を、コンパイルモード 254 ページの「B.2.68 -X[c|a|t|s]」 で決定された順に配置する。詳細は、36 ページの「2.1.2 文字定数」を参照してください。

B.2.84 -xcheck[= *o*]

スタックオーバーフローに関する実行時検査を追加し、ローカル変数を初期化します。

o には、次のいずれかを指定します。

表 B–19 -xcheck のフラグ

フラグ	意味
%none	-xcheck のチェックを実行しません。

表 B–19 -xcheck のフラグ (続き)

フラグ	意味
%all	-xcheck のチェックをすべて実行します。
stkovf	スタックオーバーフロー検査を有効にします。-xcheck=stkovf を指定すると、シングルスレッドのプログラム内のメインスレッドのスタックオーバーフローおよびマルチスレッドプログラム内のスレーブスレッドのスタックに対する実行時検査が追加されます。スタックオーバーフローが検出された場合は、SIGSEGV が生成されます。アプリケーションで、スタックオーバーフローで生成される SIGSEGV をほかのアドレス空間違反と異なる方法で処理する必要がある場合は、sigaltstack(2) を参照してください。
no%stkovf	スタックオーバーフローのチェックをオフにします。
init_local	ローカル変数を初期化します。詳細は、次の説明を参照してください。
no%init_local	ローカル変数を初期化しません。

-xcheck を指定しない場合は、コンパイラではデフォルトで -xcheck=%none が指定されます。引数を指定せずに xcheck を使用した場合は、コンパイラではデフォルトで -xcheck=%all が指定されます。

-xcheck オプションは、コマンド行で累積されません。コンパイラは、コマンドで最後に指定したものに従ってフラグを設定します。

B.2.84.1 –xcheck=init_local の初期化値

–xcheck=init_local を指定すると、次の表に示すように、コンパイラは初期設定子を指定しないで宣言されたローカル変数を事前定義された値に初期化します (これらの値は変更される可能性があるため、信頼しないでください)。

基本型

表 B–20 基本型の init_local の初期化

種類	初期化値
char, _Bool	0x85
short	0x8001
int, long, enum (-m32)	0xff80002b
long (-m64)	0xfff00031ff800033
long long	0xfff00031ff800033

表 B–20 基本型の init_local の初期化 *(続き)*

種類	初期化値
pointer	0x00000001 (-m32)
	0x0000000000000001 (-m64)
float, float _Imaginary	0xff800001
float _Complex	0xff80000fff800011
double	SPARC: 0xfff00003ff800005
	x86: 0xfff00005ff800003
double _Imaginary	0xfff00013ff800015
long double, long double _Imaginary	SPARC: 0xffff0007ff800009 / 0xfff0000bff80000d
	x86: 12 bytes (-m32): 0x80000009ff800005 / 0x0000ffff
	x86 - 16 bytes (-m64): 0x80000009ff800005 / 0x0000ffff00000000
double _Complex	0xfff00013ff800015 / 0xfff00017ff800019
long double _Complex	SPARC: 0xffff001bff80001d / 0xfff0001fff800021 / 0xffff0023ff800025 / 0xfff00027ff800029
	x86 - 12 bytes (-m32): 0x7fffb01bff80001d / 0x00007fff / 0x7fffb023ff800025 / 0x00007fff
	x86 - 16 bytes (-m64): 0x00007fff00080000 / 0x1b1d1f2100000000 / 0x00007fff00080000 / 0x2927252300000000

計算された goto で使用するために宣言されたローカル変数 (単純な void * ポインタ) は、上の表に示されたポインタの説明に従って初期化されます。

ローカル変数の型である修飾された const、register、計算された goto のラベル番号、ローカルラベル、可変長配列 (VLA) は初期化されません。

構造体、共用体、配列の初期化

基本型である struct のフィールドは、union の最初に宣言された pointer または float フィールドのように、上の表で説明されているとおりに初期化されます。この結果、未初期化の参照により目に見えるエラーが生成される可能性が最大になります。

配列要素も上の表で説明されているとおりに初期化されます。

入れ子の struct、union、配列フィールドは、ビットフィールドを含む struct、pointer または float フィールドのない union、または完全に初期化できない型の配列を除いて、上で説明されているとおりに初期化されます。これらは、double 型のローカル変数に使用される値を使用して初期化されます。可変長配列は初期化されません。

B.2.85 -xchip[= *c*]

オプティマイザ用のプロセッサを指定します。

c には次のいずれかを指定します。generic、old、super、super2、micro、micro2、hyper、hyper2、powerup、ultra、ultra2、ultra2e、ultra2i、ultra3、ultra3cu、pentium、pentium_pro

このオプションは単独でも使用できますが、-xtarget オプションが展開されたものの一部です。このオプションの主な目的は、-xtarget オプションにより指定される値を変更することです。

このオプションは、処理対象となるプロセッサを指定することによって、タイミング特性を指定します。xchip=c は次のものに影響を与えます。

- 命令の順序 (スケジューリング)
- コンパイラが分岐を使用する方法
- 意味が同じもので代用できる場合に使用する命令

次の表は、SPARC プラットフォーム向けの -xchip の値をまとめています。

表 B–21 SPARC 向けの -xchip のフラグ

フラグ	意味
generic	ほとんどの SPARC で良好なパフォーマンスとなるタイミング特性を使用します。 これはデフォルトです。ほとんどのプロセッサに良好なパフォーマンスを提供し、どのプロセッサでも著しいパフォーマンス低下が生じないような最適のタイミング特性を使用するようにコンパイラに指示します。
native	ホスト環境に対して最適化されたパラメータを設定します。
old	SuperSPARC 以前のプロセッサのタイミング特性を使用します。
sparc64vi	SPARC64 VI プロセッサ用に最適化します。
sparc64vii	SPARC64 VII プロセッサ用に最適化します。
super	SuperSPARC プロセッサのタイミング特性を使用します。
super2	SuperSPARC II プロセッサのタイミング特性を使用します。

表 B–21 SPARC 向けの -xchip のフラグ (続き)

フラグ	意味
micro	microSPARC プロセッサのタイミング特性を使用します。
micro2	microSPARC II プロセッサのタイミング特性を使用します。
hyper	hyperSPARC プロセッサのタイミング特性を使用します。
hyper2	hyperSPARC II プロセッサのタイミング特性を使用します。
powerup	Weitek PowerUP プロセッサのタイミング特性を使用します。
ultra	UltraSPARC プロセッサのタイミング特性を使用します。
ultra2	UltraSPARC II プロセッサのタイミング特性を使用します。
ultra2e	UltraSPARC IIe プロセッサのタイミング特性を使用します。
ultra2i	UltraSPARC IIi プロセッサのタイミング特性を使用します。
ultra3	UltraSPARC III プロセッサのタイミング特性を使用します。
ultra3cu	UltraSPARC III Cu プロセッサのタイミング属性を使用します。
ultra3i	UltraSPARC IIIi プロセッサのタイミング特性を使用します。
ultra4	UltraSPARC プロセッサのタイミング特性を使用します。
ultra4plus	UltraSPARC IVplus プロセッサのタイミング特性を使用します。
ultraT1	UltraSPARC T1 プロセッサのタイミング特性を使用します。
ultraT2	UltraSPARC T2 プロセッサのタイミング特性を使用します。
ultraT2plus	UltraSPARC T2+ プロセッサのタイミング特性を使用します。
ultraT3	UltraSPARC T3 プロセッサのタイミング特性を使用します。

次の表は、x86 プラットフォーム向けの -xchip の値をまとめています。

表 B–22 x86 向けの -xchip のフラグ

フラグ	意味
generic	ほとんどの x86 アーキテクチャーで良好なパフォーマンスとなるタイミング特性を使用します。 これはデフォルトです。ほとんどのプロセッサに良好なパフォーマンスを提供し、どのプロセッサでも著しいパフォーマンス低下が生じないような最適のタイミング特性を使用するようにコンパイラに指示します。
native	ホスト環境に対して最適化されたパラメータを設定します。

表 B–22 x86 向けの -xchip のフラグ (続き)

フラグ	意味
core2	Intel Core2 プロセッサ用に最適化します。
nehalem	Intel Nehalem プロセッサ用に最適化します。
opteron	AMD Opteron プロセッサ用に最適化します。
penryn	Intel Penryn プロセッサ用に最適化します。
pentium	x86 pentium アーキテクチャーのタイミング特性を使用します。
pentium_pro	x86 pentium_pro アーキテクチャーのタイミング特性を使用します。
pentium3	x86 Pentium 3 アーキテクチャーのタイミング特性を使用します。
pentium4	x86 Pentium 4 アーキテクチャーのタイミング特性を使用します。
amdfam10	AMD AMDFAM10 プロセッサ用に最適化します。

B.2.86 -xcode[= *v*]

*(SPARC)*コードアドレス空間を指定します。

注 – -xcode=pic13 または -xcode=pic32 を指定して共有オブジェクトを構築することを推奨します。-xarch=v9 -xcode=abs64 と -xarch=v8 -xcode=abs32 を指定して有効な共有オブジェクトを構築することは可能ですが、効率は悪くなります。-xarch=v9 や -xcode=abs32 または -xarch=v9、 -xcode=abs44 を指定して構築した共有オブジェクトは機能しません。

---

v には次のいずれかを指定します。

表 B–23 -xcode のフラグ

値	意味
abs32	これは 32 ビットアーキテクチャーでのデフォルトです。32 ビットの絶対アドレスを生成します。コード、データ、および bss を合計したサイズは 2**32 バイトに制限されます。
abs44	これは 64 ビットアーキテクチャーでのデフォルトです。44 ビットの絶対アドレスを生成します。コード、データ、および bss を合計したサイズは 2**44 バイトに制限されます。64 ビットアーキテクチャーだけで利用できます。

表 B–23 -xcode のフラグ (続き)

abs64	64 ビットの絶対アドレスを生成します。64 ビットのアーキテクチャーでのみ利用できます。
pic13	共有ライブラリで使用する位置独立コードを生成します。-Kpic と同義です。32 ビットアーキテクチャーでは最大 2**11 まで、64 ビットアーキテクチャーでは最大 2**10 までの固有の外部シンボルを参照できます。
pic32	共有ライブラリで使用する位置独立コード (大規模モデル) を生成します。-KPIC と同義です。32 ビットアーキテクチャーでは最大 2**30 まで、64 ビットアーキテクチャーでは最大 2**29 までの固有の外部シンボルを参照できます。

32 ビットアーキテクチャーの場合は -xcode=abs32 です。64 ビットアーキテクチャーの場合は -xcode=abs44 です。

共有動的ライブラリを作成する場合、64 ビットアーキテクチャーでは -xcode のデフォルト値である abs44 と abs32 を使用できません。-xcode=pic13 または -xcode=pic32 を指定してください。SPARC の場合、-xcode=pic13 および -xcode=pic32 では、わずかですが、次の 2 つのパフォーマンス上の負担がかかります。

- -xcode=pic13 または -xcode=pic32 のいずれかでコンパイルしたルーチンは、エントリで命令をいくつか実行することによって、共有ライブラリの大域変数や静的変数へのアクセスに使用する大域的なオフセットテーブル (_GLOBAL_OFFSET_TABLE_) を指すようにレジスタを設定します。
- 大域または静的変数へのアクセスのたびに、_GLOBAL_OFFSET_TABLE_ を使用した間接メモリー参照が 1 回余計に行われます。-xcode=pic32 でコンパイルした場合は、大域および静的メモリーへの参照ごとに命令が 2 個増えます。

こうした負担があるとしても、-xcode=pic13 あるいは -xcode=pic32 を使用すると、ライブラリコードを共有できるため、必要となるシステムメモリーを大幅に減らすことができます。-xcode=pic13 あるいは -xcode=pic32 でコンパイルした共有ライブラリを使用するすべてのプロセスは、そのライブラリのすべてのコードを共有できます。共有ライブラリ内のコードに非 pic (すなわち、絶対) メモリー参照が 1 つでも含まれている場合、そのコードは共有不可になるため、そのライブラリを使用するプログラムを実行する場合は、その都度、コードのコピーを作成する必要があります。

.o ファイルが -xcode=pic13 または -xcode=pic32 でコンパイルされたかどうかを調べるには、次のように nm コマンドを使用すると便利です。

```
% nm file.o | grep _GLOBAL_OFFSET_TABLE_ U _GLOBAL_OFFSET_TABLE_
```

位置独立コードを含む .o ファイルには、_GLOBAL_OFFSET_TABLE_ への未解決の外部参照が含まれます。このことは、英文字の U で示されます。

-xcode=pic13 または -xcode=pic32 のどちらを使用すべきか決定するときは、elfdump -c (詳細は elfdump(1) のマニュアルページを参照) を使用することによって、セクションヘッダー (sh_name: .got) を探して、大域オフセットテーブル (Global Offset Table、GOT) のサイズを調べてください。sh_size 値が GOT のサイズです。GOT のサイズが 8,192 バイトに満たない場合は -xcode=pic13、そうでない場合は -xcode=pic32 を指定します。

一般に、-xcode の使用方法の決定に際しては、次のガイドラインに従ってください。

- 実行可能ファイルを構築する場合は、-xcode=pic13 と -xcode=pic32 のどちらも使わない。
- 実行可能ファイルへのリンク専用のアーカイブライブラリを構築する場合は、-xcode=pic13 と -xcode=pic32 のどちらも使わない。
- 共有ライブラリを構築する場合は、-xcode=pic13 からスタートし、GOT のサイズが 8,192 バイトを超えたら、-xcode=pic32 を使用する。
- 共有ライブラリへのリンク用のアーカイブライブラリを構築する場合は、-xcode=pic32 のみ使用する。

B.2.87 -xcrossfile

廃止。使用しないでください。代わりに -xipo を使用します。–xcrossfile は –xipo=1 の別名です。

B.2.88 -xcsi

C コンパイラが ISO C ソース文字コードの要件に準拠しないロケールで記述されたソースコードを受け付けられるようにします。これらのロケールには、ja_JP.PCK があります。

コンパイラの翻訳段階でこのようなロケールの処理が必要になると、コンパイルの時間が非常に長くなることがあります。そのため、このオプションを使用するのは、前述のロケールのソース文字を含むソースファイルをコンパイルする場合に限定すべきです。

コンパイラは、-xcsi が発行されないかぎり、ISO C ソース文字コードの要件に準拠しないロケールで記述されたソースコードを認識しません。

B.2.89 -xdebugformat=[stabs|dwarf]

dwarf 形式のデバッグ情報を読み取るソフトウェアを保守している場合は、-xdebugformat=dwarf を指定します。このオプションを使用すると、コンパイラは dwarf 標準形式を使用してデバッグ情報を生成します。これがデフォルトです。

表 B–24 -xdebugformat のフラグ

値	意味
stabs	-xdebugformat=stabs は、stab 標準形式を使用してデバッグ情報を生成します。
dwarf	-xdebugformat=dwarf は、dwarf 標準形式を使用してデバッグ情報を生成します (デフォルト)。

-xdebugformat を指定しない場合は、コンパイラでは -xdebugformat=dwarf が指定されます。このオプションには引数が必要です。

このオプションは、-g オプションによって記録されるデータの形式に影響します。-g を指定しなくても、一部のデバッグ情報が記録されますが、その情報の形式もこのオプションによって制御されます。したがって、-g を使用しなくても、-xdebugformat は有効です。

dbx とパフォーマンスアナライザソフトウェアは、stab 形式と dwarf 形式を両方とも認識するので、このオプションを使用しても、ツールの機能にはまったく影響を与えません。

詳細は、dumpstabs(1) および dwarfdump(1) のマニュアルページも参照してください。

B.2.90 -xdepend=[yes| no]

ループの繰り返し内部でのデータ依存性を解析し、ループの交換、ループの融合、スカラー置換などの再構成をループします。

最適化レベルが -xO3 かそれ以上に設定されている場合はすべて、-xdepend のデフォルトは -xdepend=on です。-xdepend の明示的な設定を指定すると、デフォルト設定は上書きされます。

引数なしで -xdepend を指定すると、-xdepend=yes と同等であることを意味します。

依存性の解析はシングルプロセッサシステムで役立つことがあります。ただし、シングルプロセッサシステムで -xdepend を使用する場合は、-xautopar を指定するべきではありません。-xdepend 最適化は、マルチプロセッサシステム用に実行されるからです。

B.2.91 -xdryrun

このオプションは -### のマクロです。

B.2.92 -xe

ソースファイル上で構文および意味検査のみを行います。オブジェクトコードや実行可能コードは生成しません。

B.2.93 -xF[=*v*[, *v*...]]

リンカーによる関数と変数の最適な順序の並べ替えを有効にします。

このオプションは、関数とデータ変数を細分化された別々のセクションに配置するようコンパイラに指示します。それによってリンカーは、リンカーの -M オプションで指定されたマップファイル内の指示に従ってこれらのセクションの順序を並べ替えて、プログラムのパフォーマンスを最適化することができます。通常は、この最適化によって効果が上がるのは、プログラムの実行時間の多くがページフォルト時間に費やされている場合だけです。

変数を並べ替えることによって、実行時間のパフォーマンスにマイナスの影響を与える次のような問題を解決できます。

- メモリー内で関係ない変数どうしが近接しているために生じる、キャッシュやページの競合
- 関係のある変数がメモリー内で離れた位置にあるために生じる、不必要に大きな作業セットサイズ
- 使用していない weak 変数のコピーが有効なデータ密度を低下させた結果生じる、不必要に大きな作業セットサイズ

最適なパフォーマンスを得るために変数と関数の順序を並べ替えるには、次の処理が必要です。

1. -xF によるコンパイルとリンク
2. 『プログラムのパフォーマンス解析』マニュアル内の、関数のマップファイルを生成する方法に関する指示に従うか、または、『リンカーとライブラリ』内の、データのマップファイルを生成する方法に関する指示に従います。
3. リンカーの -M オプションを使用して新しいマップファイルを再リンクします。
4. アナライザで再実行して、パフォーマンスが向上したかどうかを検証します。

B.2.93.1 値

v には、次のいずれかを指定します。

表 B–25 -xF の値

値	意味
[no%]func	関数を個々のセクションに細分化します [しません]。
[no%]gbldata	大域データ (外部リンケージのある変数) を個々のセクションに細分化します [しません]。
[no%]lcldata	大域データ (外部リンケージのある変数) を個々のセクションに細分化します [しません]。
%all	関数、大域データ、局所データを細分化します。
%none	何も細分化しません。

-xF を指定しない場合のデフォルトは、-xF=%none です。引数を指定しないで -xF を指定した場合のデフォルトは、-xF=%none,func です。

-xF=lcldata を指定するとアドレス計算最適化が一部禁止されるので、このフラグは実験として意味があるときにだけ使用するとよいでしょう。

analyzer(1)、ld(1) のマニュアルページも参照してください。

B.2.94 -xhelp=*f*

オンラインヘルプ情報を表示します。

f には、flags または readme を指定してください。

-xhelp=flags は、コンパイラオプションの要約を表示します。

-xhelp=readme は、README ファイルを表示します。

B.2.95 -xhwcprof

(SPARC) コンパイラのハードウェアカウンタによるプロファイリングのサポートを有効にします。

-xhwcprof が有効な場合、ツールがプロファイリング済み負荷を関連付け、命令をデータ型および構造メンバーと一緒に (参照先の -g で生成されたシンボリック情報と組み合わせて) 格納するために役立つ情報を、コンパイラが生成します。プロファイルデータは、ターゲットの命令空間ではなく、データ空間と関連付けられ、命令のプロファイリングだけでは入手の容易でない、動作に関する詳細情報が提供されます。

指定した一連のオブジェクトファイルは、-xhwcprof を指定してコンパイルできます。ただし、-xhwcprof がもっとも役に立つのは、アプリケーション内のすべてのオブジェクトファイルに適用したときです。このオプションによって、アプリケーションのオブジェクトファイルに分散しているすべてのメモリー参照を識別したり、関連付けたりするカバレージが提供されます。

コンパイルとリンクを別々に行う場合は、-xhwcprof をリンク時にも使用してください。将来 -xhwcprof に拡張する場合は、リンク時に -xhwcprof を使用する必要があります。表 A–2 に、コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの全一覧をまとめています。

-xhwcprof=enable または -xhwcprof=disable のインスタンスは、同じコマンド行にある以前の -xhwcprof のインスタンスをすべて無効にします。

-xhwcprof はデフォルトでは無効です。引数を指定せずに -xhwcprof と指定することは、-xhwcprof=enable と指定することと同じです。

-xhwcprof では、最適化を有効にして、デバッグのデータ形式を DWARF (-xdebugformat=dwarf) に設定する必要があります。

-xhwcprof と -g を組み合わせて使用すると、コンパイラに必要な一時ファイル記憶領域は、-xhwcprof と -g を単独で指定することによって増える量の合計を超えて大きくなります。

次のコマンドは example.c をコンパイルし、ハードウェアカウンタによるプロファイリングのサポートを指定し、DWARF シンボルを使用してデータ型と構造体メンバーのシンボリック解析を指定します。

```
example% cc -c -O -xhwcprof -g -xdebugformat=dwarf example.c
```

ハードウェアカウンタによるプロファイリングの詳細については、『プログラムのパフォーマンス解析』を参照してください。

B.2.96 -xinline=*list*

list for -xinline の書式を次に示します。[{%auto ,*func_name* ,no%*func_name* }[,{%auto,*func_name*, no%*func_name*}]...]

-xinline は、オプションのリストで指定した関数だけのインライン化を試行します。リストには、空白のリストか、func_name、no%func_name、%auto をコンマで区切ったリストを指定します。ここでの *func_name* は関係数を表します。-xinline は、-xO3 以上の最適化レベルでだけ有効です。

表 B–26 -xinline のフラグ

フラグ	意味
%auto	コンパイラでソースファイル内のすべての関数を自動的にインライン化するように指定します。%auto は、-xO4 以上の最適化レベルでだけ有効です。%auto は、-xO3 以下の最適化レベルでは無視されます。
func_name	指定した関数をコンパイラでインライン化するように指定します。
no%func_name	指定した関数をコンパイラでインライン化しないように指定します。

値のリストは、左から右に累積されます。したがって、-xinline=%auto,no%foo と指定した場合は、foo 以外のすべての関数のインラインが試行されます。-xinline=%bar,%myfunc,no%bar と指定した場合は、myfunc だけのインライン化が試行されます。

最適化レベルを -xO4 以上に設定してコンパイルすると、通常はソースファイルで定義されたすべての関数参照のインライン化が試行されます。-xinline オプションを使用することで、特定の関数をインライン化しないように制限できます。引数として関数名や %auto を指定せずに、-xinline= だけを指定した場合は、ソースファイル中のルーチンはいずれもインライン化されません。リストで func_name および no%func_name を %auto なしで指定した場合は、リストで指定した関数のオンラインオプションだけが試行されます。最適化レベルが -xO4 以上のときに、-xinline オプションのリストで %auto を指定した場合は、no%func_name で明示的に除外していない関数がすべてインライン化されます。

次のいずれかの条件に該当する場合、ルーチンはインライン化されません。警告は出力されませんので注意してください。

- 最適化のレベルが -xO3 未満である。
- ルーチンが見つからない。
- iropt がルーチンのインライン化を実行できない。
- ルーチンのソースが、コンパイル対象のファイルにない (-xcrossfile を参照)。

コマンド行で -xinline を複数指定した場合は、それらは累積されません。コマンド行で最後に指定した -xinline によって、コンパイラがインライン化する関数が決定されます。

-xldscope も参照してください。

B.2.97 -xinstrument=[no%]datarace

スレッドアナライザで分析するためにプログラムをコンパイルして計測するには、このオプションを指定します。スレッドアナライザの詳細については、tha(1) を参照してください。

そうすることで、パフォーマンスアナライザを使用して計測されたプログラムを collect -r races で実行し、データ競合の検出実験を行うことができます。計測されたコードをスタンドアロンで実行できますが、低速で実行されます。

-xinstrument=no%datarace を指定して、スレッドアナライザ用のソースコードの準備をオフにすることができます。これはデフォルト値です。

-xinstrument を引数なしで指定することはできません。

コンパイルとリンクを別々に行う場合は、コンパイル時とリンク時の両方で -xinstrument=datarace を指定する必要があります。

このオプションは、プリプロセッサトークン __THA_NOTIFY を定義します。#ifdef __THA_NOTIFY を指定して、libtha(3) ルーチンへの呼び出しを保護できます。

このオプションにも、-g を設定します。

B.2.98 -xipo[= *a*]

a を 0、 1、または 2 と置き換えます。引数なしの -xipo は、 -xipo=1 に相当します。-xipo=0 はデフォルト設定であり、-xipo を無効にします。-xipo=1 を指定した場合は、すべてのソースファイルでインライン化が実行されます。

-xipo=2 を指定した場合は、コンパイラは内部手続きの別名解析と、メモリーの割り当ておよび配置の最適化を実行し、キャッシュ性能を向上させます。

このコンパイラは、内部手続き解析コンポーネントを呼び出すことにより、プログラムの一部の最適化を実行します。このオプションを指定すると、リンク段階ですべてのオブジェクトファイルを介して最適化を実行し、最適化の対象をコンパイルコマンドのソースファイルだけに限定しません。ただし、-xipo によるプログラム全体の最適化には、アセンブリ (.s) ソースファイルは含まれません。

-xipo は、コンパイル時とリンク時の両方で指定する必要があります。表 A–2 に、コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの全一覧をまとめています。

-xipo オプションは、ファイルを介して最適化を実行する際に必要な情報を追加するため、非常に大きなオブジェクトファイルを生成します。ただし、この補足情報は最終的な実行可能バイナリファイルの一部にはなりません。実行可能プログラムのサイズが拡大するのは、最適化が追加実行されるためです。このコンパイル段階で作成されたオブジェクトファイルには、内部でコンパイルされた追加の分析情報が含まれているため、リンク段階でファイル相互の最適化を実行できます。

-xipo は、大きなマルチファイルアプリケーションをコンパイルおよびリンクする際に便利です。このフラグでコンパイルされたオブジェクトファイルは、それらのファイル内でコンパイルされた解析情報を保持します。これらの解析情報は、ソースおよびコンパイル前のプログラムファイルで内部手続き解析を可能にします。

ただし、解析と最適化は、-xipo を指定してコンパイルされたオブジェクトファイルに限定され、オブジェクトファイルまたはライブラリは対象外です。

-xipoは複数の段階に分かれているため、コンパイルとリンクを個別に実行する場合、各ステップで -xipo を指定する必要があります。

-xipo に関するそのほかの重要な情報を次に示します。

- 少なくとも最適化レベル -xO4 を必要とします。
- -xipo なしでコンパイルされたオブジェクトは、-xipo でコンパイルされたオブジェクトと自由にリンクできます。

B.2.98.1 例

次の例では、コンパイルとリンクが単一ステップで実行されます。

```
cc -xipo -xO4 -o prog part1.c part2.c part3.c
```

オプティマイザは 3 つのすべてのソースファイル間でファイル間のインライン化を実行します。これは最終的なリンク手順で実行されるため、すべてのソースファイルのコンパイルを単一のコンパイル処理で実行する必要はありません。-xipo を随時指定することにより、個別のコンパイルが多数発生してもかまいません。

次の例では、個別のステップでコンパイルとリンクが実行されます。

```
cc -xipo -xO4 -c part1.c part2.c
cc -xipo -xO4 -c part3.c
cc -xipo -xO4 -o prog part1.o part2.o part3.o
```

ここでの制限事項は、-xipo でコンパイルを実行しても、ライブラリがファイル相互の内部手続き解析に含まれない点です。次の例を参照してください。

```
cc -xipo -xO4 one.c two.c three.c
ar -r mylib.a one.o two.o three.o
...
cc -xipo -xO4 -o myprog main.c four.c mylib.a
```

この例では、内部手続きの最適化は one.c、two.c および three.c 間と main.c と four.c 間で実行されますが、main.c または four.c と mylib.a のルーチン間では実行されません。最初のコンパイルは未定義のシンボルに関する警告を生成する場合がありますが、内部手続きの最適化は、コンパイル手順でありしかもリンク手順であるために実行されます。

B.2.98.2 -xipo=2 による内部手続き解析を行うべきでないケース

内部手続き解析では、コンパイラは、リンクステップでオブジェクトファイル群を操作しながら、プログラム全体の解析と最適化を試みます。このとき、コンパイラは、このオブジェクトファイル群に定義されているすべての foo() 関数 (またはサブルーチン) に関して次の 2 つのことを仮定します。

- 実行時、このオブジェクトファイル群の外部で定義されている別のルーチンによって foo() が明示的に呼び出されない。
- オブジェクトファイル群内のルーチンからの foo() 呼び出しが、そのオブジェクトファイル群の外部に定義されている別のバージョンの foo() からの割り込みを受けない。

仮定 2 が当てはまらない場合は、コンパイルで -xipo=1 および -xipo=2 を使わないでください。

1 例として、独自のバージョンの malloc() で関数 malloc() を置き換え、-xipo=2 を指定してコンパイルするケースを考えてみましょう。-xipo=2 を使ってコンパイルする場合、その独自のコードとリンクされる malloc() を参照する、あらゆるライブラリのあらゆる関数のコンパイルで -xipo=2 を使用する必要があるとともに、リンクステップでそれらのオブジェクトファイルが必要になります。システムライブラリでは、このことが不可能なことがあり、このため、独自のバージョンの malloc のコンパイルに -xipo=2 を使ってはいけません。

もう 1 つの例として、別々のソースファイルにある foo() および bar() という 2 つの外部呼び出しを含む共有ライブラリを構築するケースを考えてみましょう。また、bar() は foo() を呼び出すと仮定します。foo() が実行時に割り込みを受ける可能性がある場合、foo() および bar() のソースファイルのコンパイルで -xipo=1 や -xipo=2 を使ってはいけません。さもないと、foo() が bar() 内にインライン化され、不正な結果になる可能性があります。

B.2.99 -xipo_archive=[*a*]

新しい -xipo_archive オプションは、リンカーに渡すオブジェクトファイル、-xipo を指定してコンパイルされたオブジェクトファイル、および実行可能ファイルを生成する前にアーカイブライブラリ (.a) に存在するオブジェクトファイルを最適化できるようにします。コンパイル中に最適化されたライブラリに含まれるオブジェクトファイルはすべて、その最適化されたバージョンに置き換えられます。

a には、次のいずれかを指定します。

表 B–27 -xipo_archive のフラグ

値	意味
writeback	実行可能ファイルを生成する前に、アーカイブライブラリ (.a) に存在する -xipo でコンパイルしたオブジェクトファイルを使ってリンカーに渡すオブジェクトファイルを最適化します。コンパイル中に最適化されたライブラリに含まれるオブジェクトファイルは、すべてその最適化されたバージョンに置き換えられます。 アーカイブライブラリの共通セットを使用する並列リンクでは、最適化されるアーカイブライブラリの独自のコピーを、各リンクでリンク前に作成する必要があります。
readonly	実行可能ファイルを生成する前に、アーカイブライブラリ (.a) に存在する -xipo でコンパイルしたオブジェクトファイルを使ってリンカーに渡すオブジェクトファイルを最適化します。 -xipo_archive=readonly オプションを指定すると、リンク時に指定されたアーカイブライブラリのオブジェクトファイルで、モジュール間のインライン化と内部手続きデータフロー解析が有効になります。ただし、モジュール間のインライン化によってほかのモジュールに挿入されたコード以外のアーカイブライブラリのコードに対する、モジュール間の最適化は有効になりません。 アーカイブライブラリ内のコードにモジュール間の最適化を適用するには、-xipo_archive=writeback を指定する必要があります。これを行うと、コードが抽出されたアーカイブライブラリの内容が変更されることがあります。
none	これはデフォルト値です。アーカイブファイルの処理は行いません。コンパイラは、モジュール間のインライン化やその他のモジュール間の最適化を、-xipo を使用してコンパイルされ、リンク時にアーカイブライブラリから抽出されたオブジェクトファイルに適用しません。これを行うには、-xipo と、-xipo_archive=readonly または -xipo_archive=writeback のいずれかをリンク時に指定する必要があります。

