C ユーザーズガイド

Forte Developer 7

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A
650-960-1300

Part No. 816-4918-10
2002 年 6 月 Revision A

原典： *C User's Guide*
Part No: 816-2454-10
Revision A

目次

図目次

表目次

はじめに

このマニュアルでは、ForteDeveloper C プログラミング言語コンパイラについて説明しています。ISO C コンパイラに固有の情報もあわせて記載しています。このマニュアルは、C 言語および UNIX の一般的な知識をもつアプリケーション開発者を対象としています。

このマニュアルでは、コンパイラのコマンドリファレンスを記載しています。また、型に基づく別名解析でコードを最適化する方法、ISO/IEC 9899:1999, Programming Language - C 規格でサポートされている機能、コード検査に使用できる `lint` プログラム、コードの並列化、ISO 互換コードへの移行、インクリメンタルリンカー、対話型プログラム `cscope` などについて説明します。付録として、ISO C のデータ表現、処理系定義、Sun C (K&R) と Sun ISO C の相違点、パフォーマンスチューニング、64 ビット環境用にコンパイルするためのアプリケーションの変換が含まれています。

書体と記号について

次の表と記述は、このマニュアルで使用している書体と記号について説明しています。

表 P-1　書体の規則

書体	意味	例
AaBbCc123	コマンド、ファイル、ディレクトリの名前、画面上のコンピュータ出力例。	.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。 % メールが届きました。
AaBbCc123	コード例で、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表します。 テキストで、言語、API、ライブラリ機能名のトークンを識別します。	% **su** パスワード： **ATOMIC** 指令
AaBbCc123	書籍名、新規の用語、強調すべき用語を示します。	『User's Guide』の第 6 章を読んでください。 これらは *class* オプションと呼ばれます。 これを実行するためには、スーパーユーザである必要があります。
AaBbCc123	コマンド行プレースホルダーテキストを示します。実際の名前または値に置き換えてください。	ファイルを削除するには、**rm** *filename* と入力します。

表 P-2　コードとコマンド行の規則

コード記号	意味	表記	コード例
[ ]	省略可能な引数を囲みます。	O[*n*]	-O4,- O
{ }	必須オプションの選択肢を囲みます。	d{y|n}	-dy
|	1 つだけ選択可能な引数は“パイプ”または“バー”記号で区切ります。	B{dynamic|static}	-Bstatic
:	コンマ同様、引数を区切るために使用します。	R*dir*[:*dir*]	-R/local/libs:/U/a
...	連続した省略を示します。	-xinline=*f1*[,...*fn*]	-xinline=alpha,dos

シェルプロンプトについて

シェル	プロンプト
UNIX の C シェル	machine_name%
UNIX の Bourne シェルと Korn シェル	machine_name$
スーパーユーザー (シェルの種類を問わない)	#

Forte Developer の開発ツールとマニュアルページへのアクセス

Forte Developer の製品コンポーネントとマニュアルページは、標準の /usr/bin/ と /usr/share/man の各ディレクトリにインストールされません。Forte Developer のコンパイラとツールにアクセスするには、PATH 環境変数に Forte Developer コン

ポーネントディレクトリを必要とします。Forte Developer マニュアルページにアクセスするには、PATH 環境変数に Forte Developer マニュアルページディレクトリが必要です。

PATH 変数についての詳細は、csh(1)、sh(1) および ksh(1) のマニュアルページを参照してください。MANPATH 変数についての詳細は、 man(1) のマニュアルページを参照してください。このリリースにアクセスするために PATH および MANPATH 変数を設定する方法の詳細は、『インストールガイド』を参照するか、システム管理者にお問い合わせください。

注 – この節に記載されている情報は Forte Developer 製品が /opt ディレクトリにインストールされていることを想定しています。 Forte Developer 製品が /opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

Forte Developer コンパイラとツールへのアクセス方法

PATH 環境変数を変更して Forte Developer コンパイラとツールにアクセスできるようにする必要があるかどうか判断するには以下を実行します。

▼ PATH 環境変数を設定する必要があるかどうか判断するには

1. コマンドプロンプトで次のように入力して、PATH 変数の現在値を表示します。

```
% echo $PATH
```

2. 出力内容から /opt/SUNWspro/bin を含むパスの文字列を検索します。

   パスがある場合は、PATH 変数は Forte Developer 開発ツールにアクセスできるように設定されています。パスがない場合は、次の指示に従って、PATH 環境変数を設定してください。

▼ PATH 環境変数を設定して Forte Developer のコンパイラとツールにアクセスする

1. C シェルを使用している場合は、ホームの .cshrc ファイルを編集します。Bourne シェルまたは Korn シェルを使用している場合は、ホームの .profile ファイルを編集します。
2. 次のパスを PATH 環境変数に追加します。

   /opt/SUNWspro/bin

Forte Developer マニュアルページへのアクセス方法

Forte Developer マニュアルページにアクセスするために MANPATH 変数を変更する必要があるかどうかを判断するには以下を実行します。

▼ MANPATH 環境変数を設定する必要があるかどうか判断するには

1. コマンドプロンプトで次のように入力して、dbx マニュアルページを表示します。

```
% man dbx
```

2. 出力された場合、内容を確認します。

   dbx(1) マニュアルページが見つからないか、表示されたマニュアルページがインストールされたソフトウェアの現バージョンのものと異なる場合は、この節の指示に従って MANPATH 環境変数を設定してください。

▼ MANPATH 変数を設定して Forte Developer マニュアルページにアクセスする

1. C シェルを使用している場合は、ホームの .cshrc ファイルを編集します。Bourne シェルまたは Korn シェルを使用している場合は、ホームの .profile ファイルを編集します。
2. 次のパスを PATH 環境変数に追加します。

   /opt/SUNWspro/man

Forte Developer マニュアルへのアクセス

Forte Developer の製品マニュアルには、以下からアクセスできます。

- 製品マニュアルは、ご使用のローカルシステムまたはネットワークの製品にインストールされているマニュアルの索引から入手できます。
  `/opt/SUNWspro/docs/ja/index.html`

製品ソフトウェアが `/opt` 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

- マニュアルは、`docs.sun.com`の Web サイトで入手できます。 以下に示すマニュアルは、インストールされている製品のマニュアルの索引から入手できます (`docs.sun.com` Web サイトでは入手できません)。
  - 『Standard C++ Library Class Reference』
  - 『標準 C++ ライブラリ・ユーザーズガイド』
  - 『Tools.h++ クラスライブラリ・リファレンスマニュアル』
  - 『Tools.h++ ユーザーズガイド』

インターネットの `docs.sun.com` Web サイト (`http://docs.sun.com`) から、サンのマニュアルを読んだり、印刷したり、購入することができます。マニュアルが見つからない場合はローカルシステムまたはネットワークの製品とともにインストールされているマニュアルの索引を参照してください。

注 - Sun では、本マニュアルに掲載した第三者の Web サイトのご利用に関しましては責任はなく、保証するものでもありません。また、これらのサイトあるいはリソースに関する、あるいはこれらのサイト、リソースから利用可能であるコンテンツ、広告、製品、あるいは資料に関して一切の責任を負いません。 Sun は、これらのサイトあるいはリソースに関する、あるいはこれらのサイトから利用可能であるコンテンツ、製品、サービスのご利用あるいは信頼によって、あるいはそれに関連して発生するいかなる損害、損失、申し立てに対する一切の責任を負いません。

アクセスできる製品マニュアル

Forte Developer 7 製品マニュアルは、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マニュアルを提供しております。アクセス可能なマニュアルは以下の表に示す場所から参照することができます。製品ソフトウェアが /opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

マニュアルの種類	アクセス可能な形式と格納場所
マニュアル(サードパーティ製マニュアルは除く)	形式：HTML 場所： http://docs.sun.com
サードパーティ製マニュアル: 『Standard C++ Library Class Reference』 『標準 C++ ライブラリ・ユーザーズガイド』 『Tools.h++ クラスライブラリ・リファレンスマニュアル』 『Tools.h++ ユーザーズガイド』	形式：HTML インストール製品について 場所：file:/opt/SUNWspro/docs/ja/index.html のマニュアル索引
Readme および マニュアルページ	形式：HTML インストール製品について 場所： file:/opt/SUNWspro/docs/ja/index.html のマニュアル索引
リリースノート	製品 CD 内の HTML ファイル

関連する Forte Developer マニュアル

以下の表は、file:/opt/SUNWspro/docs/ja/index.html から参照できるマニュアルの一覧です。 製品ソフトウェアが/opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

マニュアルタイトル	内容の説明
数値計算ガイド	浮動小数点演算における数値の精度に関する問題について説明しています。

関連する Solaris マニュアル

次の表では、docs.sun.com の Web サイトで参照できる関連マニュアルについて説明します。

マニュアルコレクション	マニュアルタイトル	内容の説明
Solaris 8 Reference Manual Collection	マニュアルページの節を参照。	Solaris のオペレーティング環境に関する情報を提供しています。
Solaris 8 Software Developer Collection	リンカーとライブラリ	Solaris のリンクエディタと実行時リンカーの操作について説明しています。
Solaris 8 Software Developer Collection	マルチスレッドのプログラミング	POSIX と Solaris スレッド API、同期オブジェクトのプログラミング、マルチスレッド化したプログラムのコンパイル、およびマルチスレッド化したプログラムのツール検索について説明します。

ご意見の送付先

米国 Sun Microsystems, Inc. では、マニュアルの向上に力を注いでおり、ユーザーのご意見やご提案をお待ちしております。ご意見などがありましたら、次のアドレスまで電子メールをお送りください。

`docfeedback@sun.com`

第1章

C コンパイラの紹介

このマニュアルでは、C コンパイラとそのオペレーティング環境、準拠規格、コンパイラの構成、C 関連のプログラミングツールについて説明します。

準拠規格

本コンパイラは、次の規格に準拠しています。

- ISO/IEC 9899:1999, Programming Languages - C 規格。 実装固有の動作に関する情報については、付録 Cを参照してください。
- FIPS 160 規格。

このリリースは、次の規格で指定されているいくつかの機能もサポートします。

- ISO/IEC 9899:1999, Programming Languages - C 規格。サポートされる機能の詳細については、付録 Dを参照してください。

このコンパイラは従来の K&R C (Kernighan and 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。

このマニュアルで使用される C99 という用語は、ISO/IEC 9899:1999 プログラミング言語 C を表します。同様に C90 という用語は、ISO/IEC 9899:1990 プログラミング言語 C を表します。

日本語化について

本コンパイラからのエラー、警告などのメッセージは、原則として日本語で出力されます。ただし一部のメッセージは、技術上の制限のため英語で出力されますので、ご了承ください。

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他の指定をしない限り、この 3 つの構成要素をそれぞれ自動的に起動します。

付録 A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。

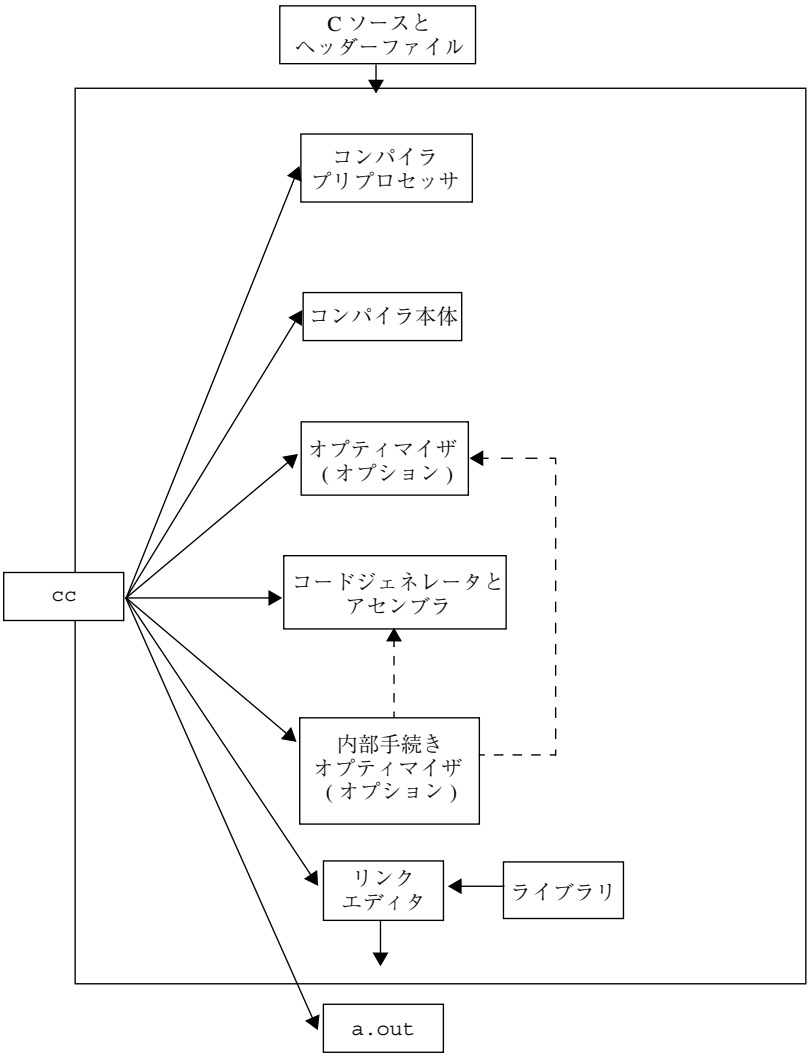

図 1-1　C コンパイルシステムの構成

次の表は、構成要素の要約を示しています。

表 1-1　C コンパイラシステムの構成要素

構成要素	説明	使用するコマンド
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 (最適化部)	*(SPARC)* -O、-x02、-x03、-x04、-x05、-fast
fbe	アセンブラ	
cg	コード生成、インライン機能、アセンブラ	*(SPARC)*
ipo	内部手続きオプティマイザ	*(SPARC)*
postopt	ポストオプティマイザ	*(SPARC)*
ir2hf	中間コード翻訳	*(INTEL)*
ube	コードジェネレータ	*(INTEL)*
ube_ipa	内部手続き解析プログラム	*(INTEL)*
ld	リンカー	
ild	インクリメンタルリンカー	*(SPARC)* -g、-xildon
mcs	コメントセクションの操作	-mr

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。

ソースの表示、デバッグ、およびパフォーマンス解析を行うツールも付属しています。詳細については、xxxiv ページの「Forte Developer マニュアルへのアクセス」を参照してください。

第2章

Sun の実装に固有の C コンパイラ情報

C コンパイラは、新しい ISO C 規格である ISO/IEC 9899-1999 に記述されている C 言語のいくつかの機能と互換性があります。旧規格である ISO/IEC 9889-1990 規格 (および改定第 1 版) と互換性のあるコードをコンパイルする場合、`-xc99=%none` を使用すると ISO/IEC 9899-1999 規格の拡張機能が無視されます。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。

環境変数

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。

OMP_NESTED

入れ子の並列化を有効または無効にします。

OMP_NUM_THREADS

実行中に使用するスレッド数を設定します。

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。

PARALLEL

(SPARC) 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、スレッドは個々のプロセッサにマップできます。プログラムを実行すると、指定したプロセッサ数のスレッドが生成され、並列化されたプログラムの各部が実行されます。

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。

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。

SUNPRO_SB_INIT_FILE_NAME

`.sbinit`(5) ファイルが格納されるディレクトリの絶対パス名。この変数は、`-xsb` または `-xsbfast` フラグが使用されている場合にのみ使用されます。

SUNW_MP_THR_IDLE

並列ジョブの分担部分を終えた後の各スレッドの状態を制御します。
`SUNW_MP_THR_IDLE` は、`spin` または `sleep` [*n* `s`|*n* `ms`] に設定できます。デフォルトは `spin` であり、これはスレッドがスピン待ちに入ることを意味します。もう一方の選択肢である `sleep` [*n* `s`|*n* `ms`] は、*n* に指定された時間だけスレッドをスピン待ち状態にし、その後休眠させることを意味します。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、`10 ms` は 10 ミリ秒を意味します。*n* に指定された時間が経過する前に新しいジョブが発生すると、スレッドはスピン待ちを中止し、新しいジョブを開始します。`SUNW_MP_THR_IDLE` に無効な値が含まれるか、あるいはこのオプションが指定されない場合、デフォルトとして `spin` が使用されます。

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 シェルの場合は以下のように入力します。

```
$ TMPDIR=<ディレクトリ>; export TMPDIR
```

C シェルの場合は以下のように入力します。

```
% setenv TMPDIR <ディレクトリ>
```

キーワード

__asm

__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。_ _asm 文は、関数の本体に配置しなければなりません。

_Restrict

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 機能の詳細については、付録 D を参照してください。

long long データ型

-xc99=%none を指定してコンパイルを行う場合、Sun 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。SPARC V8 や x86 では、long に 32 ビットの情報を格納できるのに対し、long long には 64 ビットの情報を格納できます。SPARC V9 では、long には 64 ビットの情報を格納できます。long long は -Xc モードでは使用できません。

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 データ型をもつ変数 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);
```

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。次に以下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、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。

定数

ここでは、Sun C コンパイラに固有の定数に関する情報について説明します。

整数定数

下表に示すように、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	型
u または U	unsigned
l または L	long
ll または LL	long long[1]
lu、LU、Lu、lU、ul、u、Ul、UL のいずれか	unsigned long
llu、LLU、LLu、llU、ull、ULL、uLL、Ull のいずれか	unsigned long long[1]

1. long long と unsigned long long は、-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

文字定数

エスケープシーケンスではない複数の文字からなる文字定数は、各文字が持つ数値から派生する値を持ちます。たとえば定数 '123' の持つ値は以下のようになります。

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 でない他の C では、この値は以下のようになります。

0	'1'	'2'	'3'

あるいは 0x313233 です。

インクルードファイル

C コンパイルシステムにi含まれる標準ヘッダーファイルをインクルード するには、次の書式を使用します。

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

山括弧(<>)を使用するとプリプロセッサはシステム内の標準の場所、通常は /usr/include ディレクトリにあるヘッダーファイルを検索します。

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。

```
% 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 のコマンド行では複数のオプションを指定できます。

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

-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。-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。

- コンパイラは、現在のディレクトリを検索しません。ただし、現在のディレクトリが -I 指令に明示的にリストされている場合は例外です。この効果は、書式 #include "foo.h" のインクルード文に対しても適用されます。
- 書式 #include "foo.h" のインクルード文に対し、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。
  a. -I オプションで命名されたディレクトリ (-I- の前と後の両方)
  b. /usr/include ディレクトリ
- 書式 #include <foo.h> のインクルード文に対し、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。
  a. -I- の後に配置された -I で命名されたディレクトリ (つまり、-I- の前に配置されている -I ディレクトリは検索されません)
  b. /usr/include ディレクトリ

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
```

次のコマンドは、書式 #include "foo.h" のインクルード文において、現在のディレクトリ (インクルードするファイルのディレクトリ) を検索するデフォルトの動作を示しています。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取り込みます。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 オプションは、コンパイラに対し、取り込まれたファイルのパスを出力するように指示します。

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

次のコマンドは、-I- オプションの効果を示しています。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.c の #include "c.h" を処理する際、プリプロセッサは、inc/c.h ヘッダーファイルの代わりに ./c.h ヘッダー・ファイルを取り込みます。

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

詳細については、254 ページの 「-I[-|<ディレクトリ>]」の節を参照してください。

非標準浮動小数点

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、表現可能な最小の数は次のようになると考えられます。

```
1.234567e-38
```

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、新しい値は次のようになります。

```
1.23456e-39
```

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。

前処理指令と名前

ここでは、表明、プラグマ、および事前定義名について説明します。

表明 (assertion)

以下の書式で指定します。

```
#assert <述語> (<トークン列>)
```

<トークン列> は、表明の名前領域 (マクロ定義用の領域から分離されています) 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なりません。

```
#assert <述語>
```

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 (-Xc モードを除く)。

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。

```
#assert system (unix)
#assert machine (sparc) (SPARC)
#assert machine (i386) (x86)
#assert cpu (sparc) (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 #<述語>(<空でないトークン列>)
```

たとえば以下のように指定して、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。

プラグマ

以下の書式を持つ前処理行は、各処理系が定義した処理を指定します。

```
#pragma <プリプロセッサトークン>
```

次の #pragma 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。

#pragma align <整数> (<変数>[,<変数>])

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 <整数> バイト境界に揃えられます。

- <整数> には 2 の階乗 (1 ～ 128) を指定します。有効な値は 1、2、4、8、16、32、64、128 です。
- <変数> 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なりません。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、その後で宣言されていない変数は無視されます。たとえば次のようになります。

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

#pragma does_not_read_global_data (<関数>[,<関数>])

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。

このプラグマは、指定した関数のプロトタイプを宣言した後でのみ使用できます。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。

#pragma does_not_return(<関数>[,<関数>])

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のバックエンドに表明します。この表明により、オプティマイザは、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、指定した関数のプロトタイプを宣言した後でのみ使用できます。

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

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

#pragma does_not_write_global_data(<関数>[,<関数>])

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。

このプラグマは、指定した関数のプロトタイプを宣言した後でのみ使用できます。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。

#pragma error_messages(on|off|default,<タグ>... <タグ>)

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, <タグ>... <タグ>)

  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, <タグ>... <タグ>)

  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, <タグ>... <タグ>)

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。

#pragma fini (<関数 1> [,<関数 2>...,<関数 *n*>])

main() ルーチンを呼び出した後、<関数 1> から <関数 *n*> までの関数 (終了関数) を呼び出します。この種の関数は、型が void で引数はあってはなりません。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。

#pragma ident <文字列>

実行可能プログラムの .comment セクション内に任意の <文字列> を格納します。

#pragma init (<関数 1> [,<関数 2>...,<関数 *n*>])

main() を呼び出す前に、<関数 1> から <関数 *n*> 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 で引数はあってはなりません。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。

#pragma [no_]inline(関数[,関数])

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、-xcrossfile 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、指定した関数のプロトタイプを宣言した後でのみ使用できます。

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

#pragma int_to_unsigned (<関数>)

-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。

(SPARC) #pragma MP serial_loop

詳細については、56 ページの「直列プラグマ」を参照してください。

(SPARC) #pragma MP serial_loop_nested

詳細については、56 ページの「直列プラグマ」を参照してください。

(SPARC) #pragma MP taskloop

詳細については、57 ページの「並列プラグマ」を参照してください。

(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。プラグマは現行ブロック内の次の for ループに適用されます。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。

(SPARC) #pragma no_side_effect (<関数> [,<関数>...])

<関数> には、現行の翻訳単位内の関数名を指定します。関数はプラグマより前に宣言されていなければなりません。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なりません。指定した <関数> 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、<関数> は渡された引数にだけ依存する結果の値を返します。<関数> 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。

#pragma opt *level* (<関数> [,<関数>])

指定した関数サブプログラムに対する最適化レベルを指定します。最適化レベルとしては 0～5 を選択することができます。0 に設定すると、最適化は無効になります。このプラグマを使用する前に、関数サブプログラムのプロトタイプを作成しておく必要があります。

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。

#pragma pack(*n*)

構造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せられます。*n* を指定する場合には、すべての構造体メンバーに対して最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最も厳密な整列を指示する値 (x86 では 4、SPARC v8 では 8、SPARC v9 では 16) 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 `#pragma pack` 指令の間のすべての構造体の定義に適用されます。別の翻訳単位で同じ構造体に対して異なる `#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。

注 – `#pragma pack` を使用して構造体メンバーを自然境界以外の境界で整列させると、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ることがあります。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、291 ページの「-xmemalign=ab」を参照してください。

(SPARC) `#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。プラグマは現行ブロック内の次の `for` ループに適用されます。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。

`#pragma rarely_called(<関数>[,<関数>])`

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のバックエンドに与えます。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。このプラグマは提案ですので、コンパイラのオプティマイザは、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。

次の例に示すように、#pragma rarely_called プリプロセッサ指令は、指定した関数のプロトタイプを宣言した後でのみ使用できます。

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

#pragma redefine_extname <旧外部参照名> <新外部参照名>

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 <旧外部参照名> の名前がすべて <新外部参照名> に置換されます。この結果、リンク時にリンカーは新しい名前だけを認識します。関数定義、初期設定子、または式のいずれかとして <旧外部参照名> を最初に使用した後、#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_extnmae 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__ */
```