-xipo_archive の値が指定されない場合、-xipo_archive=none に設定されます。

-xipo_archive をフラグなしで指定することはできません。

B.2.100 -xjobs=*n*

コンパイラが処理を行うために生成するプロセスの数を設定するには、-xjobs オプションを指定します。このオプションを使用すると、マルチ CPU マシン上での構築時間を短縮できます。現時点では、-xjobs とともに使用できるのは -xipo オプションだけです。-xjobs=*n* を指定すると、内部手続きオプティマイザは、さまざまなファイルをコンパイルするために呼び出すことができるコードジェネレータインスタンスの最大数として、*n* を使用します。

一般に、*n* に指定する確実な値は、使用できるプロセッサ数に 1.5 を掛けた数です。生成されたジョブ間のコンテキスト切り替えにより生じるオーバーヘッドのため、使用できるプロセッサ数の何倍もの値を指定すると、パフォーマンスが低下することがあります。また、あまり大きな数を使用すると、スワップ領域などシステムリソースの限界を超える場合があります。

`-xjobs` には必ず値を指定する必要があります。値を指定しないと、エラー診断が表示され、コンパイルは中止します。

コマンド行に複数の `-xjobs` のインスタンスがある場合、一番右にあるインスタンスが実行されるまで相互に上書きします。

次の例に示すコマンドは 2 つのプロセッサを持つシステム上で、`-xjobs` オプションを指定しないで実行された同じコマンドよりも高速にコンパイルを実行します。

```
example% cc -xipo -xO4 -xjobs=3 t1.c t2.c t3.c
```

`-xipo_archive` をフラグなしで指定することはできません。

B.2.101 -xldscope={*v*}

extern シンボルの定義に対するデフォルトのリンカースコープを変更するには、`-xldscope` オプションを指定します。デフォルトを変更すると、実装がよりうまく隠されるので、より早く、より安全に共有ライブラリを使用できます。

v には、次のいずれかを指定します。

表 B–28 `-xldscope` のフラグ

フラグ	意味
`global`	大域リンカースコープは、もっとも制限の少ないリンカースコープです。シンボルに対する参照はすべて、シンボルを定義する最初の動的モジュール内の定義に結合します。このリンカースコープは、extern シンボルの現在のリンカースコープです。
`symbolic`	シンボリックリンカースコープは、大域リンカースコープよりも制限されたスコープです。リンクしている動的モジュール内のシンボルに対する参照はすべて、モジュール内に定義されたシンボルに結合します。モジュールの外側では、シンボルは大域と同じです。このリンカースコープはリンカーオプション `-Bsymbolic` に対応します。リンカーの詳細については、`ld`(1) を参照してください。

表 B–28 -xldscope のフラグ (続き)

フラグ	意味
hidden	隠蔽リンカースコープは、シンボリックリンカースコープや大域リンカースコープよりも制限されたリンカースコープです。動的モジュール内の参照はすべて、そのモジュール内の定義に結合します。シンボルはモジュールの外側では認識されません。

-xldscope を指定しない場合は、コンパイラでは -xldscope=global が指定されます。引数を指定しないで -xldscope を指定すると、エラーが表示されます。コマンド行にこのオプションの複数のインスタンスがある場合、一番右にあるインスタンスが実行されるまで相互に上書きします。

クライアントがライブラリ内の関数をオーバーライドできるようにする場合は必ず、ライブラリの構築時に関数がインラインで生成されないようにしてください。-xinline を指定して関数名を指定した場合、-xO4 以上でコンパイルした場合 (自動的にインライン化される)、インライン指示子を使用した場合、インラインプラグマを使用した場合、または、複数のソースファイルにわたる最適化を使用している場合、コンパイラは関数をインライン化します。

たとえば、ABC というライブラリにデフォルトの allocator 関数があり、ライブラリクライアントがその関数を使用でき、ライブラリの内部でも使用されるものとします。

```
void* ABC_allocator(size_t size) { return malloc(size); }
```

-xO4 以上でライブラリを構築すると、コンパイラはライブラリ構成要素内での ABC_allocator の呼び出しをインライン化します。ライブラリクライアントが ABC_allocator を独自の allocator と置き換える場合、置き換えられた allocator は ABC_allocator を呼び出したライブラリ構成要素内では実行されません。最終的なプログラムには、この関数の相異なるバージョンが含まれることになります。

__hidden 指示子または __symbolic 指示子で宣言されたライブラリ関数は、ライブラリの構築時にインラインで生成することができます。これらの関数がクライアントからオーバーライドされることは、サポートされていません。37 ページの「2.2 リンカースコープ指示子」を参照してください。

__global 指示子で宣言されたライブラリ関数はインラインで宣言しないでください。また、-xinline コンパイラオプションを使用することによってインライン化されることがないようにしてください。

-xinline、-xO、-xcrossfile、#pragma inline も参照してください。

B.2.102 -xlibmieee

例外が起きた場合の数学ルーチンの戻り値を強制的に IEEE 754 形式にします。この場合、例外メッセージは表示されないので、errno には依存しないでください。

B.2.103 -xlibmil

実行速度を上げるため、一部のライブラリルーチンをインライン化します。オプションによって浮動小数点演算用オプションとプラットフォームに適したアセンブリ言語のインラインテンプレートが選択されます。

-xlibmil は、-xinline フラグで関数を指定しているかどうかに関係なく、関数をインライン化します。

ただし、こうした置換によって errno の値の信頼性が失われることがあります。プログラムが errno の値に依存している場合、このオプションの使用は避けてください。58 ページの「2.13 errno の値の保持」も参照してください。

B.2.104 -xlibmopt

このオプションによって、コンパイラは最適化された数学ルーチンのライブラリを利用できます。このオプションを使用するときは -fround=nearest を指定することによって、デフォルトの丸めモードを使用する必要があります。

数学ルーチンライブラリは最高のパフォーマンスが得られるように最適化されており、通常、高速なコードを生成します。この結果は、通常の数学ライブラリが生成する結果と少し異なることがあります。その場合、通常、異なるのは最後のビットです。

ただし、こうした置換によって errno の値の信頼性が失われることがあります。プログラムが errno の値に依存している場合、このオプションの使用は避けてください。58 ページの「2.13 errno の値の保持」も参照してください。

このライブラリオプションをコマンド行に指定する順序は重要ではありません。

このオプションは -fast オプションを指定した場合にも設定されます。

関連項目: -fast -xnolibmopt

B.2.105 -xlic_lib=sunperf

指定された Solaris Studio 提供のパフォーマンスライブラリにリンクします。

B.2.106 -xlicinfo

このオプションは、コンパイル時には暗黙的に無視されます。

B.2.107 -xlinkopt[= レベル]

(SPARC) 再配置可能なオブジェクトファイルのリンク時の最適化を実行するようコンパイラに指示します。このような最適化は、リンク時にオブジェクトのバイナリコードを解析することによって実行されます。オブジェクトファイルは書き換えられませんが、最適化された実行可能コードは元のオブジェクトコードとは異なる場合があります。

`-xlinkopt` をリンク時に有効にするには、少なくともコンパイルコマンドで `-xlinkopt` を使用する必要があります。`-xlinkopt` を指定しないでコンパイルされたオブジェクトバイナリについても、オプティマイザは限定的な最適化を実行できます。

`-xlinkopt` は、コンパイラのコマンド行にある静的ライブラリのコードは最適化しますが、コマンド行にある共有 (動的) ライブラリのコードは最適化しません。共有ライブラリを構築 (-G でコンパイル) する場合は、`-xlinkopt` も使用できます。

level には、実行する最適化のレベルを 0、1、2 のいずれかで設定します。最適化レベルは、次のとおりです。

表 B–29 `-xlinkopt` のフラグ

フラグ	意味
`0`	ポストオプティマイザは無効です。これがデフォルトです。
`1`	リンク時の命令キャッシュカラーリングと分岐の最適化を含む、制御フロー解析に基づき最適化を実行します。
`2`	リンク時のデッドコードの除去とアドレス演算の簡素化を含む、追加のデータフロー解析を実行します。

コンパイル手順とリンク手順を別々にコンパイルする場合は、両方の手順に `-xlinkopt` を指定する必要があります。

```
example% cc -c -xlinkopt a.c b.c
example% cc -o myprog -xlinkopt=2 a.o
```

表 A–2 に、コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの全一覧をまとめています。

レベルパラメータは、コンパイラのリンク時にだけ使用されます。前述の例では、オブジェクトバイナリが指定された 1 のレベルでコンパイルされていても、使用される最適化後のレベルは 2 です。

レベルパラメータなしで -xlinkopt を使用することは、-xlinkopt=1 を指定することと同じです。

このオプションは、プログラム全体のコンパイル時に、プロファイルのフィードバックとともに使用されると、もっとも効果的です。プロファイリングによって、コードでもっともよく使用される部分と、もっとも使用されない部分が明らかになるので、それに基づき処理を集中するよう、構築はオプティマイザに指示します。これは、リンク時に実行されるコードの最適な配置が命令のキャッシュミスを低減できるような、大きなアプリケーションにとって特に重要です。このようなコンパイルの例を次に示します。

```
example% cc -o progt -xO5 -xprofile=collect:prog file.c
example% progt
example% cc -o prog -xO5 -xprofile=use:prog -xlinkopt file.c
```

プロファイルフィードバックの使用方法についての詳細は、312 ページの「B.2.136 –xprofile=*p*」を参照してください。

-xlinkopt を指定してコンパイルする場合は、-zcombreloc リンカーオプションは指定しないでください。

このオプションを指定してコンパイルすると、リンク時間がわずかに増えます。オブジェクトファイルも大きくなりますが、実行可能ファイルのサイズは変わりません。-xlinkopt と -g を指定してコンパイルすると、デバッグ情報が取り込まれるので、実行可能ファイルのサイズが増えます。

B.2.108 -xloopinfo

並列化されているループとされていないループを示します。また、ループを並列化しない理由を簡単に説明します。-xloopinfo オプションは、-xautopar が指定されている場合にのみ有効です。指定されていない場合は、コンパイラによって警告が表示されます。

コードの実行速度を上げるには、このオプションにマルチプロセッサシステムが必要です。シングルプロセッサシステムでは、通常、生成されたコードの実行速度は低下します。

B.2.109 -xM

指定した C プログラムに対してプリプロセッサだけを実行します。その際、メイクファイルの依存関係を生成してその結果を標準出力に出力します。make ファイルと依存関係についての詳細は make(1) のマニュアルページを参照してください。

次に例を示します。

```
#include <unistd.h>
void main(void)
{}
```

この例で出力されるものは、次のとおりです。

```
e.o: e.c
e.o: /usr/include/unistd.h
e.o: /usr/include/sys/types.h
e.o: /usr/include/sys/machtypes.h
e.o: /usr/include/sys/select.h
e.o: /usr/include/sys/time.h
e.o: /usr/include/sys/types.h
e.o: /usr/include/sys/time.h
e.o: /usr/include/sys/unistd.h
```

-xM と -xMF を指定する場合、-xMF で指定したファイルに、コンパイラはすべてのメイクファイルの依存関係情報を追加します。

B.2.110 -xM1

-xM と同様にメイクファイルの依存関係を生成しますが、/usr/include ファイルは除きます。次に例を示します。

```
more hello.c
#include<stdio.h>
main()
{
    (void)printf(“hello\n”);
}
cc– xM hello.c
hello.o: hello.c
hello.o: /usr/include/stdio.h
```

-xM1 オプションを使用してコンパイルすると、ヘッダーファイルの依存関係の出力が抑制されます。

```
cc– xM1 hello.c
hello.o: hello.c
```

-Xs モードでは -xM1 は使用できません。

-xM1 と -xMF を指定する場合、-xMF で指定したファイルに、コンパイラはすべてのメイクファイルの依存関係情報を追加します。

B.2.111 -xMD

-xM と同様にメイクファイルの依存関係を生成しますが、コンパイルを続行します。**-xMD** は、指定されている場合は -o 出力ファイル名から派生したメイクファイルの依存関係情報の出力ファイルを生成します。または、ファイル名接尾辞を **.d** に置

換 (または追加) して、入力元ファイル名を生成します。**-xMD** と **-xMF** を指定する場合、**-xMF** で指定したファイルに、プリプロセッサはすべてのメイクファイルの依存関係情報を書き込みます。複数のソースファイルで **-xMD -xMF** または **-xMD -o** *filename* を使用してコンパイルすることはできず、エラーが生成されます。依存関係ファイルがすでに存在する場合は上書きされます。

B.2.112 -xMF *filename*

メイクファイルの依存関係の出力先ファイルを指定するには、このオプションを使用します。コマンド行で **-xMF** を使用して、複数の入力ファイルのファイル名を個別に指定することはできません。複数のソースファイルで **-xMD -xMF** または **-xMMD -xMF** を使用してコンパイルすることはできず、エラーが生成されます。依存関係ファイルがすでに存在する場合は上書きされます。

B.2.113 -xMMD

システムヘッダーファイルを除き、メイクファイルの依存関係を生成するには、このオプションを使用します。これは、**-xM1** と同様の機能ですが、コンパイルが続行されます。**-xMMD** は、指定されている場合は **-o** 出力ファイル名から派生したメイクファイルの依存関係情報の出力ファイルを生成します。または、ファイル名接尾辞を **.d** に置換 (または追加) して、入力元ファイル名を生成します。**-xMF** を指定する場合、コンパイラは代わりに、ユーザーが指定したファイル名を使用します。複数のソースファイルで **-xMMD -xMF** または **-xMMD -o** *filename* を使用してコンパイルすることはできず、エラーが生成されます。依存関係ファイルがすでに存在する場合は上書きされます。

B.2.114 -xMerge

データセグメントをテキストセグメントにマージします。このコンパイルで生成するオブジェクトファイルで初期化されるデータは読み取り専用なので、`ld -N` でリンクしていないかぎり、プロセスどうしで共有することができます。

3 つのオプション **-xMerge -ztext -xprofile=collect** を一緒に使用するべきではありません。**-xMerge** を指定すると、静的に初期化されたデータを読み取り専用記憶領域に強制的に配置します。 **-ztext** を指定すると、位置に依存するシンボルを読み取り専用記憶領域内で再配置することを禁止します。**-xprofile=collect** を指定すると、書き込み可能記憶領域内で、静的に初期化された、位置に依存するシンボルの再配置を生成します。

B.2.115 -xmaxopt[=*v*]

このコマンドは、pragma opt のレベルを指定されたレベルに制限します。*v* は、off、1、2、3、4、5 のいずれかです。デフォルト値は -xmaxopt=off であり、pragma opt は無視されます。引数を指定せずに -xmaxopt を指定すると、-xmaxopt=5 を指定したことになります。

-xO と -xmaxopt の両方を指定する場合、-xO で設定する最適化レベルが -xmaxopt 値を超えてはいけません。

B.2.116 -xmemalign=*ab*

(SPARC) データの境界整列についてコンパイラが使用する想定を制御するには、-xmemalign オプションを使用します。境界整列が潜在的に正しくないメモリーアクセスにつながる生成コードを制御し、境界整列が正しくないアクセスが発生したときのプログラム動作を制御すれば、より簡単に SPARC にコードを移植できます。

想定するメモリー境界整列の最大値と、境界整列に失敗したデータがアクセスされた際の動作を指定します。*a* (境界整列) と *b* (動作) の両方の値が必要です。*a* は、想定する最大メモリー境界整列です。*b* は、境界整列に失敗したメモリーへのアクセスに対する動作です。次に、-memalign の境界整列と動作の値を示します。

表 B–30 -xmemalign の境界整列と動作のフラグ

a		*b*	
1	最大 1 バイトの境界整列	i	アクセスを解釈し、実行を継続する
2	最大 2 バイトの境界整列	s	シグナル SIGBUS を発生させる
4	最大 4 バイトの境界整列	f	-xarch=v9 の不変式の場合にのみ、 4 バイト以下の境界整列に対してシグナル SIGBUS を発生させ、それ以外ではアクセスを解釈して実行を継続する。そのほかすべての -xarch 値では、f フラグは i と同じです。
8	最大 8 バイトの境界整列		
16	最大 16 バイトの境界整列		

b を i か f のいずれかに設定してコンパイルしたオブジェクトファイルにリンクする場合は、必ず、-xmemalign を指定する必要があります。表 A–2 に、コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの全一覧をまとめています。

コンパイル時に境界整列が判別できるメモリーへのアクセスの場合、コンパイラはそのデータの境界整列に適したロードおよびストア命令を生成します。

境界整列がコンパイル時に決定できないメモリーアクセスの場合、コンパイラは、境界整列を想定して、必要なロード/ストア命令のシーケンスを生成します。

-xmemalign オプションを使用すると、このような判別不可能な状況の時にコンパイラが想定するデータの最大メモリー境界整列を指定できます。-xmemalign オプションは、境界整列に失敗したメモリーへのアクセスが実行時に発生した場合に行われるエラー動作 (処理) についても指定できます。

実行時の実際のデータ境界整列が指定された整列に達しない場合、境界整列に失敗したアクセス (メモリー読み取りまたは書き込み) が行われると、トラップが発生します。このトラップに対して可能な応答は 2 つあります。

- OS がトラップを SIGBUS シグナルに変換します。プログラムがこのシグナルを捕捉しなかった場合、プログラムは異常終了します。プログラムがシグナルを捕捉しても、境界整列に失敗したアクセスが成功するわけではありません。
- 境界整列に失敗したアクセスが正常に成功したかのように OS がアクセスを解釈し、プログラムに制御を戻すことによってトラップを処理します。

次のデフォルトの値は、 -xmemalign オプションがまったく指定されていない場合にのみ適用されます。

- -xmemalgin=8i: すべての v8 アーキテクチャーに適用される。
- -xmemalign=8s すべての v9 アーキテクチャーに適用される。

次に、-xmemalign オプションが指定されているが値を持たない場合のデフォルト値を示します。

- -xmemalign=1i すべての -xarch 値に適用される。

次の表は、-xmemalign で処理できるさまざまな境界整列の状況とそれに適した -xmemalign 指定を示しています。

表 B–31 -xmemalign の例

コマンド	状況
-xmemalign=1s	境界整列されていないデータへのアクセスが多いため、トラップ処理が遅すぎる
-xmemalign=8i	コード内に境界整列されていないデータへのアクセスが意図的にいくつか含まれているが、それ以外は正しい
-xmemalign=8s	プログラム内に境界整列されていないデータへのアクセスは存在しないと思われる
-xmemalign=2s	奇数バイトへのアクセスが存在しないか検査したい

表 B–31 -xmemalign の例 (続き)

コマンド	状況
-xmemalign=2i	奇数バイトへのアクセスが存在しないか検査し、プログラムを実行したい

B.2.117 -xmodel=[*a*]

(x86) -xmodel オプションを使用すると、コンパイラで 64 ビットオブジェクトの形式を Solaris x86 プラットフォーム用に変更できます。このオプションは、そのようなオブジェクトのコンパイル時にのみ指定してください。

このオプションは、64 ビット対応の x64 プロセッサで -m64 も指定した場合にのみ有効です。

a には次のいずれかを指定します。

表 B–32 -xmodel のフラグ

値	意味
small	このオプションは、実行されるコードの仮想アドレスがリンク時にわかっていて、すべてのシンボルが 0 〜 2^31 - 2^24 - 1 の範囲の仮想アドレスに配置されることがわかっているスモールモデルのコードを生成します。
kernel	すべてのシンボルが 2^64 - 2^31 〜 2^64 - 2^24 の範囲で定義されるカーネルモデルのコードを生成します。
medium	データセクションへのシンボリック参照の範囲に関する前提がないミディアムモデルのコードを生成します。テキストセクションのサイズとアドレスは、スモールコードモデルの場合と同じように制限されます。静的データが大量にあるアプリケーションでは、-m64 を指定してコンパイルするときに、-xmodel=medium が必要になることがあります。

このオプションは累積的ではないため、コンパイラはコマンド行でもっとも右の -xmodel に従って、モデルの値を設定します。

-xmodel を指定しない場合、コンパイラは -xmodel=small とみなします。引数を指定せずに -xmodel を指定すると、エラーになります。

すべての翻訳単位をこのオプションでコンパイルする必要はありません。アクセスするオブジェクトが範囲内にあれば、選択したファイルをコンパイルできます。

すべての Linux システムが、ミディアムモデルをサポートしているわけではありません。

B.2.118 -xnolib

デフォルトのライブラリリンクを行いません。つまり ld(1) に -l オプションを渡しません。通常は、cc ドライバが -lc を ld に渡します。

-xnolib を使用する場合、すべての -l オプションをユーザーが渡さなければいけません。

B.2.119 -xnolibmil

数学ライブラリのルーチンをインライン化しません。このオプションは -fast オプションのあとに指定してください。次に例を示します。

```
% cc– fast– xnolibmil....
```

B.2.120 -xnolibmopt

前に -xlibmopt オプションが指定されている場合にその指定を無効にして、最適化された数学ライブラリをコンパイラが使用しないようにします。このオプションは、-xlibmopt を設定することによって最適化された数学ライブラリを使用可能にする -fast のあとで使用してください。

```
% cc -fast -xnolibmopt ...
```

B.2.121 -xnorunpath

実行可能ファイルに共有ライブラリへの実行時検索パスを組み込みません。

通常、cc は -R パスをリンカーにまったく渡しません。オプションの中には、-xliclib=sunperf および -xopenmp のように、-R パスをリンカーに渡すものがあります。-xnorunpath を使用すると、パスが渡されなくなります。

このオプションは、プログラムで使用される共有ライブラリへのパスが異なる顧客に出荷される実行可能ファイルの構築にお勧めします。

B.2.122 -xO[1|2| 3|4|5]

オブジェクトコードを最適化します。大文字 O のあとに数字の 1、2、3、4、5 のいずれかが続きます。一般に、最適化レベルが高いほど、実行時のパフォーマンスは向上します。しかし、最適化レベルが高ければ、それだけコンパイル時間が増え、実行可能ファイルが大きくなる可能性があります。

ごくまれに、-xO2 の方がほかの値より実行速度が速くなることがあり、-xO3 の方が -xO4 より早くなることがあります。すべてのレベルでコンパイルを行なってみて、こうしたことが発生するかどうか試してみてください。

オプティマイザによりメモリーが不足した場合は、最適化のレベルを下げて現在の関数を再試行することによって処理を続行しようとします。これ以後の関数に対してはコマンド行オプションで指定されている本来のレベルで再開します。

デフォルトでは最適化は行われません。ただし、これは最適化レベルを指定しない場合にかぎり有効です。最適化レベルを指定すると、最適化を無効にするオプションはありません。

最適化レベルを設定しないようにする場合は、最適化レベルを示すようなオプションを指定しないようにしてください。たとえば、-fast は最適化を -xO5 に設定するマクロオプションです。それ以外で最適化レベルを指定するすべてのオプションでは、最適化レベルが設定されたという警告メッセージが表示されます。最適化を設定せずにコンパイルする唯一の方法は、コマンド行またはメイクファイルから最適化レ ベルを指定するオプションをすべて削除することです。

-xO3 以下の最適化レベルで -g を指定すると、ほぼ完全な最適化と可能なかぎりのシンボル情報を取得することができます。末尾呼び出しの最適化とバックエンドのインライン化は無効です。

-xO4 以上の最適化レベルで -g を指定すると、完全な最適化と可能なかぎりのシンボル情報を取得することができます。

-g を指定したデバッグでは、-xO*n* が抑制されませんが、-xO*n* はいくつかの方法で -g を制限します。たとえば、最適化オプションを使用すると、dbx から渡された変数を表示できないなど、デバッグの機能が一部制限されます。しかし、dbx where コマンドを使用して、シンボリックトレースバックを表示することは可能です。詳細は、『dbx コマンドによるデバッグ』の第 1 章の「最適化コードのデバッグ」を参照してください。

-xO と -xmaxopt の両方を指定する場合、-xO で設定する最適化レベルが -xmaxopt 値を超えてはいけません。

-xO3 または -xO4 で (1 つの関数内のコードが数千行になるような) 大きな関数を最適化する場合、膨大な量の仮想メモリーが必要になり、マシンのパフォーマンスが低下することがあります。

B.2.122.1 SPARC での最適化について

次の表は、SPARC プラットフォームでの最適化処理について説明しています。

表 B–33 SPARC プラットフォームでの -x0 のフラグ

値	意味
-x01	最小限の局所的な最適化 (ピープホール) を行います。
-x02	基本的な局所的および大域的最適化を行います。ここでは帰納変数の削除、局所的および大域的な共通部分式の除去、算術の簡素化、コピー通達、定数通達、不変ループの最適化、レジスタの割り当て、基本ブロックのマージ、再帰的末尾の除去、無意味なコードの除去、末尾呼び出しの削除、複雑な式の展開を行います。 -x02 レベルでは、大域、外部、間接の参照または定義はレジスタに割り当てられません。これらの参照や定義は、あたかも volatile 型として宣言されたかのように取り扱われます。一般的 -x02 レベルではコードサイズはもっとも小さくなります。
-x03	-x02 に加えて、外部変数の参照または定義も最適化します。ループの展開やソフトウェアのパイプラインなども実行されます。このレベルでは、ポインタ代入の影響は追跡されません。デバイスドライバをコンパイルするとき、またはシグナルハンドラの内部から外部変数を変更するプログラムをコンパイルするときは、volatile 型の修飾子を使用してオブジェクトが最適化されないようにする必要があります。一般的に -x03 レベルではコードサイズが増大します。
-x04	-x03 に加えて、同一のファイルに含まれている関数の自動的なインライン化も行います。通常はこれによって実行速度が上がります。インライン化される関数を制御したい場合は、280 ページの「B.2.96 -xinline=*list*」を参照してください。 このレベルでは、ポインタ代入の結果が追跡され、通常はコードサイズが増大します。
-x05	最高レベルの最適化を行おうとします。この最適化アルゴリズムは、コンパイルの所要時間が長く、また実行時間が確実に短縮される保証がありません。このレベルの最適化によってパフォーマンスが改善される確率を高くするには、プロファイルのフィードバックを使用します。詳細は、312 ページの「B.2.136 –xprofile=*p*」を参照してください。

B.2.122.2 x86 での最適化について

次の表は、x86 プラットフォームでの最適化処理について説明しています。

表 B–34 x86 プラットフォームでの -x0 のフラグ

値	意味
-x01	デフォルトの最適化での 1 つの段階のほかに、メモリーからの引数の事前ロードと、クロスジャンプ (末尾融合) を行います。
-x02	高レベルと低レベルの両方の命令をスケジュールし、改良されたスピルコードの解析、ループ中のメモリー参照の除去、レジスタの寿命解析、高度なレジスタ割り当て、大域的な共通部分式の除去を行います。
-x03	レベル 2 で行う最適化のほかに、ループの削減と帰納変数の除去を行います。
-x04	-x03 レベルで行う最適化レベルに加えて、同じファイルに含まれる関数のインライン展開も自動的に行われます。インライン展開を自動的に行なった場合、通常は実行速度が速くなりますが、遅くなることもあります。一般にこのレベルを使用するとコードサイズが大きくなります。
-x05	最高レベルの最適化を行います。この最適化アルゴリズムは、コンパイルの所要時間が長く、また実行時間が確実に短縮される保証がありません。たとえば、エクスポートされた関数が局所的に呼び出されるような関数の入口を設定する、スピルコードを最適化する、命令スケジュールを向上するための解析を追加するなどがあります。

デバッグについての詳細は、『『Solaris Studio: dbx コマンドによるデバッグ』』を参照してください。最適化についての詳細は、『『Solaris Studio 12: パフォーマンスアナライザ』』マニュアルを参照してください。

`-xldscope` および `-xmaxopt` も参照してください。

B.2.123 -xopenmp[= *i*]

OpenMP 指令で明示的な並列化を有効にするには、`-xopenmp` オプションを使用します。並列化されたプログラムをマルチスレッド環境で実行するには、実行前に **OMP_NUM_THREADS** 環境変数を設定しておく必要があります。

入れ子並列を有効にするには、**OMP_NESTED** 環境変数を TRUE に設定する必要があります。入れ子並列は、デフォルトでは無効です。

次の表に *i* の値を示します。

表 B–35 -xopenmp のフラグ

値	意味
parallel	OpenMP プラグマの認識を有効にします。-xopenmp=parallel の最適化レベルは -x03 です。コンパイラは必要に応じて最適化レベルを -x03 に変更し、警告メッセージを表示します。 このフラグは、プロセッサのトークン **_OPENMP** も定義します。
noopt	OpenMP プラグマの認識を有効にします。最適化レベルが -O3 より低い場合は、最適化レベルは上げられません。 cc -O2 -xopenmp=noopt のように -O3 より低い最適化レベルを明示的に設定すると、エラーが表示されます。-xopenmp=noopt で最適化レベルを指定しなかった場合、OpenMP プラグマが認識され、その結果プログラムが並列化されますが、最適化は行われません。 このフラグは、プロセッサのトークン **_OPENMP** も定義します。
none	このフラグはデフォルトで、OpenMP プラグマの認識が有効化、プログラムの最適化レベルの変更、およびプリプロセッサトークンの事前定義を実行しません。

-xopenmp を指定しても値を設定しない場合、コンパイラは -xopenmp=parallel を仮定します。-xopenmp を指定しない場合、コンパイラは -xopenmp=none を仮定します。

dbx を指定して OpenMP プログラムをデバッグする場合、-g と -xopenmp=noopt を指定してコンパイルすれば、並列化部分にブレークポイントを設定して変数の内容を表示することができます。

-xopenmp のデフォルトは、将来変更される可能性があります。警告メッセージを出力しないようにするには、適切な最適化を明示的に指定します。

いずれかの .so を構築するときに -xopenmp を使用している場合は、実行可能ファイルをリンクするときに -xopenmp を使用する必要があります。また、実行可能ファイルに関して使用するコンパイラは、-xopenmp を指定して .so を構築するときに使用したコンパイラより古いものであってはいけません。これは、OpenMP 指令を含むライブラリをコンパイルする場合に特に重要です。表 A–2 に、コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの全一覧をまとめています。

最良のパフォーマンスを得るには、OpenMP 実行時ライブラリ libmtsk.so の最新パッチが、システムにインストールされていることを確認してください。

OpenMP の C の実装に固有の情報については、70 ページの「3.2 OpenMP に対する並列化」を参照してください。

OpenMP の詳細は、『Solaris Studio OpenMP API ユーザーズガイド』を参照してください。

B.2.124 -xP

すべての K&R C 関数のプロトタイプを出力する際、コンパイラはソースファイルに対して構文および意味検査のみ行います。このオプションは、オブジェクトコードや実行可能コードを生成しません。たとえば次のソースファイルに -xP を指定したと仮定します。

```
f()
{
}

main(argc,argv)
int argc;
char *argv[];
{
}
```

この例に対しては、次のとおりに出力します。

```
int f(void);
int main(int, char **);
```

B.2.125 -xpagesize=*n*

スタックとヒープ用の優先ページサイズを設定します。

SPARC では、次の値が有効です。4k、8K、64K 、512K、2M、4M、32M、256M、2G、16G、または default。

次の値は、86/x で有効です。4K、2M、4M、1G、または default。

有効なページサイズを指定しないと、要求は実行時に暗黙的に無視されます。ターゲットプラットフォームに対して有効なページサイズを指定する必要があります。

Solaris オペレーティングシステムでページのバイト数を調べるには、getpagesize(3C) コマンドを使用します。Solaris オペレーティングシステムでは、ページサイズ要求に従うという保証はありません。ターゲットプラットフォームのページサイズを判断するには、pmap(1) または meminfo(2) を使用します。

ターゲットプラットフォームのページサイズを判断するには、 pmap(1) または meminfo(2) を使用します。

-xpagesize オプションは、コンパイル時とリンク時に使用しないかぎり無効です。表 A–2 に、コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの全一覧をまとめています。

-xpagesize=default を指定すると、Solaris オペレーティングシステムがページサイズを設定します。

このオプションを指定してコンパイルするのは、LD_PRELOAD 環境変数を同等のオプションで mpss.so.1 に設定するか、またはプログラムを実行する前に同等のオプションを指定して Solaris コマンドの ppgsz(1) を実行するのと同じことです。詳細は Solaris のマニュアルページを参照してください。

このオプションは -xpagesize_heap と -xpagesize_stack のマクロです。これらの 2 つのオプションは -xpagesize と同じ次の引数を使用します。両方に同じ値を設定するには -xpagesize を指定します。別々の値を指定するには個々に指定します。

B.2.126 -xpagesize_heap=*n*

ヒープ用のメモリーページサイズを設定します。

このオプションは、–xpagesize と同じ値を受け入れます。有効なページサイズを指定しないと、要求は実行時に暗黙的に無視されます。

Solaris オペレーティングシステムでページのバイト数を調べるには、getpagesize(3C) コマンドを使用します。Solaris オペレーティングシステムでは、ページサイズ要求に従うという保証はありません。ターゲットプラットフォームのページサイズを判断するには、pmap(1) または meminfo(2) を使用します。

ターゲットプラットフォームのページサイズを判断するには、pmap(1) または meminfo(2) を使用します。

-xpagesize_heap=default を指定すると、Solaris オペレーティングシステムはページサイズを設定します。

このオプションを指定してコンパイルするのは、LD_PRELOAD 環境変数を同等のオプションで mpss.so.1 に設定するか、またはプログラムを実行する前に同等のオプションを指定して Solaris コマンドの ppgsz(1) を実行するのと同じことです。詳細は Solaris のマニュアルページを参照してください。

-xpagesize_heap オプションは、コンパイル時とリンク時に使用しないかぎり無効です。表 A–2 に、コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの全一覧をまとめています。

B.2.127 -xpagesize_stack=*n*

スタック用のメモリーページサイズを設定します。

このオプションは、–xpagesize と同じ値を受け入れます。有効なページサイズを指定しないと、要求は実行時に暗黙的に無視されます。

Solaris オペレーティングシステムでページのバイト数を調べるには、getpagesize(3C) コマンドを使用します。Solaris オペレーティングシステムでは、ページサイズ要求に従うという保証はありません。ターゲットプラットフォームのページサイズを判断するには、pmap(1) または meminfo(2) を使用します。

-xpagesize_stack=default を指定すると、Solaris オペレーティングシステムはページサイズを設定します。

このオプションを指定してコンパイルするのは、LD_PRELOAD 環境変数を同等のオプションで mpss.so.1 に設定するか、またはプログラムを実行する前に同等のオプションを指定して Solaris コマンドの ppgsz(1) を実行するのと同じことです。詳細は Solaris のマニュアルページを参照してください。

-xpagesize_stack オプションは、コンパイル時とリンク時に使用しないかぎり無効です。表 A–2 に、コンパイル時とリンク時の両方に指定する必要があるコンパイラオプションの全一覧をまとめています。

B.2.128 -xpch=*v*

このコンパイラオプションは、プリコンパイル済みヘッダー機能を起動します。*v* には、auto、autofirst、collect:*pch_filename*、use:*pch_filename* のいずれかを指定できます。この機能は、-xpch (303 ページの「B.2.128 -xpch=*v*」 で詳述) と -xpchstop (308 ページの「B.2.129 -xpchstop=[*file*|<include>]」 で詳述) オプションで、#pragma hdrstop 指令 (48 ページの「2.11.8 hdrstop」 で詳述) と組み合わせて利用できます。

-xpch オプションは、プリコンパイル済みヘッダーファイルを作成して、コンパイル時間を短縮するときに使用します。プリコンパイル済みヘッダーファイルは、ソースファイルが大量のソースコードを含む共通のインクルードファイル群を共有しているようなアプリケーションのコンパイル時間を低減するよう設計されています。プリコンパイル済みヘッダーを使用することによって、1 つのソースファイルから一連のヘッダーファイルに関する情報を収集し、そのソースファイルを再コンパイルする場合や、同じ一連のヘッダーを持つほかのソースファイルをコンパイルする場合に、その情報を使用することができます。コンパイラが収集する情報は、プリコンパイル済みヘッダーファイルに格納されます。

関連項目

- 308 ページの「B.2.129 -xpchstop=[*file*|<include>]」
- 48 ページの「2.11.8 hdrstop」

B.2.128.1 プリコンパイル済みヘッダーファイルの自動作成

プリコンパイル済みヘッダーファイルをコンパイラに自動的に生成させることができます。このためには、次のいずれかの方法を選択します。1 つは、ソースファイルで検出された最初のインクルードファイルからプリコンパイル済みヘッダーファイ

ルを作成する方法、もう 1 つは、最初のインクルードファイルから、最後のインクルードファイルを特定する綿密に定義された地点までの間にソースファイルで検出されたインクルードファイル群から選択してプリコンパイル済みヘッダーファイルを作成する方法です。次の 2 つのフラグのいずれかを使用して、プリコンパイル済みヘッダーの自動生成にコンパイラが使用すべき方法を指示します。

表 B–36 -xpch フラグ

フラグ	意味
-xpch=auto	プリコンパイル済みヘッダーファイルの内容は、コンパイラがソースファイルで検出した最長の活性文字列 (活性文字列の特定方法についてはこのあとを参照) に基づきます。このフラグは、もっとも多くのヘッダーファイルからなる可能性があるプリコンパイル済みヘッダーファイルを生成します。
-xpch=autofirst	このフラグは、ソースファイル内で最初に検出されたヘッダーのみからなるプリコンパイル済みヘッダーファイルを生成します。

B.2.128.2 プリコンパイル済みヘッダーファイルの手動作成

プリコンパイル済みヘッダーファイルを手動で作成する場合は、最初に -xpch を指定し、collect モードを指定します。-xpch=collect を指定するコンパイルコマンドは、ソースファイルを 1 つしか指定できません。次の例では、-xpch オプションがソースファイル a.c に基づいて myheader.cpch というプリコンパイル済みヘッダーファイルを作成します。

```
cc -xpch=collect:myheader a.c
```

有効なプリコンパイル済みヘッダーファイル名には必ず、.cpch という接尾辞が付きます。*pch_filename* を指定する場合、自分で接尾辞を追加することも、コンパイラに追加させることもできます。たとえば、cc -xpch=collect:foo a.c と指定すると、プリコンパイル済みヘッダーファイルには foo.cpch という名前が付けられます。

B.2.128.3 既存のプリコンパイル済みヘッダーファイルの処理方法

-xpch=auto または -xpch=autofirst のときにプリコンパイル済みヘッダーファイルを使用できない場合、コンパイラは新しいプリコンパイル済みヘッダーファイルを生成します。-xpch=use のときにプリコンパイル済みヘッダーファイルを使用できない場合は、警告が発行され、実際のヘッダーを使ってコンパイルが行われます。

B.2.128.4 特定のプリコンパイル済みヘッダーファイルの使用の指定

特定のプリコンパイル済みヘッダーファイルを使用するようコンパイラに指示することもできます。このためには、-xpch=use:*pch_filename* を使用します。プリコンパ

イル済みヘッダーファイルを作成するために使用されたソースファイルと同じインクルードファイルの並びを持つソースファイルであれば、いくつでも指定できます。たとえば、use モードで、次のようなコマンドがあるとします。cc -xpch=use:foo.cpch foo.c bar.c foobar.c。

次の項目が真の場合にかぎり、既存のプリコンパイル済みヘッダーファイルを使用します。次の項目で真ではないものがあれば、プリコンパイル済みヘッダーファイルを再作成する必要があります。

- プリコンパイル済みヘッダーファイルにアクセスするために使用するコンパイラは、プリコンパイル済みヘッダーファイルを作成したコンパイラと同じであること。あるバージョンのコンパイラで作成されたプリコンパイル済みヘッダーファイルは、別のバージョンのコンパイラでは使用できない場合があります。
- -xpch オプション以外で -xpch=use とともに指定するコンパイラオプションは、プリコンパイル済みヘッダーファイルが作成されたときに指定されたオプションと一致すること。
- -xpch=use で指定する一連のインクルードヘッダー群は、プリコンパイル済みヘッダーファイルが作成されたときに指定されたヘッダー群と同じであること。
- -xpch=use で指定するインクルードヘッダーの内容が、プリコンパイル済みヘッダーファイルが作成されたときに指定されたインクルードヘッダーの内容と同じであること。
- 現在のディレクトリ (すなわち、コンパイルが実行中で指定されたプリコンパイル済みヘッダーファイルを使用しようとしているディレクトリ) が、プリコンパイル済みヘッダーファイルが作成されたディレクトリと同じであること。
- -xpch=collect で指定したファイル内の #include 指令を含む前処理指令の最初のシーケンスが、-xpch=use で指定するファイル内の前処理指令のシーケンスと同じであること。

B.2.128.5 活性文字列 (Viable Prefix)

プリコンパイル済みヘッダーファイルを複数のソースファイル間で共有するために、これらのソースファイルには、最初のトークンの並びとして一連の同じインクルードファイルを使用していなければいけません。トークンはキーワードか名前、句読点のいずれかです。コンパイラは、コードおよび、#if 指令によって除外されたコードをトークンとして認識しません。この最初のトークンの並びは、活性文字列 (viable prefix) として知られています。言い替えれば、活性文字列は、すべてのソースファイルに共通のソースファイルの先頭部分です。コンパイラは、この活性文字列に基づいて、プリコンパイル済みヘッダーファイルを作成し、ソース内のプリコンパイルするヘッダーファイルを特定します。

現在のコンパイル中にコンパイラが検出する活性文字列は、プリコンパイル済みヘッダーファイルの作成に使用した活性文字列と一致する必要があります。言い替

えれば、活性文字列は、同じプリコンパイル済みヘッダーファイルを使用するすべてのソースファイル間で一貫して解釈される必要があります。

ソースファイルの活性文字列には、コメントと次に示すプリプロセッサ指令のみを指定できます。

```
#include
#if/ifdef/ifndef/else/elif/endif
#define/undef
#ident (if identical, passed through as is)
#pragma (if identical)
```

前述の任意の指令はマクロを参照する場合があります。`#else`、`#elif`、`#endif` 指令は、活性文字列内で一致している必要があります。コメントは無視されます。

`-xpch=auto` か `-xpch=autofirst` が指定されていて、次のように定義されている場合、コンパイラは活性文字列の終点を自動的に特定します。

- 最初の宣言/定義
- 最初の `#line` 指令
- `#pragma hdrstop` 指令
- 指定されたインクルードファイルのあと (`-xpch=auto` および `-xpchstop` が指定された場合)
- 最初のインクルードファイル (`-xpch=autofirst` が指定された場合)

注 – プリプロセッサ条件コンパイル文内に終点がある場合は、警告が生成され、プリコンパイル済みヘッダーファイルの自動作成は無効になります。また、`#pragma hdrstop` と `-xpchstop` オプションの両方が指定された場合、コンパイラは 2 つの停止点のうちの早い方を使用して、活性文字列を終了します。

`-xpch=collect` または `-xpch=use` の場合、活性文字列は `#pragma hdrstop` で終了します。

プリコンパイル済みヘッダーファイルを共有する各ファイルの活性文字列内では、対応する各 `#define` 指令と `#undef` 指令は同じシンボルを参照する必要があります (`#define` の場合は、各指令は同じ値を参照しなければいけません)。各活性文字列内での順序も同じである必要があります。対応する各プラグマも同じで、その順序もプリコンパイル済みヘッダーを共有するすべてのファイルで同じでなければいけません。

B.2.128.6 ヘッダーファイルの妥当性の判定

ヘッダーファイルがプリコンパイル可能となる条件。それは、複数のソースファイルに渡ってヘッダーファイルの一貫した解釈ができることです。具体的には、完全な宣言だけが含まれていることです。すなわち、どのファイルの宣言も有効な宣言

として独立している必要があります。struct S; などの不完全な型宣言は有効な型宣言です。完全な型宣言がほかのファイルに存在している可能性があります。次のヘッダーファイルの例を考えてみてください。

```
file a.h
struct S {
#include "x.h" /* not allowed */
};

file b.h
struct T; // ok, complete declaration
struct S {
   int i;
[end of file, continued in another file] /* not allowed*/
```

プリコンパイル済みヘッダーファイルに組み込まれるヘッダーファイルは、次の項目に違反しないようにしてください。これらの制限に違反するプログラムをコンパイルした場合、結果は予測できません。

- ヘッダーファイルに、__DATE__ や __TIME__ が含まれていてはいけません。
- ヘッダーファイルに、#pragma hdrstopが含まれていてはいけません。

ヘッダーに変数と関数の宣言が含まれている場合は、ヘッダーも同様にプリコンパイル可能です。

B.2.128.7 プリコンパイル済みヘッダーファイルのキャッシュ

プリコンパイル済みヘッダーファイルの自動作成では、コンパイラは、そのファイルを SunWS_cache ディレクトリに書き込みます。このディレクトリは常に、オブジェクトファイルが作成される場所に置かれます。dmake の下で適切に機能するように、ファイルの更新はロックして行われます。

強制的に再構築させる必要がある場合は、CCadmin ツールを使って、このプリコンパイル済みヘッダーファイルのキャッシュディレクトリをクリアすることができます。詳細は、CCadmin(1) のマニュアルページを参照してください。

B.2.128.8 警告

- 矛盾する -xpch フラグをコマンド行に指定しないでください。たとえば、-xpch=collect と -xpch=auto の両方を指定したり、-xpchstop=<include> を付けて -xpch=autofirst を指定したりすると、エラーになります。
- -xpch=autofirst を指定するか、-xpchstop なしで -xpch=auto を指定した場合、最初のインクルードファイル、あるいは -xpch=auto に -xpchstop を付けて指定したインクルードファイルの前に宣言や定義、#line 指令があると、エラーになり、プリコンパイル済みヘッダーファイルの自動生成が無効になります。
- -xpch=autofirst または -xpch=auto で、最初のインクルードファイルの前に #pragma hdrstop があると、プリコンパイル済みヘッダーファイルの自動生成が無効になります。

B.2.128.9 プリコンパイル済みヘッダーファイルの依存関係と make ファイル

-xpch=collect が指定されている場合、コンパイラはプリコンパイル済みヘッダーファイル用の依存関係情報を生成します。この依存関係情報を利用するには、メイクファイル内に適切な規則を作成する必要があります。次のメイクファイルの例を考えてみてください。

```
%.o : %.c shared.cpch
        $(CC) -xpch=use:shared -xpchstop=foo.h -c $<
default : a.out

foo.o + shared.cpch : foo.c
        $(CC) -xpch=collect:shared -xpchstop=foo.h foo.c -c

a.out : foo.o bar.o foobar.o
        $(CC) foo.o bar.o foobar.o

clean :
        rm -f *.o shared.cpch .make.state a.out
```

コンパイラによって生成された依存関係情報とともに、これらの make 規則があると、-xpch=collect で使用されたソースファイル、またはプリコンパイル済みヘッダーファイルを構成するヘッダーのどれかに変更があった場合、手動で作成されたプリコンパイル済みヘッダーファイルが強制的に再作成されます。これにより、古いプリコンパイル済みヘッダーファイルの使用が防止されます。

-xpch=auto または -xpch=autofirst の場合、メイクファイルに追加の make 規則を作成する必要はありません。

B.2.129 -xpchstop=[*file*|<include>]

-xpchstop=*file* オプションは、プリコンパイル済みヘッダーファイル用の活性文字列の最後のインクルードファイルを指定するときに使用します。コマンド行で -xpchstop を使用するのは、cc コマンドで指定する各ソースファイルの *file* を参照する最初のインクルード指令のあとに hdrstop プラグマを配置するのと同じことです。

<include> までのヘッダーファイルに基づくプリコンパイル済みヘッダーファイルを作成するには、-xpchstop=<include> と -xpch-auto を組み合わせます。このフラグは、活性文字列全体に含まれているすべてのヘッダーファイルを使用するデフォルトの -xpch=auto の動作に優先します。

次の例では、-xpchstop オプションは、プリコンパイル済みヘッダーファイルの活性文字列が projectheader.h をインクルードして終わるよう指定します。したがって、privateheader.h は活性文字列の一部ではありません。

```
example% cat a.c
     #include <stdio.h>
     #include <strings.h>
```

```
     #include "projectheader.h"
     #include "privateheader.h"
     .
     .
     .
example% cc -xpch=collect:foo.cpch a.c -xpchstop=projectheader.h -c
```

-xpch も参照してください。

B.2.130 - xpec[={yes|no}]

移植可能な実行可能コード (Portable Executable Code、PEC) バイナリを生成します。PEC バイナリは、自動チューニングシステム (Automatic Tuning System、ATS) と組み合わせて使用できます。ATS はチューニングとトラブルシューティングの目的で、コンパイル済み PEC バイナリを再構築する方法で動作し、下のソースコードは必要ありません。ATS の詳細は、http://cooltools.sunsource.net/ を参照してください。

-xpec を指定して構築したバイナリは通常、 -xpec なしで構築したバイナリより 5 〜 10 倍の大きさになります。

-xpec を指定しない場合は、-xpec=no に設定されます。-xpec をフラグなしで指定した場合は、コンパイラでは -xpec=yes が指定されます。

B.2.131 -xpentium

(x86) Pentium プロセッサ用のコードを生成します。

B.2.132 -xpg

gprof(1) によるプロファイルの準備として、データを収集するためのオブジェクトコードを生成します。この記録機構は実行が正常終了すると、gmon.out ファイルを作成します。

注 – -xpg を指定した場合、 -xprofile を使用する利点はありません。 これら 2 つは、他方で使用できるデータを生成せず、他方で生成されたデータを使用できません。

プロファイルは、64 ビット Solaris プラットフォームで prof(1) または gprof(1)、32 ビット Solaris プラットフォームで gprof を使用して生成され、おおよそのユーザー CPU 時間が含まれます。これらの時間は、メインの実行可能ファイルのルーチンと、実行可能ファイルをリンクするときにリンカー引数として指定した共有ライブラリのルーチンの PC サンプルデータ (pcsample(2) を参照) から導出されます。そのほかの共有ライブラリ (dlopen(3DL) を使用してプロセスの起動後に開かれたライブラリ) のプロファイルは作成されません。

32 ビット Solaris システムの場合、prof(1) を使用して生成されたプロファイルには、実行可能ファイルのルーチンだけが含まれます。32 ビット共有ライブラリのプロファイルは、-xpg で実行可能ファイルをリンクし、 gprof(1) を使用することで作成できます。

x86 システムでは、-xpg には -xregs=frameptr との互換性がないため、これらの 2 つのオプションは同時に使用できません。 -xregs=frameptr は -fast に含まれている点にも注意してください。

Solaris 10 ソフトウェアには、 -p でコンパイルされたシステムライブラリが含まれません。その結果、Solaris 10 プラットフォームで収集されたプロファイルには、システムライブラリルーチンの呼び出し回数が含まれません。

コンパイル時に -xpg を指定した場合は、リンク時にも指定する必要があります。215 ページの「A.1.2 コンパイル時とリンク時のオプション」に、コンパイル時とリンク時の両方に指定する必要があるオプションの全一覧をまとめています。

B.2.133 -xprefetch[= *val*[, *val*]]

先読みをサポートするアーキテクチャーで先読み命令を有効にします。

明示的な先読みは、測定値によってサポートされた特殊な環境でのみ使用すべきです。

val には、次のいずれかを指定します。

表 B–37 -xprefetch のフラグ

フラグ	意味
latx:factor	指定された factor によってコンパイラで使用されるロードするための先読みと、ストアするための先読みを調整します。このフラグは、-xprefetch=auto とのみ組み合わせることができます。311 ページの「B.2.133.1 先読み応答率」を参照してください。
[no%]auto	先読み命令の自動生成を有効 [無効] にします。
[no%]explicit	明示的な先読みマクロを有効 [無効] にします。
yes	廃止。使わないでください。代わりに - xprefetch=auto,explicit を使用します。
no	廃止。使わないでください。代わりに -xprefetch=no%auto,no%explicit を使用します。

デフォルトは -xprefetch=auto,explicit です。基本的に非線形のメモリーアクセスパターンを持つアプリケーションには、このデフォルトが良くない影響をもたらします。デフォルトを無効にするには、 -xprefetch=no%auto,no%explicit を指定します。

sun_prefetch.h ヘッダーファイルには、明示的な先読み命令を指定するためのマクロが含まれています。先読み命令は、実行コード中のマクロの位置にほぼ相当するところに挿入されます。

B.2.133.1 先読み応答率

先読みの応答時間とは、先読み命令を実行してから先読みされたデータがキャッシュで利用可能となるまでのハードウェアの遅延のことです。

係数には、*n.n.* という形式の正の数値を使用します。

コンパイラは、先読み命令と先読みされたデータを使用するロードまたはストア命令の距離を決定する際に先読み応答時間の値を想定します。先読みからロードまでのデフォルト応答時間は、先読みからストアまでのデフォルト応答時間と同じでない場合があります。

コンパイラは、幅広いマシンとアプリケーションで最適なパフォーマンスを得られるように先読み機構を調整します。しかし、コンパイラの調整作業が必ずしも最適であるとはかぎりません。メモリーに負担のかかるアプリケーション、特に大型のマルチプロセッサでの実行を意図したアプリケーションの場合、先読みの応答時間の値を引き上げることにより、パフォーマンスを向上できます。この値を増やすには、1 よりも大きい係数を使用します。.5 〜 2.0 の値を指定すると、ほとんどの場合は最大のパフォーマンスが実現されます。

外部キャッシュの中に完全に常駐するデータセットを持つアプリケーションの場合は、先読み応答時間の値を減らすことでパフォーマンスを向上できる場合があります。値を小さくするには、1 よりも小さな係数を使用します。

latx:factor サブオプションを使用するには、1.0 程度の係数から順にアプリケーションの性能テストを実行します。そのあと、テストの結果に応じて係数を増減し、パフォーマンステストを再実行します。係数の調整を継続し、最適なパフォーマンスに到達するまでパフォーマンステストを実行します。係数を小刻みに増減すると、しばらくはパフォーマンスに変化がなく、突然変化し、再び平常に戻ります。

B.2.134 -xprefetch_auto_type=*a*

ここで *a* は、 [no%]indirect_array_access です。

このオプションは、直接メモリーアクセスに対して先読み命令が生成されるのと同じ方法で、-xprefetch_level オプションが指示するループに対して間接先読み命令を生成するかどうかを制御します。

-xprefetch_auto_type の値が指定されていない場合、-xprefetch_auto_type=no%indirect_array_access に設定されます。

-xalias_level などのオプションは、メモリー別名を明確化する情報の生成に役立つため、間接プリフェッチ候補の計算の積極性に影響し、このため、自動的な間接プリフェッチの挿入が促進されることがあります。

B.2.135 -xprefetch_level=*l*

-xprefetch_level オプションを使用して、-xprefetch=auto で定義した先読み命令の自動挿入を調整することができます。*l* には 1、2、3 のいずれかを指定します。-xprefetch_level が高くなるほど、コンパイラはより攻撃的に、つまりより多くの先読み命令を挿入します。

-xprefetch_level に適した値は、アプリケーションでのキャッシュミス数によって異なります。-xprefetch_level の値を高くするほど、アプリケーションのパフォーマンスが向上する可能性が高くなります。

このオプションは、-xprefetch=auto を指定し、最適化レベルを 3 以上に設定して、先読みをサポートするプラットフォーム (v8plus、v8plusa、v9、v9a、v9b、generic64、native64) 用にコードを生成した場合にだけ有効です。

-xprefetch_level=1 を指定すると、先読み命令の自動生成が有効になります。-xprefetch_level=2 を指定すると、レベル 1 を上回る追加の生成が行われ、-xprefetch_level=3 を指定すると、レベル 1 を上回る追加の生成が行われます。

デフォルトは、-xprefetch=auto を指定した場合は -xprefetch_level=1 になります。

B.2.136 –xprofile=*p*

プロファイルのデータを収集したり、プロファイルを使用して最適化したりします。

p には、collect[:*profdir*]、use[:*profdir*]、または tcov[:*profdir*] を指定する必要があります。

このオプションを指定すると、実行頻度のデータが収集されて実行中に保存されます。このデータを以降の実行で使用すると、パフォーマンスを向上させることができます。プロファイルの収集は、マルチスレッド対応のアプリケーションにとって安全です。すなわち、独自のマルチタスク (-mt) を実行するプログラムをプロファイリングすることで、正確な結果が得られます。このオプションは、最適化レベルを -xO2 かそれ以上に指定するときにのみ有効になります。コンパイルとリンクを別々の手順で実行する場合は、リンク手順とコンパイル手順の両方で同じ -xprofile オプションを指定する必要があります。

collect[: *profdir*] 実行頻度のデータを集めて保存します。のちに -xprofile=use を指定した場合にオプティマイザがこれを使用します。コンパイラによって文の実行頻度を測定するためのコードが生成されます。

-xMerge、-ztext、および -xprofile=collect を一緒に使用しないでください。-xMerge を指定すると、静的に初期化されたデータを読み取り専用記憶領域に強制的に配置します。 -ztext を指定すると、位置に依存するシンボルを読み取り専用記憶領域内で再配置することを禁止します。-xprofile=collect を指定すると、書き込み可能記憶領域内で、静的に初期化された、位置に依存するシンボルの再配置を生成します。

プロファイルディレクトリ名として *profdir* を指定すると、この名前が、プロファイル化されたオブジェクトコードを含むプログラムまたは共有ライブラリの実行時にプロファイルデータが保存されるディレクトリのパス名になります。*profdir* パス名が絶対パスではない場合、プログラムがオプション -xprofile=use:*profdir* でコンパイルされるときの現在の作業用ディレクトリの相対パスとみなされます。プロファイルディレクトリ名を指定しないと、プロファイルデータは、*program*.profile という名前のディレクトリに保存されます (*program* はプロファイル化されたプロセスのメインプログラムのベース名)。

例 [1]: プログラムが構築されたディレクトリと同じディレクトリにある myprof.profile ディレクトリでプロファイルデータを収集して使用するには、次のように指定します。

```
demo: cc -xprofile=collect:myprof.profile -xO5 prog.c -o prog
demo: ./prog
demo: cc -xprofile=use:myprof.profile    -xO5 prog.c -o prog
```

例 [2]: ディレクトリ /bench/myprof.profile にプロファイルデータを収集し、収集したプロファイルデータをあとから最適化レベル -xO5 のフィードバックコンパイルで使用するには、次のように指定します。

```
demo: cc -xprofile=collect:/bench/myprof.profile
\    -xO5 prog.c -o prog
...run prog from multiple locations..
demo: cc -xprofile=use:/bench/myprof.profile
\    -xO5 prog.c -o prog
```

環境変数の SUN_PROFDATA と SUN_PROFDATA_DIR を設定して、-xprofile=collect を指定してコンパイルされたプログラムがどこにプロファイルデータを入れるかを制御できます。これらの環境変数を設定すると、-xprofile=collect データが $SUN_PROFDATA_DIR/$SUN_PROFDATA に書き込まれます。

これらの環境変数は、tcov で書き込まれたプロファイルデータファイルのパスと名前を tcov(1) マニュアルページの説明どおり、同様に制御指定します。これらの環境変数をまだ設定して

いない場合、プロファイルデータは現作業ディレクトリの *profdir* `.profile` に書き込まれます (*profdir* は実行ファイルの名前または `-xprofile=collect:` *profdir* フラグで指定された名前)。*profdir* が `.profile` ですでに終了している場合、`-xprofile` では、`profile` が *profdir* に追加されません。プログラムを複数回実行すると、実行頻度データは *profdir*`.profile` ディレクトリに蓄積されていくので、以前の実行頻度データは失われません。

別々の手順でコンパイルしてリンクする場合は、`-xprofile=collect` を指定してコンパイルしたオブジェクトファイルは、リンクでも必ず `-xprofile=collect` を指定してください。

`use`[`:` *profdir*]

`-xprofile=collect`[`:` *profdir*] でコンパイルされたコードから収集された実行頻度データを使用して、プロファイル化されたコードが実行されたときに実行された作業用の最適化が行えます。*profdir* は、`-xprofile=collect`[`:` *profdir*] でコンパイルされたプログラムを実行して収集されたプロファイルデータを含むディレクトリのパス名です。

profdir パス名は省略可能です。 *profdir* が指定されていない場合、実行可能バイナリの名前が使用されます。`-o` が指定されていない場合、`a.out` が使用されます。 *profdir* が指定されていない場合、コンパイラは、`profdir` **`.profile/feedback`**、または *a.out.profile/feedback* を探します。次に例を示します。

```
demo: cc -xprofile=collect    -o myexe prog.c
demo: cc -xprofile=use:myexe -xO5 -o myexe    prog.c
```

`-xprofile=collect` オプションを付けてコンパイルしたときに生成され、プログラムの前の実行で作成されたフィードバックファイルに保存された実行頻度データを使用して、プログラムが最適化されます。

`-xprofile` オプションを除き、ソースファイルおよびコンパイラのほかのオプションは、フィードバックファイルを生成したコンパイル済みプログラムのコンパイルに使用したものと完全に同一のものを指定する必要があります。同じバージョンのコンパイラは、収集構築と使用構築の両方に使用する必要があります。