#pragma returns_new_memory(<関数>[,<関数>])

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。この表明により、オプティマイザはポインタ値を追跡して、メモリー位置を明確にし、ループのスケジューリングやパイプライン処理、並列化処理を改善することができます。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、指定した関数のプロトタイプを宣言した後でのみ使用できます。

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

#pragma unknown_control_flow （<名前> [,<名前>]...）

#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。サンのシステム上では、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 unkown_control_flow(setjmp);
```

setjmp() のような特性を持つ他の関数も、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。

(SPARC) #pragma unroll（<展開係数>）

このプラグマは、引数 <展開係数> に正の整定数を受け入れます。プラグマは現行ブロック内の次の for ループに適用されます。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

限り、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なら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。

#pragma weak <シンボル 1> [=<シンボル 2>]

弱い大域シンボルを定義します。このプラグマは主にライブラリを構築するソースファイルの中で使用されます。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 <シンボル>
```

これは <シンボル> を弱いシンボルとして定義しています。<シンボル> 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 <シンボル 1> = <シンボル 2>
```

これは <シンボル 1> を、<シンボル 2> の別名の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、<シンボル 2> を定義した同じ変換ユニットの中に限り使用できます。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 <シンボル 1> 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、<シンボル 1> 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 <シンボル 1> 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 <シンボル 1> の弱い大域定義は使用されません。プログラムが直接 <シンボル 2> を呼び出すと、ライブラリにある定義が使用されます。<シンボル 2> の定義が重複して行われるとエラーになります。

事前定義済みの名前

下の表に示す識別子は、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 2-2　事前定義済みの識別子

識別子	説明
__STDC__	__STDC__ 1　-Xc __STDC__ 0　-Xa、-Xt 未定義　-Xs

__STDC__ が未定義の場合 (#undef __STDC__)、コンパイラは警告を出します。-Xs モードでは __STDC__ は定義されません。

事前定義されているものは次のとおりです (-Xc モードでは無効)。

- sun
- unix
- sparc *(SPARC)*
- i386 *(x86)*

次の事前定義されているものは、あらゆるモードで有効です。

- __sun
- __unix
- __SUNPRO_C=0x540
- __`uname -s`_`uname -r` (例: __SunOS_5_7)
- __sparc *(SPARC)*
- __i386 *(x86)*
- __BUILTIN_VA_ARG_INCR
- __SVR4
- __sparcv9 (-Xarch=v9、v9a)

コンパイラにより、次のオブジェクト形式のマクロが事前定義されます。

__PRAGMA_REDEFINE_EXTNAME

これにより、プラグマが認識されます。

__RESTRICT は、-Xa および -Xt モードでのみ有効です。

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;
```

後に続く 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。

上の例を別の方法で記述すると、次のようになります。

```
static const int ptrarray[] = { &&label1 - &&label1,
&&label2 - &&label1, &&label3 - &&label1 };
goto *(&&label1 + ptrarray[i]);
```

この例では必要な動的再配置を行う回数が少ないので、その結果として、データ (ptrarray の要素) を読み取り専用にすることができ、共有ライブラリコードに関してはさらに効果的です。

第3章

Sun C コードの並列化

C コンパイラは、SPARC™ メモリー共有型マルチプロセッサのマシン上で実行するコードを最適化できます。この最適化処理は「並列化」と呼ばれます。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。
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。

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に、Sun ANSI/ISO C には別名の情報をコンパイラに渡すためのプラグマおよびポインタ修飾子が用意されています。

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、272 ページの「-xautopar」の節を参照してください。

OpenMP に対する並列化

OpenMP の仕様に準拠するように コードをコンパイルできます。C 言語の OpenMP 仕様の詳細については、http://www.openmp.org/specs/ にアクセスしてください。

コンパイラ の OpenMP サポートを利用するには、コンパイラの -xopenmp オプションを発行する必要があります。296 ページの「-xopenmp[=i]」の節を参照してください。

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、34 ページの「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、付録 G を参照してください。

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
- SUNW_MP_THR_IDLE
- SUNW_MP_WARN

■ STACKSIZE

PARALLEL

マルチプロセッサ上で実行する場合は、PARALLEL 環境変数を設定してください。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。

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。

SUN_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後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 < 値 >
```

<値> 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 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。その他に、n 単位をスピン (busy-wait) してから、sleep[*n* s|*n* ms] によりスレッドをスリープ状態にすることもできます。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、10 ms 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。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、spin はデフォルトとして使用されます。

SUNW_MP_WARN

この環境変数を TRUE に設定すると、OpenMP その他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() を使用することより、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、32 ページの「OpenMP の実行時の警告の処理」を参照してください。

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 制限なし
filesize 制限なし
datasize 2097148 kbytes
stacksize 8192 kbytes <- 現在のメインスタックのサイズ
coredumpsize 0 kbytes
descriptors 256
memorysize 制限なし
% limit stacksize 65536 <- メインスタックのサイズを 64M バイトに設定
```

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 1M バイト、64 ビットアプリケーションの場合は 2M バイトです。このサイズは、STACKSIZE 環境変数で設定します。

```
% setenv STACKSIZE 8192 <- スレッドのスタックサイズを 8M バイトに設定
```

並列化されたほとんどのコードでは、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。

キーワード

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、360 ページの「C99 のキーワード」を参照してください。

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。

コード例 3-1　依存性を持つループ

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

この例では、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。コード例 3-1 のように、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場合、C コンパイラはそのループの並列化を行いません。

別の例を考えてみましょう。

コード例 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、ループ中における各繰り返しでは、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。上述のコード例 3-1 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。上述のコード例 3-2 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。

コード例 3-3 では、2 個の異なる繰り返しで、配列 a の同じ要素に書き込みが実行されるかどうかは、配列 b の要素に重複した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限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。

コード例 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];
}
```

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 (図 3-1 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。

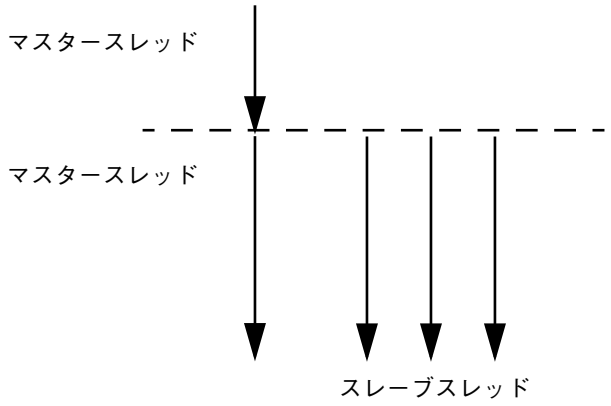

図 3-1　マスタースレッドとスレーブスレッド

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とられます。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、他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、以下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とるために必要な時間よりはるかに短くてすみます。

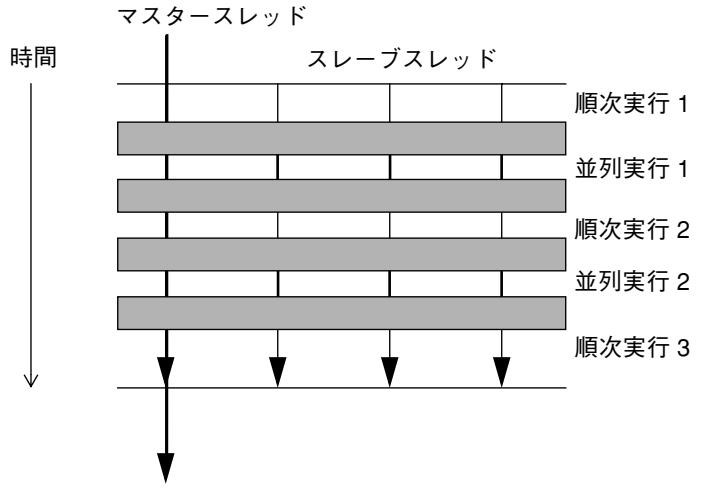

図 3-2　ループの並列実行

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。

コード例 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、以下のような文が実行されることになります。

コード例 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、以下に示します。

コード例 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 */
}
```

コード例 3-6 は、コード例 3-3 と基本的に同一なもので、それぞれのスカラー変数参照 t がここでは配列参照 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、各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。次に例を示します。

コード例 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 */
    }
}
```

コード例 3-7 では、この例では、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以下に示します。

コード例 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いいことになります。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。

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。

コード例 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 */
```

コード例 3-9 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後、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。

コード例 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、必ず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。

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。

コード例 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 */
}
```

コード例 3-11 では、ループで 2 個の配列の内積を計算して、変数 sum に格納しています。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。

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、全体の実行速度が早くならずに、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早い勢いで増加します。したがって、資源の限界に達しない限り、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。

アムダールの法則

固定されたプログラムの速度の向上度は、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、あるプログラムの並列化による速度の向上は、プログラムの順次実行部分によって制限されることを単に述べています。次は、プログラムの速度向上に関する式です。F は (全実行時間を 1 とした場合の) 順次実行部分の実行時間の割合で、残りの時間が P 個のプロセッサに均等に分散されます。この式から分かるように、式の 2 番目の項の値がゼロになると、値が固定されている 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、この概念を表しました。黒く塗ってある部分がプログラム中の順次実行部分を表現していて、この部分は 1、2、4、8 プロセッサに対して一定であるのに対し、斜線部分がプログラムの並列部分で、複数のプロセッサ間で一様に分割されています。

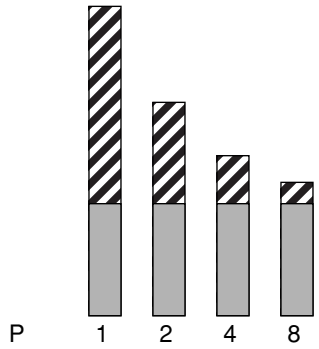

図 3-3　固定された問題の処理速度の向上

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固定であったり、そうでなかったりします。

図 3-4 には、プログラムに順次実行部分がそれぞれ 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。

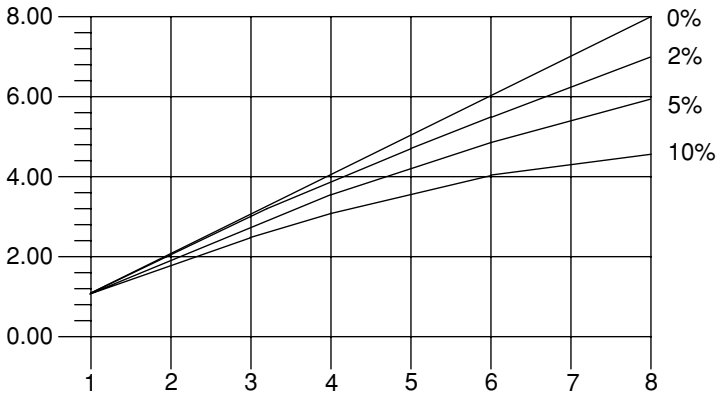

図 3-4　アムダールの法則による処理速度向上の曲線

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。

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

この式で K1 と K2 は固定された係数です。この仮定では、速度向上の曲線は図 3-5 のようになります。

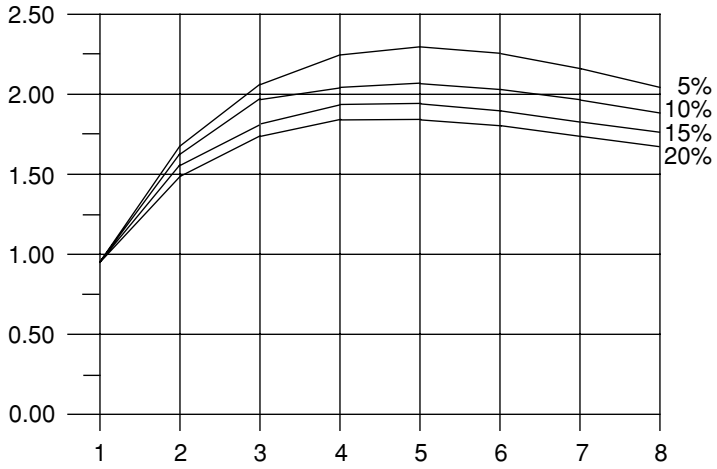

図 3-5　オーバーヘッドがある場合の速度向上の曲線

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。

ガスタフソンの法則

アムダールの法則では、実際のプログラム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。プログラムの順次実行部分に費やされる時間の割合は、プログラムサイズに依存することがあります。つまり、プログラムサイズが増加すると、速度向上が可能になる場合があります。例を使って説明します。

コード例 3-12 問題サイズの拡大によりスピードアップの可能性が増えることがある

```
/*
 * 配列を初期化
 */
for (i=0; i < n; i++) {
    for (j=0; j < n; j++) {
        a[i][j] = 0.0;
        b[i][j] = ...
        c[i][j] = ...
    }
}
/*
 * 行列の積を求める
 */
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、プログラム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。このプログラムの場合は、プログラム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。

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。

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。

コード例 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、上記の例では、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。

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。

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 (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。

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以下で説明します。

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。これを以下の例で説明します。

コード例 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 */
    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以下のようになります。

コード例 3-15 分散されたループ

```
/* L1: 並列実行ループ */
for (i=0; i < n; i++) {
    x[i] = y[i] + z[i]*w[i];              /* S1 */
    y[i] = z[i] - x[i];                   /* S3 */
}
/* L2: 順次実行ループ */
for (i=0; i < n; i++) {
    a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /* S2 */
}
```

この変換の後、上記ループ 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限りません。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。

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。

コード例 3-16 作業量の少ないループ

```
/* L1: 小さな並列ループ */
 for (i=0; i < 100; i++) {
    a[i] = a[i] + b[i];         /* S1 */
}
/* L2: 別の小さな並列ループ */
 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、以下のように安全に融合することができます。

コード例 3-17 融合された 2 つのループ

```
/* L3: 大きな並列ループ */
 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限りません。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。

コード例 3-18 安全でない融合の例

```
/* L1: 小さな並列ループ */
 for (i=0; i < 100; i++) {
    a[i] = a[i] + b[i];       /* S1 */
}
/* L2: データの依存性がある小さなループ */
 for (i=0; i < 100; i++) {
    a[i+1] = a[i] * d[i];     /* S2 */
}
```

コード例 3-18 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。

ループの交換

入れ子になっているループの最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。以下の例で説明します。

コード例 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。

コード例 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。

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。以下の例を考えてみましょう。

コード例 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、以下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限り、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、54 ページの「制限付きポインタ」の節を参照してください。

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。

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なりません。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。

以下の関数 vsq() を考えてみましょう。

コード例 3-22 2 つのポインタを持つループ

```
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、ループ内の異なるく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。以下に、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たくない場合は、次のコマンド行オプションで、関数の引数を値とするポインタを制限付きポインタとして指定することができます。

```
-xrestrict=[< 関数 1>,...,< 関数 n>]
```

関数リストが指定されている場合には、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には、C のソース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。たとえば -xrestrict=vsq を使用すると、上述の関数 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] を計算した後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。

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。Sun ISO C では、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。

直列プラグマ

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
```

#pragma MP serial_loop 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指示します。

```
#pragma MP serial_loop_nested
```

#pragma MP serial_loop_nested 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指示します。なお、serial_loop_nested のスコープは、このプラグマが適用されるループの範囲を越えることはありません。

並列プラグマ

並列プラグマは 1 つだけあります。

#pragma MP taskloop (<オプション>)

MP taskloop プラグマは、オプションとして、以下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>)

MP taskloop プラグマ 1 つに対して指定できるオプションは 1 つだけです。ただし、複数のプラグマの効果を重ねて、ソースコード内の現在のブロック内にある次の for ループに適用することができます。

```
#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、label、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限り、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。

以下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なりません。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。 変数の種類を縮約または読み取り専用にするには、明示的にプラグマで指示しなければなりません。詳しくは、63 ページの「reduction 変数」および 61 ページの「readonly 変数」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、または以下のスコープの規則にもとづいて決まります。

private 変数と shared 変数のデフォルトのスコープの規則

スレッド固有変数は、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固有変数に割り当てられた値は、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は見えません。これに対して共有変数は、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 1 つのプロセッサが共有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なりません (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、変数がスレッド固有と共有のどちらであるかを決定するために、以下の「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限り、その変数はデフォルトで共有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固有変数として扱われ、また常にストアバック変数です。

明示的に並列化された for ループ内のすべての変数を、共有、スレッド固有、縮約、または読み取り専用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することを、強くお勧めします。

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 (<スレッド固有変数リスト>)

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、それ自体が明確に共有、読み取り専用、または縮約であることが指定されていない限り、デフォルトのスコープの規則に従って、共有またはスレッド固有のどちらかに分類されます。

スレッド固有変数とは、それ自体の値がループのある繰り返しを処理するプロセッサ専用になっている変数のことです。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固有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。スレッド固有変数には、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なりません。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固有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 (<共有変数リスト>)

このプラグマは、現在のループで共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、それ自体が明確にスレッド固有、読み取り専用、または縮約であることが指定されていない限り、デフォルトのスコープの規則に従って、共有またはスレッド固有のどちらかに分類されます。

共有変数とは、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共有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>)

このプラグマは、現在のループで読み取り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。

読み取り専用変数とは、共有変数の特殊なクラスのことで、ループのどの繰り返しでも、その値を変更できません。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、現在のループでストアバック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。

ストアバック変数とは、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ストアバック変数の値が、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ストアバック変数として宣言するとよいでしょう。

なお、ストアバック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、ストアバック変数の最終的な値を保持しているプロセッサとは、異なる可能性があります。

以下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);
```

上記の例では、printf() 呼び出しによって出力されるストアバック変数 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ストアバック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以下に示します。

```
#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 (<縮約変数リスト>)

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縮約変数として扱われるために使用します。縮約変数とは、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。縮約変数リストにより、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。以下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 (和の) 縮約変数であり、i ループが (和の) 縮約ループになっています。

スケジューリングの制御

Sun 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以下に示します。

```
#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、以下のいずれかを指定できます。

■ static

静的スケジューリングでは、ループ内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++) {
...
}
```

上述の例では、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 [<チャンクサイズ>]

自己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 <チャンクサイズ> には、使用するチャンクサイズを指定します。<チャンクサイズ> は、正の整定数か、もしくは整数型の変数でなければなりません。変数の <チャンクサイズ> 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。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++) {
...
}
```

この例では、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、割り当て順に以下のようになります。

120, 120, 120, 120, 120, 120, 120, 120, 40.

■ gss [<最小チャンクサイズ>]

ガイド付き自己スケジューリング (GSS) 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 <最小チャンクサイズ> を指定すると、可変なチャンクサイズが最低でも <最小チャンクサイズ> になるように設定されます。

<最小チャンクサイズ> は、正の整定数か、もしくは整数型の変数でなければなりません。変数の <最小チャンクサイズ> 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++) {
...
}
```

この例では、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、割り当て順に以下のようになります。

250, 188, 141, 106, 79, 59, 45, 33, 25, 19, 14, 11, 10, 10, 10.

- `factoring [<最小チャンクサイズ>]`

ファクタリング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 <最小チャンクサイズ> を指定すると、可変なチャンクサイズが最低でも <最小チャンクサイズ> になるように設定されます。<最小チャンクサイズ> は、正の整定数か、もしくは整数型の変数でなければなりません。変数の <最小チャンクサイズ> 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++) {
...
}
```

上述の例では、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以下のようになります。

125, 125, 125, 125, 62, 62, 62, 62, 32, 32, 32, 32, 16, 16, 16, 16, 10, 10, 10, 10, 10, 10.

第4章

インクリメンタルリンカー (ild)

この章では、インクリメンタルリンカー (ild)、ild に固有の機能、メッセージ例、および ild オプションについて説明します。

インクリメンタルリンカーとは

ild はインクリメンタルリンカーといい、ld (通常のリンカー) の代わりにプログラムのリンク処理を行います。ild を使用すると、ld を使用するよりも、開発 (編集、コンパイル、リンク、デバッグの繰り返し) をより効率的に速く行うことができます。全体の再リンクを行わずに処理を続けるには dbx の「修正継続」機能も使用できますが、ild を使用すると処理がより速くなります。修正継続機能の詳細については『dbx コマンドによるデバッグ』の第 11 章を参照してください。

ild では変更部分だけがリンクされるので、変更していないオブジェクトファイルは再リンクを行わずに、変更したオブジェクトコードを以前に作成した実行可能ファイルに挿入することができます。再リンクに必要な時間は、変更したコードの量によって異なります。コードの変更が少ない場合は、リンク処理にかかる時間は短くなります。

初めてリンクを行う時には ild でも ld と同じくらいの時間が必要ですが、その後の ild リンクは ld リンクよりかなり時間が短縮されます。ただし、ild リンクの方が実行可能ファイルのサイズが大きくなります。

インクリメンタルリンク処理の概要

ild を使用すると、初期リンクで、さまざまなテキスト、データ、bss、例外テーブルセクションなどが後にプログラムを拡張するときのための予備スペース (パディング) とともにスペースが追加されます (図 4-1 を参照)。また、すべての再配置レコードと大域シンボルテーブルが、実行可能ファイルの新しい永久領域に保存されます。さらにインクリメンタルリンクを行うと、タイムスタンプによって、どのオブジェクトファイルが変更されたかが調べられ、変更されたオブジェクトコードが以前に作成した実行可能ファイルに追加されます。すなわち、以前のバージョンのオブジェクトファイルは無効になり、空き領域または必要に応じて実行可能ファイルのパディングセクションに、新しいオブジェクトファイルが読み込まれます。無効になったオブジェクトファイル内のシンボルに対する参照はすべて、修正された新しいオブジェクトファイルを参照するように変更されます。

すべての ld コマンドオプションが ild でサポートされているわけではありません。ild でサポートされていないコマンドオプションを指定すると、/usr/ccs/bin/ld が起動されて、リンクが行われます。インクリメンタルリンカーでサポートされていないコマンドの詳細については、86 ページの「ild で使用できない ld オプション」の節を参照してください。

ild の使用法

ild は特定の条件が揃った場合に、 ld の代わりにコンパイラによって自動的に呼び出されます。コンパイラを起動するとコンパイラドライバが起動され、ドライバは特定のオプションを渡された場合に ild を使用します。コンパイラドライバはコマンド行からオプションを読み取り、プログラムを正しい順序で実行し、渡された引数リストに従ってファイルを追加します。

たとえば、cc は最初に acomp (コンパイラのフロントエンド) を実行し、acomp が最適化コードジェネレータを実行します。続いて cc は、コマンド行に指定されている他のソースファイルに対しても同じ処理を行います。次に、指定されたオプションによって ild または ld のどちらかを呼び出し、コンパイルしたすべてのファイルと、プログラムを完成させるために必要な他のファイルやライブラリを、 ild または ld に渡します。