`-xprofile=collect:`*profdir* を付けてコンパイルした場合は、*-xprofile=use:* `profdir` のコンパイルの最適化に同じプロファイルディレクトリ名 *profdir* を使用する必要があります。

収集 (collect) 段階と使用 (use) 段階の間のコンパイル速度を高める方法については、-xprofile_ircache も参照してください。

tcov[: *profdir*]

tcov(1) を使用する基本のブロックカバレッジ分析用の命令オブジェクトファイル。

オプションの *profdir* 引数を指定すると、コンパイラは指定された場所にプロファイルディレクトリを作成します。プロファイルディレクトリに保存されたデータは、tcov(1) または -xprofile=use:*profdir* を付けたコンパイラで使用できます。オプションの *profdir* パス名を省略すると、プロファイル化されたプログラムの実行時にプロファイルディレクトリが作成されます。プロファイルディレクトリに保存されたデータは、tcov(1) でのみ使用できます。プロファイルディレクトリの場所は、環境変数 SUN_PROFDATA および SUN_PROFDATA_DIR を使用して指定できます。

profdir で指定された場所が絶対パス名ではない場合、コンパイル時に、現在のオブジェクトファイルが作成されたディレクトリの相対パスとみなされます。いずれかのオブジェクトファイルに *profdir* を指定する場合は、同じプログラムのすべてのオブジェクトファイルに対して同じ場所を指定する必要があります。場所が *profdir* で指定されているディレクトリには、プロファイル化されたプログラムを実行するときにすべてのマシンからアクセスできる必要があります。プロファイルディレクトリはその内容が必要なくなるまで削除できません。コンパイラでプロファイルディレクトリに保存されたデータは、再コンパイルする以外復元できません。

例 [1]: 1 つ以上のプログラムのオブジェクトファイルが -xprofile=tcov:/test/profdata でコンパイルされる場合、/test/profdata.profile という名前のディレクトリがコンパイラによって作成されて、プロファイル化されたオブジェクトファイルを表すデータの保存に使用されます。実行時に同じディレクトリを使用して、プロファイル化されたオブジェクトファイルに関連付けられた実行データを保存できます。

例 [2]: myprog という名前のプログラムが -xprofile=tcov でコンパイルされ、ディレクトリ /home/joe で実行されると、実行時にディレクトリ /home/joe/myprog.profile が作成されて、実行時プロファイルデータの保存に使用されます。

B.2.137 -xprofile_ircache[=*path*]

(SPARC) ンcollect 段階で保存されたコンパイルデータを再利用して use 段階のコンパイル時間を向上するには、-xprofile=collect|use で -xprofile_ircache[=*path*] を使用します。

大きなプログラムでは、中間データが保存されるため、use 段階のコンパイル時間の効率を大幅に向上させることができます。保存されたデータが必要なディスク容量を相当増やすことがある点に注意してください。

-xprofile_ircache[=*path*] を使用すると、*path* はキャッシュファイルが保存されているディレクトリを上書きします。デフォルトでは、これらのファイルはオブジェクトファイルと同じディレクトリに保存されます。collect と use 段階が 2 つの別のディレクトリで実行される場合は、パスを指定しておくと便利です。一般的なコマンドシーケンスを次に示します。

```
example% cc -xO5 -xprofile=collect -xprofile_ircache t1.c t2.c
example% a.out    // run collects feedback data
example% cc -xO5 -xprofile=use -xprofile_ircache t1.c t2.c
```

B.2.138 -xprofile_pathmap

(SPARC) -xprofile_pathmap= コマンドも指定する場合は、*-xprofile_pathmap=collect_prefix*: use_prefix オプションを使用します。次の項目がともに真で、コンパイラが -xprofile=use でコンパイルされたオブジェクトファイルのプロファイルデータを見つけられない場合は、-xprofile_pathmap を使用します。

- 前回オブジェクトファイルが -xprofile=collect でコンパイルされたディレクトリとは異なるディレクトリで、オブジェクトファイルを -xprofile=use を指定してコンパイルしている。
- オブジェクトファイルはプロファイルで共通ベース名を共有しているが、異なるディレクトリのそれぞれの位置で相互に識別されている。

collect-prefix は、オブジェクトファイルが -xprofile=collect でコンパイルされたディレクトリツリーの UNIX パス名の接頭辞です。

use-prefix は、オブジェクトファイルが -xprofile=use を指定してコンパイルされたディレクトリツリーの UNIX パス名の接頭辞です。

-xprofile_pathmap の複数のインスタンスを指定すると、コンパイラは指定した順序でインスタンスを処理します。*-xprofile_pathmap* のインスタンスで指定された各 use-prefix は、一致する *use-prefix* が識別されるか、最後に指定された *use-prefix* がオブジェクトファイルのパス名と一致しないことが確認されるまで、オブジェクトファイルのパス名と比較されます。

B.2.139 -xreduction

自動的な並列化を実行するときに、縮約の認識を有効にします。-xreduction は、-xautopar とともに指定する必要があります。それ以外の場合は、コンパイラが警告を発行します。

縮約の認識が有効な場合、コンパイラは内積、最大値発見、最小値発見などの縮約を並列化します。これらの縮約によって非並列化コードの場合とは、四捨五入の結果が異なります。

B.2.140 -xregs=*r*[,*r*...]

生成コード用のレジスタの使用法を指定します。

r には、次の 1 つまたは複数の項目をコンマで区切って指定します。appl, float,frameptr

サブオプションの前に no% を付けるとそのサブオプションは無効になります。

–xregs サブオプションは、特定のハードウェアプラットフォームでしか使用できません。

例: -xregs=appl,no%float

表 B–38 -xregs サブオプション

値	意味
appl	(SPARC) コンパイラがアプリケーションレジスタをスクラッチレジスタとして使用してコードを生成することを許可します。アプリケーションレジスタは次のとおりです。 g2、g3、g4 (32 ビットプラットフォーム) g2、g3 (64 ビットプラットフォーム) すべてのシステムソフトウェアおよびライブラリは、-xregs=no%appl を指定してコンパイルすることをお勧めします。システムソフトウェア (共有ライブラリを含む) は、アプリケーション用のレジスタの値を保持する必要があります。これらの値は、コンパイルシステムによって制御されるもので、アプリケーション全体で整合性が確保されている必要があります。 SPARC ABI では、これらのレジスタはアプリケーションレジスタと記述されています。これらのレジスタを使用すると必要なロードおよびストア命令が少なくてすむため、パフォーマンスが向上します。ただし、アセンブリコードで記述された古いライブラリプログラムとの間で衝突が起きることがあります。
float	(SPARC) コンパイラが浮動小数点レジスタを整数値用のスクラッチレジスタとして使用してコードを生成することを許可します。浮動小数点値を使用する場合は、このオプションとは関係なくこれらのレジスタを使用します。浮動小数点レジスタに対するすべての参照をコードから排除する場合は、-xregs=no%float を使用するとともに、決して浮動小数点型をコードで使わないようにする必要があります。

表 B–38 -xregs サブオプション (続き)

値	意味
frameptr	(x86) フレームポインタレジスタ (IA32 の場合 %ebp、AMD64 の場合 %rbp) を汎用レジスタとして使用することを許可します。 デフォルトは -xregs=no%frameptr です。 –features=no%except によって例外も無効になっているのでなければ、C++ コンパイラは –xregs=frameptr を無視します。–xregs=frameptr は –fast の一部ですが、–features=no%except も指定されているのでなければ C++ コンパイラによって無視される点に注意してください。 -xregs=framptr を使用すると、コンパイラは浮動小数点レジスタを自由に使用できるので、プログラムのパフォーマンスが向上します。ただし、この結果としてデバッガおよびパフォーマンス測定ツールの一部の機能が制限される場合があります。スタックトレース、デバッガ、およびパフォーマンスアナライザは、–xregs=frameptr を使用してコンパイルされた機能についてレポートできません。 さらに、Posix pthread_cancel() の C++ 呼び出しは、クリーンアップハンドラの検索に失敗します。 C、Fortran、C++ が混在しているコードで、C または Fortran 関数から直接または間接的に呼び出された C++ 関数が例外をスローする可能性がある場合、このコードは –xregs=frameptr でコンパイルできません。このような言語が混在するソースコードを –fast でコンパイルする場合は、コマンド行の –fast オプションのあとに –xregs=no%frameptr を追加します。 64 ビットのプラットフォームで使用できる多くのレジスタでは、–xregs=frameptr でコンパイルすると、64 ビットコードよりも 32 ビットコードのパフォーマンスが向上する可能性が高くなります。 -xpg も指定されている場合、コンパイラは -xregs=frameptr を無視し、警告を表示します。

SPARC のデフォルトは -xregs=appl,float です。

x86 のデフォルトは -xregs=no%frameptr です。

x86 システムでは、 -xpg には -xregs=frameptr との互換性がないため、これらの 2 つのオプションは同時に使用できません。 -xregs=frameptr は -fast に含まれている点にも注意してください。

アプリケーションにリンクする共有ライブラリ用のコードは、 -xregs=no%appl,float を指定してコンパイルすることをお勧めします。少なくとも、共有ライブラリとリ

ンクするアプリケーションがこれらのレジスタの割り当てを認識するように、共有ライブラリがアプリケーションレジスタを使用する方法を明示的に示す必要があります。

たとえば、大局的な方法で (重要なデータ構造体を示すためにレジスタを使用するなど) レジスタを使用するアプリケーションは、ライブラリと確実にリンクするため、`-xregs=no%appl` なしでコンパイルされたコードを含むライブラリがアプリケーションレジスタをどのように使用するかを正確に特定する必要があります。

B.2.141 -xrestrict[=*f*]

ポインタ値の関数引数を制限付き (restricted) ポインタとして扱います。*f* には、`%all`、`%none`、あるいは 1 つまたは複数の関数名をコンマで区切ったリストを指定します。例: {`%all`| `%none`|*fn*[,*fn*...]

関数リストの指定にこのオプションを入れると、指定された関数内のポインタ引数は制限付きとして扱われます。`-xrestrict=%all` を指定すると、C ファイル全体のすべてのポインタ引数が制限付きとして扱われます。詳細については、88 ページの「3.8.2 制限付きポインタ」を参照してください。

このコマンド行オプションは独立して使用できますが、最適化時に使用するのがもっとも適しています。たとえば、次のようなコマンドを使用します。

```
%cc -xO3 -xrestrict=%all prog.c
```

このコマンドでは、ファイル `prog.c` 内のすべてのポインタ引数を制限付きポインタとして扱います。次のようなコマンド行があるとします。

```
%cc -xO3 -xrestrict=agc prog.c
```

このコマンドでは、ファイル `prog.c` 内の関数 `agc` のすべてのポインタ引数を制限付きポインタとして扱います。

デフォルトは `%none` で、`-xrestrict` と指定すると `-xrestrict=%all` と指定した場合と同様の結果が得られます。

B.2.142 -xs

オブジェクトファイルなしに `dbx` でデバッグできるようにします。

このオプションを指定すると、すべてのデバッグ情報が実行可能ファイルにコピーされます。dbx のパフォーマンスやプログラムの実行時のパフォーマンスにはあまり影響はありませんが、より多くのディスク容量が必要となります。

B.2.143 -xsafe=mem

(SPARC) コンパイラが記憶域保護違反が発生した場合を前提とできるようにします。

このオプションを使用すると、コンパイラでは SPARC V9 アークテクチャーで違反のないロード命令を使用できます。

注 – アドレスの位置合わせが合わない、またはセグメンテーション侵害などの違反が発生した場合は違反のないロードはトラップを引き起こさないので、このオプションはこのような違反が起こる可能性のないプログラムでしか使用しないでください。ほとんどのプログラムではメモリーに関するトラップは起こらないので、大多数のプログラムでこのオプションを安全に使用できます。例外条件の処理にメモリーベースのトラップを明示的に使用するプログラムでは、このオプションは使用しないでください。

このオプションは、最適化レベルの `-xO5` と、次のいずれかの値の `-xarch` を組み合わせた場合にだけ有効です。`m32` と `m64` の両方で `sparc`、`sparcvis`、`-sparcvis2`、または `-sparcvis3`。

B.2.144 -xsb

廃止。使わないでください。ソースブラウザ機能はもうサポートされていません。

B.2.145 -xsbfast

廃止。使わないでください。ソースブラウザ機能はもうサポートされていません。

B.2.146 -xsfpconst

接尾辞のない浮動小数点定数を、デフォルトの倍精度モードではなく、単精度で表します。`-Xc` と併用することはできません。

B.2.147 -xspace

コードサイズを増やすループの最適化や並列化を行いません。

例: コードサイズが増える場合は、ループの展開や並列化は行われません。

B.2.148 -xstrconst

このオプションは非推奨であり、将来のリリースで削除される可能性があります。–xstrconst は –features=conststrings の別名です。

B.2.149 -xtarget=*t*

最適化の対象となる命令セットとシステムを指定します。

t には次の値のいずれかを指定します。native、generic、native64、generic64、*system-name*。

-xtarget に指定する値は、-xarch、-xchip、-xcache の各オプションの値に展開されます。実行中のシステムで -xtarget=native の展開を調べるには、-xdryrun コマンドを使用します。

たとえば、-xtarget=sun4/15 は次と同じです。-xarch=v8a -xchip=micro -xcache=2/16/1。

注 – 特定のホストプラットフォームで -xtarget を展開した場合、そのプラットフォームでコンパイルすると -xtarget=native と同じ -xarch、-xchip、または -xcache 設定にならない場合があります。

表 B–39 すべてのプラットフォームでの -xtarget の値

フラグ	意味
native	ホストシステムで最高のパフォーマンスが得られます。 コンパイラは、ホストシステムに対して最適化されたコードを生成します。コンパイラは自身が動作しているマシンで利用できるアーキテクチャー、チップ、キャッシュ特性を判定します。
native64	ホストシステムで 64 ビットのオブジェクトバイナリの最高のパフォーマンスが得られます。コンパイラは、ホストシステム用に最適化された 64 ビットのオブジェクトバイナリを生成します。コンパイラが動作しているマシンで使用できる 64 ビットのアーキテクチャー、チップ、キャッシュ特性を判定します。
generic	これはデフォルト値です。汎用アーキテクチャー、チップ、およびキャッシュで最高のパフォーマンスが得られます。
generic64	大多数の 64 ビットのプラットフォームのアーキテクチャーで 64 ビットのオブジェクトバイナリの最適なパフォーマンスを得るためのパラメータを設定します。

表 B–39 すべてのプラットフォームでの -xtarget の値 (続き)

フラグ	意味
システム名	指定のシステムに対して最高のパフォーマンスが得られるようにします。 対象となる実際のシステムを表すシステム名を、次のリストから選択してください。

対象となるハードウェア (コンピュータ) の正式な名前をコンパイラに指定した方がパフォーマンスが優れているプログラムもあります。プログラムのパフォーマンスが重要な場合は、対象となるハードウェアを正確に指定してください。これは特に、新しい SPARC プロセッサ上でプログラムを実行する場合に当てはまります。しかし、ほとんどのプログラムおよび廃止の SPARC システムではパフォーマンスの向上はわずかであるため、汎用的な指定方法で十分です。

B.2.149.1 SPARC プラットフォームの -xtarget の値

SPARC または UltraSPARC V9 での 64 ビット Solaris ソフトウェアのコンパイルは、-m64 オプションで指定します。-xtarget を指定し、native64 または generic64 以外のフラグを付ける場合は、-m64 オプションも次のように指定する必要があります。-xtarget=ultra ... -m64。この指定を行わない場合は、コンパイラは 32 ビットメモリーモデルを使用します。

表 B–40 SPARC での -xtarget の展開

-xtarget=	-xarch	-xchip	-xcache
generic	generic	generic	generic
ultra	v8plusa	ultra	16/32/1:512/64/1
ultra1/140	v8plusa	ultra	16/32/1:512/64/1
ultra1/170	v8plusa	ultra	16/32/1:512/64/1
ultra1/200	v8plusa	ultra	16/32/1:512/64/1
ultra2	v8plusa	ultra2	16/32/1:512/64/1
ultra2/1170	v8plusa	ultra	16/32/1:512/64/1
ultra2/1200	v8plusa	ultra	16/32/1:1024/64/1
ultra2/1300	v8plusa	ultra2	16/32/1:2048/64/1
ultra2/2170	v8plusa	ultra	16/32/1:512/64/1
ultra2/2200	v8plusa	ultra	16/32/1:1024/64/1
ultra2/2300	v8plusa	ultra2	16/32/1:2048/64/1

表 B–40 SPARC での -xtarget の展開 (続き)

-xtarget=	-xarch	-xchip	-xcache
ultra2e	v8plusa	ultra2e	16/32/1:256/64/4
ultra2i	v8plusa	ultra2i	16/32/1:512/64/1
ultra3	sparcvis2	ultra3	64/32/4:8192/512/1
ultra3cu	sparcvis2	ultra3cu	64/32/4:8192/512/2
ultra3i	sparcvis2	ultra3i	64/32/4:1024/64/4
ultra4	sparcvis2	ultra4	64/32/4:8192/128/2
ultra4plus	sparcvis2	ultra4plus	64/32/4:2048/64/4/2:32768/64/4
ultraT1	sparc	ultraT1	8/16/4/4:3072/64/12/32
ultraT2	sparcvis2	ultraT2	8/16/4:4096/64/16
ultraT2plus	sparcvis2	ultraT2plus	8/16/4:4096/64/16
ultraT3	sparcvis3	ultraT3	8/16/4:6144/64/24
sparc64vi	sparcfmaf	sparc64vi	128/64/2:5120/64/10
sparc64vii	sparcima	sparc64vii	64/64/2:5120/256/10

UltraSPARC IVplus チップ、UltraSPARC T1 チップ、UltraSPARC T2 チップのキャッシュ特性についての詳細は、266 ページの「B.2.80 -xcache[= *c*]」 を参照してください。

B.2.149.2 x86 プラットフォームの -xtarget の値

64 ビット x86 プラットフォームでの 64 ビット Solaris ソフトウェアのコンパイルは、-m64 オプションで指定します。-xtarget を指定し、native64 または generic64 以外のフラグを付ける場合は、-m64 オプションも次のように指定する必要があります。-xtarget=opteron ... -m64。この指定を行わない場合は、コンパイラは 32 ビットメモリーモデルを使用します。

表 B–41 x86 での -xtarget の展開

-xtarget=	-xarch	-xchip	-xcache
generic	generic	generic	generic
opteron	sse2a	opteron	64/64/2:1024/64/16
pentium	386	pentium	generic
pentium_pro	pentium_pro	pentium_pro	generic
pentium3	sse	pentium3	16/32/4:256/32/4

表 B–41 x86 での -xtarget の展開 (続き)

-xtarget=	-xarch	-xchip	-xcache
pentium4	sse2	pentium4	8/64/4:256/128/8
nehalem	sse4_2	nehalem	32/64/8:256/64/8:8192/64/1
penryn	sse4_1	penryn	2/64/8:6144/64/24
woodcrest	ssse3	core2	32/64/8:4096/64/16
barcelona	amdsse4a	amdfam10	64/64/2:512/64/16

B.2.150 -xtemp=*dir*

cc が使用する一時ファイルのディレクトリを *dir* に設定します。このオプション文字列の中にはスペースを入れてはいけません。このオプションを指定しない場合、一時ファイルは /tmp に配置されます。-xtemp は TMPDIR 環境変数より優先します。

B.2.151 -xthreadvar[=*o*]

スレッドローカルな変数の実装を制御するには -xthreadvar を指定します。コンパイラのスレッドローカルな記憶領域機能を利用するには、このオプションを __thread 宣言指示子と組み合わせて使用します。__thread 指示子を使用してスレッド変数を宣言したあとは、-xthreadvar を指定して動的 (共有) ライブラリの位置に依存しないコード (PIC 以外のコード) でスレッド固有の記憶領域を使用できるようにします。__thread の使用方法の詳細については、37 ページの「2.3 スレッドローカルな記憶領域指示子」を参照してください。

o には、次のいずれかを指定します。

表 B–42 -xthreadvar のフラグ

フラグ	意味
[no%]dynamic	動的ロード用の変数をコンパイルします [しません]。-xthreadvar=no%dynamic を指定すると、スレッド変数へのアクセスは非常に早くなりますが、動的ライブラリ内のオブジェクトファイルは使用できません。すなわち、実行可能ファイル内のオブジェクトファイルだけが使用可能です。

-xthreadvar を指定しない場合、コンパイラが使用するデフォルトは位置独立コード (PIC) が有効になっているかどうかによって決まります。位置独立コードが有効になっている場合、オプションは -xthreadvar=dynamic に設定されます。位置独立コードが無効になっている場合、オプションは -xthreadvar=no%dynamic に設定されます。

値を指定しないで -xthreadvar を指定すると、オプションは -xthreadvar=dynamic に設定されます。

動的ライブラリ内に位置に依存するコードがある場合、-xthreadvar を指定する必要があります。

リンカーは、動的ライブラリ内の位置依存コード (非 PIC) スレッド変数と同等のスレッド変数はサポートできません。非 PIC スレッド変数は非常に高速なため、実行可能ファイルのデフォルトとして指定できます。

異なるバージョンの Solaris ソフトウェアでスレッド変数を使用するには、コマンド行で異なるオプションを指定する必要があります。

- Solaris 8 ソフトウェアでは、__thread を使用するオブジェクトは -mt を指定してコンパイルし、-mt -L/usr/lib/lwp -R/usr/lib/lwp を指定してリンクする必要があります。
- Solaris 9 ソフトウェアでは、__thread を使用するオブジェクトはコンパイルとリンクに -mt を指定する必要があります。

関連項目: -xcode、-KPIC、-Kpic

B.2.152 -xtime

コンパイルの各構成要素が占有した実行時間と資源を報告します。

B.2.153 -xtransition

K&R C と Solaris Studio ISO C との間の相違に対して警告を出します。

-xtransition オプションを、-Xa または -Xt オプションとともに使用すると警告を出します。異なる動作に関するすべての警告メッセージは適切なコーディングを行うことによって取り除くことができます。次の警告は、-xtransition オプション を使用していなければ表示されません。

- \a は ISO C の「警告」文字です
- \x は ISO C の 16 進エスケープです
- 無効な 8 進数
- 型の種類は実際には *type tag* です: *name*
- コメントが "##" で置き換えられます
- コメントがトークンを連結していません
- ISO C では新しい型で置き換えてしまう型の宣言です: *type tag*
- 文字定数中のマクロ置換
- 文字列リテラル中のマクロ置換
- 文字定数中のマクロ置換は行われません

- 文字列リテラル中のマクロ置換は行われません
- オペランドが符号なしとして処理されました
- 3 文字表記シーケンスが置き換えられました
- ISO C は定数を unsigned 型として扱います: *operator*
- semantics of *operator* change in ISO C; use explicit cast

B.2.154 -xtrigraphs

-xtrigraphs オプションは、コンパイラが ISO 規格で定義されている三文字表記シーケンスを認識するかどうかを決定します。

デフォルトにより、コンパイラは -xtrigraphs=yes を仮定し、コンパイル単位をとおしてすべての三文字表記シーケンスを認識します。

コンパイラが文字表記シーケンスとして解釈している疑問符 (?) の入ったリテラル文字列がソースコードにある場合は、-xtrigraph=no サブオプションを使用して文字表記シーケンスの認識をオフにすることができます。-xtrigraphs=no オプションは、コンパイル単位全体をとおしてすべての三文字表記シーケンスの認識をオフにします。

次の例は、ソースファイル trigraphs_demo.c を示しています。

```
#include <stdio.h>

int main ()
{

  (void) printf("(\?\?) in a string appears as (??)\n");

  return 0;
}
```

このコードに -xtrigraphs=yes を指定してコンパイルした場合の出力は、次のとおりです。

```
example% cc -xtrigraphs=yes trigraphs_demo.c
example% a.out
(??) in a string appears as []
```

このコードに -xtrigraphs=no を指定してコンパイルした場合の出力は、次のとおりです。

```
example% cc -xtrigraphs=no trigraphs_demo.c
example% a.out
(??) in a string appears as (??)
```

B.2.155 -xunroll=*n*

ループを *n* 回展開するようオプティマイザに指示します。*n* は正の整数です。*n* が 1 のときはコマンドとなり、コンパイラはループを展開しません。*n* が 2 以上のとき、-xunroll=*n* は *n* 回ループを展開することをコンパイラに知らせます。

B.2.156 -xustr={ascii_utf16_ushort|no}

ISO10646 UTF-16 文字列リテラルを使用する国際化アプリケーションをサポートする必要がある場合は、このオプション使用します。言い替えれば、このオプションは、オブジェクトファイル内で UTF-16 文字列に変換したい文字列リテラルがコードに含まれる場合に使用します。このオプションが指定されていない場合、コンパイラは 16 ビット文字列リテラルの生成、認識のいずれも行いません。このオプションは、U"*ASCII_string*" という書式の文字列リテラルを unsigned short int 型の配列として認識できるようにします。このような文字列は標準として規定されていないので、このオプションは標準に準拠しない C++ の認識を可能にします。

U"*ASCII_string*" 文字列リテラルのコンパイラによる認識を無効にすることができます。-xustr=noこのオプションのコマンド行の右端にあるインスタンスは、それまでのインスタンスをすべて上書きします。

デフォルトは -xustr=no です。引数を指定しないで -xustr を指定した場合、コンパイラはこの指定を受け付けず、警告を出力します。C または C++ 規格で構文の意味が定義されると、デフォルト値が変わることがあります。

-xustr=ascii_utf16_ushort を指定して U"*ASCII_string*" 文字列リテラルを指定しなくても、エラーにはなりません。

すべてのファイルを、このオプションによってコンパイルしなければならないわけではありません。

次の例では、U を付加した二重引用符で文字列リテラルを示します。また、-xustr を指定するコマンド行も示します。

```
example% cat file.c
const unsigned short *foo = U"foo";
const unsigned short bar[] = U"bar";
const unsigned short *fun() { return foo;}
example% cc -xustr=ascii_utf16_ushort file.c -c
```

8 ビットの文字列リテラルに u を付加して、unsigned short 型を持つ 16 ビットの UTF-16 文字を形成できます。次に例を示します。

```
const unsigned short x = U’x’;
const unsigned short y = U’\x79’;
```

B.2.157 -xvector[= *a*]

ベクタライブラリ関数の呼び出しの自動生成や、SIMD (Single Instruction Multiple Data) 命令の生成ができます。このオプションを使用するときは -fround=nearest を指定することによって、デフォルトの丸めモードを使用する必要があります。

a は、次の指定と同じです。

表 B–43 -xvector のフラグ

値	意味
[no%]lib	コンパイラは可能な場合はループ内の数学ライブラリへの呼び出しを、同等のベクトル数学ルーチンへの単一の呼び出しに変換します [しません]。大きなループカウントを持つループでは、これによりパフォーマンスが向上します。(Solaris のみ)
[no%]simd	コンパイラはネイティブ x86 SSE SIMD 命令を使用して特定のループのパフォーマンスを向上させます [させません]。ストリーミング拡張機能は、x86 で最適化レベルが 3 かそれ以上に設定されている場合にデフォルトで使用されます。サブオプション no%simd を使用すると、この機能を無効にできます。 コンパイラは、ストリーミング拡張機能がターゲットのアーキテクチャーに存在する場合、つまりターゲットの ISA が SSE2 以上である場合にのみ SIMD を使用します。たとえば、最新のプラットフォームで -xtarget=woodcrest、–xarch=generic64、-xarch=sse2、-xarch=sse3、または -fast を指定して使用できます。ターゲットの ISA にストリーミング拡張機能がない場合、このサブオプションは無効です。
yes	これは非推奨であり、代わりに -xvector=lib を指定します。
no	これは非推奨であり、代わりに -xvector=none を指定します。

デフォルトは、x86 では -xvector=simd で、SPARC プラットフォームでは -xvector=%none です。サブオプションなしで -xvector を指定すると、コンパイラでは、x86 では -xvector=simd,lib、SPARC (Solaris) では -xvector=lib、および -xvector=simd (Linux) が使用されます。

–xvector オプションを指定するには、最適化レベルが —xO3 かそれ以上に設定されていることが必要です。最適化レベルが指定されていない場合や **–x03** よりも低い場合はコンパイルは続行されず、メッセージが表示されます。

B.2.158 -xvis

(SPARC) VIS instruction-set Software Developers Kit (VSDK) に定義されているアセンブリ言語のテンプレートを使用する場合は、-xvis=[yes|no] コマンドを使用します。デフォルトは -xvis=no です。-xvis と指定すると -xvis=yes と指定した場合と同様の結果が得られます。

VIS 命令セットは、SPARC v9 命令セットの拡張です。UltraSPARC プロセッサは 64 ビットでも、多くの場合、特にマルチメディアアプリケーションではデータサイズが 8 ビットまたは 16 ビットに制限されています。VIS 命令では、1 命令で 4 つの 16 ビットデータを処理できるので、画像、線形代数、信号処理、オーディオ、ビデオ、ネットワーキングなどの新しいメディアを扱うアプリケーションのパフォーマンスが大幅に向上します。

VSDK の詳細については、http://www.sun.com/processors/vis/ を参照してください。

B.2.159 -xvpara

OpenMP を使用する場合に正しくない結果をもたらす可能性のある、並列プログラミングに関連する潜在的な問題に関して、警告を発行します。-xopenmp および OpenMP API 指令とともに使用します。

次の状況が検出された場合は、コンパイラは警告を発行します。

- 異なるループ繰り返し間でデータに依存関係がある場合に、MP 指令を使用して並列化されたループ。
- OpenMP データ共有属性節に問題がある場合。たとえば、「shared」と宣言された変数に、OpenMP 並列領域からアクセスするとデータ競合が発生する可能性がある場合や、並列領域の中に値を持つ変数を「private」と宣言し、並列領域よりあとでその変数を使用する場合です。

すべての並列化命令が問題なく処理される場合、警告は表示されません。

次に例を示します。

```
cc -xopenmp -vpara any.c
```

注 – Solaris Studio のコンパイラは OpenMP 2.5 API の並列化をサポートします。そのため、MP プラグマ命令は非推奨で、サポートされません。OpenMP API への移植については、『OpenMP API ユーザーズガイド』を参照してください。

B.2.160 -Y*c*, *dir*

c を検索するための新しい *dir* を指定します。*c* は -W オプションで示したコンパイラ構成要素を表わす文字です。

構成要素の検索が指定されている場合、ツールのパス名は *dir/tool* になります。2 つ以上の `-Y` オプションが 1 つの項目に適用されている場合には、最後に現れたものが有効です。

B.2.161 `-YA,` *dir*

コンパイラのすべての構成要素の検索場所にするディレクトリ *dir* を指定します。*dir* 内で構成要素が見つからない場合は、コンパイラがインストールされているディレクトリに戻って検索されます。

B.2.162 `-YI,` *dir*

インクルードファイルを検索するデフォルトのディレクトリを変更します。

B.2.163 `-YP,` *dir*

ライブラリファイルを検索するデフォルトのディレクトリを変更します。

B.2.164 `-YS,` *dir*

起動用のオブジェクトファイルを検索するデフォルトのディレクトリを変更します。

B.2.165 `-Zll`

(SPARC) `lock_lint` 用にプログラムデータベースを作成しますが、実行可能なコードは生成しません。詳細は、`lock_lint`(1) のマニュアルページを参照してください。

B.3 リンカーに渡されるオプション

`cc` は `-a`、`-e`、`-r`、`-t`、`-u`、`-z` を認識し、これらのオプションとその引数を `ld` に渡します。`cc` は認識できないオプションを警告付きで `ld` に渡します。Solaris プラットフォームでは、**`-i`** オプションとその引数もリンカーに渡されます。

◆ ◆ ◆ 付録 C

C ISO/IEC C 99 の処理系定義の動作

『Programming Language - C (ISO/IEC 9899:1999)』規格には、C 言語で記述されたプログラムの構文と解釈が規定されています。この付録では、それらの動作を詳しく説明します。ISO/IEC 9899:1999 規格そのものとすぐに比較できるよう、この付録では、すべての節の見出しにセクション番号を付記しています。

- 各節の見出しには、ISO 規格にあるものと同じ *letter.number* 識別子を使用しています。
- 各節では、処理系で定義すべきこととして ISO 規格に規定されている要件を示しています。この要件のあとに、Sun の処理系の説明があります。

C.1 処理系定義の動作 (J.3)

この従属節で一覧になっている各分野での動作の選択肢を文書化するには、処理系に従う必要があります。処理系定義の動作は次のとおりです。

C.1.1 翻訳 (J.3.1)

- (3.10, 5.1.1.3) 診断の識別方法

  エラーメッセージは次の書式です。

  filename, *line number*: *message*

  filename は、そのエラーまたは警告があるファイルの名前です

  line number はエラーまたは警告が検出された行の番号、*message* は診断メッセージです。
- (5.1.1.2) 翻訳段階 3 で改行以外の非空の空白文字の連続を保持するか、またはスペース文字 1 つに置き換えるかどうか。

  タブ (\t) やフォームフィード (\f)、垂直タブ (\v) からなる非空の文字の連続をスペース文字 1 文字に置き換えます。

C.1.2 環境 (J.3.2)

- (5.1.1.2) 翻訳段階 1 における物理ソースファイルの複数バイト文字とソース文字セットのマッピング。

  ASCII 部分では、1 文字に 8 ビットです。ロケール固有の拡張文字部分では、ロケール固有の 8 ビットの倍数です。
- (5.1.2.1) 自立した環境でプログラム起動時に呼び出す関数の名前と種類。

  ホスト環境に実装されています。
- (5.1.2.1) 自立した環境でのプログラム終了の処理。

  ホスト環境に実装されています。
- (5.1.2.2.1) `main` 関数を定義する代わりの方法。

  規格に定義されている以外の `main` の定義方法はありません。
- (5.1.2.2.1) main の `argv` 引数が指し示す文字列に与える値。

  `argv` は、コマンド行引数へのポインタからなる配列です。`argv[0]` はプログラム名を表します (該当する場合)。
- (5.1.2.3) 対話型デバイスを構成するもの。

  対話型デバイスにはシステムライブラリコールの `isatty()` が 0 以外の値を返します。
- (7.14) シグナルとその意味、デフォルトの処理。

  次の表に `signal` 関数が認識する各シグナルの意味を示します。

表 C–1 `signal` 関数のシグナルの意味

シグナル番号	デフォルトのイベント	シグナルの意味
`SIGHUP 1`	終了	ハングアップ
`SIGINT 2`	終了	割り込み (rubout)
`SIGQUIT 3`	コア	終了 (ASCII FS)
`SIGILL 4`	コア	不当な命令 (捕捉されてもリセットされない)
`SIGTRAP 5`	コア	トレーストラップ (捕捉されてもリセットされない)
`SIGIOT 6`	コア	IOT 命令
`SIGABRT 6`	コア	異常終了時に使用
`SIGEMT 7`	コア	EMT 命令
`SIGFPE 8`	コア	浮動小数点の例外
`SIGKILL 9`	終了	強制終了 (捕捉または無視できない)

表 C–1 signal 関数のシグナルの意味 (続き)

シグナル番号	デフォルトのイベント	シグナルの意味
SIGBUS 10	コア	バスエラー
SIGSEGV 11	コア	セグメンテーション違反
SIGSYS 12	コア	システムコールへの引数誤り
SIGPIPE 13	終了	読み手のないパイプ上への書き込み
SIGALRM 14	終了	アラームクロック
SIGTERM 15	終了	プロセスの終了によるソフトウェアの停止
SIGUSR1 16	終了	ユーザー定義のシグナル 1
SIGUSR2 17	終了	ユーザー定義のシグナル 2
SIGCLD 18	無視	子プロセス状態の変化
SIGCHLD 18	無視	子プロセス状態の変化の別名 (POSIX)
SIGPWR 19	無視	電源障害による再起動
SIGWINCH 20	無視	ウィンドウサイズの変更
SIGURG 21	無視	ソケットの緊急状態
SIGPOLL 22	終了	ポーリング可能なイベント発生
SIGIO 22	Sigpoll	ソケット入出力可能
SIGSTOP 23	停止	停止 (キャッチまたは無視できない)
SIGTSTP 24	停止	tty より要求されたユーザーストップ
SIGCONT 25	無視	停止していたプロセスの継続
SIGTTIN 26	停止	バックグラウンド tty の読み込みを試みた
SIGTTOU 27	停止	バックグラウンド tty の書き込みを試みた
SIGVTALRM 28	終了	仮想タイマーの時間切れ
SIGPROF 29	終了	プロファイリングタイマーの時間切れ
SIGXCPU 30	コア	CPU の限界をオーバー
SIGXFSZ 31	コア	ファイルサイズの限界をオーバー
SIGWAITING 32	無視	スレッド処理コードで使われていた予約シグナル
SIGLWP 33	無視	スレッド処理コードで使われていた予約シグナル
SIGFREEZE 34	無視	チェックポイント一時停止
SIGTHAW 35	無視	チェックポイント再開

表 C–1 signal 関数のシグナルの意味 (続き)

シグナル番号	デフォルトのイベント	シグナルの意味
SIGCANCEL 36	無視	スレッドライブラリで使われている取り消しシグナル
SIGLOST 37	無視	リソースがない (レコードロックがない)
SIGXRES 38	無視	リソース制御超過 (setrctl(2) を参照)
SIGJVM1 39	無視	Java Virtual Machine 用に予約 1
SIGJVM2 40	無視	Java Virtual Machine 用に予約 2

- (7.14.1.1) SIGFPE、SIGILL、および SIGSEGV 以外の、演算例外に対応するシグナル値

  SIGILL、SIGFPE、SIGSEGV、SIGTRAP、SIGBUS、SIGEMT。表 C–1 を参照。

- プログラムの起動時に signal に相当するもの (sig、SIG_IGN) が実行される際のシグナル (7.14.1.1)。

  SIGILL、SIGFPE、SIGSEGV、SIGTRAP、SIGBUS、SIGEMT。表 C–1 を参照。

- (7.20.4.5) 環境名および、getenv 関数が使用する環境リストの変更方法。

  マニュアルページの environ(5) に環境名の一覧を記載しています。

- (7.20.4.6) system 関数による文字列の実行方法。

  system(3C) のマニュアルページからの抜粋

  system() 関数は、端末からコマンドとして入力されかのように *string* を入力としてシェルに渡します。この呼び出し側は、シェルが完了するのを待ち、waitpid(2) が指定する形式でシェルの終了ステータスを返します。

  string が null ポインタの場合、system() はシェルが存在し、実行可能かどうかを調べます。シェルが使用可能な場合は、system () によってゼロ以外の値を返し、そうでない場合は、0 を返します。

C.1.3 識別子 (J.3.3)

- (6.4.2) 識別子に追加で使用される複数バイト文字とその汎用文字名との対応。

  なし

- (5.2.4.1, 6.4.2) 識別子の有効初期文字数。

  1023

C.1.4 文字 (J.3.4)

- (3.6) 1 バイトのビット数。

  1 バイトは 8 ビットです。
- (5.2.1) 実行文字セットのメンバーの値。

  ASCII 部分では、配置はソース文字と実行文字と同様です。
- (5.2.2) 各標準の英字エスケープシーケンス用に生成される実行文字セットのメンバーの固有値。

表 C–2 標準の英字エスケープシーケンスの固有値

エスケープシーケンス	固有値
\a (アラート)	7
\b (バックスペース)	8
\f (フォームフィード)	12
\n (改行)	10
\r (復帰)	13
\t (水平タブ)	9
\v (垂直タブ)	11

- (6.2.5) 基本実行文字セットのメンバー以外の文字が格納されている char オブジェクトの値。

  char オブジェクトに割り当てられている文字に関連付けられている下位 8 ビットの数値です。
- (6.2.5, 6.3.1.1) signed char または unsigned char のどちらが単純 char と同じ範囲、表現、および動作を持つか。

  signed char が通常の char として処理されます。
- (6.4.4.4, 5.1.1.2) ソース文字セット (文字定数と文字列リテラル) のメンバーの実行文字セットメンバーへのマッピング。

  ASCII 部分では、配置はソース文字と実行文字と同様です。
- (6.4.4.4) 複数の文字、または単一バイトの実行文字にマッピングされていない文字またはエスケープシーケンスを含む整数文字定数の値。

  エスケープシーケンスの発生しない複数バイト文字セットの値は、各文字の示す数値から派生しています。
- (6.4.4.4) 複数の複数バイト文字、または拡張実行文字セットで表現されていない複数バイト文字またはエスケープシーケンスを含むワイド文字定数の値。

エスケープシーケンスではない複数文字のワイド文字定数は、各文字の数値から得られる値を持ちます。

- (6.4.4.4) 拡張実行文字セットのメンバーにマッピングする単一複数バイト文字からなるワイド文字定数を、対応するワイド文字コードに変換するために使用する標準ロケール。

  LC_ALL、LC_CTYPE、 LANG 環境変数のいずれかで指定したロケールが標準で使用されます。

- (6.4.5) ワイド文字列リテラルを、対応するワイド文字コードに変換するのに使用する標準ロケール。

  LC_ALL、LC_CTYPE、LANG 環境変数のいずれかで指定されたロケールが標準で使用されます。

- (6.4.5) 拡張実行文字セットで表現されていない複数バイト文字またはエスケープシーケンスを含む文字列リテラルの値。

  複数バイト文字の各バイトが文字列リテラルの 1 文字を表し、この文字は、複数バイト文字のそのバイトの数値に等しい値を持ちます。

C.1.5 整数 (J.3.5)

- (6.2.5) 実装に存在する拡張整数型。

  なし

- (6.2.6.2) 符号付き整数型を符号と絶対値、2 の補数、1 の補数のどれで表現するか、また規格外の値をトラップ表現または通常値のどちらにするか。

  符号付き整数型は 2 の補数で表します。規格外の値は通常値になります。

- (6.3.1.1) 任意の拡張整数型と同じ精度を持つ別の拡張整数型との相対的なランク。

  この実装に該当するものはありません。

- (6.3.1.3) 整数を符号付き整数型に変換し、値がその型のオブジェクトで表現できない場合の結果、または立てられるシグナル。

  整数がより短い符号付き整数に変換される場合は、長い方の整数の下位ビットが短い方の符号付き整数に複写されます。結果は負になることがあります。

  符号なし整数が同サイズの符号付き整数に変換される場合は、符号なし整数の下位ビットが符号付き整数に複写されます。結果は負になることがあります。

- (6.5) 符号付き整数に対するビット単位演算の結果。

  ビット単位演算を符号付きの型に適用すると、符号ビットを含むオペランドのビット単位演算となります。その結果の各ビットは、両オペランドの対応するビットが設定されていた場合にのみ設定されます。

C.1.6 浮動小数点 (J.3.6)

- (5.2.4.2.2) 浮動小数点の結果を返す浮動小数点演算、および <math.h> と <complex.h> のライブラリ関数の精度。

  浮動小数点演算の精度は FLT_EVAL_METHOD の設定に合わせられます。<math.h> および <complex.h> のライブラリ関数の精度は、libm(3LIB) のマニュアルページに示されているとおりです。

- (5.2.4.2.2) FLT_ROUNDS の規格外の値に対する丸め動作。

  この実装に該当するものはありません。

- (5.2.4.2.2) FLT_EVAL_METHOD の規格外の負の値に対する評価方法。

  この実装に該当するものはありません。

- (6.3.1.4) 整数を、元の値を正確に表現できない浮動小数点数に変換したときの丸め方向。

  そのとき有効な丸め方向モードに従います。

- (6.3.1.5) 浮動小数点数を短い浮動小数点数に変換した場合の丸め方向。

  そのとき有効な丸め方向モードに従います。

- (6.4.4.2) 特定の浮動小数点定数を表現する方法 (表現可能な最近似値か、最近似値にもっとも近い最大または最小値)。

  浮動小数点定数はつねに表現可能な最近似値に丸められます。

- (6.5) FP_CONTRACTプラグマが許可していない場合に浮動小数点式を短縮するかどうか、また、短縮する場合はその方法。

  この実装に該当するものはありません。

- (7.6.1) FENV_ACCESS プラグマのデフォルトの状態。

  -fsimple=0 の場合、デフォルトは ON です。それ以外の -fsimple のほかのすべての値で FENV_ACCESS のデフォルト値は OFF です。

- (7.6, 7.12) 追加の浮動小数点例外、丸めモード、環境、分類、マクロ名。

  この実装に該当するものはありません。

- (7.12.2) FP_CONTRACTプラグマのデフォルトの状態。

  -fsimple=0 の場合、デフォルトは OFF です。それ以外の -fsimple のほかのすべての値で FP_CONTRACT のデフォルト値は ON です。

- (F.9) IED 60559 準拠の実装で丸め結果が実際には数学的な演算結果に等しくない場合に「不正確」の浮動小数点例外が立てられるかどうか。

  結果の判定は不可能です。

- (F.9) IEC 60559 準拠の実装で結果が小さいが不正確でない場合に、アンダーフロー (および「不正確」) の浮動小数点例外を立てられるかどうか。

  アンダーフロー時のトラップが無効 (デフォルト) の場合、このようなケースでハードウェアはアンダーフローや不正確の例外を立てません。

C.1.7 配列とポインタ (J.3.7)

- (6.3.2.3) 整数または -Xarch=v9への変換またはその逆の変換結果。

  ポインタおよび整数の変換では、ビットパターンは変化しません。整数またはポインタ型で結果を表現できない場合を除きますが、結果は定義されていません。
- (6.5.6) 同じ配列の要素への 2 つのポインタを減算した結果のサイズ。

  stddef.h で定義されているとおり int 型です。 -Xarch=v9 の場合は long 型です。

C.1.8 ヒント (J.3.8)

- (6.7.1) レジスタ記憶クラス指示子で行う推奨を有効にする範囲。

  有効なレジスタ宣言の数は使用パターンおよび各関数における定義に依存し、割り当て可能なレジスタ数に制限されます。コンパイラやオプティマイザは、レジスタ宣言に従う必要はありません。
- (6.7.4) inline 関数指示子で行う推奨を有効にする範囲。

  inline キーワードは、最適化でコードのインライン化が発生し、インライン化のメリットがあるとオプティマイザが判断したときにのみ有効です。最適化のオプションのリストについては、213 ページの「A.1.1 最適化とパフォーマンスのオプション」を参照してください。

C.1.9 構造体、共用体、列挙型、およびビットフィールド (J.3.9)

- (6.7.2, 6.7.2.1) 単純な int 型ビットフィールドを signed int 型 ビットフィールドまたは unsigned int ビットフィールドのどちらにみなすか。

  unsigned int とみなされます。
- (6.7.2.1) _Bool、signed int、および unsigned int 以外に使用可能なビットフィールドの型。

  ビットフィールドは任意の整数型として宣言できます。
- (6.7.2.1) ビットフィールドが記憶装置の境界を越えられるかどうか。

  ビットフィールドは記憶装置の境界を越えません。
- (6.7.2.1) ユニット内のビットフィールドの割り当て順序。

  ビットフィールドは、記憶装置内で高位から低位の順に割り当てられます。
- (6.7.2.1) 構造体のビットフィールド以外のメンバーの整列条件。1 つの実装で書き込まれたバイナリデータが別の実装で読み取られないかぎり、このことは問題になりません。

表 C–3 構造体メンバーのパディングと整列

種類	整合の境界	バイト境界
`char` と `_Bool`	byte	1
`short`	halfword	2
`int`	word	4
`long -m32`	word	4
`long -m64`	doubleword	8
`float`	word	4
`double -m64`	doubleword	8
`double` (SPARC) `-m32`	doubleword	8
`double` (x86) `-m32`	doubleword	4
`long double` (SPARC) `-m32`	doubleword	8
`long double` (x86) `-m32`	word	4
`long double -m64`	quadword	16
`pointer -m32`	word	4
`pointer -m64`	quadword	8
`long long –m64`	doubleword	8
`long long` (x86) `-m32`	word	4
`long long` (SPARC) `-m32`	doubleword	8
`_Complex float`	word	4
`_Complex double -m64`	doubleword	8
`_Complex double` (SPARC) `-m32`	doubleword	8
`_Complex double` (x86) `-m32`	doubleword	4
`_Complex long double -m64`	quadword	16
`_Complex long double` (SPARC) `-m32`	quadword	8
`_Complex long double` (x86) `-m32`	quadword	4
`_Imaginary float`	word	4
`_Imaginary double -m64`	doubleword	8

表 C–3 構造体メンバーのパディングと整列 (続き)

種類	整合の境界	バイト境界
`_Imaginary double` (x86) `-m32`	doubleword	4
`_Imaginary` (SPARC) `-m32`	doubleword	8
`_Imaginary long double` (SPARC) `-m32`	doubleword	8
`_Imaginary long double -m64`	quadword	16
`_Imaginary long double` (x86) `-m32`	word	4

- (6.7.2.2) 各列挙型と互換性のある整数型。

  int 型です。

C.1.10 修飾子 (J.3.10)

- (6.7.3) volatile 修飾型のオブジェクトへのアクセス。

  オブジェクト名を参照するたびに、そのオブジェクトへアクセスされます。

C.1.11 前処理指令 (J.3.11)

- (6.4.7) 両方の形式のヘッダー名のシーケンスをヘッダーまたは外部ソースファイル名にマッピングする方法。

  ソースファイルの文字は対応する ASCII の値に配置されます。
- (6.10.1) 条件付きのインクルードを制御する定数式の文字定数の値が、実行文字セット中の同一の文字定数の値に一致するかどうか。

  前処理命令内の文字定数はほかの式のものと同一の数値を持ちます。
- (6.10.1) 条件付きのインクルードを制御する定数式の単一文字の文字定数が負の値をとることがあるかどうか。

  この場合の文字定数は負の値を取ることがあります。
- (6.10.2) インクルードする、< > 区切りのヘッダーの検索場所と、その場所の指定方法、ほかのヘッダーの識別方法。

  ヘッダーファイルの場所は、コマンド行のオプションの指定と、`#include` 指令内に現れるファイルに依存します。詳細は、62 ページの「2.16 インクルードファイルを指定する方法」を参照してください。
- (6.10.2) インクルードする " " 区切りのヘッダー内での指定されたソースファイルの検索方法。

ヘッダーファイルの場所は、コマンド行のオプションの指定と、#include 指令内に現れるファイルに依存します。詳細は、62 ページの「2.16 インクルードファイルを指定する方法」を参照してください。

- (6.10.2) #include 指令内の前処理トークン (マクロ展開で生成されることもある) からヘッダー名を形成する方法。

  62 ページの「2.16 インクルードファイルを指定する方法」で説明しているように、ヘッダー名 (空白を含む) を形成するすべてのトークンは、ヘッダーを検索する際に使用するファイルパスとみなされます。

- (6.10.2) #include 処理の入れ子制限。

  コンパイラによる制限はありません。

- (6.10.3.2) 文字定数または文字列定数に # 演算子があるとき、汎用文字名で始まる \ 文字の前に \ 文字を挿入するかどうか。

  いいえ。

- (6.10.6) 非 STDC の #pragma 指令が認識されたときの動作。

  非 STDC の#pragma 指令が認識されたときの動作については、45 ページの「2.11 プラグマ」を参照してください。

- (6.10.8) 翻訳の日付と時刻がわからないときの __DATE__ と __TIME__ の定義。

  これらのマクロは常に使用できます。

C.1.12 ライブラリ関数 (J.3.12)

- (5.1.2.1) 第 4 節で規定されている最小セット以外に自立したプログラムから使用可能なライブラリ機能。

  ホスト環境に実装されています。

- (7.2.1.1) 表明マクロが出力する診断の形式。

  診断は次のような形式になっています。

  Assertion failed: *statement*. file *filename*, *line number*, *function name*

  statement は表明が失敗した文、*filename* は __FILE__ の値です。*line number* は __LINE__、*function name* は __func__ の値です。

- (7.6.2.2) fegetexceptflag 関数が格納する浮動小数点ステータスフラグの表現。

  fegetexceptflag によってステータスフラグに格納された各例外は、定数のすべての組み合わせのビット単位 OR が明確な値を持つように、値を持つ整定数式に展開されます。

- (7.6.2.3) 「オーバーフロー」または「アンダーフロー」浮動小数点例外のほかに feraiseexcept 関数によって「不正確」浮動小数点例外が立てられるかどうか。

  「不正確」の例外は立てられません。

- (7.11.1.1) setlocale 関数への第 2 引数として渡すことが可能な、「C」および "" 以外の文字列。

  意図的に空白にします。

- (7.12) FLT_EVAL_METHODマクロの値がゼロ未満か 2 より大きい場合に float_t および double_t に定義される型。

  - SPARC の場合、型は次のとおりです。

    ```
    typedef float float_t;

    typedef double double_t;
    ```

  - x86 の場合、型は次のとおりです。

    ```
    typedef long double float_t;

    typedef long double double_t;
    ```

  (7.12.1) この国際規格で規定されている以外の、数学関数のドメインエラー。

  入力引数が 0 か +/--Inf、 NaN のどれかの場合、ilogb()、ilogbf()、および ilogbl() は不正の例外を立てます。

- (7.12.1) ドメインエラー時に数学関数が返す値。

  完全な C99 モード (-xc99=%all,lib) でのドメインエラー時に返される『Programming Language - C (ISO/IEC 9899:1999)』の Annex F の仕様どおりです。

- アンダーフロー範囲エラー時に数学関数が返す値と、整数式の math_errhandling & MATH_ERRNO がゼロ以外の場合に errno に ERANGE マクロの値が設定されるかどうか、また整数式の math_errhandling & MATH_ERREXCEPT がゼロ以外の場合に「アンダーフロー」浮動小数点例外が立てられるかどうか。 (7.12.1)

  アンダーフロー範囲エラーについて: 値が非正規数の可能性がある場合は、その非正規数が返され、それ以外の場合は、適宜 +-0 が返されます。

  整数式の math_errhandling & MATH_ERRNO がゼロ以外の場合に errno に ERANGE マクロの値が設定されるかどうかについて: Sun の実装では、(math_errhandling & MATH_ERRNO) == 0 のため、この部分は該当しません。

  整数式の math_errhandling & MATH_ERREXCEPT がゼロ以外の場合に「アンダーフロー」浮動小数点例外が立てられるかどうかについて: 浮動小数点アンダーフローとともに精度が失われた場合に例外が立てられます。

- (7.12.10.1) fmod 関数の第 2 引数が 0 の場合に、ドメインエラーとなるか、0 が返されるか。

  ドメインエラーが発生します。

- (7.12.10.3) 商を約分する際に remquo 関数が使用する係数の 2 を底とする対数。

  31。

- (7.14.1.1) シグナルハンドラを呼び出す前に signal(sig, SIG_DFL); 相当のものを実行するかどうか、また実行されない場合は、実行されるシグナルのブロック処理。

シグナルハンドラを呼び出す前に signal(sig, SIG_DFL); 相当を実行します。

- (7.17) マクロ NULL を展開したときの null ポインタ定数。

  NULL は 0 に展開されます。

- (7.19.2) テキストストリームの最終行で改行文字による終了を必要とするか。

  最終行を改行文字で終了する必要はありません。

- (7.19.2) テキストストリームへの書き出しでスペース文字が改行文字の直前にあった場合、そのスペース文字が読み込みで表示されるかどうか。

  ストリームが読み込まれるときにはすべての文字が表示されます。

- (7.19.2) バイナリストリームに書き込まれるデータに追加することのできる null 文字の数。

  バイナリストリームには null 文字を追加しません。

- (7.19.3) 当初、アペンドモードのストリームのファイル位置指示子がファイルの始まりと終わりのどちらに置かれるか。

  ファイル位置指示子は最初にファイルの終わりに置かれます。

- (7.19.3) テキストストリームへの書き込みを行うと、関係するファイルの内容が書き込み点以降切り捨てられるか。

  ハードウェアの命令がないかぎり、テキストストリームへの書き込みによって書き込み点以降の関連ファイルが切り捨てられることはありません。

- (7.19.3) ファイルバッファリングの特性。

  標準エラーストリーム (stderr) を除く出力ストリームは、デフォルトでは、出力がファイルの場合にはバッファリングされ、出力が端末の場合には行バッファリングされます。標準エラー出力ストリーム (stderr) は、デフォルトではバッファリングされません。

  バッファリングされた出力ストリームは多くの文字を保存し、その文字をブロックとして書き込みます。バッファリングされなかった出力ストリームは宛先ファイルあるいは端末に迅速に書き込めるように情報の待ち行列を作ります。行バッファリングされた出力は、その行が完了するまで (改行文字が要求されるまで) 行単位の出力待ち行列に入れられます。

- (7.19.3) 長さゼロのファイルが実際に存在するかどうか。

  ディレクトリエントリを持つという意味ではゼロ長ファイルは存在します。

- (7.19.3) 有効なファイル名の作成規則。

  有効なファイル名は 1 から 1,023 文字までの長さで、NULL 文字とスラッシュ (/) 以外のすべての文字を使用することができます。

- (7.19.3) 同一のファイルを同時に複数回開くことが可能か。

  同一のファイルを何回も開くことができます。

- (7.19.3) ファイル内の複数バイト文字に使用する符号化方式の性質と選択方法。

  複数バイト文字に使用する符号化方式は、ファイルごとに同じです。

- (7.19.4.1) 開いたファイルに対する remove 関数の処理。

  ファイルを閉じる最後の呼び出しによりファイルが削除されます。すでに除去されたファイルをプログラムが開くことはできません。

- (7.19.4.2) rename 関数を呼び出す前に新しい名前を持つファイルがあった場合の処理。

  そのようなファイルがあれば削除され、新しいファイルが元のファイルの上に書き込まれます。

- (7.19.4.3) プログラムの異常終了時に開いていた一時ファイルが削除されるかどうか。

  ファイルの作成とリンク解除の間にプロセスが終了した場合は、ファイルがそのまま残ることがあります。freopen(3C) のマニュアルページを参照してください。

- (7.19.5.4) 許されるモードの変更とその状況。

  ストリームの基になっているファイル記述子のアクセスモードに従って、次のモード変更が許されます。

  - + が指定されている場合、ファイル記述子モードは O_RDWR である必要があります。
  - r が指定されている場合、ファイル記述子モードは O_RDONLY か O_RDWR である必要があります。
  - a または w が指定されている場合、ファイル記述子モードは O_WRONLY か O_RDWR である必要があります。

    freopen(3C) のマニュアルページを参照してください。

  (7.19.6.1, 7.24.2.1) 無限または NaN の出力に使用する形式と NaN で出力する n-char または n-wchar シーケンスの意味。

  [-]Inf、[-]NaN です。F 変換指示子がある場合は、[-]INF、[-]NAN になります。

- (7.19.6.1, 7.24.2.1) fprintf または fwprintf 関数における %p 変換の出力。

  %p の出力は %x と等しくなります。

- (7.19.6.2, 7.24.2.1) fscanf() または fwscanf() 関数における %[変換のスキャンリストで、先頭文字でも最終文字でもなく、また先頭文字が ^ の場合に 2 番目の文字でもない - 文字の解釈。

  - がスキャンリストにあり、先頭文字でも最終文字でもなく、^ が先頭文字の場合に 2 番目の文字でない場合は、一致とみなす文字の範囲を示します。

  fscanf(3C) のマニュアルページを参照してください。

- (7.19.6.2, 7.24.2.2) fscanf() または fwscanf() 関数の %p 変換で一致とみなされるシーケンスと、対応する入力項目の解釈。

対応する printf(3C) 関数の %p 変換で生成されるシーケンスと同じシーケンスを一致とみなします。対応する引数は、void 型ポインタへのポインタである必要があります。入力項目が同じプログラムの実行中の以前に変換された値の場合、生成されるポインタはその値に等しいとみなされます。それ以外の場合の %p 変換の動作は定義されていません。

fscanf(3C) のマニュアルページを参照してください。

- (7.19.9.1, 7.19.9.3, 7.19.9.4) エラー時に fgetpos、fsetpos、ftell がマクロ errno に設定する値。
  - EBADF ストリームの基になっているファイル記述子が不正です。fgetpos(3C) のマニュアルページを参照してください。
  - ESPIPE ストリームの基になっているファイル記述子がパイプか FIFO、ソケットに関連付けられています。fgetpos(3C) のマニュアルページを参照してください。
  - EOVERFLOW fpos_t 型のオブジェクトでは、ファイル位置の現在の値を正しく表現できません。fgetpos(3C) のマニュアルページを参照してください。
  - EBADF ストリームの基になっているファイル記述子が不正です。fsetpos(3C) のマニュアルページを参照してください。
  - ESPIPE ストリームの基になっているファイル記述子がパイプか FIFO、ソケットに関連付けられています。 fsetpos(3C) のマニュアルページを参照してください。
  - EBADF ストリームの基になっているファイル記述子が開いているファイルの記述子ではありません。ftell(3C) のマニュアルページを参照してください。
  - ESPIPE ストリームの基になっているファイル記述子がパイプか FIFO、ソケットに関連付けられています。ftell(3C) のマニュアルページを参照してください。
  - EOVERFLOW long 型のオブジェクトでは、現在のファイルオフセット値を正しく表現できません。ftell(3C) のマニュアルページを参照してください。

  (7.20.1.3, 7.24.4.1.1) strtod()、 strtof()、 strtold()、 wcstod()、 wcstof()、または wcstold() 関数によって変換される NaN を表す文字列内の n-char または n-wchar シーケンスの意味。

  n-char シーケンスには特別な意味は与えられていません。
- (7.20.1.3, 7.24.4.1.1) アンダーフローが発生したとき、strtod、strtof、strtold、wcstod、wcstof、または wcstold 関数が errno に ERANGE を設定するかどうか。