次の図に、インクリメンタルリンク処理の例を示します。

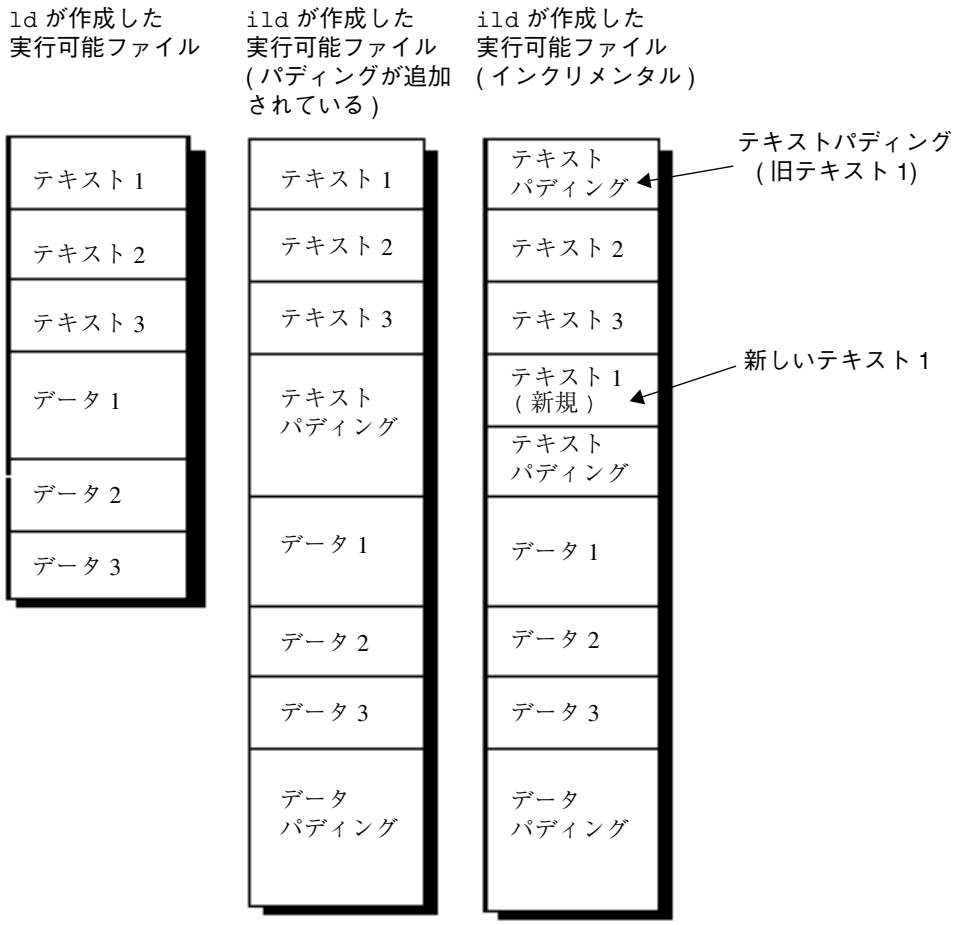

図 4-1　インクリメンタルリンク処理の例

ild と ld のどちらを使用するかは、コンパイラのオプションによって次のように制御されます。

- -xildon オプションを指定すると ild が使用されます
- -xildoff オプションを指定すると ld が使用されます

注 – -xildon と -xildoff が両方とも指定された場合、最後に指定されたオプションを使用してリンカーが選択されます。

- -g オプション
  -xildoff と -G が両方とも指定されていない場合でリンクのみを行う場合 (コマンド行上にソースファイルは存在しません) に、ild を使用します。-g オプションの詳細については、252 ページの「-g」を参照してください。

- -G オプション
  -G オプションが指定されていると、-g オプションによるリンカーの選択が無効になります。-G オプションの詳細については、252 ページの「-G」を参照してください。

デフォルトのメークファイル構造 (リンクコマンド行内に -g オプションのようなコンパイルオプションを含む) で -g オプションを指定してデバッグを行うと、自動的に ild が使用されます。

ild の動作

初期リンク時には以下の情報が保存されます。

- 参照するすべてのオブジェクトファイル
- 作成した実行可能ファイル用のシンボルテーブル
- コンパイル時に解決されなかったすべてのシンボル参照

ild の初期リンクでは、ld リンクとほぼ同じ時間がかかります。

インクリメンタルリンクでは以下の処理が行われます。

- 変更されたファイルの検出
- 変更されたオブジェクトファイルの再リンク
- 保存されている情報を使用して、プログラムの残りの部分で変更されたシンボル参照を修正

ild のインクリメンタルリンクは ld リンクより高速です。

通常は、一回だけ初期リンクを行い、その後のリンクはすべてインクリメンタルリンクを実行します。

たとえば、ild ではコード内でシンボル foo を参照しているすべての場所が保存されます。foo の値を変更するインクリメンタルリンクを行うと、foo を参照している値もすべて変更する必要が生じます。

そこで ild は、プログラムの構成要をいくつかに分散し、実行可能ファイルの各セクションにパディングを追加します。パディングによって、ld リンクを実行したときよりも、実行可能モジュールのサイズが大きくなります。インクリメンタルリンクを何度か繰り返すうちに、オブジェクトファイルのサイズは徐々に大きくなっていくので、パディングがすべて使用されてしまうことがあります。このような場合には、メッセージが表示され、実行可能ファイル全体に対して完全再リンクが行われます。

たとえば、図 5-1 の 3 つの図は、それぞれリンク済み実行可能ファイルのテキストとデータの並びを示しています。左の図は、ld がリンクした実行可能ファイルのテキストとデータを示します。中央の図は、ild がリンクした実行可能ファイルにテキストとデータのパディングが追加された様子を示します。ソースファイルのテキスト 1 に対して、テキスト 1 以外のセクションのサイズには影響を与えずに、テキスト 1 のサイズだけが増えるような変更が行われたとします。右の図では、テキスト 1 の元の位置がテキストパディングに置き換えられています (テキスト 1 は無効になります)。元のテキスト 1 はテキストパディング領域の一部分に移動されています。

-xildon オプションを指定せずにコンパイラドライバ (たとえば、cc または CC) を起動すると ld が呼び出されるので、インクリメンタルでないリンクが行われ、サイズがより小さな実行可能ファイルが作成されます。

dbx デバッガは、ild によってプログラムの間に挿入されたパディングを認識することができるので、ild で作成された実行可能ファイルのデバッグを行うことができます。

ild が認識できないコマンド行オプションを指定すると、ild は ld (/usr/ccs/bin/ld) を呼び出します。ild は ld (in /user/ccs/bin/ld) と互換性があります。詳細については、78 ページの「ild オプション」を参照してください。

ild でしか使用できないファイルはありません。

ild の制限事項

共有オブジェクトを作成する場合に ild を起動しても、ld が起動されてリンクが実行されます。

オブジェクトファイルに多くの変更を加えると、ild の処理速度が低下することがあります。ild は、ファイルに多くの変更が加えられていることを検出すると、自動的に完全再リンクを行います。

最終的に製品となるコードを作成するときには ild (-xildon オプション) を使用しないでください。ild を使用すると、パディングによってプログラムが分散されるため、作成されるファイルのサイズが大きくなり、リンクに余分な時間がかかります (-g オプションがある場合は -xildoff を使用してください)。

小さいプログラムでは ild を使用しても、リンク処理の時間はあまり短縮されません。また、大きなプログラムよりも、実行可能ファイルのサイズが増加する割合が高くなることがあります。

実行可能ファイルを操作する他社のツールを ild で作成したバイナリに使用すると、予想外の結果が出ることがあります。

実行可能ファイルを修正するプログラム (たとえば、strip または mcs など) を使用すると、ild のインクリメンタルリンク機能に影響を及ぼすことがあります。このような場合には、メッセージが表示されて完全再リンクが実行されます。完全再リンクの詳細については、73 ページの「完全再リンクが行われる場合」を参照してください。

完全再リンクが行われる場合

以下に、リンクを実行するために ild が ld を呼び出す例を示します。

ild 先送りリンクメッセージ

メッセージ「ild: calling ld to finish link」は、ild がリンクを実行できないため ld に先送りしてリンクを実行することを意味します。このようなメッセージはデフォルトでは出力されますが、-z i_quiet オプションを使って出力されないようにすることもできます。

次のメッセージは ild が暗黙的に (-g オプションで) 要求されている場合は抑制できますが、コマンド行で -xildon が指定されている場合は常に表示されます。このメッセージは -z i_verbose オプションを指定すると常に表示され、-z i_quiet オプションを指定すると表示されません。

```
ild: calling ld to finish link -- cannot handle shared libraries in
archive <ライブラリ名>

<訳>、
ld によるリンクを実行します  --  アーカイブ <ライブラリ名> の共有ライブラリを
                                処理できません
```

以下に、-z i_verbose メッセージのその他の例を示します。

```
ild: calling ld to finish link -- cannot handle keyword <キーワード名>
ild: calling ld to finish link -- cannot handle -d <キーワード名>
ild: calling ld to finish link -- cannot handle -z <キーワード名>
ild: calling ld to finish link -- cannot handle argument <キーワード名>

<訳>
ld によるリンクを実行します  --  キーワード <キーワード名> を処理できません
ld によるリンクを実行します  --  キーワード名 -d を処理できません
ld によるリンクを実行します  --  -z <キーワード名> を処理できません
ld によるリンクを実行します  --  引数 <キーワード名> を処理できません
```

ild 再リンクメッセージ

メッセージ「ild: (Performing full relink)」は、何らかの理由で ild がインクリメンタルリンクを実行できないため、完全再リンクを実行しなければならないことを示します。これはインクリメンタルリンクより時間がかかることを示すメッセージであり、エラーではありません (詳細については、70 ページの「ild の動作」を参照)。ild のメッセージは -z i_quiet オプションと -z i_verbose オプションで制御できます。一部のメッセージには詳細情報モードがあり、より詳しい情報が示されます。

これらのメッセージはすべて -z i_quiet オプションで抑制できます。デフォルトメッセージが詳細情報モードの場合はメッセージの最後に [...] と表示され、詳細情報があることを示します。詳細情報は、-z i_verbose オプションを使用して参照できます。以下の例では、-z i_verbose オプションが選択されています。

例 1: 内部空き領域の不足

完全再リンクでよく表示されるメッセージとして、「internal free space exhausted」があります。

 # test1.o を作成
最小限のデバッグ情報を a.out に出力
1 行でコンパイルとリンクを を行い、デバッグ情報をすべて a.out に出力

```
$ cat test1.c
int main() { return 0; }
$ rm a.out
$ cc -xildon -c -g test1.c
$ cc -xildon -z i_verbose -g test1.o

$ cc -xildon -z i_verbose -g test1.c


ild: (Performing full relink) internal free
space in output file exhausted (sections)
$
<訳>
完全再リンクを実行します
-- 出力ファイルの容量が不足しています
```

この例からわかるように、1 行コンパイルから 2 行コンパイルに変更すると、実行可能ファイル内のデバッグ情報が増え、このために領域が不足し、完全再リンクが実行されます。

例 2: strip の実行

strip を実行すると、別の問題が発生します。次の例は例 1 からの続きです。

#a.out を取り除く
#インクリメンタルリンクを行おうとする

```
$ strip a.out
$ cc -xildon -z i_verbose -g test1.c

ild: (Performing full relink) a.out has been
altered since the last incremental link --
maybe you ran strip or mcs on it?
$
<訳>
完全再リンクを実行します
--  最後のインクリメンタルリンク以降に strip または
    mcs を実行したために、a.out が変更されています。
```

例 3: ild のバージョン

旧バージョンの ild で作成した実行可能ファイルに対して、新バージョンの ild を実行すると、以下のようなエラーメッセージが表示されます。

#old_executable は ild の以前のバージョンで作成

```
$ cc -xildon -z i_verbose foo.o -o old_executable

ild: (Performing full relink) an updated ild
has been installed since a.out was last linked
(2/16)
<訳>
完全再リンクを実行します
--  最後の a.out のリンク以降に、更新された ild がイ
    ンストールされています (2/16)
```

注 – 数字 (2/16) は、バグレポート用に使用されます。

例 4: 変更されたファイルが多い

インクリメンタルリンクより完全再リンクを行う方が、処理時間が短くなる場合があります。以下に例を示します。

```
$ rm a.out
$ cc -xildon -z i_verbose \
     x0.o x1.o x2.o x3.o x4.o x5.o x6.o x7.o x8.o test2.o
```

```
$ touch x0.o x1.o x2.o x3.o x4.o x5.o x6.o x7.o x8.o
$ cc -xildon -z i_verbose \
     x0.o x1.o x2.o x3.o x4.o x5.o x6.o x7.o x8.o test2.o
ild: (Performing full relink) too many files changed
<訳>
完全再リンクを実行します  -- 変更されたファイルが多すぎます
```

ファイル x0.o から x8.o が変更されているので、touch コマンドを使用すると、9 つのオブジェクトファイルをすべてインクリメンタルリンクするより、完全再リンクを行う方が処理時間が短縮されます。

例 5: 新たな完全再リンク

例 4 は、これから行うリンクで完全再リンクを実行する例でしたが、その次に行うリンクで初期リンクを実行してしまう不具合があります。

次にこのプログラムをリンクしようとすると、以下のメッセージが表示されます。

以前のエラーを検出し完全再リンクを行う

```
$ cc -xildon -z i_verbose broken.o
ild: (Performing full relink) cannot do incremental
relink due to problems in the previous link
<訳>
完全再リンクを実行します
--  前回のリンクに問題があるため、インクリメンタルリンクを実行
    できません
```

新たに完全再リンクが行われます。

例 6: 新たな作業用ディレクトリ

現在の作業ディレクトリでの初期リンク。/tmp で行う場合と同じ。

インクリメンタルリンク。現在の作業ディレクトリは /tmp/junk になっている。

```
% cd /tmp
% cat y.c
    int main(){ return 0;}
% cc -c y.c
% rm -f a.out
% cc -xildon -z i_verbose y.o -o a.out


% mkdir junk
% mv y.o y.c a.out junk
% cd junk
% cc -xildon -z i_verbose y.o -o a.out


ild: (Performing full relink) current directory has
changed from ‘/tmp’ to ‘/tmp/junk’
%
<訳>
完全再リンクを実行します
--  現在の作業ディレクトリが /tmp から /tmp/junk に変更さ
    れました
```

ild オプション

この節では、リンカー制御オプション (直接コンパイラが受け取るオプションと、コンパイラによって ild に渡されるオプション) について説明します。

-a

静的モードのみで、実行可能オブジェクトファイルを作成します。未定義の参照にエラーを出します。静的モードのデフォルト動作です。

-B dynamic | static

ライブラリの取り込みを制御するオプションです。-Bdynamic は、動的モードでのみ有効です。この 2 つのオプションはコマンド行で何度でも指定でき、そのたびにモードを切り換えることができます。-Bstatic を指定している場合に共有オブジェクトを処理するには、-Bdynamic に切り替える必要があります。79 ページの「-lx」を参照してください。

-d y|n

-dy (デフォルト) を指定すると、動的リンク処理が実行されます。-dn を指定すると、静的リンク処理が実行されます。78 ページの「-B dynamic | static」の節を参照してください。

-e *epsym*

出力ファイルの入口点アドレスをシンボル *epsym* のアドレスに設定します。

-g

-g オプション (デバッグ情報の出力) を指定すると、コンパイラは ld の代わりに ild を実行します。次の条件が 1 つでも満たされている場合は -g オプションは使用できません。

- -G オプション (共有ライブラリの作成) が指定されている
- -xildoff オプションが指定されている

■ コマンド行内にソースファイル名が指定されている

-I <名前>

実行可能ファイルを作成するときは、プログラムヘッダーに書き込まれるインタープリタのパス名として <名前> を使用します。静的モードでのデフォルトはインタープリタなしです。動的モードでは、デフォルトは実行時リンカー /usr/lib/ld.so.1 の名前です。どちらの場合も、-I <名前> で無効にすることができます。exec システムコールは、a.out を読み込むときにこのインタープリタを読み込み、インタープリタに制御を渡します。a.out に直接渡すことはしません。

-i

LD_LIBRARY_PATH の設定を無視します。LD_LIBRARY_PATH の設定が有効なときに、実行中のリンク処理に干渉する実行時ライブラリの検索を変更する場合に使用します (これは LD_LIBRARY_PATH_64 の設定にも適用されます)。

-L<パス>

ライブラリ検索ディレクトリに <パス> を追加します。ライブラリは、最初にオプション -L で指定されたディレクトリで検索され、次に標準ディレクトリで検索されます。このオプションは、同じコマンド行上の -l オプションの前に指定した場合だけ有効です。環境変数 LD_LIBRARY_PATH と LD_LIBRARY_PATH_64 でも、ライブラリ検索パスを追加できます (84 ページの「LD_LIBRARY_PATH」を参照)。

-l*x*

ライブラリ lib*x*.so または lib*x*.a を検索します。これらはそれぞれ共有オブジェクトとアーカイブライブラリの一般的な名前です。動的モード、つまり -Bstatic オプションが指定されていないときは、ライブラリ検索パスに指定された各ディレクトリで、ファイル lib*x*.so またはファイル lib*x*.a を検索します。どちらかのファイルが含まれている最初のディレクトリで、検索は停止します。lib*x*.so と lib*x*.a という形式のファイルが両方ともある場合は、拡張子 .so が付いているファイルが使用されます。lib*x*.so がない場合は、lib*x*.a が使用されます。静的モードのとき、つまり -Bstatic オプションが指定されているときは、拡張子 .a が付いているファイルだけを使用します。ライブラリは、名前が検出された時に検索されるので、-l を指定するのは重要な意味があります。

-m

標準出力の入出力セクションのメモリーマップまたはリストを作成します。

-o <出力ファイル>

<出力ファイル> という名前の出力オブジェクトファイルを作成します。デフォルトのオブジェクトファイル名は a.out です。

-Q y|n

-Qy を指定すると、出力ファイルの .comment セクションに、出力ファイルの作成に使用したリンカーのバージョンを識別する ident 文字列が追加されます。この結果、複数回リンク処理を行なった場合は、ld -r を使用したときと同じように複数の ld idents が追加されます。これは cc コマンドのデフォルト処理と同じです。オプション -Qn を指定すると、バージョンを識別する ident 文字列は追加されません。

-R<パス>

実行時リンカーに渡されるライブラリ検索ディレクトリを、コロン (:) で区切られたリストで指定します。<パス> が存在しかつ空でない場合は、<パス> は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。このオプションを複数回に分けて使用すると、指定された検索ディレクトリは連結されてコロンで区切られます。

-s

シンボル情報を出力ファイルに出力しません。デバッグ情報および対応する再配置エントリが出力されなくなります。再配置可能ファイルまたは共有オブジェクトファイル以外の出力ファイルには、シンボルテーブルと文字列テーブルのセクションも出力されなくなります。

-t

複数回定義され、しかもサイズが異なるシンボルに関する警告を出力しません。

-u <シンボル名>

<シンボル名> をシンボルテーブルに未定義シンボルとして記入します。これは、アーカイブライブラリをまるごと読み込むのに便利です。シンボルテーブルははじめは空であり、最初のルーチンを読み込むために未解決の参照が必要だからです。コマンド行でのこのオプションを指定する位置は重要です。このオプションは、シンボルを定義するライブラリの前に置かなければなりません。

-V

使用している ild のバージョンに関するメッセージを出力します。

-xildoff

インクリメンタルリンカーを使用せず、強制的に ld を実行します。-g オプションを指定しない場合、または -G オプションを指定した場合は、このオプションがデフォルトです。デフォルトを受け入れたくない場合は -xildon を指定してください。

-xildon

強制的に、ild を使用してインクリメンタルリンカーを実行します。-g オプションを指定した場合は、このオプションがデフォルトです。デフォルトを受け入れたくない場合は -xildoff を指定してください。

-YP,<ディレクトリのリスト>

(cc のみ) ライブラリを検索するデフォルトディレクトリを変更します。オプション <ディレクトリのリスト> には、ディレクトリのパスをコロンで区切って指定します。

注 – -z <名前> という形は、ild で特殊なオプションを指定する場合に使用します。-z オプションの接頭辞として i_ を付けたものは、ild に固有のオプションとして識別されます。

-z allextract|defaultextract|weakextract

このオプションの後に続くアーカイブから、オブジェクトの抽出基準を変更します。デフォルトでは、未定義の参照を解決し、一時的な定義がデータ定義になるようにアーカイブメンバーが抽出されます。弱いシンボル参照のためには、抽出は行われません。-z allextract を指定すると、アーカイブからすべてのメンバーが抽出されます。-z weakextract を指定すると、弱い参照によってアーカイブ抽出が発生します。-z defaultextract は、それまでに使用されていた抽出オプションからデフォルトに戻ることを意味します。

-z defs

リンク終了時に未定義シンボルが残っている場合に、致命的エラーを出力します。実行可能ファイルの作成時はこれがデフォルトです。共有オブジェクトを作成し、シンボル参照がすべてオブジェクト内部で解決されるようにする場合に、このオプションを使用します。

-z i_dryrun

(ild のみ) リンクされるファイルのリストを出力して、処理を終了します。

-z i_full

(ild のみ) インクリメンタルモードで全体の再リンクを行います。

-z i_noincr

(ild のみ) ild をインクリメンタルでないモードで実行します (ユーザーによる使用はお勧めできません。テスト用にだけ使用してください)。

-z i_quiet

((ild のみ) ild 再リンクメッセージをすべて表示しません。

-z i_verbose

(ild のみ) 詳細情報モードを持つ ild 再リンクメッセージを拡張して、より詳細な情報を出力します。

-z nodefs

未定義シンボルがあってもエラーにしません。共有オブジェクトの作成時はこれがデフォルトです。実行可能ファイルの作成時には、未定義シンボルは参照されません。

コンパイラから ild に渡されるオプション

以下のオプションは ild によって認識されますが、コンパイラから ild に渡すときは次の形式を使用する必要があります。

-Wl,<引数>,<引数> (cc の場合)

-a

静的モードのみで、実行可能オブジェクトファイルを作成します。未定義の参照にエラーを出します。静的モードのデフォルト動作です。オプション -a は、オプション -r と同時には指定できません。

-e *epsym*

出力ファイルのエントリポイントアドレスをシンボル *epsym* のアドレスに設定します。

-I <名前>

実行可能ファイルを作成するときは、プログラムヘッダーに書き込まれるインタープリタのパス名として <名前> を使用します。静的モードでのデフォルトはインタープリタなしです。動的モードでは、デフォルトは実行時リンカー /usr/lib/ld.so.1 の名前です。どちらの場合も、-I <名前> で無効にすることができます。exec システムコールは、a.out を読み込むときにこのインタープリタを読み込み、インタープリタに制御を渡します。a.out に直接渡すことはしません。

-m

標準出力の入出力セクションのメモリーマップまたはリストを作成します。

-t

複数回定義され、しかもサイズが異なるシンボルに関する警告を出力しません。

-u <シンボル名>

<シンボル名> をシンボルテーブルに未定義シンボルとして記入します。これは、アーカイブライブラリをまるごと読み込むのに便利です。シンボルテーブルははじめは空であり、最初のルーチンを読み込むために未解決の参照が必要だからです。コマンド行でのこのオプションを指定する位置は重要です。このオプションは、シンボルを定義するライブラリの前に置かなければなりません。

環境変数

LD_LIBRARY_PATH
  -l オプションで指定されたライブラリを検索するディレクトリのリストです。複数のディレクトリはコロンで区切ります。通常は、1 つのセミコロンで区切られた 2 つのディレクトリのリストが指定されています。

```
  <ディレクトリのリスト1>; <ディレクトリのリスト2>
```

  以下のように、-L を任意の回数指定して ild を呼び出すとします。

```
  ild ...-L<パス1> ...  -L<パスn> ...
```

この場合、検索パスの順序は以下のようになります。

```
  <ディレクトリのリスト 1> <パス 1> ...  <パス n> <ディレクトリのリスト 2>
LIBPATH
```

ディレクトリのリストにセミコロンが入っていないときは、次のように解釈されます。

```
<ディレクトリのリスト 2>
```

LD_LIBRARY_PATH は、実行時リンカーにライブラリ検索ディレクトリを指定するのにも使用します。つまり、LD_LIBRARY_PATH が指定されていると、実行時リンカーは、実行時にプログラムとリンクされる共有オブジェクトを、LD_LIBRARY_PATH に指定されたディレクトリで検索してから、デフォルトのディレクトリで検索します。

注 – set-user-ID プログラムまたは set-group-ID プログラムを実行するときは、実行時リンカーは /usr/lib 内のライブラリと、実行可能ファイル内に指定された絶対パス名を検索します。この絶対パス名は、実行可能ファイルが作成されるときに指定された実行時パスです。相対パス名で指定されたライブラリ関係は暗黙的に無視されます。

LD_LIBRARY_PATH_64

Solaris 7 と Solaris 8 で使用されます。この環境変数は LD_LIBRARY_PATH に似ていますが、64 ビットの依存関係を検索する場合に LD_LIBRARY_PATH を無効にします。

Solaris 7 または Solaris 8 を SPARC プロセッサ上で実行し、32 ビットモードでリンクする場合、LD_LIBRARY_PATH_64 は無視されます。LD_LIBRARY_PATH しか定義しなかった場合は、32 ビットリンクと 64 ビットリンクの両方に LD_LIBRARY_PATH が使用されます。LD_LIBRARY_PATH と LD_LIBRARY_PATH_64 の両方を定義すると、32 ビットリンクは LD_LIBRARY_PATH を使用して行われ、64 ビットリンクは LD_LIBRARY_PATH_64 を使用して行われます。

LD_OPTIONS

ild のデフォルトのオプションです。LD_OPTIONS の値は、ild を起動するコマンドの直後に入力されたものとして解釈されます。

```
ild $LD_OPTIONS  ... <その他の引数> ...
```

LD_PRELOAD

実行時リンカーによって解釈される共有オブジェクトのリストです。指定された共有オブジェクトは、実行中のプログラムの後、プログラムが参照する他の共有オブジェクトの前にリンクされます。

注 – set-user-ID プログラムまたは set-group-ID プログラムを実行するときは、このオプションは暗黙的に無視されます。

LD_RUN_PATH
リンカーに実行時パスを指定するもうひとつの方法です (-R オプションを参照)。LD_RUN_PATH と -R オプションの両方を指定すると、-R で指定したパスが使用されます。

LD_DEBUG
(ild ではサポートされていません) 実行時リンカーによって標準エラーにデバッグ情報が出力されるように、トークンのリストを指定します。トークンとして help と指定すると、使用できるすべてのトークンが表示されます。

注 – LD_ で始まる環境変数名は、ld の将来の拡張用に予約されています。ILD_ で始まる環境変数名は、ild の将来の拡張用に予約されています。

---

ild で使用できない ld オプション

以下のオプションはコンパイラに指定することはできますが、現在 ild でサポートされていません。

-B symbolic

動的モードのみで使用します。定義が使用できる場合は、共有オブジェクトの作成時に、大域シンボルの参照をオブジェクト内の定義に結合します。通常、共有オブジェクト内の大域シンボルの参照は、定義を使用できても実行時まで結合されないので、実行可能ファイルまたは他の共有オブジェクト内の同じシンボルの定義によって、オブジェクト自体の定義が無効になることがあります。未定義シンボルがあると ld によって警告メッセージが出力されます (-z defs を指定すると警告メッセージは出力されません)。

-b

動的モードのみで、実行可能ファイルを作成するときに、共有オブジェクト内のシンボルを参照する再配置用の特殊処理を行わないようにします。-b オプションを指定しないと、リンカーは、共有オブジェクト内に定義された関数参照のために位置独立の特殊な再配置コードを作成し、共有オブジェクト内に定義されたデータオブジェクトが実行可能ファイルのメモリーイメージ内に実行時リンカーによってコピーされるようにします。-b オプションを指定すると、出力コードは効率的になりますが、共有性は低下します。

-G

動的モードのみで使用し、共有オブジェクトを作成します。未定義シンボルも使用できます。

-h <名前>

動的モードのみで使用します。共有オブジェクトの作成時に、共有オブジェクトの動的セクション内に <名前> を記録します。<名前> は、オブジェクトの UNIX ファイル名としてではなく、オブジェクトとリンクされる実行可能ファイル内に記録されます。そのため <名前> は、実行時リンカーが実行時に検索する共有オブジェクト名として使用されます。

-z muldefs

複数のシンボル定義を行うことができるようにします。デフォルトでは、再配置可能オブジェクトの間に複数のシンボル定義があると致命的エラーになります。このオプションを指定すると、複数のシンボル定義をエラーにせず、最初のシンボル定義が使用されます。

-z text

動的モードのみで使用します。書き込み禁止の割り当て可能セクションに対する再配置が残っている場合に、致命的エラーにします。

サポートされないその他のコマンド

また、以下のオプションは直接 ld に渡され、ild にはサポートされていません。

-D <トークン>,<トークン>, ...

各トークンで指定したように、デバッグ情報を標準エラーに出力します。トークンとして help を指定すると、使用できるすべてがトークンが表示されます。

-F <名前>

共有オブジェクトの作成時のみ使用します。共有オブジェクトのシンボルテーブルが、<名前> で指定した共有オブジェクトのシンボルテーブルの「フィルタ」として使用されるように指定します。

-M <マップファイル>

<マップファイル> を、ld に対する指示テキストファイルとして読み取ります。マップファイルについての詳細は、『リンカーとライブラリ』を参照してください。

-r

再配置可能なオブジェクトファイルを組み合わせ、1 つのオブジェクトファイルを作成します。ld は解釈されない参照があってもエラーとしません。このオプションは、動的モードでは使用できません。また、-a オプションと同時に使用することはできません。

ild で使用するファイル

- libx.a ライブラリ
- a.out 出力ファイル

第5章

lint ソースコード検査プログラム

この章では lint プログラムの使用方法について説明します。lint プログラムを使用すると実行時にコンパイルエラーや予期しない結果をもたらす可能性のあるコードが C コードにないか検査することができます。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、第 6 章を参照してください。

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、他の方法では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

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 <ファイル 1>.c <ファイル 2>.c
```

拡張 lint は、-Nlevel または -Ncheck オプションを使用して、以下のように呼び出します。

```
% lint -Nlevel=3 <ファイル 1>.c <ファイル 2>.c
```

lint は、2 つのパスでコードの検査をします。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 <ファイル 1>.c <ファイル 2>.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 <ファイル 1>.c と <ファイル 2>.c 間の定義および使用の矛盾に関する情報を収集します。そして、それを <ファイル 1>.ln および <ファイル 2>.ln と名付けられた中間ファイルとして作成します。

```
% ls
<ファイル 1>.c
<ファイル 1>.ln
<ファイル 2>.c
<ファイル 2>.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様に動作します。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。

.ln ファイルが lint で生成されると、第 2 のパスが実行されます。

```
% lint <ファイル 1>.ln <ファイル 2>.ln
```

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。次のコマンド行は、<ファイル 3>.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 <ファイル 1>.ln <ファイル 2>.ln <ファイル 3>.c
```

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 オプションを使用することができます。11 ページの「インクルードファイル」の節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 -I<ディレクトリ 1> -I<ディレクトリ 2> <ファイル 1>.c <ファイル 2>.c
```

このコマンドは lint に下記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、特定の処理を実行し、特定の条件について報告するためのオプションが数多くあります。

lint のオプション

lint プログラムは静的アナライザです。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注釈) が役に立ちます。以下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 のオプションを以下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後、106 ページの表 5-7 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。

-#

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。

-###

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。

-a

一定のメッセージを抑制します。106 ページの表 5-7 を参照してください。

-b

一定のメッセージを抑制します。106 ページの表 5-7 を参照してください。

-C<ファイル名>

指定されたファイル名を持った .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。<ファイル名> は絶対パス名でもかまいません。

-c

コマンド行で指定された .c ファイルごとに、lint の第 2 の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。

-dirout=<ディレクトリ>

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。

-err=warn

-err=warn は -errwarn=%all のマクロです。98 ページの「-errwarn=t」の節を参照してください。

-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 5-1　-errchk の引数

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。これがデフォルトです。
[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。

表 5-1　-errchk の引数

値	意味
[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。
[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。

-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 5-2　-errfmt の値

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。これがデフォルトです。

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。

-errhdr=*h*

-Ncheck で検査対象になるヘッダーファイルを限定します。*h* には、<ディレクトリ>、no%<ディレクトリ>、%all、%none、%user の 1 つまたは複数をコンマで区切って指定します。

表 5-3　-errhdr の値

値	意味
<ディレクトリ>	<ディレクトリ> で使用されているヘッダーファイルを検査します。
no%<ディレクトリ>	<ディレクトリ> で使用されているヘッダーファイルを検査しません。
%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。これがデフォルトです。
%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。以下に例を示します。

```
% 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。

-erroff=<タグ>(,<タグ>)

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、<タグ>、no%<タグ>、all、%none の 1 つまたは複数をコンマで区切って指定します。

表 5-4　-erroff の値

値	意味
<タグ>	<タグ> 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%<タグ>	<タグ> 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。これがデフォルトです。

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

```
% 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。

-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。

-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、<タグ>、no%<タグ>、%all、%none のいずれかをコンマで区切って指定します。指定する順序は重要です。たとえば、「%all,no%<タグ>」と指定した場合、<タグ> 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 の値を以下に示します。

表 5-5　-errwarn の値

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。<タグ> 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。 <タグ> 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、<タグ> 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%<タグ> を使用すると、警告メッセージによってはこの動作が発生しないように指定できます。
%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 と指定したことと同じになります。

-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。

-fd

古い形式の関数定義または宣言について報告します。

-flagsrc=<ファイル>

<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。

-h

一定のメッセージを抑制します。106 ページの表 5-7 を参照してください。

-I<ディレクトリ>

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。

-k

/*LINTED[<メッセージ>]*/ 指令または注釈 NOTE(LINTED(<メッセージ>)) の動作を変更します。通常 lint は、上述のような指令の後にコードが続く場合、警告メッセージを抑制します。-k オプションを指定した場合は、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。

-L<ディレクトリ>

-l と共に使用し、<ディレクトリ> の lint ライブラリを検索します。

-l*x*

lint ライブラリ llib-l*x*.ln にアクセスします。

-m

一定のメッセージを抑制します。106 ページの表 5-7 を参照してください。

-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 5-6　-Ncheck の値

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 <ファイル 1>.c と <ファイル 1>.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。これがデフォルトです。
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。

-Nlevel=*n*

問題報告の解析レベルを指定します。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、1、2、3、4 のいずれかの数値です。デフォルトは -Nlevel=2 です。-Nlrevel は -Nlevel=4 と同義になります。

-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。

-Nlevel=2

デフォルトです。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。

-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、他の集合 (定数伝播の形式) を作成するときの基準になります。その後、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。

-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。

-n

デフォルトの lint 標準 C ライブラリとの互換性検査を抑制します。

-o*x*

llib-l*x*.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 lib-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、後で 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。

-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。

-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使用不可能にします。

-s

複合メッセージを単一メッセージに変換します。

-u

一定のメッセージを抑制します。106 ページの表 5-7 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。

-V

製品名とリリース時期を標準エラーに書き込みます。

-v

一定のメッセージを抑制します。106 ページの表 5-7 を参照してください。

-W<ファイル>

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。

-x

一定のメッセージを抑制します。106 ページの表 5-7 を参照してください。

-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。

-Xalias_level [=*l*]

I には、any、basic、weak、layout、strict、std、または 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 A-7 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、第 6 章を参照してください。

-Xarch=v9

__sparcv9 マクロを事前に定義して、v9 版の lint ライブラリを検索します。

-Xexplicitpar=*a*

(SPARC) lint に #pragma MP 指令を認識するよう指定します。*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 -Xexplicitpar=no です。-Xexplicitpar だけを指定すると、-Xexplicitpar=yes を指定するのと同じことになります。

-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。

-Xtemp=<ディレクトリ>

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。

-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。

-Xtransition=*a*

K&R C と Sun 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 のいずれかを指定します。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。

-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。

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他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 (120 ページの「lint ライブラリ」の節を参照) 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 オプションを指定して 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。

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 オプションの後に 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。

表 5-7 に lint のメッセージを抑制するオプションを示します。

表 5-7　lint のオプションメッセージを抑制される

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=<タグ>	<タグ> で指定した 1 つまたは複数の lint のメッセージ
-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

lint メッセージの形式

lint プログラムに一定のオプションを指定すると、エラーが発生した行位置を示すポインタを伴った詳細なソースファイル行を表示することができます。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 は以下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)
       |             ^  2 行目, Test1.c
       |
       |         cpv(argv[0], argc, strlen(argv[0]));
       |                          ^  10 行目, Test1.c
ポインタ/整数の組み合わせは不適切です: 2 番目の引数
       |
       |static void cpv(char *s, char* v, unsigned n)
       |                                  ^  2 行目, Test1.c
       |
       |        cpv(argv[0], argc, strlen(argv[0]));
       |                          ^  10 行目, Test1.c
       |
       |         *v++ = *s++;
       |          ^  5 行目, Test1.c
疑わしい方法で作成されたポインタを使用しています
    v, 定義された場所: Test1.c(2):: Test1.c(5)
    呼び出しスタック:
           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、上記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);
       |        ^  11 行目, Test2.c
       |
       |#define B(c,d) c+AA(d)
       |                 ^  2 行目, Test2.c
       |
       |#define AA(b) AR[b+l]
       |                   ^  1 行目, Test2.c
未定義のシンボル: l
       |
       | return B(y,z);
       |        ^  11 行目, Test2.c
       |
       |#define B(c,d) c+AA(d)
       |                 ^  2 行目, Test2.c
       |
       |#define AA(b) AR[b+l]
       |                   ^  1 行目, Test2.c
変数が設定以前に使用されている可能性があります: l
lint: Test2.c 中のエラー; 出力ファイルは作成されません
lint: 2 回目のパスは実行しません - Test2.c 中のエラー
```

lint の指令

事前定義された値

次の事前定義はすべてのモードで有効です。

__sun
__unix
__lint
__SUNPRO_C=0x540
__‘uname -s‘_‘uname -r‘ (例: __SunOS_5_7)
__RESTRICST (-Xa および -Xt モードのみ)
__sparc *(SPARC)*
__i386 *(x86)*

__BUILTIN_VA_ARG_INCR
__SVR4
__sparcv9 (-Xarch=v9)

次の事前定義は -Xc モード以外で有効です。

- sun
- unix
- sparc *(SPARC)*
- i386 *(x86)*
- lint

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。

以下のようにファイル 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。Sun 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、Sun C のソースコードを検査する lint は、/usr/lib/note と /OPT/SUNWspro/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}: <ディレクトリ>
```

表 5-8 に、lint 指令と動作を示します。

表 5-8 lint 指令と動作

指令	動作
NOTE(ALIGNMENT(<関数>,*n*)) *n*=1, 2, 4, 8, 16, 32, 64, 128	lint に、関数結果を *n* バイト境界で整列させます。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。
NOTE(ARGSUSED(*n*)) /*ARGSUSED*n*/	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。 以下のメッセージが抑制されます。 指令の後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、必ず *n* の指定が必要です。 • 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)。 以下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。 以下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 "||" • 指令の後に来る言語構造が対象となります。

表 5-8　lint 指令と動作 (続き)

指令	動作
NOTE(EMPTY) /*EMPTY*/	if 文に続く空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。 文が帰結していません: if (if の条件式とセミコロンの間に挿入された場合) 以下のメッセージが抑制されます。 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)
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。 以下のメッセージが抑制されます。 指令の後に来る 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

表 5-8　lint 指令と動作 (続き)

指令	動作
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、指令の後に来るコード行の下記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 <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。
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。 以下のメッセージが抑制されます。 到達できない文です 指令の後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 指令の後の 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

表 5-8　lint 指令と動作 (続き)

指令	動作
NOTE(PRINTFLIKE(*n*)) NOTE(PRINTFLIKE(<関数>,*n*))/*PRINTFLIKE*n*/	指令の後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、下記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
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 ファイルに書き込みます。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(<関数>,*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 5-8　lint 指令と動作 (続き)

指令	動作
NOTE(VARARGS(*n*)) NOTE(VARARGS(<関数>,*n*)) /*VARARGS*n**/	指令の後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、末尾には省略記号 (...) を使用することをお勧めします。 この指令の直後で定義されている関数に関しては、以下のメッセージが抑制されます。 • 可変数の引数で関数が呼び出されています *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。

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。

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。

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後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。

a. そのファイルで宣言された関数と変数が他の場所で定義または使用された。

b. そのファイルで使用された関数と変数が他の場所で定義されていた。

c. そのファイルで定義された関数と変数が他の場所で使用された。

a の場合を抑制するには -x オプションを、b と c 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。

移植性の検査

lint は、デフォルトでいくつかの移植不能コードを知らせます。lint が -p または -Xc を指定して呼び出されると、さらに多くのケースが診断されます。-Xc により、lint は ISO C 規格に一致しない言語構造を検査します。-p と -Xc のもとで発行されるメッセージに関しては、120 ページの「lint ライブラリ」の節を参照してください。次に例を示します。

- いくつかの C 言語処理系では、signed や unsigned と明示的に宣言されない文字変数は、一般に –128 から 127 の範囲の符号付きデータとして扱われます。別の処理系では、これらは一般に 0 から 255 の範囲の負でないデータとして扱われます。そこで EOF が値 -1 を持つ以下のテストは、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。

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

-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。しかし上記の例では、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ある点に注意してください。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最も一般的な例です。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データとして扱うマシンでは、int x:3 に対し許容される値は 0 から 7 の範囲で、符号付きデータとして扱うマシンでは、–4 から 3 の範囲です。int 型を宣言した 3 ビットフィールドは後者のマシンでは値 4 を持つことはできません。-p を指定して呼び出された lint は、unsigned int または signed int 以外のすべてのビットフィールド型にフラグを立てます。unsigned int と signed int 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、lint は、デフォルトでこの様な代入すべてを知らせます。

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

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、他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、120 ページの「lint ライブラリ」の節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。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。

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

-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、他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 * を使用すれば他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他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 & が論理演算子 && の代わりに誤って使用されると、この診断を引き起こします。

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

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。以下に例を示します。

- 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常に失敗します。

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

一方、次の 2 つのテストは同等です。

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

```
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、副作用が期待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。以下に例を示します。

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

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)) ...
```

-h を指定して lint を呼び出すと、この診断は抑制されます。

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-l*x*.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。その他の 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。すなわち次のコマンド行は、<ファイル 1>.c と <ファイル 2>.c の関数と変数の使用法について、lint ライブラリ llib-l*x*.ln との互換性を検査するよう lint に指示します。

```
% lint -lx <ファイル 1>.c <ファイル 2>.c
```

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 オプションと共にそのファイルに対して lint を実行してください。

```
% lint -ox <ファイル 1>.c <ファイル 2>.c
```

上記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 の -o と cc の -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。

```
% lint -y -ox <ファイル 1>.c <ファイル 2>.c
```

上記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 -L<ディレクトリ> -lx <ファイル 1>.c <ファイル 2>.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 /opt/SUNWspro/prod/src/lintlib; make
```

ここで、/opt/SUNWspro/prod 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。

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時に出力されるコメントです。

/* LINTED [<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、111 ページの表 5-8 を参照してください。

第6章

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新しい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、262 ページの「-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、103 ページの「-Xalias_level[=l]」を参照してください。

型に基づいた解析の概要

-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最も役立ちます。

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最も役立ちます。

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参照の後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	262 ページの「`-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。

#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他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)
```

#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;
```

#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他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)
```

#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)
```

#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;
```

#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他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他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)
```

#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他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)
```

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。lint -Xalias_level コマンドの詳細については、103 ページの「-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

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、警告メッセージが生成されます。

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

struct foo *f;
int *p;

void main()
{
    f = (struct foo *)p; /* 構造体ポインタへのスカラーポインタのキャスト
警告 */
}
```

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; /* 構造体ポインタへの void ポインタのキャスト
警告 */
}
```

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 foo.b のアドレスが integer 型のポインタとしてキャストされた後、p 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、警告メッセージが生成されます。

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

struct foo *f;
int *p;

void main()
{
    p = (int *)&f->b; /* 構造体ポインタへの構造体フィールドのキャスト警告
*/
}
```

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) でない限り、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; /* 明示的な別名設定に必要な警告*/
}
```

メモリ参照の制限の例

この 節では、ソースファイルで使用される可能性の高いコードを例に挙げて説明します。それぞれの例の後に、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。

次のコードを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。

コード例 6-1

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

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

int *ip;
short *sp;
```

コード例 6-1 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

コード例 6-1 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。

コード例 6-1 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。

コード例 6-1 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。

コード例 6-1 が -xalias_level=strict オプションでコンパイルされた場合、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。

コード例 6-1 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。

コード例 6-1 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。

次の例のソースコードを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。

コード例 6-2

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

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

コード例 6-2 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。

コード例 6-2 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。

コード例 6-2 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。

コード例 6-2 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。

コード例 6-2 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。

コード例 6-2 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。

コード例 6-2 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。

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、262 ページの表 A-7 を参照してください。

コード例 6-3

```
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);
```

コード例 6-3 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アクセスは同じメモリ位置に接触します。

- fp->f2 および bp->b2 は、同じメモリ位置にアクセスします。
- fp->f3 および bp->b3 は、同じメモリ位置にアクセスします。
- fp->f4 および bp->b4 は、同じ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った後で、コンパイルを実行してください。

```
struct foo {
        int f1;
        struct bar fb;   /* 変更行 */
#define f2 fb.b2         /* 変更行 */
#define f3 fb.b3         /* 変更行 */
#define f4 fb.b4         /* 変更行 */
        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アクセスは同じメモリ位置に接触します。

- fp->f2 および bp->b2
- fp->f3 および bp->b3
- fp->f4 および bp->b4

前出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。

次の例のソースコードについて考えてみましょう。

コード例 6-4

```
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;
```

コード例 6-4 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、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限り、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、それらのメモリ参照が別名設定できないと仮定します。

コード例 6-4 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、次の別名情報だけを仮定します。

- 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。

コード例 6-4 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、次の別名情報だけを仮定します。

- 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。

コード例 6-4 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、次の別名情報だけを仮定します。

- 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。

次の例のソースコードを考えてみましょう。

コード例 6-5

```
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。

- コード例 6-5 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。
- コード例 6-5 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。
- コード例 6-5 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。
- コード例 6-5 が-xalias_level=std オプションでコンパイルされる場合、相互に別名設定できるフィールドはありません。

次の例のソースコードを考えてみましょう。

コード例 6-6

```
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。

- コード例 6-6 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。
- コード例 6-6 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。
- コード例 6-6 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。タグが削除された後も、2 つの構造体の型が異なるからです。
- コード例 6-6 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ないからです。

次の例のソースコードを考えてみましょう。

コード例 6-7

```
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7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 9899:1999 ISO/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。

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。デフォルトのモードは `-Xa` 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。

`-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。

-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。

-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。

-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。

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。

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、148 ページの「併用に関する考慮点」の節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なりません。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、もっともありがちな C プログラミングの誤りを防ぐことができます。

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、上記の 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。

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なりません。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なりません。

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。上記 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 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 (154 ページの「拡張: 符号なし保存と値の保持」の節を参照)。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 を使用するべきです。
- -Xa と -Xc では int を使用するべきです。

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、関数を入力した後でも、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 の使用に気をつけてください。次の例を考えてください。

```
#define status 23
    void my_exit(int status);* 通常、スコープはプロトタイプで始まり、*/
                                /* プロトタイプで終わる */
```

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;
      {
      /* .  .  .  */
      }
      /* .  .  .  */
```

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なりませんでした。

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。このパラメータの後には、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後には、(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上記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>
```

その後で 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 /* 注:セミコロンなし */
#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なりません。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、Sun 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);
    /* 固定引数を抽出する */
    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]);
```

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。「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。

"QUIET CHANGE ". A program that depends on unsigned preserving arithmetic conversions will behave differently, probably without complaint. This is considered to be the most serious change made by the Committee to a widespread current practice.