  アンダーフロー時、errno には ERANGE が設定されます。
- (7.20.3) 要求サイズがゼロの場合、calloc、malloc、および realloc 関数が null ポインタまたは割り当てオブジェクトへのポインタのどちらを返すか。

  null ポインタか free() に渡すことが可能な一意のポインタが返されます。

malloc(3C) のマニュアルページを参照してください。

- (7.20.4.1, 7.20.4.4) abort または _Exit 関数が呼び出されたときに、バッファー内にまだ書き出されていないデータがある開いているストリームをフラッシュするか、開いているストリームを閉じるか、一時ファイルを削除するか。

  異常終了処理では、開いているすべてのストリームに対して少なくとも fclose(3C) の処理が行われます。abort(3C) のマニュアルページを参照してください。

  開いているストリームが閉じられます。フラッシュはされません。_Exit(2) のマニュアルページを参照してください。

- (7.20.4.1, 7.20.4.3, 7.20.4.4) abort、exit、または _Exit 関数がホスト環境に返す終了ステータス。

abort によって wait(3C) または waitpid(3C) から使用できるようにされたステータスが、SIGABRT シグナルで終了されたプロセスのステータスになります。abort(3C)、exit(1)、および _Exit(2) のマニュアルページを参照してください。

exit または _Exit によって返される終了ステータスは、呼び出し側のプロセスの親プロセスが行なっていた処理によって異なります。

呼び出し側プロセスの親プロセスが wait(3C) か wait3(3C)、waitid(2)、waitpid(3C) のどれかを実行していて、その SA_NOCLDWAIT フラグの設定がなく、かつ SIGCHLD を SIG_IGN に設定していなかった場合、親プロセスは呼び出し側プロセスの終了の通知を受け、ステータスの下位 8 ビット (すなわち、ビット 0377) を使用できるようになります。親が待ち状態ではない場合は、そのあと、親が wait()、wait3()、waitid()、waitpid() のどれかを実行した時点で子のステータスを利用できるようになります。

- (7.20.4.6) 引数が null ポインタ以外の場合に system 関数によって返される値。

  waitpid(3C) によって指定された形式でシェルの終了ステータスが返されます。

- (7.23.1) 現地時間帯と夏時間。

  現地時間帯は、環境変数 TZ で設定します。

- (7.23) clock_t および time_t で表現可能な時間の範囲と精度。

  clock_t および time_t の精度は 100 万分の 1 秒です。範囲は、x86 および SPARC v8 で -2147483647-1 〜 4294967295 (100 万分の 1 秒単位) です。SPARC v9 では、-9223372036854775807LL-1 〜 18446744073709551615 です。

- (7.23.2.1) clock 関数の経過時間。

  clock 関数の経過時間は、プログラム実行開始時を原点とする時間経過として表現されます。

- (7.23.3.5, 7.24.5.1) 「C」ロケールにおける、strftime および wcsftime 関数に対する %Z 指示子の置換文字列。

  時間帯の名前か短縮名。また、時間帯の判定ができない場合は、置換文字なしです。

- (F.9) IEC 60559 準拠の実装で trigonometric、hyperbolic、e を底とする指数、e を底とする対数、エラー、ログガンマ関数が「不正確」の浮動小数点例外を立てるかどうか、また立てる場合はそのタイミング。

  一般に「不正確」の例外は、結果が正確に表現できない場合に立てられます。また「不正確」の例外は、結果が正確に表現可能な場合にも立てられることがあります。

- (F.9) IEC 60559 準拠の実装で <math.h> の関数が丸め方向モードに従うかどうか。

  <math.h> のどの関数についても、デフォルトの丸め方向モードの強制は試みられません。

C.1.13 アーキテクチャー (J.3.13)

- (5.2.4.2, 7.18.2, 7.18.3) ヘッダーの <float.h>、<limits.h>、および <stdint.h> に指定されているマクロに割り当てられている値または式。
  - <float.h> に指定されているマクロに対する値または式は次のとおりです。

```
#define CHAR_BIT 8 /* max # of bits in a “char” */
#define SCHAR_MIN (-128) /* min value of a “signed char” */
#define SCHAR_MAX 127 /* max value of a “signed char” */
#define CHAR_MIN SCHAR_MIN /* min value of a “char” */
#define CHAR_MAX SCHAR_MAX /* max value of a “char” */
#define MB_LEN_MAX 5
#define SHRT_MIN (-32768) /* min value of a “short int” */
#define SHRT_MAX 32767 /* max value of a “short int” */
#define USHRT_MAX 65535 /* max value of “unsigned short int” */
#define INT_MIN (-2147483647-1) /* min value of an “int” */
#define INT_MAX 2147483647 /* max value of an “int” */
#define UINT_MAX 4294967295U /* max value of an “unsigned int” */
#define LONG_MIN (-2147483647L-1L)
#define LONG_MAX 2147483647L /* max value of a “long int” */
#define ULONG_MAX 4294967295UL /* max value of “unsigned long int” */
#define LLONG_MIN (-9223372036854775807LL-1LL)
#define LLONG_MAX 9223372036854775807LL
#define ULLONG_MAX 18446744073709551615ULL

#define FLT_RADIX 2
#define FLT_MANT_DIG 24
#define DBL_MANT_DIG 53
#define LDBL_MANT_DIG 64

#if defined(__sparc)
#define DECIMAL_DIG 36
#elif defined(__i386)
#define DECIMAL_DIG 21
#endif
#define FLT_DIG 6
#define DBL_DIG 15
#if defined(__sparc)
#define LDBL_DIG 33
#elif defined(__i386)
#define LDBL_DIG 18
```

```
#endif

#define FLT_MIN_EXP (-125)
#define DBL_MIN_EXP (-1021)
#define LDBL_MIN_EXP (-16381)

#define FLT_MIN_10_EXP (-37)
#define DBL_MIN_10_EXP (-307)
#define LDBL_MIN_10_EXP (-4931)

#define FLT_MAX_EXP (+128)
#define DBL_MAX_EXP (+1024)
#define LDBL_MAX_EXP (+16384)

#define FLT_EPSILON 1.192092896E-07F
#define DBL_EPSILON 2.2204460492503131E-16

#if defined(__sparc)
#define LDBL_EPSILON 1.92592994438723585305597794258492731 9E-34L
#elif defined(__i386)
#define LDBL_EPSILON 1.08420217248550443400 75E-19L
#endif

#define FLT_MIN 1.175494351E-38F
#define DBL_MIN 2.2250738585072014E-308

#if defined(__sparc)
#define LDBL_MIN 3.3621031431120935062626778173217526 03E-4932L
#elif defined(__i386)
#define LDBL_MIN 3.362103143112093506262 7E-4932L
#endif
```

<limits.h> に指定されているマクロに対する値または式は次のとおりです。

```
#define INT8_MAX (127)
#define INT16_MAX (32767)
#define INT32_MAX (2147483647)
#define INT64_MAX (9223372036854775807LL)

#define INT8_MIN (-128)
#define INT16_MIN (-32767-1)
#define INT32_MIN (-2147483647-1)
#define INT64_MIN (-9223372036854775807LL-1)

#define UINT8_MAX (255U)
#define UINT16_MAX (65535U)
#define UINT32_MAX (4294967295U)
#define UINT64_MAX (18446744073709551615ULL)

#define INT_LEAST8_MIN INT8_MIN
#define INT_LEAST16_MIN INT16_MIN
#define INT_LEAST32_MIN INT32_MIN
#define INT_LEAST64_MIN INT64_MIN

#define INT_LEAST8_MAX INT8_MAX
#define INT_LEAST16_MAX INT16_MAX
#define INT_LEAST32_MAX INT32_MAX
#define INT_LEAST64_MAX INT64_MAX
```

```
#define UINT_LEAST8_MAX UINT8_MAX
#define UINT_LEAST16_MAX UINT16_MAX
#define UINT_LEAST32_MAX UINT32_MAX
#define UINT_LEAST64_MAX UINT64_MAX
```

- <stdint.h> に指定されているマクロに対する値または式は次のとおりです。

```
#define INT_FAST8_MIN INT8_MIN
#define INT_FAST16_MIN INT16_MIN
#define INT_FAST32_MIN INT32_MIN
#define INT_FAST64_MIN INT64_MIN

#define INT_FAST8_MAX INT8_MAX
#define INT_FAST16_MAX INT16_MAX
#define INT_FAST32_MAX INT32_MAX
#define INT_FAST64_MAX INT64_MAX

#define UINT_FAST8_MAX UINT8_MAX
#define UINT_FAST16_MAX UINT16_MAX
#define UINT_FAST32_MAX UINT32_MAX
#define UINT_FAST64_MAX UINT64_MAX
```

- (6.2.6.1) この国際規格に明示的に規定されていないオブジェクトのバイト数と順序、符号化方式。

1999 C 規格に明示的に規定されていないオブジェクトの処理系定義のバイト数と順序、符号化方式は、この章の別の場所で定義する必要があります。

- (6.5.3.4) sizeof 演算子の結果の値。

  次の表は、sizeof の結果を一覧表示します。

表 C–4 sizeof 演算子の結果 (バイト単位)

種類	バイト単位のサイズ
char と _Bool	1
short	2
int	4
long	4
long -m64	8
long long	8
float	4
double	8
long double (SPARC)	16
long double (x86) -m32	12
long double (x86) -m64	16

表 C–4 sizeof 演算子の結果 (バイト単位)　(続き)

種類	バイト単位のサイズ
pointer	4
pointer -m64	8
_Complex float	8
_Complex double	16
_Complex long double (SPARC)	32
_Complex long double (x86) -m32	24
_Complex long double (x86) -m64	32
_Imaginary float	4
_Imaginary double	8
_Imaginary long double (SPARC)	16
_Imaginary long double (x86) -m32	12
_Imaginary long double (x86) -m64	16

C.1.14 ロケール固有の動作 (J.4)

ホスト環境の次の特性はロケール依存で、処理系で文書化する必要があります。

- (5.2.1) 基本文字セット以外のソースおよび実行文字セットの追加メンバー。
  ロケール依存です。C ロケールでは、文字セットの拡張はありません。
- (5.2.1.2) 基本文字セット以外の実行文字セットの追加の複数バイト文字の有無と意味、表現。
  デフォルトまたは C ロケールでは、実行文字セットに存在する複数バイト文字はありません。
- (5.2.1.2) 複数バイト文字の符号化で使用するシフト状態。
  シフト状態はありません。
- (5.2.2) 連続する印刷文字の出力方向。
  常に左から右に印刷されます。
- (7.1.1) 小数点の文字。
  ロケール依存です (C ロケールでは、ピリオド「.」)。
- (7.4, 7.25.2) 印刷文字セット。
  ロケール依存です (C ロケールでは、ピリオド「.」)。
- (7.4, 7.25.2) 制御文字セット。

制御文字セットは、水平タブ、垂直タブ、フォームフィード、アラート、バックスペース、復帰、改行で構成されます。

- `isalpha`、 `isblank`、`islower`、`ispunct`、 `isspace`、`isupper`、`iswalpha`、`iswblank`、`iswlower`、`iswpunct`、 `iswspace`、`iswupper` 関数によるテスト対象の文字セット。

  `isalpha` () および `iswalpha`() と、前述の関係するマクロについては、`isalpha`(3C) および `iswalpha`(3C) のマニュアルページを参照してください。これらの動作はロケールを変更することによって変更されることがあります。

- (7.11.1.1) ネイティブ環境。

  `setlocale`(3C) マニュアルページで説明しているように、ネイティブ環境は `LANG` および `LC_*` 環境変数で指定します。ただし、これらの環境変数が設定されていない場合は、C ロケールに設定されます。

- (7.20.1, 7.24.4.1) 数値変換関数が受け付ける追加の変換対象シーケンス。

  基数文字はプログラムのロケールで定義され (`LC_NUMERIC` カテゴリ)、ピリオド (.) 以外のものに定義できます。

- 実行文字セットの照合シーケンス (7.21.4.3, 7.24.4.4.2)。

  ロケール依存です。C ロケールでは、照合順序は ASCII の照合シーケンスと同じです。

- (7.21.6.2) `strerror` 関数が作成するエラーメッセージ文字列の内容。

  アプリケーションが `-lintl` を付けてリンクされた場合、この関数が返すメッセージは `LC_MESSAGES` ロケールカテゴリで指定されたネイティブ言語になります。それ以外の場合は、C ロケールです。

- (7.23.3.5, 7.24.5.1) 時刻と日付の書式。

  ロケール依存です。C ロケールでの形式を次の表にまとめます。

  月の名前は次のとおりです。

表 C–5 月の名前

January	May	September
February	June	October
March	July	November
April	August	December

曜日の名前は次のとおりです。

表 C–6 曜日の名前と省略名

曜日名	省略名
Sunday Thursday	Sun Thu
Monday Friday	Mon Fri
Tuesday Saturday	Tue Sat
Wednesday	Wed

時間の書式は次のとおりです。

`%H:%M:%S`

日付の書式は次のとおりです。

`%m/%d/ -Xc` モード

午前/午後を指定する書式は、次のとおりです。AM PM

- (7.25.1) `towctrans` 関数がサポートする文字マッピング。

  プログラムのロケール (`LC_CTYPE` カテゴリ) 内の文字マッピング情報で定義される符号化文字セットの規則で、`tolower` および `toupper` 以外の文字マッピングを規定できます。使用可能なロケールとその定義の詳細は、『*Solaris Internationalization Guide For Developers*』を参照してください。
- (7.25.1) `iswctype` 関数がサポートする文字分類。

  使用可能なロケールと規格外の予約文字の分類についての詳細は、『*Solaris Internationalization Guide For Developers*』を参照してください。

◆ ◆ ◆ 付 録 D

C99でサポートされている機能

この付録では、『Programming Language - C (ISO/IEC 9899:1999)』規格でサポートされている機能について説明します。

`-xc99` フラグは、実装されている機能をコンパイラで認識させるかどうかを設定します。`-xc99` の構文の詳細については、265 ページの「B.2.79 -xc99[= *o*]」を参照してください。

注 – コンパイラは、デフォルトでは以降で説明している C99 の機能をサポートしていますが、Solaris が提供する標準のヘッダーファイル (`/usr/include` にある) は、1999 ISO/IEC C 規格にまだ準拠していません。エラーメッセージが生成される場合は、`-xc99=none` を指定して、ヘッダー用に 1990 ISO/IEC C 規格を使用してみてください。

D.1 説明と使用例

この付録では、サポートされている機能のうちの一部を詳細に説明し、使用例を示します。

- 5.2.4.2.2 項 浮動小数点型 `<float.h>` の特性
- 6.2.5 項 _Bool
- 6.2.5 項 _Complex 型
- 6.3.2.1 項 左辺値に限定されないポインタへの配列の変換
- 6.4.1 項 キーワード
- 6.4.2.2 項 定義済み識別子
- 6.4.3 項 汎用文字名
- 6.4.4.2 項 16 進数浮動小数点リテラル
- 6.4.9 項 コメント
- 6.5.2.2 項 関数呼び出し
- 6.5.2.5 項 複合リテラル
- 6.7.2 項 型指示子

- 6.7.2.1 項 構造体および共用体の指示子
- 6.7.3 項 型修飾子
- 6.7.4 項 関数指示子
- 6.7.5.2 項 配列宣言子
- 6.7.8 項 初期化
- 6.8.2 項 複合文
- 6.8.5 項 繰り返し文
- 6.10.3 項 マクロ置換
- 6.10.6 項 STDC プラグマ
- 6.10.8 項 __STDC_IEC_559 マクロおよび __STDC_IEC_559_COMPLEX マクロ
- 6.10.9 項 プラグマ演算子

D.1.1 浮動小数点評価における精度

5.2.4.2.2 項 浮動小数点型 <float.h> の特性

浮動小数点オペランドを持つ演算の値、および通常の算術変換および浮動小数点定数両方の影響を受ける値は、その型が必要とするより大きくすることが可能な範囲および精度を持つ形式で評価されます。使用する評価形式は、FLT_EVAL_METHOD の実装定義値によって決まります。

表 D–1 FLT_EVAL_METHOD の値

値	意味
-1	判定不能
0	コンパイラは、すべての演算および定数を正確にその型の範囲と精度で評価します。
1	コンパイラは、float および double 型の演算および定数を double 型の範囲および精度で評価します。long double 型の演算および定数は long double 型の範囲と精度で評価します。
2	コンパイラは、すべての演算および定数を long double 型の範囲と精度で評価します。

SPARC アーキテクチャーで float.h をインクルードすると、デフォルトでは、FLT_EVAL_METHOD は }0 に展開され、すべての浮動小数点式はその型に従って評価されます。

x86 アーキテクチャーで float.h をインクルードすると、デフォルトでは、FLT_EVAL_METHOD は -1 に展開され (-xarch=sse2 または -xarch=amd64 の場合を除く)、すべての浮動小数点定数式はその型として、ほかのすべての浮動小数点式は long double として評価されます。

-flteval=2 を指定して、float.h をインクルードすると、FLT_EVAL_METHOD は 2 に展開され、すべての浮動小数点式は long double として評価されます。詳細については、237 ページの「B.2.21 -flteval[={any|2}]」を参照してください。

-xarch=sse2 (または sse3、ssse3、sse4_1、sse4_2 など、SSE2 プロセッサファミリのそれ以降のバージョン) または -m64 を x86 で指定して、float.h をインクルードすると、FLT_EVAL_METHOD は 0 に展開され、すべての浮動小数点式はその型に従って評価されます。

浮動小数点式が double として評価されても、-Xt オプションがFLT_EVAL_METHOD の展開内容に影響することはありません。詳細については、254 ページの「B.2.68 -X[c|a|t|s]」を参照してください。

-fsingle オプションを指定すると、浮動小数点式が単精度で評価されます。詳細については、242 ページの「B.2.30 -fsingle」を参照してください。

-xarch=sse2 (または sse3、ssse3、sse4_1、sse4_2 など、SSE2 プロセッサファミリのそれ以降のバージョン) または -m64 を使用して x86 アーキテクチャーで -fprecision を指定して、float.h をインクルードすると、FLT_EVAL_METHOD が -1 に展開されます。

D.1.2 C99 のキーワード

6.4.1 キーワード

C99 の規格では、次のキーワードが追加されました。-xc99=none を指定した場合に、これらのキーワードを識別子として使用すると、コンパイラは警告を生成します。-xc99=none を指定していない場合は、これらのキーワードが識別子として使用されているときに、コンパイラがコンテキストに応じて警告またはエラーメッセージを生成します。

- inline
- _Imaginary
- _Complex
- _Bool
- restrict

D.1.2.1 restrict キーワードの使用

restrict で修飾されたポインタを使用してオブジェクトにアクセスするには、そのオブジェクトへのすべてのアクセスで、直接的または間接的にそのポインタの値を使用する必要があります。ほかの方法によってそのオブジェクトにアクセスすると、定義されていない動作が発生する可能性があります。restrict 修飾子は、コンパイラで最適化を行うための想定を可能にするために使用します。

restrict 修飾子を効果的に使用する例および方法については、88 ページの「3.8.2 制限付きポインタ」を参照してください。

D.1.3 __func__ のサポート

6.4.2.2 定義済み識別子

コンパイラで、定義済み識別子 __func__ がサポートされました。__func__ は、__func__ のあるコードでの現在の関数の名前を格納する文字配列として定義されています。

D.1.4 汎用文字名 (UCN)

6.4.3 項 汎用文字名

UCN では、C のソースで英字ばかりでなく、任意の文字を使用することができます。UCN の形式は次のとおりです。

- \u 4 桁の 16 進値
- \U 8 桁の 16 進値

UCN では、0024 ($)、0040 (@)、0060 (?) を除く 00A0 未満の値や、D800 ～ DFFF の範囲内の値を指定してはいけません。

識別子や文字定数、文字列リテラルで UCN を使用して、C の基本文字セットにはない文字を表すことができます。

UCN の \unnnnnnnn は、8 桁の短い識別子が nnnnnnnn の文字を表します (ISO/IEC 10646 で規定)。同様に、汎用文字名の \unnnn は、4 桁の短い識別子が nnnn (8 桁の短い識別子は 0000nnnn) の文字を表します。

D.1.5 // を使用したコードのコメント処理

6.4.9 コメント

// から復帰改行までのすべての複数バイト文字 (復帰改行そのものは除く) は、コメントとして処理されます。ただし、文字定数、文字列リテラル、コメント内で // が使用されている場合は、コメントとして処理されません。

D.1.6 暗黙の int および暗黙の関数宣言の禁止

6.5.2.2 関数呼び出し

暗黙の宣言は、1990 C 規格の場合とは異なり、1999 C 規格では許可されなくなりました。以前のバージョンの C コンパイラでは、-v (冗長形式) を指定した場合にだけ、暗黙の定義についての警告メッセージが生成されていました。暗黙の定義についてのこれらのメッセージおよび新しい警告は、識別子が暗黙に int または関数として宣言されている場合は常に生成されます。

多数の警告メッセージが生成されることがあるため、この変更はたいていの場合はすぐにわかります。よくある原因としては、たとえば、printf では <stdio.h> をインクルードする必要があるように、使用する関数を宣言する適切なシステムヘッダーファイルをインクルードしていないことが考えられます。1990 C 規格に従って暗黙の宣言を許可するには、-xc99=none を指定します。

C コンパイラは、暗黙の関数宣言に対して警告を生成するようになりました。

```
example% cat test.c
void main()
{
  printf("Hello, world!\n");
}
example% cc test.c
"test.c", line 3: warning: implicit function declaration: printf
example%
```

D.1.7 暗黙の int を使用した宣言

6.7.2 型指示子

少なくとも 1 つの型指示子を、各宣言の宣言指示子で指定します。358 ページの「D.1.6 暗黙の int および暗黙の関数宣言の禁止」も参照してください。

暗黙の int 宣言を使用した場合は、C コンパイラは次の例のように警告を生成します。

```
example% more test.c
volatile i;
const foo()
{
  return i;
}
example% cc test.c
  "test.c", line 1: warning: no explicit type given
  "test.c", line 3: warning: no explicit type given
example%
```

D.1.8 柔軟な配列のメンバー

6.7.2.1 構造体および共用体の指示子

「struct hack」とも呼びます。構造体の最後のメンバーを、int foo[]; などのゼロ長の配列にすることができます。このような構造体は、malloc で割り当てられたメモリーにアクセスするためのヘッダーとして一般的に使用されます。

たとえば、struct s { int n; double d[]; } S; では、配列 d が不完全な配列型です。C コンパイラは、この S のメンバーのメモリーオフセットをカウントしません。つまり、sizeof(struct s) は S.n のオフセットと同一になります。

d は、通常の配列メンバーと同様に、S.d[10] = 0; のように使用することができます。

C コンパイラが不完全な配列型をサポートしていない場合は、次の例の DynamicDouble のような構造体を定義および宣言します。

```
typedef struct { int n; double d[1]; ) DynamicDouble;
```

ここで、配列 d は不完全な配列型ではなく、1 つのメンバーを指定して宣言されています。

次に、ポインタ dd を宣言してメモリーを割り当てます。

```
DynamicDouble *dd = malloc(sizeof(DynamicDouble)+(actual_size-1)*sizeof(double));
```

そのあとで、次のように S.n にオフセットのサイズを格納します。

```
dd->n = actual_size;
```

コンパイラが不完全な配列型をサポートしているため、1 つのメンバーを指定して配列を宣言することなく、同一の結果を得ることができます。

```
typedef struct { int n; double d[]; } DynamicDouble;
```

ここで、ポインタ dd を宣言し、前の場合と同様にメモリーを割り当てます。ただし、ここでは actual_size から 1 を引く必要はありません。

```
DynamicDouble *dd = malloc (sizeof(DynamicDouble) + (actual_size)*sizeof(double));
```

前の場合と同様に、オフセットは次のように S.n に保存されます。

```
dd->n = actual_size;
```

D.1.9 べき等修飾子

6.7.3 型修飾子

同一の指示子と修飾子のリストで、直接または typedef により同一の修飾子が複数ある場合は、動作は型修飾子が 1 つだけの場合と同様になります。

C90 では、次のコードではエラーが発生します。

```
%example cat test.c

const const int a;

int main(void) {
  return(0);
```

```
}

%example cc -xc99=none test.c
"test.c", line 1: invalid type combination
```

ただし、C99 では、C コンパイラで複数の修飾子を使用できます。

```
%example cc -xc99 test.c
%example
```

D.1.10 `inline` 関数

6.7.4 関数指示子

1999 C ISO 標準で定義されているインライン関数は、完全にサポートされています。

C 標準によると、インラインは C コンパイラに対する推奨に過ぎません。C コンパイラは、何もインライン化しないこと、または実際の関数への呼び出しをコンパイルすることを選択できます。

最適化レベルを **-x03** かそれ以上に設定してコンパイルする場合以外は、Solaris Studio C コンパイラは C の関数呼び出しをインライン化しません。なおかつ、オプティマイザのヒューリスティック (発見的手法) によってインライン化に利点があると判断された場合のみ、インライン化します。C コンパイラは、関数のインライン化を強制する方法を用意していません。

static インライン関数は単純です。インライン関数指定子によって定義された関数を、参照時にインライン化するか、実際の関数を呼び出すかのどちらかです。コンパイラは、参照ごとにどちらを実行するかを選択できます。コンパイラは **-x03** 以上の場合、インライン化に利点があるかどうかを判定します。インライン化の利点がない場合 (または最適化が **-x03** 未満の場合)、実際の関数への参照がコンパイルされ、関数定義がコンパイルされてオブジェクトコードの一部になります。プログラムが関数のアドレスを使用している場合は、実際の関数がコンパイルされてオブジェクトコードの一部になり、インライン化されないことに注意してください。

extern インライン関数はより複雑です。extern インライン関数には 2 つの種類があります。インライン定義および *extern* インライン関数です。

インライン定義とは、キーワード `inline` を使い、`static` または `extern` キーワードなしで定義された関数です。ソース (またはインクルードファイル) 内に含まれるあらゆるプロトタイプにも、`static` または `extern` キーワードなしでキーワード `inline` が含まれます。インライン定義に対して、コンパイラは関数のグローバル定義を作成してはいけません。このことは、インライン化されないインライン定義へのあらゆる参照は、どこか他の場所で定義されたグローバル関数への参照になることを意味します。言い換えると、この変換ユニット (ソースファイル) をコンパイルすることで作成されたオブジェクトファイルには、インライン定義に対応するグローバルシ

ンボルは含まれません。インライン化されない関数へのあらゆる参照は、リンク時にほかのオブジェクトファイルまたはライブラリから提供される extern (大域) シンボルになります。

extern インライン関数は、extern 記憶クラス指定子 (つまり、関数定義とプロトタイプの一方または両方) を持つファイルスコープ宣言によって宣言されます。extern インライン関数の場合、コンパイラはその関数に対応するグローバル定義を、結果として生成されるオブジェクトファイルの中で提供します。コンパイラは、関数定義の提供元である変換ユニット (ソースファイル) の中で観察された関数へのすべての参照をインライン化するか、グローバル関数を呼び出すかを選択できます。

関数呼び出しが実際にインライン化されるかどうかに依存する任意のプログラムの動作は、未定義です。

外部リンケージを持つインライン関数が、変換ユニットのどこであっても静的変数を宣言または参照してはいけないことにも注意してください。

D.1.10.1 インライン関数に対する Solaris Studio C コンパイラと gcc の互換性

ほとんどのプログラムで、extern インライン関数に関する gcc の実装に対して Solaris Studio C コンパイラが互換性を維持できるようにするには、**`-features=no%extinl`** フラグを使用します。このフラグが指定されている場合は、Solaris Studio C コンパイラはあたかも static インライン関数と宣言されたかのように、関数を処理します。

この手法が互換性を維持できない状況の 1 つは、関数のアドレスを取る場合です。gcc ではこのアドレスはグローバル関数のアドレスになりますが、Sun のコンパイラではローカルの static 定義アドレスが使用されます。

D.1.11 配列宣言子で使用可能な `Static` およびそのほかの型修飾子

6.7.5.2 配列宣言子

関数宣言子内のパラメータの配列宣言子で `static` キーワードを使用することが可能になりました。この場合は、コンパイラが、宣言する関数に多数の要素が引き渡されることを少なくとも認識することができます。これにより、オプティマイザで従来は不可能だった想定が可能になります。

C コンパイラは、配列パラメータをポインタに変換します。したがって、`void foo(int a[])` は `void foo(int *a)` と同義になります。

`void foo(int * restrict a);` などの型修飾子を指定すると、C コンパイラはそれを配列文 `void foo(int a[restrict]);` で表現します。これは、実質的には制限付きポインタを宣言するのと同義です。

C コンパイラは、配列サイズに関する情報を保持するためにも static 修飾子を使用します。たとえば、void foo(int a[10]) を指定した場合でも、コンパイラはこれを void foo(int *a) と表現します。ポインタが NULL ではなく、少なくとも 10 個の要素を持つ整数配列へのポインタであることをコンパイラに認識させるには、void foo(int a[static 10]) のように static 修飾子を使用します。

D.1.12 可変長配列 (VLA)

6.7.5.2 配列宣言子

VLA は、alloca 関数を呼び出した場合と同様に、スタックに割り当てられます。VLA の有効期間は、有効範囲に関係なく、alloca の呼び出しによりスタックに割り当てられたデータと同様に、関数から復帰するまでです。割り当てられた領域は、VLA が割り当てられた関数から復帰してスタックが解放されるときに同時に解放されます。

可変長配列では、一部の制約がまだ有効になっていません。制約に違反すると、定義されていない結果になります。

```
#include <stdio.h>
void foo(int);

int main(void) {
  foo(4);
  return(0);
}

void foo (int n) {
  int i;
  int a[n];
  for (i = 0; i < n; i++)
    a[i] = n-i;
  for (i = n-1; i >= 0; i--)
    printf("a[%d] = %d\n", i, a[i]);
}

example% cc test.c
example% a.out
a[3] = 1
a[2] = 2
a[1] = 3
a[0] = 4
```

D.1.13 指示付きの初期化子

6.7.8 初期化

指示付き初期化子は、数値プログラミングで一般的なスパース配列を初期化する仕組みです。

指示付き初期化子によって、システムプログラミングで一般的なスパース構造体を初期化したり、先頭メンバーであるかどうかに関係なく、任意のメンバーを使って共用体を初期化したりできます。

例を挙げて考えてみます。最初の例は、指示付き初期化子を使って配列を初期化する方法を示しています。

```
enum { first, second, third };
         const char *nm[] = {
         [third] = "third member",
         [first] = "first member",
         [second] = "second member",
         };
```

次の例は、指示付き初期化子を使用して struct オブジェクトのフィールドを初期化する方法を示しています。

```
division_t result = { .quot = 2, .rem = -1 };
```

次の例は、指示付き初期化子を使用して、これ以外の方法では誤解を生むおそれがある複雑な構造体を初期化する方法を示しています。

```
struct { int z[3], count; } w[] = { [0].z = {1}, [1].z[0] = 2 };
```

1 つの指示子で両端から配列を作成することができます。

```
int z[MAX] = {1, 3, 5, 7, 9, [MAX-5] = 8, 6, 4, 2, 0};
```

MAX が 10 より大きい場合、この配列の中央には値ゼロの要素が含まれます。MAX が 10 より小さい場合、最初の 5 つの初期化子が提供する値の一部は、2 つ目の 5 つの値によって置き換えられます。

共用体のすべてのメンバーを初期化することができます。

```
union { int i; float f;} data = { .f = 3.2 };
```

D.1.14 型宣言とコードの混在

6.8.2 複合文

C コンパイラで、次の例のように型宣言と実行可能コードを混在させることが可能になりました。

```
#include <stdio.h>

int main(void){
  int num1 = 3;
  printf("%d\n", num1);
```

```
  int num2 = 10;
  printf("%d\n", num2);
  return(0);
}
```

D.1.15 for ループ文での宣言

6.8.5 繰り返し文

C コンパイラで、for ループ文の最初の式として、型宣言を使用できるようになりました。

```
for (int i=0; i<10; i++){ //loop body };
```

for ループの初期化文で宣言した変数の有効範囲は、ループ全体になります (制御式と繰り返し式を含む)。

D.1.16 可変数の引数をとるマクロ

6.10.3 マクロ置換

C コンパイラで、次の形式の #define プリプロセッサ指令を使用することができます。

```
#define identifier (...) replacement_list
#define identifier (identifier_list, ...) replacement_list
```

マクロ定義で *identifier_list* が省略符号で終わる場合は、マクロ定義でのパラメータよりも呼び出しの引数の方が多いことを示します (省略符号を除く)。それ以外の場合は、マクロ定義でのパラメータ数 (プリプロセッサトークンのない引数を含む) が引数の個数と一致します。引数に省略符号表記を使用する #define のプリプロセッサ指令の置換リストで、識別子 __VA_ARGS__ を使用します。次のコードは、マクロの可変引数リスト機能の例です。

```
#define debug(...) fprintf(stderr, __VA_ARGS__)
#define showlist(...) puts(#__VA_ARGS__)
#define report(test, ...) ((test)?puts(#test):\
                          printf(__VA_ARGS__))
debug(“Flag”);
debug(“X = %d\n”,x);
showlist(The first, second, and third items.);
report(x>y, “x is %d but y is %d”, x, y);
```

この結果は、次のようになります。

```
fprintf(stderr, “Flag”);
fprintf(stderr, “X = %d\n”, x);
puts(“The first, second, and third items.”);
((x>y)?puts(“x>y”):printf(“x is %d but y is %d”, x, y));
```

D.1.17 _Pragma

6.10.9 プラグマ演算子

_Pragma (*string-literal*) という形式の単項演算子の式は、次のように処理されます。

- 文字列定数の L 接頭辞がある場合は削除されます。
- 前および後ろの二重引用符は削除されます。
- エスケープシーケンス ' は、二重引用符に置換されます。
- エスケープシーケンス \\ は、1 つの \ に置換されます。

生成されたプリプロセッサトークンのシーケンスは、プラグマの指令でのプリプロセッサトークンと同様に処理されます。

単項演算子の式にある元の 4 つのプリプロセッサトークンは削除されます。

_Pragma は、#pragma と比較して、マクロ定義で使用可能であるという利点があります。

_Pragma("string") は、#pragma 文字列とまったく同一になります。次の例を考えてみましょう。最初の例は、プリプロセッサの使用前のソースコードです。次の例は、プリプロセッサの使用後のソースコードです。

```
example% cat test.c

#include <omp.h>
#include <stdio.h>

#define Pragma(x) _Pragma(#x)
#define OMP(directive) Pragma(omp directive)

void main()
{
  omp_set_dynamic(0);
  omp_set_num_threads(2);
  OMP(parallel)
  {
  printf("Hello!\n");
  }
}

example% cc test.c -P -xopenmp -x03
example% cat test.i
```

次は、プリプロセッサ終了後のソースコードです。

```
void main()
{
  omp_set_dynamic(0);
  omp_set_num_threads(2);
  # pragma omp parallel
  {
    printf("Hellow!\n");
```

```
  }
}

example% cc test.c -xopenmp -->
example% ./a.out
Hello!
Hello!
example%
```

◆ ◆ ◆ 付録 E

ISO/IEC C90 の処理系定義の動作

『Programming Language - C (ISO/IEC 9899:1999)』規格には、C 言語で記述されたプログラムの構文と解釈が規定されています。この付録では、それらの動作を詳しく説明します。各項は ISO/IEC 9899:1990 規格そのものと簡単に比較できるようになっています。日本の対応規格は、JIS X 3010 - 1993 です。

- ISO 規格と同様の文を用いて各動作を説明しています。
- 各動作の説明の前に ISO 規格で対応するセクション番号を付けています。

E.1 ISO 規格との実装の比較

E.1.1 翻訳 (G.3.1)

括弧内の数は、ISO/IEC 9899/1990 規格のセクション番号に対応しています。

E.1.1.1 (5.1.1.3) 診断の認識

エラーメッセージは次の書式です。

filename, `line` *line number: message*

警告メッセージは次の書式です。

filename, `line` *line number: warning message*

ここで

- ファイル名とはエラーまたは警告があったファイルの名前です
- 行番号とはエラーまたは警告が検出された行の番号です
- メッセージとは診断メッセージです

E.1.2 環境 (G.3.2)

E.1.2.1 (5.1.2.2.1) `main` の引数の意味

```
int main (int argc, char *argv[])
{
....
}
```

`argc` はプログラムの呼び出しに伴うコマンド行引数の数です。シェルによって展開されたあとは、`argc` は必ず 1 以上、つまりプログラム名が 1 つ以上になります。

`argv` はコマンド行引数へのポインタ配列です。

E.1.2.2 (5.1.2.3) 対話型デバイスを構成するもの

対話型デバイスにはシステムライブラリコールの `isatty()` が 0 以外の値を返します。

E.1.3 識別子 (G.3.3)

E.1.3.1 (6.1.2) 外部リンケージのない識別子の先頭から (31 を超える) 有意文字の数

最初の 1,023 文字が有意です。識別子は大文字と小文字を別の文字として扱います。

(6.1.2) 外部リンケージのある識別子の先頭から (6 を超える) 有意文字の数

最初の 1,023 文字が有意です。識別子は大文字と小文字を別の文字として扱います。

E.1.4 文字 (G.3.4)

E.1.4.1 (5.2.1) ソースと実行の文字セットについて (規格に明確に規定されているものを除く)

どちらの文字セットも ASCII 文字セットやロケール固有の拡張文字と同一です。

E.1.4.2 (5.2.1.2) 複数バイト文字を符号化するためのシフト状態について

シフト状態はありません。

E.1.4.3 (5.2.4.2.1) 実行文字セットで 1 文字のビット数

ASCII 部分では、1 文字に 8 ビットです。ロケール固有の拡張文字部分では、ロケール固有の 8 ビットの倍数です。

E.1.4.4 (6.1.3.4) ソース文字セット (文字と文字列リテラル) メンバーの実行文字セットメンバーへの配置

ASCII 部分では、配置はソース文字と実行文字と同様です。

E.1.4.5 (6.1.3.4) 基本の実行文字セット、またはワイド文字定数用の拡張文字セットのどちらにも表現されていない文字や、エスケープシーケンスを含む整数文字定数の値

右端の文字が示す数値です。たとえば、'\q' は 'q' に等しくなります。このようなエスケープシーケンスが発生すると警告が発行されます。

E.1.4.6 (6.1.3.4) 2 つ以上の文字を含む整数文字定数の値、または 2 つ以上の複数バイト文字を含むワイド文字定数の値

エスケープシーケンスの発生しない複数バイト文字セットの値は、各文字の示す数値から派生しています。

E.1.4.7 (6.1.3.4) 複数バイト文字を対応するワイド文字 (コード) に変換するのに使用される現ロケール (locale)

有効なロケールは LC_ALL、LC_CTYPE、または LANG 環境変数のいずれかで指定されたものです。

E.1.4.8 (6.2.1.1) 何も付いていない char は、signed char と、unsigned char のどちらと同じ範囲の値を持つか

char は、signed char とみなされます。

E.1.5 整数 (G.3.5)

E.1.5.1 (6.1.2.5) 整数の型の表現と値について

表 E–1 整数の表現と値

整数	ビット数	最小値	最大値
`char`	8	-128	127
`signed char`	8	-128	127
`unsigned char`	8	0	255
`short`	16	-32768	32767
`signed short`	16	-32768	32767
`unsigned short`	16	0	65535
`int`	32	-2147483648	2147483647
`signed int`	32	-2147483648	2147483647
`unsigned int`	32	0	4294967295
`long -m32`	32	-2147483648	2147483647
`long -m64`	64	-9223372036854775808	9223372036854775807
`signed long -m32`	32	-2147483648	2147483647
`signed long -m64`	64	-9223372036854775808	9223372036854775807
`unsigned long -m32`	32	0	4294967295
`unsigned long -m64`	64	0	18446744073709551615
`long long`	64	-9223372036854775808	9223372036854775807
`signed long long`[1]	64	-9223372036854775808	9223372036854775807
`unsigned long long`[1]	64	0	18446744073709551615

[1] `-Xc` モードでは無効です。

E.1.5.2 (6.2.1.2) 値を表現できない場合に整数をより短い符号付き整数に変換した結果、また符号なしの整数を同じ長さの符号付き整数に変換した結果

整数がより短い `signed` 整数に変換される場合は、長い方の整数の下位ビットが短い方の `signed` 整数に複写されます。結果は負になることがあります。

符号なし整数が同サイズの signed 整数に変換される場合は、unsigned 整数の下位ビットが signed 整数に複写されます。結果は負になることがあります。

E.1.5.3 (6.3) 符号付き整数におけるビット単位演算の結果

ビット単位演算を signed の型に適用すると、sign ビットを含むオペランドのビット単位演算となります。その結果の各ビットは、両オペランドの対応するビットが設定されていた場合にのみ設定されます。

E.1.5.4 (6.3.5) 整数の除算における剰余の符号について

結果は被除数と同じ符号になります。たとえば、-23/4 の剰余は -3 となります。

E.1.5.5 (6.3.7) 負の値を持つ符号付き整数型を右シフトした結果

右シフトの結果は signed の右シフトとなります。

E.1.6 浮動小数点 (G.3.6)

E.1.6.1 (6.1.2.5) 浮動小数点数の型の表現と値

表 E–2 float の値

float	
ビット数	32
最小値	1.17549435E-38
最大値	3.40282347E+38
イプシロン	1.19209290E-07

表 E–3 double の値

double	
ビット数	64
最小値	2.2250738585072014E-308
最大値	1.7976931348623157E+308
イプシロン	2.2204460492503131E-16

表 E–4 long double の値

long double	
ビット数	128 *(SPARC)* 80 *(x86)*
最小値	3.362103143112093506262677817321752603E-4932 (SPARC) 3.36210314311209350626 27E-4932 *(x86)*
最大値	1.189731495357231765085759326628007016E+4932 *(SPARC)* 1.18973149535723176502 13E4932 *(x86)*
イプシロン	1.925929944387235853055977942584927319E-34 *(SPARC)* 1.08420217248550443400 75E-19 *(x86)*

E.1.6.2 (6.2.1.3) 整数値が元の値を完全には表現できない浮動小数点数に変換された場合の切り捨ての指示

数値は元の値の近似値に丸められます。

E.1.6.3 (6.2.1.4) 浮動小数点数が短い浮動小数点数に変換された場合の切り捨てまたは丸めの指示

数値は元の値の近似値に丸められます。

E.1.7 配列とポインタ (G.3.7)

E.1.7.1 (6.3.3.4, 7.1.1) 配列の最大サイズを維持するのに必要な整数型。すなわち、sizeof 演算子の size_t の型

stddef.h において定義されている unsigned int です (–m32 の場合)。

unsigned long (-m64 の場合)

E.1.7.2 (6.3.4) ポインタを整数に cast で型変換した結果、またはその逆の結果

ポインタおよび int、long、unsigned int、unsigned long 型の値ではビットパターンは変わりません。

E.1.7.3 (6.3.6, 7.1.1) 同じ配列のメンバーへの 2 つのポインタの相違 ptrdiff_t を維持するのに必要な整数型

stddef.h において定義されている int です (–m32 の場合)。

long (-m64 の場合)

E.1.8 レジスタ (G.3.8)

E.1.8.1 (6.5.1) register 記憶クラス指定子を使用して、オブジェクトを実際に入れることのできるレジスタの数

有効なレジスタ宣言の数は使用パターンおよび各関数における定義に依存し、割り当て可能なレジスタ数に制限されます。コンパイラやオプティマイザは、レジスタ宣言に従う必要はありません。

E.1.9 構造体、共用体、列挙型、およびビットフィールド (G.3.9)

E.1.9.1 (6.3.2.3) 共用体のオブジェクトのメンバーはほかの型のメンバーを使用してアクセスされる

共用体のメンバーに記憶されているビットパターンがアクセスされ、アクセスしたメンバーの型に従って値が解釈されます。

E.1.9.2 (6.5.2.1) 構造体のメンバーのパディングと整列条件

表 E–5 構造体メンバーのパディングと整列

種類	整合の境界	バイト境界
char と _Bool	バイト	1
short	ハーフワード	2
int	ワード	4
long -m32	ワード	4
long -m64	ダブルワード	8
long long -m32	ダブルワード (SPARC) ワード (x86)	8 (SPARC) 4 (x86)

表 E–5 構造体メンバーのパディングと整列 (続き)

種類	整合の境界	バイト境界
`long long -m64`	ダブルワード	8
`float`	ワード	4
`double -m32`	ダブルワード (SPARC) ワード (x86)	8 (SPARC) 4 (x86)
`double -m64`	ダブルワード	8
`long double -m32`	ダブルワード (SPARC) ワード (x86)	8 (SPARC) 4 (x86)
`long double -m64`	クワドワード	16
`pointer -m32`	ワード	4
`pointer -m64`	クワドワード	8
`float _Complex`	ワード	4
`double _Complex -m32`	ダブルワード (SPARC) ワード (x86)	8 (SPARC) 4 (x86)
`double _Complex -m64`	ダブルワード	8
`long double _Complex -m32`	ダブルワード (SPARC) ワード (x86)	8 (SPARC) 4 (x86)
`long double _Complex -m64`	クワドワード	16
`float _Imaginary`	ワード	4
`double _Imaginary -m32`	ダブルワード (SPARC) ワード (x86)	8 (SPARC) 4 (x86)
`double _Imaginary -m64`	ダブルワード	8
`long double _Imaginary -m32`	ダブルワード (SPARC) ワード (x86)	8 (SPARC) 4 (x86)
`long double _Imaginary -m64`	ダブルワード	16

各要素が適切な境界上に並ぶように、構造体のメンバーが自動的に埋め込まれます。

構造体自身の整列条件はそのメンバーの整列条件と同一です。たとえば、`chars` 型だけの `struct` は整列の制限がありませんが、`–m64` を使用してコンパイルされた `double` 型を含む `struct` は 8 バイトの境界上に並びます。

E.1.9.3 (6.5.2.1) 単なる int のビットフィールドは signed int ビットフィールドとみなされるか、unsigned int ビットフィールドとみなされるか

unsigned int とみなされます。

E.1.9.4 (6.5.2.1) int 内のビットフィールドの割り当て順序

ビットフィールドは、記憶装置内で高位から低位の順に割り当てられます。

E.1.9.5 (6.5.2.1) ビットフィールドは記憶装置の境界を越えることができるか

ビットフィールドは記憶装置の境界を越えません。

E.1.9.6 (6.5.2.2) 列挙型の値を表現するための整数型

int 型です。

E.1.10 修飾子 (G.3.10)

E.1.10.1 (6.5.5.3) volatile 修飾子型を持つオブジェクトへのアクセス方法

オブジェクト名を参照するたびに、そのオブジェクトへアクセスされます。

E.1.11 宣言子 (G.3.11)

E.1.11.1 (6.5.4) 算術演算、構造体、または共用体の型が修正可能な宣言子の最大数

コンパイラによる制限はありません。

E.1.12 文 (G.3.12)

E.1.12.1 (6.6.4.2) switch 文中の case 値の最大個数

コンパイラによる制限はありません。

E.1.13 プリプロセッサ指令 (G.3.13)

E.1.13.1 (6.8.1) 条件付きのインクルードを制御する定数式のシングルキャラクタ文字定数の値は、実行文字セット中の同一の文字定数の値に一致するか

前処理命令内の文字定数はほかの式のものと同一の数値を持ちます。

E.1.13.2 (6.8.1) そのような文字定数は負の値をとり得るか

この場合の文字定数は負の値を取ることがあります。

E.1.13.3 (6.8.2) インクルード可能なソースファイルの位置を知る方法

最初に、ファイル名が < > によって区切られたファイルを、`-I` オプションによって指定されたディレクトリの中で検索します。次に、標準ディレクトリの中を検索します。異なるデフォルト位置を指定するのに `-YI` オプションが使用されていないかぎり、標準ディレクトリは `/usr/include` です。

最初に、ファイル名が引用符によって区切られたファイルを、`#include` 文のあるソースファイルのディレクトリ内で検索します。次に、`-I` オプションによって指定されたディレクトリの中を検索し、最後に標準ディレクトリ内を検索します。

< > や二重引用符で囲まれたファイル名が / で始まっている場合は、そのファイル名はルートディレクトリで始まるパス名であると解釈されます。このファイルの検索はルートディレクトリの中でのみ行われます。

E.1.13.4 (6.8.2) インクルード可能なソースファイルの引用符付きの名前のサポート

`include` 命令の引用符付きのファイル名はサポートされます。

E.1.13.5 (6.8.2) ソースファイルの文字シーケンスの配置

ソースファイルの文字は対応する ASCII の値に配置されます。

E.1.13.6 (6.8.6) 認識された `#pragma` 命令の動作

次に示すプラグマがサポートされています。詳細は、45 ページの「2.11 プラグマ」を参照してください。

- `align` *integer (variable[, variable])*
- `c99 (“implicit” | “no%implicit”)`
- `does_not_read_global_data` *(funcname [, funcname])*
- `does_not_return` *(funcname[, funcname])*

- `does_not_write_global_data` *(funcname[, funcname])*
- `error_messages` *(on|off|default, tag1[tag2... tagn])*
- `fini` *(f1[, f2..., fn])*
- `hdrstop`
- `ident` *string*
- `init` *(f1[, f2..., fn])*
- `inline` *(funcname[, funcname])*
- `int_to_unsigned` *(funcname)*
- MP `serial_loop`
- MP `serial_loop_nested`
- MP `taskloop`
- `no_inline` *(funcname[, funcname])*
- `no_warn_missing_parameter_info`
- `nomemorydepend`
- `no_side_effect` *(funcname[, funcname])*
- `opt_level` *(funcname[, funcname])*
- `pack`*(n)*
- `pipeloop`*(n)*
- `rarely_called` *(funcname[, funcname])*
- `redefine_extname` *old_extname new_extname*
- `returns_new_memory` *(funcname[, funcname])*
- `unknown_control_flow` *(name[, name])*
- `unroll` *(unroll_factor)*
- `warn_missing_parameter_info`
- `weak` *symbol1 [= symbol2]*

E.1.13.7 (6.8.8) 翻訳の日付と時間がわからないときの `__DATE__` と `__TIME__` の定義

これらのマクロは常に使用できます。

E.1.14 ライブラリ関数 (G.3.14)

E.1.14.1 (7.1.6) マクロの `NULL` を拡張した null ポインタ定数

`NULL` は 0 になります。

E.1.14.2 (7.2) `assert` 関数によって出力される診断と assert 関数の終了動作

診断は次のようになります。

`Assertion failed:` *statement*`. file` *filename*`, line` *number*

ここで

- *statement* は表明に失敗した文です
- *filename* は障害を持ったファイルの名前です
- *line number* は障害が発生した行の番号です

E.1.14.3 (7.3.1) `isalnum`、`isalpha`、`iscntrl`、`islower`、`isprint`、および `isupper` 関数によってテストされる文字セット

表 E–6 `isalpha`、`islower` などによりテストされる文字セット

`isalnum`	ASCII 文字の A から Z、a から z、0 から 9
`isalpha`	ASCII 文字の A から Z、a から z、およびロケール固有の単一バイト文字
`iscntrl`	0 から 31 までと 127 の値を持つ ASCII 文字
`islower`	ASCII 文字の a から z
`isprint`	ロケール固有の単一バイトの出力可能文字
`isupper`	ASCII 文字の A から Z

E.1.14.4 (7.5.1) ドメインエラーの数値演算関数によって返される値

表 E–7 ドメインエラーの場合の戻り値

エラー	数値演算関数	コンパイラモード	
		`-Xs, -Xt`	`-Xa, -Xc`
DOMAIN	acos(|x|>1)	0.0	0.0
DOMAIN	asin(|x|>1)	0.0	0.0
DOMAIN	atan2(+-0,+-0)	0.0	0.0
DOMAIN	y0(0)	-HUGE	-HUGE_VAL
DOMAIN	y0(x<0)	-HUGE	-HUGE_VAL
DOMAIN	y1(0)	-HUGE	-HUGE_VAL
DOMAIN	y1(x<0)	-HUGE	-HUGE_VAL
DOMAIN	yn(n,0)	-HUGE	-HUGE_VAL
DOMAIN	yn(n,x<0)	-HUGE	-HUGE_VAL
DOMAIN	log(x<0)	-HUGE	-HUGE_VAL
DOMAIN	log10(x<0)	-HUGE	-HUGE_VAL
DOMAIN	pow(0,0)	0.0	1.0

表 E–7 ドメインエラーの場合の戻り値 (続き)

エラー	数値演算関数	コンパイラモード	
		-Xs, -Xt	-Xa, -Xc
DOMAIN	pow(0,neg)	0.0	-HUGE_VAL
DOMAIN	pow(neg,non-integal)	0.0	NaN
DOMAIN	sqrt(x<0)	0.0	NaN
DOMAIN	fmod(x,0)	x	NaN
DOMAIN	remainder(x,0)	NaN	NaN
DOMAIN	acosh(x<1)	NaN	NaN
DOMAIN	atanh(|x|>1)	NaN	NaN

E.1.14.5 (7.5.1) アンダーフローエラーの場合に、数値演算関数が整数式 errno をマクロ ERANGE の値に設定するかどうか

アンダーフローが検出された場合、scalbn を除いた数値演算関数は errno を ERANGE に設定します。

E.1.14.6 (7.5.6.4) fmod 関数の第 2 引数が 0 を持つ場合に、ドメインエラーとなるか、0 が返されるか

この場合は、ドメインエラーとして第 1 引数が返されます。

E.1.14.7 (7.7.1.1) signal 関数に対するシグナルの設定

次の表に signal 関数が認識する各シグナルの意味を示します。

表 E–8 signal シグナルの意味

シグナル	いいえ。	デフォルト	イベント
SIGHUP	1	終了	ハングアップ
SIGINT	2	終了	interrupt
SIGQUIT	3	コア	quit
SIGILL	4	コア	不当な命令 (捕捉されてもリセットされない)
SIGTRAP	5	コア	トレーストラップ (捕捉されてもリセットされない)
SIGIOT	6	コア	IOT 命令
SIGABRT	6	コア	異常終了時に使用

表 E–8 signal シグナルの意味 *(続き)*

シグナル	いいえ。	デフォルト	イベント
SIGEMT	7	コア	EMT 命令
SIGFPE	8	コア	浮動小数点の例外
SIGKILL	9	終了	強制終了 (捕捉または無視できない)
SIGBUS	10	コア	バスエラー
SIGSEGV	11	コア	セグメンテーション違反
SIGSYS	12	コア	システムコールへの引数誤り
SIGPIPE	13	終了	読み手のないパイプ上への書き込み
SIGALRM	14	終了	アラームクロック
SIGTERM	15	終了	プロセスの終了によるソフトウェアの停止
SIGUSR1	16	終了	ユーザー定義のシグナル 1
SIGUSR2	17	終了	ユーザー定義のシグナル 2
SIGCLD	18	無視	子プロセス状態の変化
SIGCHLD	18	無視	子プロセス状態の変化の別名
SIGPWR	19	無視	電源障害による再起動
SIGWINCH	20	無視	ウィンドウサイズの変更
SIGURG	21	無視	ソケットの緊急状態
SIGPOLL	22	終了	ポーリング可能なイベント発生
SIGIO	22	終了	ソケット入出力可能
SIGSTOP	23	停止	停止 (キャッチまたは無視できない)
SIGTSTP	24	停止	tty より要求されたユーザーストップ
SIGCONT	25	無視	停止していたプロセスの継続
SIGTTIN	26	停止	バックグラウンド tty の読み込みを試みた
SIGTTOU	27	停止	バックグラウンド tty の書き込みを試みた
SIGVTALRM	28	終了	仮想タイマーの時間切れ
SIGPROF	29	終了	プロファイリングタイマーの時間切れ
SIGXCPU	30	コア	CPU の限界をオーバー
SIGXFSZ	31	コア	ファイルサイズの限界をオーバー

表 E–8 signal シグナルの意味 (続き)

シグナル	いいえ。	デフォルト	イベント
SIGWAITINGT	32	無視	プロセスの LWP がブロックされた

E.1.14.8 (7.7.1.1) signal 関数によって認識される各 signal のデフォルトの取扱い、およびプログラムのスタートアップ時における取扱い

上記を参照してください。

E.1.14.9 (7.7.1.1) シグナルハンドラを呼び出す前に signal(sig, SIG_DFL); 相当のものが実行されない場合は、どのシグナルがブロックされるか

signal(sig,SIG_DFL) 相当のものは、常に実行されます。

E.1.14.10 (7.7.1.1) SIGILL 関数に指定されたハンドラにより SIGILL シグナルが受信された場合は、デフォルト処理はリセットされるか

SIGILL ではデフォルト処理はリセットされません。

E.1.14.11 (7.9.2) テキストストリームの最終行で、改行文字による終了を必要とするか

最終行を改行文字で終了する必要はありません。

E.1.14.12 (7.9.2) 改行文字の直前でテキストストリームに書き出されたスペース文字は読み込みの際に表示されるか

ストリームが読み込まれるときにはすべての文字が表示されます。

E.1.14.13 (7.9.2) バイナリストリームに書かれたデータに追加することのできる null 文字の数

バイナリストリームには null 文字を追加しません。

E.1.14.14 (7.9.3) アペンドモードのストリームのファイル位置指示子は、最初にファイルの始まりと終わりのどちらに置かれるか

ファイル位置指示子は最初にファイルの終わりに置かれます。

E.1.14.15 (7.9.3) テキストストリームへの書き込みを行うと、書き込み点以降の関連ファイルが切り捨てられるか

ハードウェアの命令がないかぎり、テキストストリームへの書き込みによって書き込み点以降の関連ファイルが切り捨てられることはありません。

E.1.14.16 (7.9.3) ファイルのバッファリングの特徴

標準エラーストリーム (stderr) を除く出力ストリームは、デフォルトでは、出力がファイルの場合にはバッファリングされ、出力が端末の場合にはラインバッファリングされます。標準エラー出力ストリーム (stderr) は、デフォルトではバッファリングされません。

バッファリングされた出力ストリームは多くの文字を保存し、その文字をブロックとして書き込みます。バッファリングされなかった出力ストリームは宛先ファイルあるいは端末に迅速に書き込めるように情報の待ち行列を作ります。行バッファリングされた出力は、その行が完了するまで (改行文字が要求されるまで) 行単位の出力待ち行列に入れられます。

E.1.14.17 (7.9.3) ゼロ長ファイルは実際に存在するか

ディレクトリエントリを持つという意味ではゼロ長ファイルは存在します。

E.1.14.18 (7.9.3) 有効なファイル名を作成するための規則

有効なファイル名は 1 から 1,023 文字までの長さで、null 文字とスラッシュ (/) 以外のすべての文字を使用することができます。

E.1.14.19 (7.9.3) 同一のファイルを何回も開くことができるか

同一のファイルを何回も開くことができます。

E.1.14.20 (7.9.4.1) 開いたファイルへの remove 関数の効果

ファイルを閉じる最後の呼び出しによりファイルが削除されます。すでに除去されたファイルをプログラムが開くことはできません。

E.1.14.21 (7.9.4.2) rename 関数を呼び出す前に新しい名前を持つファイルがあった場合、そのファイルはどうなるか

そのようなファイルがあれば削除され、新しいファイルが元のファイルの上に書き込まれます。

E.1.14.22 (7.9.6.1) fprintf 関数における %p 変換の出力

%p の出力は %x と等しくなります。

E.1.14.23 (7.9.6.2) fscanf 関数における %p 変換の入力

%p の入力は %x と等しくなります。

E.1.14.24 (7.9.6.2) fscanf 関数における %[変換のための走査リストで最初の文字でも最後の文字でもないハイフン文字 - の解釈

- 文字は包含的範囲を意味します。すなわち、[0-9] は [0123456789] に等しくなります。

E.1.15 ロケール固有の動作 (G.4)

E.1.15.1 (7.12.1) 現地時間帯と夏時間の設定

現地時間帯は環境変数 TZ で設定します。

E.1.15.2 (7.12.2.1) clock 関数の経過時間

clock 関数の経過時間は、プログラム実行開始時を原点とする時間経過として表現されます。

ホスト環境については次のようなロケール固有の性質があります。

E.1.15.3 (5.2.1) 必要なメンバー以外の実行文字セットの内容

ロケール依存です。C ロケールでは、文字セットの拡張はありません。

E.1.15.4 (5.2.2) 印刷方向

常に左から右に印刷されます。

E.1.15.5 (7.1.1) 10 進小数点を表わす文字

ロケール依存です (C ロケールでは、ピリオド「.」)。

E.1.15.6 (7.3) 処理系ごとに定義される文字テストおよびケース配置関数の項目

「4.3.1」と同義です。

E.1.15.7 (7.11.4.4) 実行文字セットの照合シーケンス

ロケール依存です。C ロケールでは、照合順序は ASCII の照合シーケンスと同じです。

E.1.15.8 (7.12.3.5) 時間と日付の書式

ロケール依存です。C ロケールでの形式を次の表にまとめます。月の名前は次のとおりです。

表 E–9 月の名前

January	May	September
February	June	October
March	July	November
April	August	December

曜日の名前は次のとおりです。

表 E–10 曜日の名前と省略名

曜日名		省略名	
Sunday	Thursday	Sun	Thu
Monday	Friday	Mon	Fri
Tuesday	Saturday	Tue	Sat
Wednesday		Wed	

時間の書式は次のとおりです。

```
%H:%M:%S
```

日付の書式は次のとおりです。

```
%m/%d/%y
```

午前/午後を指定する書式は、次のとおりです。AM PM

◆ ◆ ◆ 付録 F

ISO C データ表現

この付録では、ISO C の記憶装置におけるデータ表現と、関数に引数を渡す仕組みについて説明します。この付録では、C 言語以外の言語でモジュールを記述したり使用したい場合に、これらのモジュールに C 言語コードとのインタフェースを持たせるための手引きとして書かれたものです。

F.1 記憶装置の割り当て

データ型とその表現方法について次の表にまとめます。

注 – スタックへの記憶装置の割り当て (内部リンクつまり自動リンクを伴う識別子を使用) は、2G バイト以下に制限すべきです。

表 F–1 データ型の記憶装置の割り当て (バイト単位のサイズ)