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。

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他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。

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。

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;
}
```

上記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。

```
6 行目 : 警告 : ISO C では "<" の意味が変わります。明示的なキャストを使用してください。
```

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。

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。

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。

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。9 ページの「整数定数」を参照してください。

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。

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。

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。

1. 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 7-1　3 文字シーケンス

3 文字シーケンス	変換後の文字
??=	#
??-	~
??(	[

表 7-1　3 文字シーケンス

3 文字シーケンス	変換後の文字
??)	]
??!	|
??<	{
??>	}
??/	\
??'	^

ISO C コンパイラは上記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。

```
/* コメント *??/
/* コメントの続き ? */
```

??/ 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。

```
/* コメント */* コメントの続き ? */
```

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。

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。

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の次の行には、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。

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。次の例を見てください。

```
#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。

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 (ǂ 印) 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!"ǂ
str(x y)
```

したがって、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、上記マクロは文字列リテラル「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)
```

上記コードは、2 つの文字列リテラル「x y」と「!」を生成し、連結した後、同じ「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') ǂ
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。

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 ǂ
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、上記の呼び出しは、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、上記の呼び出しの内 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なりません。## 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。

上記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、他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。

const と volatile

キーワード const は C++ の機能の 1 つで、ISO C に取り入れられました。ISO C 委員会が類似キーワード volatile を導入したとき、「型修飾子」カテゴリが作成されました。

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、右辺値が左辺値になるとき) 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、lvalue 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。

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。この 2 つの宣言の効果は上記宣言と同じです。

```
int const five = 5;
```

```
const five = 5;
```

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。上記宣言において、const が * の後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;
```

上記の宣言は、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;
```

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、上記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。

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、同じプログラムの他の並行呼び出しが、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。

volatile は文字通りの解釈を意味する

これまでの例ではすべて const を使用してきましたが、これは const が概念的に簡単であるためです。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。

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ます。

```
flag = 1;
while (flag);
```

これは、flag が volatile 修飾型を持つ間は有効です。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。それ以外の場合、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は上記ループを、完全に flag の値を無視する本当の無限ループに変更できます。

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。

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。

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。

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。

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なりません (完全な中国語には

65,000 以上もの表意文字があります)。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 (その逆もあります)。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。

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。

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。表記文字 \ 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'};
```

上記の例では、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。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。

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。

標準ヘッダー

標準ヘッダーは次のとおりです。

表 7-2　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、上記ヘッダーだけです。

これらヘッダーの一部の内容については、他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 `#define` 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、その前後で他のヘッダーを `#include` 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 `#include` 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 `#include` 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。

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、下線で始まる他のすべての名前が (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。

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 7-3 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]

表 7-3　拡張用の予約名

ファイル	予約名のパターン
signal.h	(SIG|SIG_)[A-Z].*
stdlib.h	str[a-z].*
string.h	(str|mem|wcs)[a-z].*

上記リストにおいて、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。

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、その後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 #include 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。

国際化

166 ページの「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。

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。他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、他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。

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なりません。

```
#include <locale.h>
/*...*/
setlocale(LC_ALL, "");
```

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

上記の任意のマクロを 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_< カテゴリ名 >, NULL);
if (setlocale(LC_< カテゴリ名 >, "new") != 0)
{
        /* use temporarily changed locale */
    (void)setlocale(LC_< カテゴリ名 >, 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。

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上記の関数は、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。

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()	照合のために文字列を変換する

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、他にも 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 リリース 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。

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、上記 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();
```

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。上記式の副作用とは、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、値を他の値や演算子と結合させる方法です。上記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なりません。上記の式の場合、`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なりません。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。

K&R C の再配置ライセンス

上記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 の再配置の権利が上記式に適用されます。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() };
```

上記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、上記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、上記 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、上記 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();
```

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なりません。上記の式は、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。

括弧

ISO C では、不充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。

上記の式は、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。

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限り、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。

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。

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、その他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他の 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。

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。

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。他のすべての宣言は、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なりません。`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。

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限り、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。

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 では他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なりません。

```
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、すべて上記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。

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 typedefs 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 {
    List *next; . . .
};
```

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。プログラムの他の部分は、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。

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。

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。

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) も一致しなければなりません。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。

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なりません。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。複合型については後で説明します。

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` を持つ型であることを示します。他のすべての型は、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、未修飾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。

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なりません。ポインタの修飾子は * の後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;
```

上記 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。

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なりません。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なりません。つまり、不完全な配列型 (179 ページの「不完全な型」の節を参照) は、他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。

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なりません。

- 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なりません。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なりません。
- 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 (...) であってはなりません。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なりません。
- 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 (...) であってはなりません。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なりません。
- 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後、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後に、互換性を持たなければなりません。

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なりません。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。

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、複合型の最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すべての使用可能な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8章

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方が便利です。このため、この付録では、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、この付録では、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。

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 8-1 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。標準的な C の変換規則の働きも、データ型のサイズの変更の影響を受けます。意図したこと正しく示すには、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。

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。

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/type.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。

<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。

<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
- 制限値
- 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、int_16t、int32_t、int64_t などの符号付き整数型と、uint8_t、uint_16t、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_least16_t、int_least32_t、int_least64_t、uint_least8_t、uint_least16_t、uint_least32_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整数を使用することは問題ありません。配列インデックスにロング整数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

uninptr_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、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。

unitptr_t 型は、unsigned 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 unsigned long と pointer 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、他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 unsigned 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 という文字列を付加します。たとえば、INT64_C(l) 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、194 ページの「LP64 データ型モデルへの変換」の節で説明している定数型を指定する際に大変役立ちます。これらのマクロは、194 ページの「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。

制限値

<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。190 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最も適しています。

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。また、C コンパイラに -v オプションを使用すると、より厳密な意味検査も行われます。-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 7 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。

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。

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 `-Xarch=v9` 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(`<任意のメッセージ>`))` 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、リンクや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(`<任意のメッセージ>`))` 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`(1) のマニュアルページを参照してください。

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。

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、unintptr_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);
%
警告: ポインタの変換でビットが失われます
```

修正版は、次のようになります。

```
char *p;
p = (char *) ((uintptr_t)p & PAGEOFFSET);
```

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;

%
警告: 64 ビット整数を 32 ビット整数に代入します
```

符号の拡張

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、符号の拡張はよく問題になります。符号の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。

符号の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;  /* ここで符号拡張する ! */
  printf("addr 0x%lx\n", addr);

  addr = (unsigned int)(a.base << 13); /* 符号拡張しない ! */
  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、unsigned 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。

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);

%
警告: ポインタの変換でビットが失われます
```

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;
```

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、long 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、long 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。long 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 */
```

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;
```

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;  /* RHS が整数式のため j は 0 になる */
```

修正版は次のようになります。

```
int i = 32;
long j = 1L << i;
```

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);
}

%
警告: ポインタ/整数の組み合わせは不適切です: 演算子 "="
警告: 32 ビット整数からポインタにキャストしています
int を返すように暗黙的に宣告されます
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);
}
```

sizeof() は unsigned long

LP64 データ型モデルでは、sizeof() の有効な型は unsigned long です。sizeof() は、ときには 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);


%
警告: 代入によって 62 ビット定数が 8 ビットに切り捨てられました
警告: 初期設定子が適合していないか範囲を超えています: 0x190
```

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。

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);


%
警告: 関数への引数の型と初期とが整合していません
  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);

%
警告: 64 ビット整数から 32 ビット整数にキャストしています
警告: 64 ビット整数から 32 ビット整数にキャストしています
警告: 64 ビット整数から 32 ビット整数にキャストしています
```

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);
```

その他の注意事項

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。

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。

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。

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。

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 #ifdef を使用する必要があります。

呼び出し規則の変更

構造体を値によって渡し、SPARC V9 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 (構造体がそう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。

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。

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、多数の基本派生型とともに `_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。
- `-errchk=longptr64`、`signext` と `-D_sparcv9` フラグを使用して `lint` を実行し、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限り、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9章

cscope: 対話的な C プログラムの検査

cscope は、C、lex、または yacc の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 言語の識別子と予約語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。

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、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できます。

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、指定したコードが存在するファイル名、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。リストを表示した後は、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。さらに他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「基本的な使用方法」で説明します。

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なりません。この場合、次の手順で実行します。

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 (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<端末名> 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:

```
$ TERM=<端末名>; export TERM
```

C シェル:

```
% setenv TERM <端末名>
```

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 を使用したくない場合は、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 シェルの場合は以下のように入力します。

```
$ EDITOR=emacs; export EDITOR
```

C シェルの場合は以下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なりません。詳細については、226 ページの「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したい (編集は使用しない) 場合は、VIEWER 環境変数を 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。220 ページの「ビューパス (Viewpath)」を参照してください。

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 <ファイル 1>.c <ファイル 2>.c <ファイル 3>.h
```

cscope の他の起動方法については、217 ページの「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。その後 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。

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、カーソルは下に移動し (画面の一番下まで移動すると、先頭に戻ります)、^p (Ctrl キーと p キー) を押すと上に移動します。また、上矢印と下矢印キーも使用できます。以下の単一キーコマンドを使用すれば、メニュー操作とその他の作業が行えます。

表 9-1　cscope メニュー操作コマンド (1 / 2)

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

表 9-1　cscope メニュー操作コマンド (2 / 2)

^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上記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 番目の項目は他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、221 ページの「cscope の使用例」の節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:
```

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後に使用可能なコマンドを以下に示します。

表 9-2　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
+	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上記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)));
}

/* メモリーの割り当て失敗を検査する */

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」の後に 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()
{
    /* 表示される参照行数の最大を計算する */
    lastdispline = FLDLINE - 4;
    mdisprefs = lastdispline - REFLINE + 1;
    if (mdisprefs > 9) {
    mdisprefs = 9;
    }
    /* 表示される行の配列を割り当てる */
    displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L  /* 参照ページを表示する */

void
display()
{
    char file[PATHLEN + 1];   /* ファイル名 */
    char function[PATLEN + 1];/* 関数名 */
    char linenum[NUMLEN + 1]; /* 行番号 */
    int screenline; /* 画面行番号 */
    int width;      /* ソース行表示幅 */
    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。

ステップ 4：コードの編集

ウィンドウ端末上では、任意のサイズで複数のウィンドウを開くことができます。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、もっと分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() 関数を以下のように編集します。

cscope 関数: 問題箇所の修正

```
void
dispinit()
{
        /* 表示される参照行数の最大を計算する */
    lastdispline = FLDLINE - 4;
    mdisprefs = lastdispline - REFLINE + 1;
    if (mdisprefs > 9) {
    mdisprefs = 9;
    }
        /* 表示される行の配列を割り当てる */
    displine = (int *) mymalloc(mdisprefs * sizeof(int));
}
^L/* 参照ページを表示する */

void
display()
{
    char file[PATHLEN + 1];   /* ファイル名 */
    char function[PATLEN + 1];/* 関数名 */
    char linenum[NUMLEN + 1]; /* 行番号 */
    int screenline; /* 画面行番号 */
    int width;      /* ソース行表示幅 */
    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分かりやすいエラーメッセージを出力した後に終了するようになります。

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、およびソースファイルのシンボル相互参照表を作成します。すなわち、次の 2 つのコマンドは等価です。

```
% cscope
```

```
% 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 <ファイル 1>.c <ファイル 2>.c <ファイル 3>.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 オプションの後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 <ディレクトリ 1>,<ディレクトリ 2>,<ディレクトリ 3>
```

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 <ファイル>
```

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、以下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 > <ファイル>
% cscope -i <ファイル>
```

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は、指定されたファイル以外については無視されるので注意してください。

-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。11 ページの「インクルードファイル」の節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、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 の後の *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、以下のコマンドによって検索結果のリストに表示されるパス名は、common/<ファイル 1>.c や common/<ファイル 2>.c のようになります。

```
% cscope -p2
```

表示したい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後で cscope を終了することができます。このとき、作業メニューは表示されません。パイプを使用して、cscope -b の出力を batch(1) コマンドの入力につなげると、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限り、次のように指定するだけで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。

```
% cscope
```

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、ソースファイルの変更が検査され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。

他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。

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。各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、以下のように入力します。

```
$ VPATH=/usr/you:/fs1/ofc; export VPATH
```

C シェルの場合は、以下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、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 はユーザーの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) でなければなりません。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なりません。

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、他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。

cscope の使用例

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 メニューの他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。

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

```
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; /* 百分率にする */

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):
```

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。また、`read.c` 内の `0100` と `err.c` 内の `100.0` (リストの 4、 5 行目) は、変更すべきでないこともわかります。変更する行を指定するには、以下の単一キーコマンドを使用します。

表 9-3　変更行選択コマンド

`1-9`	変更行をマークまたはマーク解除する
`*`	表示されている行をすべて変更対象としてマークまたはマーク解除する
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。代わりに各行の行番号の後に > (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; /* 百分率にする */

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; /* 百分率にする */

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。

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 を使用すると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 function 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。

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。

エディタのコマンド行構文

cscope はデフォルトで vi エディタを使用しています。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、208 ページの「ステップ 1：環境設定」で述べた通りです。ただし cscope は、使用するエディタのコマンド行構文が vi と同様に次のような形式であるとみなします。

```
% editor +<行番号> <ファイル>
```

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。

Bourne シェルの場合は以下のように入力します。

```
$ EDITOR=myedit; export EDITOR
```

C シェルの場合は以下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。

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 (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"
```

このメッセージが表示されたら、208 ページの「ステップ 1：環境設定」で説明した手順に従って、TERM 環境変数を設定してエクスポートしてください。

付録 A

C コンパイラオプション

この章では、C コンパイラのオプションについて説明します。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、353 ページの「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 C プログラムを ISO C に移植する場合は、260 ページの「`-X[c|a|t|s]`」を特に参照してください。こ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、第 7 章の移行に関する説明も参照してください。

オプションの構文

`cc` コマンドの構文を以下に示します。

```
% cc [<オプション>] <ファイル名> [<ライブラリ>] . . .
```

- <オプション> は、239 ページの「`cc` オプション」で説明している各種のオプション (複数指定可) です。
- <ファイル名> は、実行可能プログラムの作成に使用するファイル名 (複数指定可) です。

`cc` は <ファイル名> 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。

cc は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、cc 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。

- <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。

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、-YP,<ディレクトリ> を参照してください。<ディレクトリ> には、複数のパスをコロンで区切って指定します。cc のデフォルトのライブラリ検索は、次の順序で行われます。

```
/opt/SUNWspro/prod/lib

/usr/ccs/lib

/usr/lib
```

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) のマニュアルページも参照してください。

オプションの一覧

この節では、 簡単に参照できるように、コンパイラオプションを機能ごとに分けて示します。オプションの詳細については、この後の節を参照してください。次の表は、cc コンパイラオプションを機能ごとに要約したものです。一部のフラグは複数の目的で使用されているため、複数箇所に記載されています。

表 A-1　機能別コンパイラオプション

ライセンス	オプションフラグ
ライセンスシステムについての情報を返す	`-xlicinfo`
最適化とパフォーマンス	**オプションフラグ**
速度が最も速くなるコンパイルオプションの組み合わせを選択する	`-fast`
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する	`-p`
80386 プロセッサ用に最適化する	`-x386`
80486プロセッサ用に最適化する	`-x486`
コンパイラが型ベースの別名の分析と最適化を実行す。	`-xalias_level`
標準ライブラリ関数を呼び出すコードの最適化機能が向上する	`-xbuiltin`
複数ソースファイルに渡る最適化とインライン化を有効にする	`-xcrossfile`
ループの繰り返し内部でのデータ依存の解析およびループ再構成を実行する	`-xdepend`
アナライザを使用した実行可能ファイルのパフォーマンス解析ができるように準備する	`-xF`
指定された関数だけをインライン化する	`-xinline`
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する	`-xipo`
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する	`-xlibmil`

表 A-1　機能別コンパイラオプション (続き)

ライセンス	オプションフラグ
指定された Sun 提供のパフォーマンスライブラリにリンクする	`-xlic_lib=sunperf`
`pragma opt` のレベルを指定されたレベルに限定する	`-xmaxopt`
数学ライブラリルーチンをインライン化しない	-xnolibmil
オブジェクトコードを最適化する	`-x`
Pentium™ プロセッサ用に最適化する	`-xpentium`
先読み命令を有効にする	`-xprefetch`
先読み命令の自動挿入の優先度を制御する	-xprefecth_level
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する	`-xprofile`
ポインタ値の関数引数を制限付きポインタとして扱う	`-xrestrict`
メモリーに関するトラップが発生しないことを前提とする	`-xsafe`
コードサイズを増やすループの最適化や並列化を行わない	`-xspace`
ループを *n* 回展開するようオプティマイザに指示する	`-xunroll`
データ境界整列	**オプションフラグ**
複数文字から成る定数の文字を指定されたバイト順序で配置して、整定数を生成する	`-xchar_byte_order`
想定する最大の境界整列と、境界整列が不正な場合の動作を指定する	`-xmemalign`
数値と浮動小数点	**オプションフラグ**
浮動小数点演算ハードウェアの非標準の初期化を行う	`-fnonstd`
SPARC 非標準の浮動小数点モードに切り替える	`-fns`
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する	`-fprecision`
プログラム初期化中に、実行時に確立される IEEE 754 丸めモードを設定する	`-fround`

表 A-1　機能別コンパイラオプション (続き)

ライセンス	オプションフラグ
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する	-fsimple
float 式を倍精度ではなく単精度で評価する	-fsingle
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する	-fstore
起動時に有効になる IEEE 754 トラップモードを設定する	-ftrap
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ない	-nofstore
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 形式にする	-xlibmieee
接尾辞のない浮動小数点定数を単精度で表す	-xsfpconst
ベクトルライブラリ関数を自動的に呼び出す	-xvector
並列化	**オプションフラグ**
-D_REENTRANT-lthread に展開されるマクロオプション	-mt
複数プロセッサの自動並列化を有効にする	-xautopar
スタックオーバーフロー用に実行時検査を追加する	-xcheck
反復相互のデータ依存関係についてループを分析し、ループの再構築を実行します。	-xdepend
#pragma MP 指令の指定にもとづいて並列化コードを生成する	-xexplicitpar
並列化されているループとされていないループを示す	-xloopinfo
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。	-xopenmp
ループを、コンパイラで自動的に並列化するとともに、プログラマの指定によって明示的に並列化する	-xparallel
自動並列化中の縮約の認識を有効にする	-xreduction

表 A-1　機能別コンパイラオプション (続き)

ライセンス	オプションフラグ
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。	-xrestrict
ループが正しく並列化指定されていない場合に、#pragra MP 指令が指定されているループについて警告を出す	-xvpara
lock_lint 用にプログラムデータベースを作成するが、コンパイルは行わない	-Zll
ソースコード	**オプションフラグ**
<名前> を述語として <トークン> と関連付ける。#assert 前処理指令を実行するのと同様	-A
C プリプロセッサがコメントを削除しないようにする。ただし前処理指令の行にあるコメントは削除される	-C
#define 前処理指令が行うように、<名前> を <トークン> に関連付ける。	-D
ソースファイルの前処理だけを行い、結果を stdout に出力する	-E
K&R 形式の関数の宣言や定義を報告する	-fd
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する	-H
ディレクトリのリストに <ディレクトリ> を追加する。このディレクトリは相対ファイル名で指定されるインクルードファイルを検索する時のディレクトリである	-I
ソースファイルのプリプロセッサ処理のみを行う	-P
初期定義されているプリプロセッサシンボル <名前> をすべて削除する	-U
C++ 形式のコメントを受け入れる	-xCC
サポートされている C99 機能に対するコンパイラの認識状況を制御します。	-xc99
文字が符号なしと定義されるシステムから移行をヘルプする	-xchar

表 A-1　機能別コンパイラオプション (続き)

ライセンス	オプションフラグ
C コンパイラは、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ができます。	`-xcsi`
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する。その際、メー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する	`-xM`
`-xM` と同様に依存関係を収集するが、`/usr/include` ファイルは除く	`-xM1`
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する	`-xP`
`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する	`-xpg`
ソースブラウザ用のシンボルテーブル情報を生成する	`-xsb`
ソースブラウザ用のデータベースを作成する	`-xsbfast`
三重字シーケンスの認識状況を判定します。	`-xtrigraphs`
コンパイル済みコード	**オプションフラグ**
`ld`(1) によるリンクを行わず、ソースファイルごとに `.o` ファイルを作成する	`-c`
出力ファイルに名前を付ける	`-o`
アセンブリソースファイルを作成するが、アセンブルは行わない	`-S`
コンパイルモード	**オプションフラグ**
冗長モードでコンパイラを動作させる。呼び出された各構成要素が表示される	`-#`
呼び出された各構成要素が表示されるが、実行はされない	`-###`
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する	`-keeptmp`
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する	`-V`

表 A-1　機能別コンパイラオプション (続き)

ライセンス	オプションフラグ
引数を C コンパイルシステムの構成要素に渡します。	-W
ANSI/ISO C に準拠する度合いを指定する	-X
オンラインヘルプ情報を表示する	-xhelp
cc が使用する一時ファイルの <ディレクトリ> を設定する	-xtemp
コンパイルの各構成要素が使用した実行時間と資源を報告する	-xtime
C 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。	-Y
構成要素検索時のデフォルトディレクトリを変更します。	-YA
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。	-YI
ライブラリ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
起動オブジェクトファイルのデフォルトディレクトリを変更します。	-YS
診断	**オプションフラグ**
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る	-errfmt
コンパイラからの警告メッセージを出力しない	-erroff
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する	-errshort
各警告メッセージのメッセージタグを表示する	-errtags
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する	-errwarn
より厳しい意味検査を行い、lint に似たその他の検査を可能にする	-v
コンパイラからの警告メッセージを出力しない	-w

表 A-1　機能別コンパイラオプション (続き)

ライセンス	オプションフラグ
ソースファイル上で構文および意味検査のみを行う。オブジェクトコードや実行可能コードは生成しない	-xe
K&R C と Sun ANSI/ISO C との間の相違に対して警告を出す	-xtransition
#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す	-xvpara
デバッグ	**オプションフラグ**
スタックオーバーフロー用に実行時検査を追加する	-xcheck
デバッガ用に追加のシンボルテーブル情報を作成する	-g
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する	-s
dbx のための自動読み取りを無効にする	-xs
リンクとライブラリ	**オプションフラグ**
ライブラリのリンクを静的と動的のどちらにするかを指定する	-B
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する	-d
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する	-G
共有動的ライブラリに <名前> をつける。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る	-h
LD_LIBRARY_PATH の設定を無視するオプションをリンカーへ渡す	-i
リンカーがライブラリを検索するリストに <ディレクトリ> を付け加える	-L
ld がオブジェクトライブラリ lib<名前>.so、または lib<名前>.a をリンクの対象とする	-l

表 A-1　機能別コンパイラオプション (続き)

ライセンス	オプションフラグ
オブジェクトファイルの .common セクションから重複している文字列を削除する	-mc
オブジェクトファイルの .common セクションからすべての文字列を削除し、<文字列> を挿入する	-mr
出力ファイルに識別情報を入れるかどうかを設定する	-Q
実行時リンカーが使用するライブラリ検索ディレクトリを指定する	-R
データセグメントをテキストセグメントにマージする	-xMerge
コードアドレス空間を指定する	-xcode
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する	-xstrconst
インクリメンタルリンカーを使用せず、強制的に ld を起動する	-xildoff
強制的に、インクリメンタルリンカー (ild) をインクリメンタルモードで起動する	-xildon
共有ライブラリが Java[tm] プログラミング言語で記述されたコードとインタフェースをもつようにするため、インタフェース情報をオブジェクトファイルと後続の共有ライブラリに組み込む	- xnativeconnect
デフォルトのライブラリをリンクしない	-xnolib
数学ライブラリのルーチンをインライン化しない	-xnolibmil
対象プラットフォーム	**オプションフラグ**
命令セットアーキテクチャを指定する	-xarch
オプティマイザ用のキャッシュ特性を定義する	-xcache
-xarch、-xchip、および -xcache の値を指定する	-xcg