C の型	LP64 (-m64) サイズ	LP64 整列	ILP32 (-m32) サイズ	ILP 32 整列
Integral				
_Bool char signed char unsigned char	1	1	1	1
short signed short unsigned short	2	2	2	2

表 F–1 データ型の記憶装置の割り当て(バイト単位のサイズ) (続き)

C の型	LP64 (-m64) サイズ	LP64 整列	ILP32 (-m32) サイズ	ILP 32 整列
int signed int unsigned int enum	4	4	4	4
long signed long unsigned long	8	8	4	4
long long signed long long unsigned long long	8	8	8	4 (x86) / 8 (SPARC)
Pointer				
任意の型 * 任意の型 (*) ()	8	8	4	4
Floating Point				
float	4	4	4	4
double	8	8	8	4 (x86) / 8 (SPARC)
long double	16	16	12 (x86) / 16 (SPARC)	4 (x86) / 8 (SPARC)
Complex				
float _Complex	8	4	8	4
double _Complex	16	8	16	4 (x86) / 8 (SPARC)
long double _Complex	32	16	24 (x86) / 32 (SPARC)	4 (x86) / 16 (SPARC)
Imaginary				
float _Imaginary	4	4	4	4
double _Imaginary	8	8	8	4 (x86) / 8 (SPARC)
long double _Imaginary	16	16	12 (x86) / 16 (SPARC)	4 (x86) / 16 (SPARC)

F.2 データ表現

使用しているアーキテクチャーによってデータ要素のビット番号の割り当てが異なります。SPARCstation ではビット 0 を最下位有効ビット、バイト 0 を最上位有効バイトとしてそれぞれ使用します。次の表に表現方法を示します。

F.2.1 整数表現

ISO C で使用されている整数型は `short`、`int`、`long`、および `long long` です。

表 F–2 `short` の表現

ビット数	内容
8- 15	バイト 0 (SPARC) バイト 1 *(x86)*
0- 7	バイト 1 (SPARC) バイト 0 (x86)

表 F–3 `int` の表現

ビット数	内容
24- 31	バイト 0 (SPARC) バイト 3 (x86)
16- 23	バイト 1 (SPARC) バイト 2 (x86)
8- 15	バイト 2 (SPARC) バイト 1 (x86)
0- 7	バイト 3 (SPARC) バイト 0 (x86)

表 F–4 `long` の表現と `-m32` でのコンパイル

ビット数	内容
24- 31	バイト 0 (SPARC) バイト 3 (x86)

表 F–4 long の表現と -m32 でのコンパイル　(続き)

ビット数	内容
16- 23	バイト 1 (SPARC) バイト 2 (x86)
8- 15	バイト 2 (SPARC) バイト 1 (x86)
0- 7	バイト 3 (SPARC) バイト 0 (x86)

表 F–5 long (-m64) および long long (-m32 と -m64 の両方) の表現

ビット数	内容
56- 63	バイト 0 (SPARC) バイト 7 (x86)
48- 55	バイト 1 (SPARC) バイト 6 (x86)
40- 47	バイト 2 (SPARC) バイト 5 (x86)
32- 39	バイト 3 (SPARC) バイト 4 (x86)
24- 31	バイト 4 (SPARC) バイト 3 (x86)
16- 23	バイト 5 (SPARC) バイト 2 (x86)
8- 15	バイト 6 (SPARC) バイト 1 (x86)
0- 7	バイト 7 (SPARC) バイト 0 (x86)

F.2.2 浮動小数点表現

float、double、long double のデータ要素は、ISO IEEE 754-1985 規格に従って次の式のように表現されます。

$(-1)^{s} * 2^{(e - bias)} * [j.f]$

ここで

- *s* = 符号
- *e* = バイアス付きの指数
- *j* = 先行ビット。*e* の値によって決まる。`long double` *(x86)* では、先行ビットは明示的。そのほかは暗黙的。
- *f* = 仮数部 (23 ビット)
- *u* は、ビットが 0 でも 1 でも良いことを意味します (次の表で使用)。

IEEE Single および Double の場合、*j* は常に暗黙的です。バイアス付きの指数が 0 の場合、*f* が 0 でない限り、*j* は 0 で、結果として生成される数値は非正規数です。バイアス付きの指数が 0 より大きい場合、数値が有限である限り *j* は 1 です。

Intel 80 ビット拡張の場合、*j* は常に明示的です。

各ビットの位置は次の表のとおりです。

表 F–6 `float` の表現

ビット数	名
31	符号
23- 30	バイアス付きの指数
0- 22	仮数部

表 F–7 `double` の表現

ビット数	名
63	符号
52- 62	バイアス付きの指数
0- 51	仮数部

表 F–8 `long double` の表現 (SPARC)

ビット数	名
127	符号
112- 126	バイアス付きの指数
0- 111	仮数部

表 F–9 long double の表現 (x86)

ビット数	名
80- 95	使用されない
79	符号
64- 78	バイアス付きの指数
63	先行ビット
0- 62	仮数部

詳細については、『数値計算ガイド』を参照してください。

F.2.3 極値表現

正規化された float と double の数は「隠された」ビットまたは暗黙のビットを持つと言われます。それにより、精度を 1 ビット分高めることができます。long double の場合は、先行ビットは暗黙的 (SPARC) または明示的 (x86) のいずれかになります。このビットは正規数に対しては 1、非正規数に対しては 0 になります。

表 F–10 float の表現

正規数 (0<e<255):	$(-1)^s 2^{(e-127)} 1.f$
非正規数 (e=0, f!=0):	$(-1)^s 2^{(-126)} 0.f$
ゼロ (e=0, f=0):	$(-1)^s 0.0$
シグナルを発生する NaN	s=u, e=255(最大値); f=.0uuu〜uu (少なくとも 1 ビットは 0 以外)
シグナルを発生しない NaN	s=u, e=255(最大値); f=.1uuu〜uu
無限大	s=u, e=255(最大値); f=.0000〜00 (すべてが 0)

表 F–11 double の表現

正規数 (0<e<2047):	$(-1)^s 2^{(e-1023)} 1.f$
非正規数 (e=0, f!=0):	$(-1)^s 2^{(-1022)} 0.f$
ゼロ (e=0, f=0):	$(-1)^s 0.0$
シグナルを発生する NaN	s=u, e=2047(最大値); f=.0uuu〜uu (少なくとも 1 ビットは 0 以外)
シグナルを発生しない NaN	s=u, e=2047(最大値); f=.1uuu〜uu

表 F–11 double の表現 *(続き)*

無限大	s=u, e=2047(最大値); f=.0000〜00 (すべてが 0)

表 F–12 long double の表現

正規数 (0<e<32767):	$(-1)^s 2^{(e-16383)} 1.f$
非正規数 (e=0, f!=0):	$(-1)^s 2^{(-16382)} 0.f$
ゼロ (e=0, f=0):	$(-1)^s 0.0$
シグナルを発生する NaN	s=u, e=32767(符号); f=.0uuu〜uu (少なくとも 1 ビットは 0 以外)
シグナルを発生しない NaN	s=u, e=32767(最大値); f=.1uuu〜uu
無限大	s=u, e=32767(最大値); f=.0000〜00 (すべてが 0)

F.2.4 重要な数の 16 進数表現

よく使用される数値の 16 進数表現を次の表にまとめます。

表 F–13 重要な数の 16 進数表現 (SPARC)

値	float	double	long double
+0	00000000	0000000000000000	00000000000000000000000000000000
-0	80000000	8000000000000000	80000000000000000000000000000000
+1.0	3F800000	3FF0000000000000	3FFF0000000000000000000000000000
-1.0	BF800000	BFF0000000000000	BFFF0000000000000000000000000000
+2.0	40000000	4000000000000000	40000000000000000000000000000000
+3.0	40400000	4008000000000000	40080000000000000000000000000000
プラス無限大	7F800000	7FF0000000000000	7FFF0000000000000000000000000000
マイナス無限	FF800000	FFF0000000000000	FFFF0000000000000000000000000000
NaN	7FBFFFFF	7FF7FFFFFFFFFFFF	7FFF7FFFFFFFFFFFFFFFFFFFFFFFFFFF

表 F–14 重要な数の 16 進数表現 *(x86)*

値	float	double	long double
+0	00000000	0000000000000000	00000000000000000000
-0	80000000	0000000080000000	80000000000000000000

表 F–14　重要な数の 16 進数表現 *(x86)*　　*(続き)*

値	float	double	long double
+1.0	3F800000	000000003FF00000	3FFF8000000000000000
-1.0	BF800000	00000000BFF00000	BFFF8000000000000000
+2.0	40000000	0000000040000000	40008000000000000000
+3.0	40400000	0000000040080000	4000C000000000000000
プラス無限大	7F800000	000000007FF00000	7FFF8000000000000000
マイナス無限	FF800000	00000000FFF00000	FFFF8000000000000000
NaN	7FBFFFFF	FFFFFFFF7FF7FFFF	7FFFBFFFFFFFFFFFFFFF

詳細については、『数値計算ガイド』を参照してください。

F.2.5　ポインタ表現

C 言語におけるポインタは 4 バイトを使用します。C でのポインタは、SPARC v9 アーキテクチャーでは 8 バイトを占有します。NULL 値のポインタはゼロと等価です。

F.2.6　配列の格納

配列は、それぞれの要素が決められた記憶順序で格納されます。各要素は実際には記憶要素の一次元の列に格納されます。

C 言語の配列は行の並びを優先して格納されます。この順序では、多次元配列における右端の添字がもっとも速く変化します。

文字列データ型は char 要素の配列になります。連結後、文字列リテラルまたはワイド文字列リテラルに指定できる最大の文字数は、4,294,967,295 個です。

スタックに割り当てられた記憶領域のサイズ制限については、387 ページの「F.1 記憶装置の割り当て」を参照してください。

表 F–15　配列の型と最大の大きさ

種類	-m32 の要素の最大数	-m64 の要素の最大数
char	4,294,967,295	2,305,843,009,213,693,951
short	2,147,483,647	1,152,921,504,606,846,975
int	1,073,741,823	576,460,752,303,423,487

表 F–15 配列の型と最大の大きさ *(続き)*

種類	-m32 の要素の最大数	-m64 の要素の最大数
long	1,073,741,823	288,230,376,151,711,743
float	1,073,741,823	576,460,752,303,423,487
double	536,870,911	288,230,376,151,711,743
long double	268,435,451	144,115,188,075,855,871
long long	536,870,911	288,230,376,151,711,743

静的および大域配列にはさらに多くの要素を格納することができます。

F.2.7 極値の算術演算

この節では、浮動小数点の極値と通常値を組み合わせたものに基本算術演算を適用して得られる結果について説明します。トラップやその他の例外は起こらないものとします。

次の表で、略語の意味を説明します。

表 F–16 略語の使用法

略語	意味
Num	非正規のまたは正規化された数字
Inf	無限大 (正または負)
NaN	数字ではない
Uno	順序不定

次の表は、異なるタイプのオペランドを組み合わせて行なった算術演算から得られた値のタイプを示しています。

表 F–17 加算と減算の結果

	右側のオペランド: 0	右側のオペランド: Num	右側のオペランド: Inf	右側のオペランド: NaN
左側のオペランド: 0	0	Num	Inf	NaN
左側のオペランド: Num	Num	を参照してください。[1]	Inf	NaN

[1] Num + Num は、結果が大きすぎる (オーバーフロー) 場合は Num ではなく Inf になることがあります。無限量が逆の sign の場合は、Inf + Inf = NaN になります。

表 F–17 加算と減算の結果 *(続き)*

	右側のオペランド: 0	右側のオペランド: Num	右側のオペランド: Inf	右側のオペランド: NaN
左側のオペランド: Inf	Inf	Inf	[1]を参照してください。	NaN
左側のオペランド: NaN	NaN	NaN	NaN	NaN

[1] Num + Num は、結果が大きすぎる (オーバーフロー) 場合は Num ではなく Inf になることがあります。無限量が逆の `sign` の場合は、Inf + Inf = NaN になります。

表 F–18 乗算結果

	右側のオペランド: 0	右側のオペランド: Num	右側のオペランド: Inf	右側のオペランド: NaN
左側のオペランド: 0	0	0	NaN	NaN
左側のオペランド: Num	0	Num	Inf	NaN
左側のオペランド: Inf	NaN	Inf	Inf	NaN
左側のオペランド: NaN	NaN	NaN	NaN	NaN

表 F–19 除算結果

	右側のオペランド: 0	右側のオペランド: Num	右側のオペランド: Inf	右側のオペランド: NaN
左側のオペランド: 0	NaN	0	0	NaN
左側のオペランド: Num	Inf	Num	0	NaN
左側のオペランド: Inf	Inf	Inf	NaN	NaN
左側のオペランド: NaN	NaN	NaN	NaN	NaN

表 F–20 比較結果

	右側のオペランド: 0	右側のオペランド: +Num	右側のオペランド: +Inf	右側のオペランド: +NaN
左側のオペランド: 0	=	<	<	Uno

表 F–20 比較結果 (続き)

	右側のオペランド: 0	右側のオペランド: +Num	右側のオペランド: +Inf	右側のオペランド: +NaN
左側のオペランド: +Num	>	比較の結果	<	Uno
左側のオペランド: +Inf	>	>	=	Uno
左側のオペランド: +NaN	Uno	Uno	Uno	Uno

注 – NaN と比較した NaN は順序不定で、結果は不等価になります。+0 は -0 と比較結果が等しくなります。

F.3 引数を渡す仕組み

本節では ISO C における引数の渡し方について説明します。

- C の関数への引数は、すべて値渡しされます。
- 実引数は関数の宣言において宣言されるのと逆の順序で渡されます。
- 実引数が式の場合、関数参照の前に評価されます。その後、式の結果がレジスタに置かれるかスタックにプッシュされます。

F.3.1 32 ビット SPARC

関数は `integer` 型の結果をレジスタ `%o0` に返します。`float` 型の結果はレジスタ `%f0` に、`double` 型の結果はレジスタ `%f0` と `%f1` に返します。

`long long` 型 整数は上位ワードは `%oN` 下位ワードは `%o(N+1)` というようにレジスタに渡されます。 レジスタ内の結果は同様の順序で `%o0` と `%o1` に返されます。

`double` および long double 型を除くすべての引数は 4 バイトの値として渡されます。`double` 型は 8 バイトの値として渡されます。先頭 6 個の 4 バイト値 (`double` を 8 と数える) は `%o0` から `%o5` までのレジスタに渡され、残りはスタック経由で渡されます。構造体の場合は、構造体のコピーが作成され、ポインタがそのコピーに渡されます。`long double` は構造体と同様に渡されます。

前述のレジスタは、呼び出し側から見えます。

F.3.2 64 ビット SPARC

すべての整数の引数は、8 バイト値として引き渡されます。

浮動小数点引数は可能なかぎり、浮動小数点レジスタに渡されます。

F.3.3 x86/x64

Intel 386 psABI および AMD64 psABI を遵守しています。

関数は次のレジスタで結果を返します。

表 F–21 型を返すために x86 関数が使用するレジスタ

レジスタ	返される型
int	%eax
long long	%edx と %eax
float、double、long double	%st(0)
float _Complex	%eax (実数部) と %edx (虚数部)
double _Complex と long double _Complex	対応する浮動小数点型の 2 つの要素を含む構造体と同じ

詳細は、http://www.x86-64.org/documentation/abi.pdf で AMD64 psABI についての説明を参照してください。

struct、union、long long、double、long double を除くすべての引数は 4 バイト値として渡されます。long long は 8 バイト値として、double は 8 バイト値として、また long double は 12 バイト値としてそれぞれ渡されます。

struct と union はスタックにコピーされます。サイズは 4 の倍数バイトに丸められます。struct と union を返す関数は、その struct や union を格納する場所を指す隠された最初の引数に渡されます。

関数から戻ったあと、スタックから引数をポップするのは呼び出し側の責任です。呼び出された関数によってポップされる struct や union の余分な引数を除きます。

◆ ◆ ◆ 付録 G

パフォーマンスチューニング

この付録では、C プログラムでのパフォーマンスチューニングについて説明します。『Solaris Studio パフォーマンスアナライザ』マニュアルも参照してください。

G.1 libfast.a Library (SPARC)

libfast.a は、C 標準ライブラリのうち、32 ビットの SPARC 固有バージョンであり、シングルスレッドの単体で実行可能なアプリケーション用に最適化されたメモリー割り当てを実現します。これはオプションであるため、標準 C ライブラリでは使用できないようなアルゴリズムやデータ表現を使用することができ、ほとんどのアプリケーションのパフォーマンスを改善することができます。

次のチェックリストを参考にして、自分のアプリケーションのパフォーマンスが libfast.a によって向上するかどうかを判断してください。その際、プロファイリングを使用します。

- メモリー割り当てのパフォーマンスが重要で、通常もっとも多く割り当てられるメモリーのサイズが 2 のべき乗に近い場合は libfast.a を使用してください。重要なルーチン: malloc()、free()、realloc()
- ブロック移動またはブロック塗りつぶしのパフォーマンスが重要な場合は libfast.a を使用してください。重要なルーチン: bcopy()、bzero()、memcpy()、memmove()、memset()
- アプリケーションがマルチスレッド対応である場合は、libfast.a を使用しないでください。

アプリケーションをリンクする際には、cc コマンドの後ろに -lfast オプションを加えてください。cc コマンドは標準の C ライブラリよりも先に libfast.a にあるルーチンをリンクします。

◆ ◆ ◆ 付 録 H

K&R Solaris Studio C と Solaris Studio ISO C の違い

この付録では、従来の K&R Solaris Studio C と Solaris Studio ISO C の違いを説明します。

詳細は、30 ページの「1.5 準拠規格」を参照してください。

H.1 Solaris Studio ISO C との K&R Solaris Studio C の互換性

表 H–1 Solaris Studio ISO C との K&R Solaris Studio C の互換性

項目	Solaris Studio C (K&R)	Solaris Studio ISO C
`main()` の `envp` 引数	`main()` の 3 番目の引数として `envp` を使用できる。	3 番目の引数として使用できるが、この使用法は厳密には ISO C 規格に準拠しない。
キーワード	識別子、`const`、`volatile`、`signed` を普通の識別子として扱う。	`const`、`volatile`、`signed` はキーワードである。
ブロック内の `extern` と `static` 関数宣言	これらの関数宣言をファイルスコープに拡張する。	ISO 規格は、ブロックスコープ関数宣言がファイルスコープに拡張されることを保証しない。
識別子	識別子でドル記号 (`$`) を使用できる。	`$` は使用できない。
`long float` 型	`long float` 宣言を受け入れ、`double` として処理する。	このような宣言は使用できない。

複数バイト文字定数	`int mc = 'abcd';`
	は、次を生成する。
	`abcd`
		`int mc = 'abcd';`
		は、次を生成する。
		`dcba`

整数定数	8 進数のエスケープシーケンスで、8 または 9 を使用できる。	8 進数のエスケープシーケンスで、8 または 9 を使用できない。

表 H–1 Solaris Studio ISO C との K&R Solaris Studio C の互換性 (続き)

項目	Solaris Studio C (K&R)	Solaris Studio ISO C
代入演算子	次の演算子の組み合わせを 2 つのトークンとして処理するため、演算子の間に空白を使用できる。 `*=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=`	1 つのトークンとして処理するため、演算子の間に空白を使用できない。
式の符号なし保存の意味解釈	符号なし保存をサポートする。つまり、`unsigned char/shorts` は `unsigned int` に変換される。	値の保持をサポートする。つまり、`unsigned char/short` は `int` に変換される。
単精度計算と倍精度計算	浮動小数点式のオペランドを `double` に拡張する。 `float` を返すように宣言された関数の戻り値は、常に `double` に拡張される。	`float` の演算を単精度計算で行うことができる。 このような関数に `float` の戻り型を使用できる。
`struct/union` のメンバーの名前空間	`struct` と `union` を使用できるほか、メンバー選択演算子 ('.' , '->') を使用する演算の型に基づき、他の `struct` または `union` のメンバーを操作できる。	すべての一意な `struct` または `union` は、独自の一意な名前空間を持たなければならない。
左辺値 (`lvalue`) としてのキャスト	値 (`lvalue`) としての整数型およびポインタ型のキャストをサポートしている。次に例を示します。 `(char *)ip = &char;`	この機能はサポートしない。

暗黙の `int` 宣言	明示的な型指示子なしの宣言をサポートする。`num;` などの宣言は、暗黙の `int` として処理される。次に例を示します。
	`num; /*num は暗黙の int*/`
	`int num2; /* num2 は*/`
	`/* 明示的に宣言された int */`
		`num;` 宣言 (明示的な型指示子 `int` なし) はサポートされず、構文エラーとなる。

空の宣言	空の宣言を使用できる。 `int;`	タグを除いて、空の宣言を使用できない。

型定義の型指示子	`typedef` 宣言で `unsigned`、`short`、`long` などの型指示子を使用できる。次に例を示します。
	`typedef short small;`
	`unsigned small x;`
		`typedef` 宣言は型指示子で変更できない。

表 H–1 Solaris Studio ISO C との K&R Solaris Studio C の互換性 *(続き)*

項目	Solaris Studio C (K&R)	Solaris Studio ISO C
ビットフィールドで使用できる型	すべての整数型のビットフィールドを使用できる (名前なしビットフィールドも含む)。 ABI は、名前なしビットフィールドとほかの整数型のサポートを必要とする。	型 `int`、`unsigned int`、および `signed int` だけのビットフィールドをサポートする。ほかの型は未定義。

不完全な宣言におけるタグの処理	不完全な型宣言を無視する。次の例では、`f1` は外側の `struct` を参照する。
	`struct x { . . . } s1;`
	`{struct x; struct y {struct x f1; } s2; struct x`
	`{ . . . };}`
		ISO 準拠の実装では、不完全な `struct` または `union` 型指示子は、同じタグで囲んだ宣言を隠す。

`struct`、union、または enum 宣言での不一致	入れ子にされた `struct` または `union` 宣言において、タグの struct、enum、union 型の不一致を許可する。次の例では、2 番目の宣言は `struct` として処理される。
	`struct x {. . . }s1;`
	`{union x s2;. . .}`
		外側のタグを隠し、内側の宣言を新しい宣言として処理する。

式内のラベル	ラベルを `(void *) lvalue` として処理する。	式内ではラベルを使用できない。
`switch` 条件型	`int` に変換することで、`float` と `double` を使用できる。	整数型 (`int`、`char`、列挙型) だけを switch 条件型として評価する。
条件付きインクルード指令の構文	プリプロセッサは `#else` または `#endif` 指令のあとにあるトークンを無視する。	このような構文は使用できない。
トークンの結合と `##` プリプロセッサ演算子	`##` 演算子を認識しない。トークンの結合を行うには、結合される 2 つのトークンの間にコメントを置く。 `#define PASTE(A,B) A/*任意のコメント*/B`	`##` をトークンの結合を実行するプリプロセッサ演算子として定義する。次に例を示します。 `#define PASTE(A,B) A##B` さらに、Solaris Studio ISO C プリプロセッサは Solaris Studio C メソッドを認識しません。その代わりに、2 つのトークン間のコメントを空白として処理する。

表 H–1 Solaris Studio ISO C との K&R Solaris Studio C の互換性 (続き)

項目	Solaris Studio C (K&R)	Solaris Studio ISO C

プリプロセッサの再走査	プリプロセッサは再帰的に置換する。
	`#define F(X) X(arg)`
	`F(F)`
	は、次を生成する。
	`arg(arg)`
		再走査中に置換リストに見つかったマクロは置換されない。
		`#define F(X)X(arg)F(F)`
		は、次を生成する。
		`F(arg)`

仮パラメータリスト内の `typedef` 名	関数宣言中、`typedef` 名を仮パラメータ名として使用できる。つまり、`typedef` 宣言を隠す。	`typedef` 名として宣言された識別子を仮パラメータとして使用できない。

実装固有の集合体の初期化	中括弧内で部分的に省略された初期設定子を構文解析および処理するときは、ボトムアップアルゴリズムを使用する。
	`struct{ int a[3]; int b; }\ w[ ]={{1},2};`
	は、次を生成する。
	`sizeof(w)=16`
	`w[0].a=1,0,0`
	`w[0].b=2`
		構文解析には、トップダウンアルゴリズムを使用する。次に例を示します。
		`struct{int a[3];int b;}\`
		`w[]={{1},2};`
		は、次を生成する。
		`sizeof(w)=32w[0].a=1,0,0w[0].=0w[1].a=2,0,0w[1].b=0`

`include` ファイルをまたがるコメント	`#include` ファイルで始まり、最初のファイルをインクルードしたファイルで終了するコメントを使用できる。	コンパイルの翻訳段階で、つまり、`#include` 指令が処理される前に、コメントは空白文字に置換される。

文字定数内の仮引数の置換	置換リストマクロと一致したとき、文字定数内の文字を置換する。
	`#define charize(c)'c'`
	`charize(Z)`
	は、次を生成する。
	'Z'
		文字は置換されない。
		`#define charize(c) 'c'charize(Z)`
		は、次を生成する。
		'c'

表 H–1 Solaris Studio ISO C との K&R Solaris Studio C の互換性 *(続き)*

項目	Solaris Studio C (K&R)	Solaris Studio ISO C

文字列定数内の仮引数の置換	プリプロセッサは文字列定数内の囲まれた仮引数を置換する。
	`#define stringize(str) 'str'`
	`stringize(foo)`
	は、次を生成する。
	`"foo"`
		プリプロセッサ演算子 # を使用しなければならない。
		`#define stringize(str) 'str'`
		`stringize(foo)`
		は、次を生成する。
		`"str"`

コンパイラの「フロントエンド」に組み込まれたプリプロセッサ	コンパイラは、cpp(1) を呼び出し、指定したオプションに従って、コンパイルシステムのほかのすべてのコンポーネントを処理する。	ISO C の変換フェーズ 1 ～ 4 (プリプロセッサ指令の処理を含む) は acomp に直接組み込まれる。したがって、-Xs モードの場合を除き、cpp はコンパイル中に直接呼び出されることはない。
バックスラッシュによる行の連結	行の連結では、バックスラッシュ文字を認識しない。	改行文字の直前にバックスラッシュ文字を指定しなければならない。
文字列リテラル内の 3 文字表記	この ISO C の機能はサポートしない。	
`asm` キーワード	`asm` はキーワードである。	`asm` は通常の識別子として処理される。

識別子のリンケージ	初期化されていない `static` 宣言を仮定義として処理しない。この結果、2 番目の宣言が「再宣言」エラーを生成する。次に例を示します。
	`static int i = 1;`
	`static int i;`
		初期化されていない `static` 宣言を仮定義として処理する。

名前空間	`struct`、`union`、`enum` のタグ、struct、union、enum のメンバー、および、そのほかすべての 3 つだけを識別する。	ラベル名、タグ (キーワード `struct`、`union`、`enum` のあとに続く名前)、`struct`、`union`、`enum` のメンバー、および、通常の識別子の 4 つの名前空間を認識する。
`long double` 型	サポートしない。	`long double` 型の宣言を使用できる。
浮動小数点定数	浮動小数点の接尾辞 `f`、`l`、`F`、`L` はサポートされない。	
接尾辞なしの整数定数は異なる型を持つことができる。	整数定数の接尾辞 `u` と `U` はサポートされない。	

表 H–1 Solaris Studio ISO C との K&R Solaris Studio C の互換性 (続き)

項目	Solaris Studio C (K&R)	Solaris Studio ISO C
ワイド文字定数	ワイド文字定数についての ISO C 構文を使用できない。次に例を示します。 `wchar_t wc = L'x';`	この構文をサポートする。
'\a'および'\x'	文字 'a' と 'x' として処理する。	特別なエスケープシーケンス'\a'と'\x'として処理する。
文字列リテラルの連結	ISO C の隣接する文字列リテラルの連結はサポートしない。	
ワイド文字の文字列リテラル構文	ISO C のワイド文字の文字列リテラル構文はサポートしない。次に例を示します。 `wchar_t *ws = L"hello";`	この構文をサポートする。
ポインタ: `void *` と `char *`	ISO C の `void *` 機能をサポートする。	
単項プラス演算子	この ISO C の機能はサポートしない。	
関数のプロトタイプ - 省略記号	サポートしない。	ISO C は可変引数パラメータリストを示すための省略記号「...」の使用を定義する。
型定義	`typedef` は、同じ型名を持つ別の宣言により、内側のブロックで再宣言できない。	`typedef` は、同じ型名を持つ別の宣言により、内側のブロックで再宣言できる。
`extern` 変数の初期化	明示的に `extern` と宣言した変数の初期化はサポートしない。	明示的に `extern` と宣言した変数の初期化を定義として処理する。
集合体の初期化	ISO C の共用体または自動構造体の初期化はサポートしない。	
プロトタイプ	この ISO C の機能はサポートしない。	
前処理指令の構文	第 1 桁に `#` がある指令だけを認識する。	ANSI/ISO では、`#` 指令の前に空白文字を使用できる。
プリプロセッサ演算子 `#`	ISO C のプリプロセッサ演算子 `#` はサポートしない。	
`#error` 指令	この ISO C の機能はサポートしない。	
プリプロセッサ指令	`#ident` 指令とともに、2 つのプラグマ `unknown_control_flow` と `makes_regs_inconsistent` をサポートする。プラグマを認識できないとき、プリプロセッサは警告を発行する。	認識できないプラグマに対する動作は指定されていない。

表 H–1 Solaris Studio ISO C との K&R Solaris Studio C の互換性 *(続き)*

項目	Solaris Studio C (K&R)	Solaris Studio ISO C

事前定義されたマクロ名	次の ISO C 定義のマクロ名は定義されていない。
	`__STDC__`
	`__DATE__`
	`__TIME__`
	`__LINE__`	

H.2 キーワード

次の表は、ISO C 規格、Solaris Studio ISO C コンパイラ、および Solaris Studio C コンパイラのキーワードのリストです。

次の表は、ISO C 規格で定義されたキーワードのリストです。

表 H–2 ISO C 規格のキーワード

`_Bool`[1]	`_Complex`[1]	`_Imaginary`[1]	`auto`
`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`
`else`	`enum`	`extern`	`float`
`for`	`goto`	`if`	`inline`[1]
`int`	`long`	`register`	`restrict`
`return`	`short`	`signed`	`sizeof`
`static`	`struct`	`switch`	`typedef`
`union`	`unsigned`	`void`	`volatile`
`while`			

[1] `-xc99=all` でのみ定義されます。

C コンパイラは、追加のキーワードとして asm を定義しています。しかし、asm は `-Xc` モードではサポートされません。

次に、Solaris Studio C のキーワードのリストを示します。

表 H–3 Solaris Studio C (K&R) のキーワード

asm	auto	break	case
char	continue	default	do
double	else	enum	extern
float	for	fortran	goto
if	int	long	register
return	short	sizeof	static
struct	switch	typedef	union
unsigned	void	while	

索引

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X