表 A-1　機能別コンパイラオプション (続き)

ライセンス	オプションフラグ
オプティマイザ用のプロセッサを定義する	-xchip
生成されたコードでのレジスタの使用方法を指定する	-xregs
最適化と命令セットの対象となるシステムを指定する	-xtarget

cc オプション

この項では cc オプションを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。オプションおよび引数で使用している表記方法については、xxx ページの「書体と記号について」 を参照し てください。

-#

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。

-###

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を表示します。

-A<名前> [(<トークン>)]

<名前> を述語として <トークン> と関連付けます。#assert 前処理指令を実行するのと同様です。

事前表明 (preassertion): -Xc モードの場合、以下の事前表明は有効になりません。

- system(unix)
- machine(sparc) *(SPARC)*
- machine(i386) *(x86)*
- cpu(sparc) *(SPARC)*
- cpu(i386) *(x86)*

-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、-lx 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。

-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。

-c

ld(1) を使用するリンクを行わずに、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。各 .i または .c 入力ファイルのオブジェクトコードを作成する場合、コンパイラは常に現在の作業ディレクトリ内にオブジェクト (.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。

-D<名前> [=<トークン>]

#define 前処理指令が行うように、 <名前> を <トークン> に関連付けます。「=<トークン>」を省略した場合はトークンとして 1 が使用されます。

事前定義: 次の事前定義は -Xc モードの場合は無効です。

- sun
- unix
- sparc *(SPARC)*
- i386 *(x86)*

次の事前定義はあらゆるモードにおいて有効です。

- __sparcv9 (-xarch=v9,v9a,v9b)
- __sun
- __unix
- __SUNPRO_C=0x540
- __`uname -s`_`uname -r` (例: __SunOS_5_7)
- __sparc *(SPARC)*
- __i386 *(x86)*
- __BUILTIN_VA_ARG_INCR
- __SVR4
- __RESTRICT 次の事前定義および -xa および -xt モードにおいて有効です。

コンパイラでは、__PRAGMA_REDEFINE_EXTNAME のようなオブジェクト形式のマクロを定義しておくことにより、プラグマを認識させることができます。

-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。

このオプションと引数は ld(1) に渡されます。

注 – Solaris 7 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、このオプションと -Xarch=v9 を組み合わせると、致命的なエラーになります。

-dalign

-dalign は、-xmemalign=8s を指定することと同じです。291 ページの「-xmemalign=ab」を参照してください。

-E

ソースファイルの前処理だけを行い、結果を stdout に出力します。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 モードの場合を除きます)。プリプロセッサの行番号付け情報も含みます (257 ページの「-P」を参照してください)。

-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 A-2　The -errfmt の値

値	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、コンパイラでは -errfmt=no%error が指定されます。-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。

-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。

t には、以下の 1 つまたは複数の項目をコンマで区切って指定します。

<タグ>、no%<タグ>、%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%<タグ>」と指定すると、<タグ> 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 A-3　-erroff の引数

値	意味
<タグ>	<タグ> 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 オプションを使用すればメッセージのタグを表示することができます。
no%<タグ>	<タグ> 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。18 ページの「#pragma error_messages(on | off | default, <タグ>... <タグ>)」を参照してください。

-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、以下のいずれかを指定します。

表 A-4　The -errshort の値

値	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。

-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他のコンポーネントから出力されるメッセージにはえら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 を指定します。デフォルトは -errtags=no です。-errtags だけを指定した場合は、-errtags=yes と同義になります。

-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 を使用します。

t には、*tag*、no%*tag*、%all、%none の組み合わせをコンマで区切って指定します。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 A-5 -errwarn の引数

<タグ>	<タグ> 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。<タグ> が発行されない場合は無効です。
no%<タグ>	<タグ> 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。<タグ> 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、<タグ> 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 1 つでも発行されると cc は致命的ステータスを返して終了します。%all に続いて no%<タグ> 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 cc が致命的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 とだけ指定することは、-errwarn=%all と指定することと同じです。

-fast

ベンチマークアプリケーションの基本的な最適化オプションを選択します。これらの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。-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。

-fast は、実行ファイルの実行時の性能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オプションです。-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実行することをお勧めします。

```
cc -fast -xtarget=ultra ...
```

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 の後に -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 が設定されなかったり、あるいは 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 A-6　-fast の拡張値

オプション	SPARC	x86
-dalign	—	○
-fns	○	○
-fsimple=2	○	—
-fsingle	○	○
-ftrap=%none	○	○
-nofstore	—	○

表 A-6　-fast の拡張値

-xalias_level=basic	○	—
-xarch	○	○
-xbuiltin=%all	○	○
-xdepend	○	○
-xlibmil	○	○
-xmemalign=8s	○	—
-xO5	○	○
-xprefetch=auto,explicit	○	—

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 の後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。

前回のリリースでは、-fast のマクロオプションには -fnonstd が含まれていましたが、今回のリリースでは、-fns に置き換えられています。

-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、math.h が、<math.h> でプロトタイプ化されたいくつかの数学ルーチンに対し、次のようなパフォーマンス関連プラグマを表明します。

- #pragma does_not_read_global_data
- #pragma does_not_write_global_data
- #pragma no_side_effect

matherr(3M) マニュアルページで説明されているように、自分のコードが例外的な事例で errno の戻り値に依存する場合、-fast オプションの後に -U__MATHERR_ERRNO_DONTCARE マクロを発行することにより、__MATHERR_ERRNO_DONTCARE マクロをオフにする必要があります。

このオプションを使用すれば、大部分のプログラムのパフォーマンスを向上させることができます。

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。

マクロオプションの展開を表示する方法については、239 ページの「-#」および 239 ページの「-###」を参照してください。

-fd

K&R 形式の関数の宣言や定義を報告します。

-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。

-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。-fnonstd オプションを使用すると、浮動小数点のオーバーフロー、ゼロによる除算、不正演算例外に対し、ハードウェアによるトラップが可能になります。これらの例外は SIGFPE シグナルに変換されます。プログラムが SIGFPE ハンドラを持っていないときは、メモリーダンプを行なってプログラムを終了します。

デフォルトでは、IEEE 754 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 (詳細については 14 ページの「非標準浮動小数点」を参照)。

(SPARC) -fns と -ftrap=common を指定することと同等です。

-fns[={no,yes}]

(SPARC) 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、SPARC 標準浮動小数点モードです。

オプションの =yes または =no を使用すると、-fast のように、-fns を含む他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。このフラグを設定すると、プログラムが実行を開始するときに、非標準の浮動小数点モードが有効になります。デフォルトでは、非標準の浮動小数点モードは自動的に有効になりません。

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。段階的アンダーフローと非正規数をハードウェアでサポートしていない SPARC システムでこのオプションを使用すると、プログラムによってはパフォーマンスが飛躍的に上がることがあります。

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。

-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。

注 – 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。

-fround=*r*

プログラム初期化中の実行時に確立される IEEE 754 小数点丸めモードを設定します。

r は、nearest、tozero、negative、positive のうちのいずれかです。

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、メインプログラムのコンパイル時に使用する場合だけ有効です。

-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。

n を指定する場合は、0、1、2 のいずれかでなければなりません。デフォルトは次のとおりです。

- -fsimple[=*n*] を指定しない場合は、-fsimple=0 が使用されます。
- =*n* のない -fsimple だけを指定すると、-fsimple=1 が使用されます。

-fsimple=0

前提事項の単純化を行えないようにします。厳密に IEEE 754 に準拠します。

-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 に厳密には準拠しませんが、大部分のプログラムの数値結果は変化しません。

-fsimple=1 を指定すると、オプティマイザは以下の事項を前提とします。

- IEEE 754 のデフォルトの丸めとトラップモードは、プロセスの初期化後は変化しない。
- 浮動小数点の数値例外以外には、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 をもつ演算は、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。
- 演算はゼロの符号に依存しない。

-fsimple=1 を指定した場合は、オプティマイザは必ず四捨五入や数値例外に応じた処理を行います。特に浮動小数点演算は、実行時に定数化される四捨五入モードでは異なる結果を生じる演算と、置き換えることはできません。-fast マクロフラグを使用すると、-fsimple=1 に設定されます。

`-fsimple=2`

高度な浮動小数点最適化を行うことができ、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、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` 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。

-fsingle

(`-Xt` モードまたは `-Xs` モードの場合に限り) `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。

`-fsimple=2` を指定しても、本来浮動小数点例外を生成しないプログラムには、オプティマイザは浮動小数点例外を導入しません。

-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。

-ftrap=*t*

起動時に有効になる IEEE 754 トラップモードを設定します。

t には、以下の 1 つまたは複数の項目をコンマで区切って指定します。

`%all`、`%none`、`common`、`[no%]invalid`、`[no%]overflow`、`[no%]underflow`、`[no%]division`、`[no%]inexact`

デフォルトは `-ftrap=%none` です。

このオプションは、プログラム初期化時に設定される IEEE 754 トラップモードを設定します。項目は左から右への順に評価されます。common を指定した場合は、定義により、無効、0 除算、オーバーフローの各例外のトラップモードがオンになります。

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。

意味は、次の項目を除き、ieee_flags サブルーチンの場合と同じです。

- %all はすべてのトラップモードをオンにします。
- %none はデフォルトで、すべてのトラップモードをオフにします。
- no% を先頭につけると、そのトラップモードだけをオフにします。

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。

-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令します。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

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追加シンボルテーブル情報を生成します。

このオプションは、インクリメンタルリンカーを呼び出します。284 ページの「-xildoff」および 285 ページの「-xildon」を参照してください。-G または -xildoff オプションを使用していない場合、あるいはコマンド行にソースファイルの名前を指定していない場合は、ld の代わりに ild を呼び出します。

-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限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になります。

-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限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、Forte Developer 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解析機能は -g オプション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表示するには、-g オプションでコンパイルする必要があります。詳細については、analyzer(1) 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』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。er_src(1) コマンドを使用すると、これらのメッセージを表示できます。これらのメッセージは、ソースコードに挿入されています。

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。

-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。以下で、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
```

-h<名前>

共有動的ライブラリに <名前> をつけます。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 の後の <名前> は、-o オプションの後に指定するファイル名と同じです。-h と名前の間の空白は任意です。

リンカーは指定された <名前> 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 <名前> 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。共有ライブラリのイントリンシック名がない場合、代わりにリンカーは共有ライブラリファイルのパス名をコピーします。

-I [- | <ディレクトリ>]

-I<ディレクトリ>は、相対ファイル名 (つまり、/ (スラッシュ) から始まらないファイル名) の付いた #include ファイルを検索するディレクトリのリストに <ディレクトリ> を追加します。-I の値は累積します。インクルードファイルの検索順序については、11 ページの「インクルードファイル」を参照してください。

-I- は、インクルードファイル検索時に使用されるアルゴリズムに対し、制御を強化します。-I- の値は累積しません。この節では、最初にデフォルトの検索アルゴリズムについて説明し、次にそれらのアルゴリズムに対する -I- の効果について説明します。

コンパイラの検索パターンの詳細については、11 ページの「インクルードファイル」を参照してください。

-i

(SPARC) オプションをリンカーへ渡して、LD_LIBRARY_PATH または LP_LIBRARY_PATH_64 の設定を無視します。（SPARC）

-KPIC

(SPARC) -KPIC は、-xcode=pic32 と同義です。279 ページの「-xcode=v」も参照してください。

(x86) -KPIC は -Kpic と同じです。

-Kpic

(SPARC) `-Kpic` は、`-xcode=pic13` と同義です。279 ページの「`-xcode=v`」を参照してください。

(x86) 共用ライブラリ (小型モデル) で使用するために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。

-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。

-L<ディレクトリ>

`ld`(1) がライブラリを検索するディレクトリのリストに <ディレクトリ> 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。

-l<名前>

`ld` がオブジェクトライブラリ `lib` <名前> `.so`、または `lib` <名前> `.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> 引数の後に指定してください。

-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。

-misalign

(SPARC) `-misalign` は、`-xmemalign=1i` と同義です。291 ページの「`-xmemalign=ab`」を参照してください。

-misalign2

(SPARC) -misalign2 は、-xmemalign=2i と同義です。291 ページの「-xmemalign=ab」を参照してください。

-mr[,<文字列>]

-mr は、.comment セクションからすべての文字列を削除します。このフラグを使用すると mcs -d -a が呼び出されます。

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 <文字列> を挿入します。<文字列> に空白が含まれている場合は二重引用符で囲みます。<文字列> 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 -da<文字列> として mcs に渡されます。

-mt

-D_REENTRANT-lthread に展開されるマクロオプションです。ユーザー固有のマルチスレッドコーディングを行なっている場合は、コンパイルとリンクのときに必ず -mt オプションを使用してください。マルチプロセッサシステム上でこのオプションを使用すると、生成された実行可能ファイルの実行速度が早くなります。シングルプロセッサシステム上では、実行速度は通常よりも遅くなります。

-native

このオプションは、-xtarget=native と同義です。

-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。251 ページの「-fstore」も参照してください。

-O

-xO2 と同じです。

-o <出力ファイル>

出力ファイルに (デフォルトの a.out の代わりに) <出力ファイル> 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、<出力ファイル> には <ソースファイル> と同じ名前は使用できません。このオプションと引数は、ld(1) に渡されます。

-P

ソースファイルのプリプロセッサ処理のみを行い、.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 (242 ページの「-E」を参照してください)。

-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。プログラムを実行すると、実行時の記録機構が起動されます。プログラムが正常終了すると、この記録機構によって mon.out ファイルが作成されます。

-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。-Qy がデフォルトです。-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。

-qp

-p と同じです。

-R<ディレクトリ>[:<ディレクトリ>]

実行時リンカーが使用するライブラリ検索ディレクトリを渡します。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。

-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。

-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て ld(1) に渡します。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。

-U<名前>

プリプロセッサシンボル <名前> 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 <名前> 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 < 名前 > が指定された場合、オプションの配置順に関係なく、名前の定義が削除されます。次の例で、-U は __sun の定義を削除します。

```
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、241 ページの「-D<名前>[=<トークン>]」を参照してください。

-V

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。

-V

より厳しい意味検査および他の lint に似た検査を行います。以下は支障なくコンパイルと実行ができるコードです。

```
#include <stdio.h>
main(void)
{
     printf("Hello World.\n");
}
```

-v を使用すると、コンパイルは行われますが以下の警告が表示されます。

"hello.c", 5 行目: 警告: 関数中に return 文がありません: 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 で上記の例を実行すると確認することができます。

-W*c*,<引数>

指定されたコンパイラ構成要素 *c* に、<引数> を渡します。各引数はコンマで区切ります。すべての -W 引数は、通常のコマンド行の引数の後に渡されます。コンマの前にバックスラッシュ (\) 文字を置いてエスケープすることにより、コンマを引数の一部にできます。*c* は以下のいずれかです。すべての -W 引数は、通常のコマンド行引数の後に渡されます。

たとえば、-Wa,-o,<オブジェクトファイル> は、-o と <オブジェクトファイル> を (この順番で) アセンブラに渡します。また、-Wl,-I,<名前> 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。

その他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。

要素のリストについては、表 1-1 を参照してください。*c* には、以下のいずれかを指定します。

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[1]

h	中間コード翻訳 (ir2hf)(*x86*)
i	手続き間の分析 (ube_ipa)(*x86*)
l	リンクエディタ (ld)
m	mcs
o	手続き間オプティマイザ
p	プリプロセッサ (cpp)
u	C コードジェネレータ (ube)(*x86*)
0	コンパイラ (acomp) (ssbd, *SPARC*)
2	オプティマイザ (iropt) *(SPARC)*

1. この章に示されている cc オプションは -wd を使用して C コンパイラに渡すことはできません。

-w

コンパイラの警告メッセージを抑制します。

このオプションは error_messages プラグマを無効にします。

-X[c|a|t|s]

-X (大文字の X) オプションは、ISO C 規格への準拠の程度を指定します。-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、付録 D 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、付録 F 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 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` 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 = transition) 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の変更は行いません。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、K&R C に準拠した解釈を行います。`-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` 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。

+-x386

(x86) 80386 プロセッサ用に最適化します。

-x486

(x86) 80486 プロセッサ用に最適化します。

-xa

このオプションは、旧形式になりました。代わりに `-xprofile=tcov` を使用してください。

-xalias_level[=*l*]

(SPARC) 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 A-7 別名明確化のレベル

用語	意味
any	コンパイラは、すべてのメモリ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、他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他のあらゆる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フロート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 * 型のポイントが int * 型のポインタで参照される同一メモリ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。

表 A-7　別名明確化のレベル

用語	意味
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なりません。 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他の型に関連するメモリ参照を別名設定すると仮定します。
layout	-xalias_level=layout オプションを使用すると、コンパイラは、メモリ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 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他の型に関連するメモリ参照を別名設定できると仮定します。

表 A-7　別名明確化のレベル

用語	意味
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、タグを削除した後も同一にならない型に関連するメモリ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なりません。 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他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他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限り、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。

-xarch=*isa*

命令セットアーキテクチャ (Instruction Set Architecture、ISA) を指定します。

-xarch のキーワード、*isa* に指定できるアーキテクチャを次の表 2-4 に示します。

表 A-8　-xarch ISA のキーワード

プラットフォーム	有効な -xarch キーワード
SPARC	generic、native、v7、v8a、v8、v8plus、v8plusa、v8plusb、v9、v9a、v9b
x86	generic、386、pentium_pro

-xarch は単独で使用できますが、本来は -xtarget オプションの展開の一部です。特定の -xtarget オプションで設定されている -xarch の値を上書きするために使用することもできます。次に例を示します。

```
% cc -xtarget=ultra2 -xarch=v8plusb ...
```

この例では、-xtarget=ultra2 によって設定されている -xarch=v8 が -xarch=v8plusb によって上書きされます。

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を選択すると、そのアーキテクチャ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。不適切なアーキテクチャを選択すると、バイナリプログラムがその対象プラットフォーム上で実行できなくなることがあります。

SPARC のみ

次の表は、指定された -xarch オプションでコンパイルされた後さまざまな SPARC プロセッサで実行される実行可能ファイルのパフォーマンスを示しています。この表は、特定の対象マシン上の実行可能ファイルに最も適した -xarch オプションを調べるために利用してください。初めにマシンの範囲を決め、続いて複数のバイナリを管理する手間と、より新しいマシンから最大限のパフォーマンスを引き出す効果を比較してみるとよいでしょう。

表 A-9 -xarch の組み合わせ

		SPARC マシンの命令セット					
		V7	V8a	V8	V9 (Sun 以外のプロセッサ	V9 (Sun のプロセッサ)	V9b
-xarch コンパイルオプション	v7	N	S	S	S	S	S
	v8a	PD	N	S	S	S	S
	v8	PD	PD	N	S	S	S
	v8plus	NE	NE	NE	N	S	S
	v8plusa	NE	NE	NE	**	N	S
	v8plusb	NE	NE	NE	**	NE	N
	v9	NE	NE	NE	N	S	S
	v9a	NE	NE	NE	**	N	S
	v9b	NE	NE	NE	**	NE	N

** 注: この命令セットでコンパイルされる実行可能ファイルは、Sun 以外の V9 プロセッサチップ上で仕様どおり機能するか、あるいはまったく機能しません。実行可能ファイルがその対象マシンで動作するかどうかについては、ハードウェアベンダーに問い合わせてください。

- N は、「仕様どおりのパフォーマンス」を意味します。プロセッサの命令セットを十分に利用してプログラムの実行が行われます。
- S は、「十分なパフォーマンス」を意味します。プログラムは実行されますが、提供されているプロセッサ命令の一部を利用できない場合があります。
- PD は、「パフォーマンスの低下」を意味します。プログラムは実行されますが、使用されている命令によってはパフォーマンスの低下が発生する場合があります (低下の程度はさまざま)。このパフォーマンス低下は、プロセッサが実装していない命令がカーネルでエミュレートされる場合に起きます。

- NE は、「実行不可能」を意味します。カーネルがプロセッサが実装していない命令をエミュレートしないため、プログラムは実行されません。

`v8plus` または `v8plusa` 命令セットを使用して実行可能ファイルをコンパイルしようと考えている場合は、代わりに `v9` または `v9a` によるコンパイルを考慮してください。`v8plus` と `v8plusa` オプションは、64 ビットプログラム対応の Solaris 7 がリリースされる以前に、プログラムで SPARC V9 と UltraSPARC 機能の一部が利用できるように提供されたものです。v8plus または v8plusa オプションでコンパイルされたプログラムは、SPARC V8 以前のマシンには移植できません。これらのプログラムは、`v9` または `v9a` でそれぞれコンパイルし直すと、SPARC V9 と UltraSPARC の全機能を十分利用できるようになります。`v8plus` と `v8plusa` の制限については、ホワイトペーパー『The V8+ Technical Specification』(パーツ番号 802-7447。購入先から入手可) で説明されています。

- SPARC 命令セットアーキテクチャ V7、V8、および V8a はすべてバイナリ互換性があります。

- `v8plus` と `v8plusa` でそれぞれコンパイルされたオブジェクトバイナリファイル (`.o`) は、一緒にリンクおよび実行できます。ただし、SPARC V8plusa 互換プラットフォーム上で実行する場合に限られます。

- `v8plus`、`v8plusa`、および `v8plusb` でそれぞれコンパイルされたオブジェクトバイナリファイル (`.o`) は、一緒にリンクおよび実行できます。ただし、SPARC V8plusb 互換プラットフォーム上で実行する場合に限られます。

- `-xarch` の値、`v9`、`v9a`、および `v9b` は、UltraSPARC 64 ビット対応 Solaris 環境にしか使用できません。

- `v9` と `v9a` でそれぞれコンパイルされたオブジェクトバイナリファイル (.o) は、一緒にリンクおよび実行できます。ただし、SPARC V9a 互換プラットフォーム上で実行する場合に限られます。
- `v9`、`v9a`、および `v9b` でそれぞれコンパイルされたオブジェクトバイナリファイル (`.o`) は、一緒にリンクおよび実行できます。ただし、SPARC V9b 互換プラットフォーム上で実行する場合に限られます。

どの値を使用しても、生成された実行可能ファイルはアーキテクチャが古くなるほど実行速度が遅くなります。また、これらの命令セットアークテクチャの多くで 4 倍精度 (`REAL*16` と `long double`) の浮動小数点命令を使用できますが、コンパイラは生成するコード内ではこれらの命令を使用しません。

-xarch のキーワード、isa に指定できるアーキテクチャを次の表 2-4 に示します。

表 A-10 SPARC プラットフォームの -xarch 値

isa の値	意味
generic	ほとんど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。 これはデフォルトです。このオプションは、どのプロセッサでも大きく性能を落とさず、また、ほとんどのプロセッサで良好なパフォーマンスを得られるような最良の命令セットを使用します。「最良な命令セット」の内容は、新しいリリースごとに調整される可能性があります。
native	現在のシステムで良好な性能を得られるようにコンパイルします。 これは -fast オプションのデフォルトです。現在コンパイラを実行しているシステムのプロセッサに最も適した設定を選択します。
v7	SPARC-V7 ISA 用にコンパイルします。 V7 ISA 上で良好なパフォーマンスを得るためのコードを生成します。これは、V8 ISA 上で最良なパフォーマンスを得るための最良の命令セットと同じですが、整数の mul と div 命令、および fsmuld 命令は含まれていません。 例: SPARCstation 1、SPARCstation 2
v8a	V8a 版の SPARC-V8 ISA 用にコンパイルします。 定義上、V8a は V8 ISA を意味します。ただし、fsmuld 命令は含まれていません。 V8a ISA 上で良好なパフォーマンスを得るためのコードを生成します。 例: microSPARC I チップアーキテクチャに基づくすべてのシステム
v8	SPARC-V8 ISA 用にコンパイルします。 V8 アーキテクチャ上で良好なパフォーマンスを得るためのコードを生成します。 例: SPARCstation 10

表 A-10 SPARC プラットフォームの -xarch 値 (続き)

isa の値	意味
v8plus	V8plus 版の SPARC-V9 ISA 用にコンパイルします。 定義上、V8plus は V9 ISA を意味します。ただし、V8plus ISA 仕様で定義されている 32 ビットサブセットに限定されます。さらに、VIS (Visual Instruction Set) と実装に固有な ISA 拡張機能は含まれていません。 • V8plus ISA 上で良好なパフォーマンスを得るためのコードを生成します。 • 生成されるオブジェクトコードは SPARC-V8 + ELF32 形式であり、Solaris UltraSPARC 環境でのみ実行できます。つまり、V7 または V8 のプロセッサ上では実行できません。 例: UltraSPARC チップアーキテクチャに基づくすべてのシステム
v8plusa	V8plusa 版の SPARC-V9 ISA 用にコンパイルします。 定義上、V8plusa は V8plus アーキテクチャ + VIS (Visual Instruction Set) バージョン 1.0 + UltraSPARC 拡張機能を意味します。 • UltraSPARC アーキテクチャ上で良好なパフォーマンスを得るためのコードを生成します。ただし、V8plus 仕様で定義されている 32 ビットサブセットに限定されます。 • 生成されるオブジェクトコードは SPARC-V8 + ELF32 形式であり、Solaris UltraSPARC 環境でのみ実行できます。つまり、V7 または V8 のプロセッサ上では実行できません。 例: UltraSPARC チップアーキテクチャに基づくすべてのシステム
v8plusb	UltraSPARC-III 拡張機能を持つ、V8plusb 版の SPARC-V8 plus ISA 用にコンパイルします。 UltraSPARC アーキテクチャ + VIS (Visual Instruction Set) バージョン 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。 • 生成されるオブジェクトコードは SPARC-V8 + ELF32 形式です。Solaris UltraSPARC-III 環境でのみ実行できます。 • UltraSPARC-III アーキテクチャ上で良好なパフォーマンスを得るための最良のコードを使用します。

表 A-10 SPARC プラットフォームの -xarch 値 (続き)

isa の値	意味
v9	SPARC-V9 ISA 用にコンパイルします。 V9 SPARC アーキテクチャ上で良好なパフォーマンスを得るためのコードを生成します。 • 生成される .o オブジェクトファイルは ELF64 形式です。このファイルは同じ形式の SPARC-V9 オブジェクトファイルとしかリンクできません。 • 生成される実行可能ファイルは、64 ビット対応の Solaris オペレーティング環境が動作する、64 ビットカーネルを持つ UltraSPARC プロセッサ上でしか実行できません。 • -xarch=v9 は、64 ビット対応の Solaris オペレーティング環境でコンパイルする場合にのみ使用できます。
v9a	UltraSPARC 拡張を持つ SPARC-V9 ISA 用にコンパイルします。 SPARC-V9 ISA に VIS (Visual Instruction Set) と UltraSPARC プロセッサに固有の拡張機能を追加します。V9 SPARC アーキテクチャ上で良好なパフォーマンスを得るためのコードを生成します。 • 生成される .o オブジェクトファイルは ELF64 形式です。このファイルは同じ形式の SPARC-V9 オブジェクトファイルとしかリンクできません。 • 生成される実行可能ファイルは、64 ビット対応の Solaris オペレーティング環境が動作する、64 ビットカーネルを持つ UltraSPARC プロセッサ上でしか実行できません。 • -xarch=v9a は、64 ビット対応の Solaris オペレーティング環境でコンパイルする場合にのみ使用できます。

表 A-10 SPARC プラットフォームの -xarch 値 (続き)

isa の値	意味
v9b	UltraSPARC-III 拡張機能を持つ SPARC-V9 ISA 用にコンパイルします。V9a 版の SPARC-V9 ISA に UltraSPARC-III 拡張と VIS バージョン 2.0 を追加します。Solaris UltraSPARC-III 環境で良好なパフォーマンスを得るためのコードを生成します。 • 生成される .o オブジェクトファイルは SPARC-V9 ELF64 形式です。このファイルは同じ形式の SPARC-V9 オブジェクトファイルとしかリンクできません。 • 生成される実行可能ファイルは、64 ビット対応の Solaris オペレーティング環境が動作する、64 ビットカーネルを持つ UltraSPARC-III プロセッサ上でしか実行できません。 • -xarch=v9b は、64 ビット対応の Solaris オペレーティング環境でコンパイルする場合にのみ使用できます。

x86 のみ

表 A-11 x86 の -xarch 値

isa の値	意味
generic	命令セットを x86 アーキテクチャに限定します。386 オプションと同義です。
native	現在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を実行します。この値は、-fast オプションのデフォルトです。コンパイラは、コンパイルが行われる現在のシステムプロセッサについて適切な設定を選択します。
386	命令セットを Intel 386/486 アーキテクチャに限定します。
pentium_pro	命令セットを pentium_pro アーキテクチャに限定します。

-xautopar

(SPARC) 複数プロセッサ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。

使用できるプロセッサの数を調べるには、psrinfo コマンドを使用してください。

```
% psrinfo
0   on-linesince 01/12/95 10:41:54
1   on-linesince 01/12/95 10:41:54
3   on-linesince 01/12/95 10:41:54
4   on-linesince 01/12/95 10:41:54
```

複数のプロセッサを使用するには、PARALLEL 環境変数を設定してください。デフォルトは 1 です。

- 使用できる数より多くのプロセッサを指定しないでください。
- マシン上のプロセッサの数を *n* とした場合、単一ユーザーがマルチプロセッサシステムを使用するには PARALLEL=*n*-1 を指定してください。

-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 を指定しなければなりません。

-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。

-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 に設定されます。

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。ユーザ定義の 静的関数はインライン化しません。

-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。"//" がコメントの開始を示すために使えるようになります。

-xc99[=*o*]

-xc99 フラグは、C99 規格 (ISO/IEC 9899:1999, Programming Language - C) 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、次の値のいずれかが入ります：%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。

引数を取らずに -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、デフォルトでは付録 D で説明している C99 の機能になりますが、Solaris が提供する標準のヘッダーファイル (/usr/include にあります) 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。

-xcache[=*c*]

オプティマイザ用のキャッシュ特性を定義します。*c* には以下のいずれかを指定します。

- `generic`
- *s1/l1/a1*
- *s1/l1/a1:s2/l2/a2*
- *s1/l1/a1:s2/l2/a2:s3/l3/a3*

si、*li*、*ai* の定義は以下のとおりです。

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

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ことが主な目的です。

このオプションは、オプティマイザが使用できるキャッシュ特性を指定します。特定のキャッシュ特性が必ず使用されるわけではありません。次に、`xcache` の値を示します。

表 A-12 `-xcache` の値

値	意味
`generic`	ほとんどの x86、SPARC の各アーキテクチャで良好なパフォーマンスを得るためのキャッシュ特性を定義します。 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。

表 A-12 -xcache の値

値	意味
s1/l1/a1	レベル 1 のキャッシュ特性を定義します。
s1/l1/a1:s2/l2/a2	レベル 1 と 2 のキャッシュ特性を定義します。
s1/l1/a1:s2/l2/a2:s3/l3/a3	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、以下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
32 バイトの行サイズ	32 バイトの行サイズ
4 面結合	直接マップ結合

-xcg[89|92]

(SPARC)

-xcg89 は、「-xarch=v7 -xchip=old -xcache=64/32/1」を意味するマクロです。

-xcg92 は、「-xarch=v8 -xchip=super -xcache=16/32/4:1024/32/1」を意味するマクロです。

-xchar[=*o*]

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を明示的に示すように記述しなおす必要があるのは、char 型の符号が問題になるコードだけです。

o には、以下のいずれかを指定します。

表 A-13 -xchar の値

値	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最も一般的です。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。デフォルトでは、C コンパイラは Solaris ABI に従い char を符号付きとして定義します。-xchar を指定しても、これは変更されません。したがって、このようなライブラリを使用する場合は、このオプションも使用するか、char 型で引き渡される値または戻り値の符号を明示的に指定することが必要です。

-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、次の値のいずれかを指定できます。

- `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 `-X[a|c|s|t]` で決定された順に配置する。詳細は、84 ページの「文字定数」を参照してください。

-xcheck[=*o*]

(SPARC)

`-xcheck=stkovf` を指定してコンパイルすると、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が、実行時にチェックされます。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。アプリケーションにおける、スタックオーバーフローで生成される `SIGSEGV` を他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。

o には、以下のいずれかを指定します。

表 A-14 `-xcheck` の値

値	意味
`%none`	`-xcheck` のチェックを実行しません。
`%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のチェックをオンにします。
`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。

`-xcheck` を指定しない場合は、コンパイラでは `-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 `-xcheck=%all` がデフォルトで指定されます。この場合は、スタックオーバーフローの実行時チェックがオンになります。

`-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。

-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、generic、old、super、super2、micro、micro2、hyper、hyper2、powerup、ultra、ultra2、ultra2e、ultra2i、ultra3、ultra3cu、386、486、pentium、pentium_pro のいずれかを指定します。

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。

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。

xchip=*c* は以下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
- 同義の代替命令が使用できる場合に使用する命令

表 A-15 -xchip の値

値	意味
generic	ほとんどの x86、および 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 SPARC プロセッサに良好なパフォーマンスを提供し、どの SPARC 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。
old	SuperSPARC 以前のプロセッサのタイミング特性を使用する。
super	SuperSPARC プロセッサのタイミング特性を使用する。
super2	SuperSPARC II プロセッサのタイミング特性を使用する。
micro	microSPARC プロセッサのタイミング特性を使用する。
micro2	microSPARC II プロセッサのタイミング特性を使用する。
hyper	hyperSPARC プロセッサのタイミング特性を使用する。
hyper2	hyperSPARC II プロセッサのタイミング特性を使用する。
powerup	Weitek® PowerUP プロセッサのタイミング特性を使用する。

表 A-15 -xchip の値 (続き)

値	意味
ultra	UltraSPARC プロセッサのタイミング特性を使用する。
ultra2	UltraSPARCII プロセッサのタイミング特性を使用する。
ultra2e	UltraSPARCIIe プロセッサのタイミング特性を使用する。
ultra2i	UltraSPARCIIi プロセッサのタイミング特性を使用する。
ultra3	UltraSPARCIII プロセッサのタイミング特性を使用する。
ultra3cu	UltraSPARC III Cu プロセッサのタイミング属性を使用する。
386	Intel 386 アーキテクチャのタイミング特性を使用する。
486	Intel 486 アーキテクチャのタイミング特性を使用する。
pentium	Intel Pentium アーキテクチャのタイミング特性を使用する。
pentium_pro	Intel Pentium Pro アーキテクチャのタイミング特性を使用する。

-xcode=*v*

(SPARC) コードアドレス空間を指定します。*v* は次のいずれか 1 つでなければなりません。

abs32	32 ビット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。これは 32 ビットアーキテクチャ (-xarch=generic, v7, v8, v8a, v8plus, v8plusa) でデフォルトです。
abs44	44 ビット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 (-xarch=v9, v9a) だけで利用できます。

abs64	64 ビット絶対アドレスを生成します。64 ビットアーキテクチャ (-xarch=v9, v9a) だけで利用できます。
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では最大 2**11 まで、64 ビットアーキテクチャでは最大 2**10 までの固有の外部シンボルを参照できます。 -xcode=pic13 コマンドは、大域オフセットテーブルのサイズが 8K バイトに制限されることを除けば、-xode=pic32 に似ています。
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では最大 2**30 まで、64 ビットアーキテクチャでは最大 2**29 までの固有の外部シンボルを参照できます。 大域データへの参照はそれぞれ、大域オフセットテーブル内のポインタの間接参照として生成されます。各関数呼び出しは、手続きリンケージテーブルを使用して PC の相対アドレッシングモードで生成されます。ごくまれに -xcode=pic32 で大域データオブジェクトが多くなり過ぎることがありますが、その場合は大域オフセットテーブルは 32 ビットアドレス範囲も使用します。

SPARC V7 と V8 の場合のデフォルトは -xcode=abs32 です。SPARC V9 と UltraSPARC V9 (-xarch=v9|v9a) の場合のデフォルトは -xcode=abs64 です。

64 ビットの Solaris 環境で -xarch=v9、v9a、v9b のいずれかを指定して共有動的ライブラリを構築する場合は、-xcode=pic13 または -xcode=pic32 を指定することができます (必須ではありません)。

-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 および -xcode=pic32 のいずれかでコンパイルしたルーチンは、共有ライブラリの大域または静的変数へのアクセスに使用されるテーブル (_GLOBAL_OFFSET_TABLE_) を指し示すようレジスタを設定するために、入口で命令を数個余計に実行します。
- 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、大域および静的変数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<ファイル名>.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、英字の U で示されます。

-xcode=pic13 または -xcode=pic32 を使用すべきかどうかを判断するには、nm を使用して、共有ライブラリで使用または定義されている明確な大域および静的変数の個数を確認します。_GLOBAL_OFFSET_TABLE_ のサイズが 8,192 バイトより小さい場合は、-Kpic を使用できます。そうでない場合は、-xcode=pic32 を使用する必要があります。

-xcrossfile[=*n*]

(SPARC) ソースファイル間の最適化とインライン化を有効にします。*n* を指定する場合は、0、1、2 のいずれかです。

通常、コンパイラの解析のスコープは、コマンド行上の各ファイルに制限されます。たとえば、-xO4 の自動インライン化は、同じソースファイル内で定義および参照されるサブプログラムに制限されます。

-xcrossfile を指定すると、コンパイラは、コマンド行に指定したすべてのファイルを (1 つのソースファイルに連結したように) 解析します。-xcrossfile は、-xO4 または -xO5 と併用したときだけ有効です。

このコンパイルから生成されるファイルは、インライン化のため相互に依存しています。したがって、1 つのプログラムにリンクするときには、1 つの単位として使用しなければなりません。任意の 1 つのルーチンを変更し、そのファイルを再コンパイルした場合は、すべてのファイルを再コンパイルしなければなりません。つまり、このオプションを使用すると、メークファイルの構成にも影響があります。

デフォルトは `xcrossfile=0` で、ファイル間の最適化は行われません。`-xcrossfile` は `-xcrossfile=1` と同義です。

-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` などがあります。

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、上記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` が発行されない限り、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。

-xdepend

(SPARC) 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。ループの再構成には、ループの交換、ループの融合、スカラーの置換、無意味な配列への代入の除去が含まれます。最適化が `-xO3` 以上でない場合、`-xO3` に上げられ、警告が出されます。

`-xautopar` または `-xparallel` には依存性の解析も含まれています。依存性の解析はコンパイル時に実行されます。

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に `-xdepend` を試みる場合は、`-xautopar` または `-xexplicitpar` を使用しないでください。これらのいずれかが有効になっていると、`-xdepend` の最適化はマルチプロセッサシステムについて実行されます。

-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。

-xexplicitpar

(SPARC) `#pragma MP` 指令の指定にもとづいて並列化コードを生成します。ユーザー自身が、依存性の解析を行い、ループの繰り返し内部でのデータの依存性を解析および指定します。ソフトウェアは指定されたループを並列化します。最適化が `-xO3` 以上でない場合、`-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、`-xexplicitpar` を使用しないでください。

コードの実行速度を高めたければ、このオプションにはマルチプロセッサシステムが必要です。単一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。

ユーザーが指定したループに依存関係が含まれていると、正しい結果が得られないことがあります。しかも、結果が実行するごとに異なることがあります。なお、警告は出力されません。縮約ループには、明示的な並列プラグマは適用しないでください。明示された並列化は行われますが、ループの縮約は適用されないため、誤った結果が生じる場合があります。

要約すると、明示的な並列化には次の操作を実行します。

- ループを解析して、安全に並列化できるループを見つける。
- `#pragma MP` を挿入してループを並列化する。詳細については、56 ページの「明示的な並列化およびプラグマ」を参照してください。
- `-xexplicitpar` オプションを使用する。

ループの直前に並列プラグマを挿入する例を示します。

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

-xexplicitpar を使用してコンパイルとリンクを一度に実行する場合、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。-xexplicitpar 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 -xexplicitpar を指定しなければなりません。

-xexplicitpar と -xopenmp を一緒に使用しないでください。

-xF

パフォーマンスアナライザによる関数の順序の変更を許可します (analyzer(1) のマニュアルページを参照)。-xF オプションを指定してコンパイルし、パフォーマンスアナライザを実行すると、関数の順序が最適化されたマップファイルを生成することができます。リンカーの -Mmapfile オプションを使用することで、実行可能ファイルを構築するそれ以降のリンクが最適化されたマップファイルを使用するように指定することができます。このオプションは、実行可能ファイルの各関数を別々のセクションに配置します。たとえば、関数 foo() および bar() は、それぞれ .text%foo および .text%bar に配置されます。また、このオプションを使用した場合は、データ収集に必要なデバッグ情報がオブジェクトファイルでアセンブラによって生成されます。

-xhelp=*f*

オンラインヘルプ情報を表示します。

f には、flags、readme、errors のいずれか 1 つ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。

-xildoff

インクリメンタルリンカーを使用せず、強制的に ld を起動します。-g オプションを使用しない場合、-G オプションを使用した場合、またはコマンド行にソースファイルが存在する場合は、このオプションがデフォルトになります。このデフォルトを使用したくない場合は、-xildon オプションを指定してください。

-xildon

強制的に、インクリメンタルリンカー (ild) をインクリメンタルモードで起動します。
-g オプションを使用し 、-G オプションを使用せず、かつ、コマンド行にソースファイルを指定しない場合は、このオプションがデフォルトです。このデフォルトを使用したくない場合は、-xildoff オプションを指定してください。

-xinline=<リスト>

-xinlineのリストの書式を以下に示します。
[{%auto,*func_name*,no%*func_name*}[,{%auto,*func_name*,no%f*unc_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* は関数名を表します) 。
-xinline は、-xO3 以上の最適化レベルでだけ有効です。

表 A-16 -xinline の引数

値	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。
値なし	コンパイラでソースファイル内のすべての関数を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化が試行されます。
-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 -x04 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インライン化だけが試行されます。最適化レベルが -x04 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。

-xipo[=a]

a は 0、1 または 2 と置き換えます。引数を取らない -xipo は、-xipo=1 と同じ働きをします。-xipo=0 はデフォルトの設定であり、-xipo をオフにします。

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、プログラム全体の最適化を実行します。-xcrossfile と異なり、-xipo は、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。-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、コンパイラは手続き間の別名解析と、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。

-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、この追加情報は最終的な実行可能バイナリファイルの一部とはなりません。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。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、大型の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特に便利です。このフラグでコンパイルしたオブジェクトファイルには、内部でコンパイルされた分析情報が含まれているため、ソースファイルとコンパイル前のプログラムファイルを介して内部手続き解析が有効になります。

ただし、解析と最適化の対象は、-xipo でコンパイルされたオブジェクトファイルに限定され、ライブラリのオブジェクトファイルにまで拡張されません。

-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。

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 つのソースファイルを介してファイル相互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
```

ここで、one.c、two.c、および three.c の間、および main.c と four.c の間で内部手続きの最適化が実行されますが、main.c または four.c および mylib.a のルーチンの間では内部手続きの最適化が実行されません (初回コンパイルで未定義のシンボルについて警告が発せられることがありますが、コンパイルとリンクの作業であるために内部手続きの最適化は実行されます)。

-xipo に関するその他の重要な情報：

- 少なくとも最適化レベル -x04 を必要とします。
- -xcrossfile と競合します。両方を使用した場合、コンパイルエラーが発生します。
- -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。

-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。

-xlibmil

libm 用のインライン展開テンプレートをインクルードします。この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。

-xlic_lib=sunperf

(SPARC) Sun 提供のパフォーマンスライブラリにリンクします。

-xlicinfo

使用されているライセンスファイル、許可されているライセンストークン、シリアル番号、RTU、試用ライセンス、および有効期限についての情報を戻します。このオプションは、コンパイルの要求やライセンスのチェックは実行しません。

-xloopinfo

(SPARC) 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`、`-xparallel`、または `-xexplicitpar` 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、コンパイラは警告を出します。

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。

-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、makefile 用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 (makefile 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)。

例:

```
#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
```

-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。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
```

-xMl 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
```

-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限り、プロセスどうしで共有することができます。

-xmaxopt=[*v*]

v は、off、1、2、3、4、5 のいずれかです。 このコマンドは、pragra opt のレベルを指定されたレベルに限定します。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。

-xmemalign=*ab*

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 A-17 -xmemalign の境界整列と動作の値

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	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
8	最大 8 バイトの境界整列		
16	最大 16 バイトの境界整列		

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。

コンパイル時に境界整列が判別できないメモリーへのアクセスの場合、コンパイラは必要なロードおよびストア命令を生成するための境界整列を想定する必要があります。

-xmemalign フラグを使用すると、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 フラグは、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。

次に、-xmemalign フラグがまったく指定されていない場合にのみ適用される -xmemalign のデフォルト値を示します。

- -xmemalign=4s: -xarch の値が generic、v7、v8、v8a、v8plus、v8plusa のいずれかの場合に適用される。バグ 4340650 により、ILP32 がリストから削除されました。
- -xmemalign=8s: -xarch の値が v9 と v9a のどちらかの場合に適用される

次に、-xmemalign フラグ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 A-18 `-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
`-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

-xnativeconnect[=*a*[,*a*]...]

`-xnativeconnect` オプションを使用して、オブジェクトファイル内の情報とそれ以降の共有ライブラリを取り込み、共有ライブラリを Java[tm] プログラミング言語で記述したコード (Java コード) から使用可能にすることができます。また、共有ライブラリを構築するときに、`cc -G` を使用して `-xnativeconnect` を含める必要があります。

`-xnativeconnect` を指定した場合は、ネイティブコードのインタフェースの外部に対する可視性が最大になります。ネイティブコネクタツール (NCT) を使用して、C 標準ライブラリを Java コードから呼び出すことができるように、Java コードおよび Java[tm] Native Interface (JNI) コードを自動的に生成することができます。NCT の使用方法の詳細については、Forte Developer のオンラインヘルプを参照してください。

a には、`%all`|`%none`|`[no%]interfaces` のいずれかを指定します。

`-xnativeconnect` を指定しない場合は、コンパイラでは `-xnativeconnect=%none` が指定されます。`-xnativeconnect` だけを指定した場合は、コンパイラでは `-xnativeconnect=interfaces` が指定されます。

-xnativeconnect=interfaces は、バイナリインタフェース記述子 (BIDS) を生成します。

-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なりません。次に例を示します。

```
% cc test.c -xnolib -Bstatic -lm -Bdynamic -lc
```

このように指定すると、libm は静的にリンクされ、その他のライブラリは動的にリンクされます。

-xnolibmil

数学ライブラリのルーチンをインライン化しません。このオプションは -fast オプションの後に指定してください。以下に例を示します。

```
% cc -fast -xnolibmil...
```

-xO[1|2|3|4|5]

オブジェクトコードを最適化します。O が大文字であることに注意してください。-xO オプションと -g オプションを組み合わせて使用すると、特定の範囲だけをデバッグすることができます。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

最適化のレベルは 1 から 5 のうちのいずれかです。使用するプラットフォームによって変わります。

(SPARC)

`-xO1`

最小限の局所的な最適化 (ピープホール) を行います。

`-xO2`

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。

`-xO2` 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。一般的にコードサイズは最も小さくなります。

`-xO3`

`-xO2` 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。`-xO3` レベルではポインタ割り当ての結果を追跡しません。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 `-xO3` レベルではコードサイズが増大します。

`-xO4`

`-xO3` 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指定したい場合は、285 ページの「`-xinline=<リスト>`」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。

`-xO5`

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より長く、または実行時間が確実に短縮化されるわけではないという短所がありますが、プロファイルフィードバックを伴って実行するとパフォーマンスをより向上させやすくなります。300 ページの「`-xprofile=p`」を参照してください。

(x86)

`-xO1`

デフォルトの最適化での 1 つの段階の他に、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。

`-xO2`

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。

`-xO3`

レベル 2 で行う最適化の他に、ループの削減と帰納変数の除去を行います。

`-xO4`

レベル 2 と 3 で行う最適化の他に、ループの展開と、可能であればスタックフレーム生成の回避、および同一ファイルに含まれる関数の自動インライン化を行います。この最適化を行うと、`adb` と `dbx` のスタックトレースに誤りが発生する可能性があるので注意してください。

`-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。

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。

`-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。

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。最適化の詳細については、『プログラムのパフォーマンス解析』を参照してください。

-xopenmp [=*i*]

(SPARC)

ここで *i* は、parallel、stubs、または none です。-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。

-xopenmp=parallel は、OpenMP プラグマの認識を有効にし、SPARC のみに適用されます。-xopenmp=parallel の最適化レベルは -x03 です。プログラムの最適化レベルを下位レベルから -x03 に引き上げると、警告メッセージが発せられます。-xopenmp=parallel は、_OPENMP プリプロセッサトークンを事前定義します。

-xopenmp=stubs は、OpenMP API ルーチンのスタブルーチンとリンクします。シリアルに実行するアプリケーションをコンパイルする場合に、このオプションを使用します。-xopenmp=stubs も _OPENMP プリプロセッサトークンを事前定義します。

-xopenmp=none は、OpenMP プラグマの認識の有効化、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 -xexplicitpar または -xparallel と同時に指定しないでください。

OpenMP 準拠のプログラムのコンパイル方法の詳細については、32 ページの「OpenMP に対する並列化」を参照してください。

この OpenMP の実装固有の情報については、付録 G を参照してください。

-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。

```
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 **);
```

-xparallel

(SPARC) ループを、コンパイラで自動的に並列化するとともに、プログラマの指定によって明示的に並列化します。-xparallel オプションはマクロで、-xautopar、-xdepend、-xexplicitpar の 3 つをすべて指定することと同じです。ループの明示的な並列化では、誤った結果が生まれる危険性があります。最適化が -xO3 以上でない場合、-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、-xparallel を使用しないでください。-xopenmp を発行する場合、-xparallel を発行しないでください。-xparallel は、-xopenmp を指定する際に発行すべきでない -xexplicitpar を設定するからです。

コードの実行速度を高めたければ、このオプション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、生成されたコードの実行速度は低下します。

コンパイルとリンクを 1 度で実行すると、-xparallel によりマイクロタスキングライブラリとスレッドに対して安全な実行時ライブラリがリンクに含まれます。-xparallel オプション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 -xparallel オプションを指定しなければなりません。

-xpentium

(x86) Pentium プロセッサ用に最適化を行います。

-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。-xpg はプログラム実行時に記録機構を起動します。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。

-xprefetch[=<値>],<値>

(SPARC) 先読みをサポートするアーキテクチャ (UltraSPARC II など) で先読み命令を有効にします (-xarch=v8plus、v9plusa、v9、v9a のいずれか)。

明示的な先読み命令の使用は、パフォーマンスが実際に向上する特別な場合に限定してください。

< 値 > には、次のいずれかを指定します。

表 A-19 -xprefetch の引数

値	意味
latx:factor	factor で指定した係数で、先読みから読み込みまで、および先読みから格納までの想定応答時間を調整します。298 ページの「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	-xprefetch=auto,explicit と同じ
no	-xprefetch=no%auto,no%explicit と同じ

-xprefetch が指定されない場合のデフォルトは、-xprefetch=no%auto,explicit です。値なしで -xprefetch を指定すると、-xprefetch=auto,explicit と同じ意味になります。

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。

先読み応答率

先読み応答時間は、先読み命令が実行されてから、先読みされたデータがキャッシュに格納されるまでのハードウェア遅延を示します。

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、先読み命令および先読みしたデータを使用するロード命令やストア命令を実行するまでの時間を特定する際に、先読み応答時間の値を使用します。先読みからロードまでの想定応答時間は、先読みからストアまでの想定応答時間とは同一でない場合があります。

コンパイラは、さまざまなマシンおよびアプリケーションで性能の最適化を行うため、先読みメカニズムを調整します。このチューニングが最適でない場合があります。メモリーを多用するアプリケーション、特に大規模なマルチプロセッサで実行するアプリケーションでは、先読み応答時間の値を大きくすることで、性能が向上することがあります。値を大きくするには、1 よりも大きな係数を使用します。.5 ～ 2.0 の値を指定すると、ほとんどの場合は最大の性能が実現されます。

完全に外部キャッシュ内に収まるデータセットを使用するアプリケーションでは、先読み応答時間の値を小さくすることで、性能が向上することがあります。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。必要に応じて係数を変更し、性能テストを再実行します。最適な性能になるまで、係数の変更と性能テストの実行を繰り返します。係数をわずかな値だけ変更していくと、最初は性能の差がはっきりせず、数回変更して初めて明確な性能の差が生じ、その後再び性能の変化が横ばいになります。

-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性能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` を指定した場合は、先読み命令の自動生成が有効になります。`-xprefetch_level=2` では、レベル 1 よりも多くの先読み命令が生成されます。`-xprefetch_level=3` では、レベル 2 よりも多くの先読み命令が生成されます。

デフォルトは、`-xprefetch=auto` を指定した場合は `-xprefetch_level=1` になります。

-xprofile=*p*

(SPARC)

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。

p には、collect[:<名前>]、use[:<名前>]、または tcov のいずれか 1 つを指定します。

このオプションを使用すると実行中に実行頻度データを収集して保存することができ、後 続の実行ではそのデータを使用してパフォーマンスを向上させることができます。このオプションは、`-x02` 以上の最適化のレベルを指定した場合にのみ有効です。

- `collect`[:<名前>]

  実行後に `-xprofile=use` を指定してオプティマイザで使用するために、実行頻度データを収集して保存します。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。

  <名前> は、解析の対象となるプログラムの名前です。この名前はオプションです。<名前> の指定を省略すると、`a.out` が実行可能ファイルの名前とみなされます。

  環境変数 `SUN_PROFDATA` および `SUN_PROFDATA_DIR` を設定すると、`-xprofile=collect` でコンパイルされたプログラムがプロファイルデータを格納する場所を制御できます。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も指定します。tcov(1) マニュアルページを参照してください。

  これらの環境変数を設定しない場合、プロファイルデータは現在のディレクトリの *name*`.profile/feedback` に書き込まれます。ここで *name* には、実行可能ファイルの名前、または `-xprofile=collect:name` フラグで指定された名前が入ります。*name* に拡張子 `.profile` がすでに含まれている場合、*name* に `.profile` は付加されません。プログラムを複数回実行する場合、実行頻度のデータがフィードバックファイルに蓄積されます。つまり、過去のプログラム実行の出力が失われることはありません。

注 - -xprofile=collect を指定して共有ライブラリをコンパイルすることはできません。

- use[:<名前>]

  実行頻度データを使用して、効果的に最適化を行います。

  collect:<名前> と同様に、<名前> はオプションです。これにより、プログラム名を指定できます。

  -xprofile=collect でコンパイルしたプログラムを前回実行したときに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のデータにもとづいて、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他のコンパイラのオプションは、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。収集ビルドと使用ビルドの両方に対し、同じバージョンのコンパイラを使用する必要があります。-xprofile=collect:<名前> を使用してコンパイルする場合は、最適化コンパイルでも同じ名前 (-xprofile=use:<名前>) が使用されていなければなりません。

- tcov

  新しい形式の tcov を使用した基本ブロックカバレージ解析です。

  -xprofile=tcov オプションは、新しい形式の tcov 用基本ブロックプロファイリングです。 機能は -xa オプションと類似していますが、ヘッダーファイルにソースコードがあるプログラムまたは C++ テンプレートを使用するプログラムのデータを正確に収集します。古い形式のプロファイリングについては 「-xa」の節、tcov(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』を参照してください。

  時間計測コードの組み込みは -xa オプションの場合と同様に実行されますが、.d ファイルは生成されません。その代わりに、最終的な実行可能ファイルにもとづいた名前をもつファイルが 1 つだけ生成されます。たとえば、プログラムが /foo/bar/myprog.profile から実行されると、データファイルは /foo/bar/myprog.profile/myprog.tcovd に格納されます。

-xprofile=tcov と -xa オプションは、同じ実行可能ファイル内に指定することができます。すなわち、-xprofile=tcov でコンパイルされたファイルと -xa でコンパイルされたファイルが両方含まれたプログラムをリンクすることができます。1 つのファイルを両方のオプションでコンパイルすることはできません。

tcov を実行する時点で、新しい形式のデータを使用させるように -x オプションを渡さなければなりません。これを渡さないと、古い.d ファイルがまだ存在する場合に tcov はデフォルトで古いファイルからデータを使用するため、予想に反した出力が生成されます。

-xa オプションの場合とは異なり、TCOVDIR 環境変数はコンパイル時には影響力を持ちません。ただし、その値はプログラムの実行時に使用されます。詳細については
tcov(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』を参照してください。

注 – -xO4 または -xinline によるルーチンのインライン化が存在する場合、tcov カバレージ解析のデータが不正確になることがあります。

-xprofile=collect を指定してプロファイルの収集用にコンパイルを行い、-xprofile=use を指定してプロファイルのフィードバック用にコンパイルを行う場合は、ソースファイルおよび -xprofile=collect と -xprofile=use 以外のコンパイラオプションが両方のコンパイルで同一である必要があります。

-xprofile=use:name オプションで指定するプロファイルのフィードバックのディレクトリ名を複数指定した場合は、コンパイラの 1 回の呼び出しですべて使用できます。たとえば、プロファイル対象のバイナリ a、b、c を実行した結果、プロファイルのディレクトリ a.profile、b.profile、c.profile が生成されたとします。

```
cc -O -c foo.c -xprofile=use:a -xprofile=use:b -xprofile=use:c
```

このとき、3 つのプロファイルのディレクトリがすべて使用されます。特定のオブジェクトファイルに関係する有効なプロファイルのフィードバックデータは、オブジェクトファイルのコンパイル時に、指定したフィードバックのディレクトリから累積されます。

-xprofile=collect と -xprofile=use の両方を同一のコマンド行で指定した場合は、コマンド行で一番右側の -xprofile オプションは以下のように適用されます。

- 一番右側の -xprofile オプションが -xprofile=use の場合は、-xprofile=use で指定したすべてのプロファイルフィードバックディレクトリ名がフィードバック指定の最適化で使用され、このオプションよりも前の -xprofile=collect オプションは無視されます。
- 一番右側の -xprofile オプションが -xprofile=collect の場合は、-xprofile=use で指定したすべてのプロファイルフィードバックディレクトリ名は無視され、プロファイル生成の計測が有効になります。

-xreduction

(SPARC) 自動並列化の間に縮約を認識させます。-xreduction オプションは、-xautopar または -xparallel のいずれかが指定されている場合にのみ有効です。

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。

-xregs=*r*[,*r*...]

(SPARC) 使用するレジスタを指定します。

r には、以下の 1 つまたは複数の項目をコンマで区切って指定します。

[no%]appl、[no%]float

no には % を付けてください。

例: `-xregs=appl, no%float`

表 A-20 `-xregs` の値

値	意味
`appl`	`g2`、`g3`、`g4` レジスタ (`v8a`、`v8`、`v8plus`、`v8plusa`、`v8plusb` の場合)、または `g2`、`g3` レジスタ (`v9`、`v9a`、`v9b` の場合) を使用できるようにします。SPARC の命令セットの詳細については、265 ページの「`-xarch=isa`」を参照してください。 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呼ばれます。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。
`no%appl`	アプリケーションレジスタ g2, g3, g4 (v8a, v8, v8plus, v8plusa, v8plusb) g2, g3 (v9, v9a, v9b) を使用しません。すべてのシステムソフトウェアおよびライブラリは、`-xreg=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。
`float`	SPARC ABI で規定されている浮動小数点レジスタを使用できるようにします。これらのレジスタは、プログラムに浮動小数点のコードがない場合でも使用することができます。
`no%float`	浮動小数点レジスタを使用しません。 このオプションを使用すると、ソースプログラムに浮動小数点コードを記述することはできません。

デフォルトは `-xregs=appl, float` です。

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、リンクするアプリケーションでのレジスタ処理に問題がないように、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。

-xrestrict= [=*f*]

(SPARC) 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。

{%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、54 ページの「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最も適しています。以下に例を示します。

```
% 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。

```
% 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。

-xs

dbx での .o ファイルの自動読み込みを無効にします。このオプションは .o ファイルを保存しておくことができない場合に使用します。-s オプションはアセンブラに渡されます。

旧来の方法 (自動読み取りなし) では、シンボルテーブルのロードで次の処理が行われます。dbx 用のすべてのシンボルテーブルを実行可能ファイルに書き込むため、リンカーの結合と dbx の初期設定に時間がかかります。

新しい方法では、シンボルテーブルのロードの際に自動読み取りがデフォルトで稼働します。自動読み取りによって、シンボルテーブルの情報が .o ファイルに分散され、必要なときだけ dbx がシンボルテーブル情報をロードします。そのため、リンカーの結合と dbx の初期設定が速くなります。

-xs を指定すると、実行可能ファイルを他のディレクトリに移動し、dbx を使用する必要が生じたときに、オブジェクト (.o) ファイルを無視することができます。

-xs を指定しない場合は、実行可能ファイルを移動する際に、ソースファイルとオブジェクト (.o) ファイルの両方を移動するか、dbx pathmap または use コマンドでパスを設定しなければなりません。

-xsafe=mem

(SPARC) メモリーに関するトラップが発生しないことを前提とします。

このオプションによって、V9 マシン上で投機的ロード命令を使用することが許可されます。これは、-xO5 最適化と、-xarch=v8plus|v8plusa|v9|v9a を指定する場合だけ有効です。

注 - ロードそのものが完了した場合は、アドレスの不正な整列やセグメンテーション違反などの障害が発生してもトラップが実行されないため、このような障害が発生することがないプログラムでは、このオプションを使用してください。メモリーベースのトラップが関係するプログラムはほとんどないため、多くのプログラムではこのオプションを使用しても問題はありません。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。

-xsb

ソースブラウザ用のシンボルテーブル情報を生成します。このオプションは、コンパイラの -Xs モードと併用することはできません。

-xsbfast

ソースブラウザ用のデータベースを作成します。ソースファイルはオブジェクトファイルにはコンパイルされません。このオプションは、コンパイラの -Xs モードと併用することはできません。

-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。

-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。

-xstrconst

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。重複する文字列は削除され、それ以外の部分はコード中の参照間で共有されます。

-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 の値は native、generic、SPARC または x86 のシステム名のいずれかでなければなりません。

-xtarget オプションは、実際のシステムに合わせて、-xarch、-xchip、-xcache の組み合わせを手早く簡単に指定することができます。-xtarget の意味は = の後に指定した値を展開したものにあります。

表 A-21 -xtarget の展開

値	意味
native	ホストシステムに対してパフォーマンスを最適化します。 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、チップ、キャッシュ特性を判定します。
generic	一般的なアーキテクチャ、チップ、キャッシュ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。 コンパイラは -xtarget=generic を次のように展開します。 -xarch=generic -xchip=generic -xcache=generic これはデフォルトです。
<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。 このオプションはマクロです。表 A-22 に示す、実際のシステム名と機種番号のリストから、システム名を選択してください。

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の名前を正式に指定してください。これは、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。ただし、ほとんどのプログラムと、より旧式の SPARC プロセッサ間では、パフォーマンス向上はごくわずかであり、generic を指定することで十分です。

-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。表 A-22 を参照してください。

例: -xtarget=sun4/15 と指定することは、
-xarch=V8a -xchip=micro -xcache=2/16/1 と指定することと同じです。

表 A-22 -xtarget の展開

-xtarget の値	-xarch	-xchip	-xcache
generic	generic	generic	generic
cs6400	v8	super	16/32/4:2048/64/1
entr150	v8	ultra	16/32/1:512/64/1
entr2	v8plusa	ultra	16/32/1:512/64/1
entr2/1170	v8plusa	ultra	16/32/1:512/64/1
entr2/1200	v8plusa	ultra	16/32/1:512/64/1
entr2/2170	v8plusa	ultra	16/32/1:512/64/1
entr2/2200	v8plusa	ultra	16/32/1:512/64/1
entr3000	v8plusa	ultra	16/32/1:512/64/1
entr4000	v8plusa	ultra	16/32/1:512/64/1
entr5000	v8plusa	ultra	16/32/1:512/64/1
entr6000	v8plusa	ultra	16/32/1:512/64/1
sc2000	v8	super	16/32/4:2048/64/1
solb5	v7	old	128/32/1
solb6	v8	super	16/32/4:1024/32/1
ss1	v7	old	64/16/1
ss10	v8	super	16/32/4
ss10/20	v8	super	16/32/4
ss10/30	v8	super	16/32/4
ss10/40	v8	super	16/32/4
ss10/402	v8	super	16/32/4
ss10/41	v8	super	16/32/4:1024/32/1
ss10/412	v8	super	16/32/4:1024/32/1
ss10/50	v8	super	16/32/4
ss10/51	v8	super	16/32/4:1024/32/1
ss10/512	v8	super	16/32/4:1024/32/1
ss10/514	v8	super	16/32/4:1024/32/1

表 A-22 -xtarget の展開 (続き)

-xtarget の値	-xarch	-xchip	-xcache
ss10/61	v8	super	16/32/4:1024/32/1
ss10/612	v8	super	16/32/4:1024/32/1
ss10/71	v8	super2	16/32/4:1024/32/1
ss10/712	v8	super2	16/32/4:1024/32/1
ss10/hs11	v8	hyper	256/64/1
ss10/hs12	v8	hyper	256/64/1
ss10/hs14	v8	hyper	256/64/1
ss10/hs21	v8	hyper	256/64/1
ss10/hs22	v8	hyper	256/64/1
ss1000	v8	super	16/32/4:1024/32/1
ss1plus	v7	old	64/16/1
ss2	v7	old	64/32/1
ss20	v8	super	16/32/4:1024/32/1
ss20/151	v8	hyper	512/64/1
ss20/152	v8	hyper	512/64/1
ss20/50	v8	super	16/32/4
ss20/502	v8	super	16/32/4
ss20/51	v8	super	16/32/4:1024/32/1
ss20/512	v8	super	16/32/4:1024/32/1
ss20/514	v8	super	16/32/4:1024/32/1
ss20/61	v8	super	16/32/4:1024/32/1
ss20/612	v8	super	16/32/4:1024/32/1
ss20/71	v8	super2	16/32/4:1024/32/1
ss20/712	v8	super2	16/32/4:1024/32/1
ss20/hs11	v8	hyper	256/64/1
ss20/hs12	v8	hyper	256/64/1
ss20/hs14	v8	hyper	256/64/1
ss20/hs21	v8	hyper	256/64/1
ss20/hs22	v8	hyper	256/64/1

表 A-22 -xtarget の展開 (続き)

-xtarget の値	-xarch	-xchip	-xcache
ss2p	v7	powerup	64/32/1
ss4	v8a	micro2	8/16/1
ss4/110	v8a	micro2	8/16/1
ss4/85	v8a	micro2	8/16/1
ss5	v8a	micro2	8/16/1
ss5/110	v8a	micro2	8/16/1
ss5/85	v8a	micro2	8/16/1
ss600/120	v7	old	64/32/1
ss600/140	v7	old	64/32/1
ss600/41	v8	super	16/32/4:1024/32/1
ss600/412	v8	super	16/32/4:1024/32/1
ss600/51	v8	super	16/32/4:1024/32/1
ss600/512	v8	super	16/32/4:1024/32/1
ss600/514	v8	super	16/32/4:1024/32/1
ss600/61	v8	super	16/32/4:1024/32/1
ss600/612	v8	super	16/32/4:1024/32/1
sselc	v7	old	64/32/1
ssipc	v7	old	64/16/1
ssipx	v7	old	64/32/1
sslc	v8a	micro	2/16/1
sslt	v7	old	64/32/1
sslx	v8a	micro	2/16/1
sslx2	v8a	micro2	8/16/1
ssslc	v7	old	64/16/1
ssvyger	v8a	micro2	8/16/1
sun4/110	v7	old	2/16/1
sun4/15	v8a	micro	2/16/1
sun4/150	v7	old	2/16/1
sun4/20	v7	old	64/16/1

表 A-22 -xtarget の展開 (続き)

-xtarget の値	-xarch	-xchip	-xcache
sun4/25	v7	old	64/32/1
sun4/260	v7	old	128/16/1
sun4/280	v7	old	128/16/1
sun4/30	v8a	micro	2/16/1
sun4/330	v7	old	128/16/1
sun4/370	v7	old	128/16/1
sun4/390	v7	old	128/16/1
sun4/40	v7	old	64/16/1
sun4/470	v7	old	128/32/1
sun4/490	v7	old	128/32/1
sun4/50	v7	old	64/32/1
sun4/60	v7	old	64/16/1
sun4/630	v7	old	64/32/1
sun4/65	v7	old	64/16/1
sun4/670	v7	old	64/32/1
sun4/690	v7	old	64/32/1
sun4/75	v7	old	64/32/1
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
ultra2e	v8plusa	ultra2e	16/32/1:256/64/4

表 A-22 -xtarget の展開 (続き)

-xtarget の値	-xarch	-xchip	-xcache
ultra2i	v8plusa	ultra2i	16/32/1:512/64/1
ultra3	v8plusa	ultra3	64/32/4:8192/256/1
ultra3cu	v8plusa	ultra3cu	64/32/4:8192/512/2

次の表は、x86アーキテクチャの場合の -xtarget の値です。

表 A-23 x86 アーキテクチャでの -xtarget の展開

-xtarget=	-xarch	-xchip	-xcache
generic	generic	generic	generic
386	386	386	generic
486	386	486	generic
pentium	386	pentium	generic
pentium_pro	pentium_pro	pentium_pro	generic

- generic または native
- 386 (-386 オプションと等価) または 486 (-486 オプションと等価)
- pentium (-pentium オプションと等価) または pentium_pro

-xtemp=<ディレクトリ>

cc が使用する一時ファイルの <ディレクトリ> を設定します。このオプション文字列の中にはスペースを入れてはなりません。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。-xtemp は、TMPDIR 環境変数より優先します。

-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。

-xtransition

K&R C と Sun 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 オプションと共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 <型名> です: <名前>
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: <型名>
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
文字定数中では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: <演算子>
ISO C では <演算子> の意味が変わります。明示的なキャストを使用してください。
```

-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。

ソースコードのリテラル文字列に三文字表記シーケンスとして解釈される疑問符 (?) が含まれている場合、-xtrigraph=no サブオプションを使用すると、三文字表記シーケンスの認識をオフにできます。-xtrigraph=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;
}
```

-xtrigraphs=yes でこのコードをコンパイルした場合の出力を次に示します。

```
example% cc -xtrigraphs=yes trigraphs_demo.c
example% a.out
(??) in a string appears as (]
```

-xtrigraphs=no でこのコードをコンパイルした場合の出力を次に示します。

```
example% cc -xtrigraphs=no trigraphs_demo.c
example% a.out
(??) in a string appears as (??)
```

-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。

-xvector[={yes|no}]

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。

-xvector=yes が指定されると、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。大きなループカウントを持つループでは、この変換によりパフォーマンスが向上します。

-xvector が指定されない場合のデフォルトは、-xvector=no です。-xvector=no は、前に指定した-xvector=yes を取り消します。値のない -xvector が指定された場合のデフォルトは、-xvector=yes です。

あらかじめ -xdepend を指定せずにコマンド行で -xvector を指定すると、-xdepend が自動的に呼び出されます。また、最適化レベルが指定されないか、-xO3 以上でない場合は、最適化レベルが -xO3 に上げられます。

コンパイラは、リンク時に libmvec ライブラリを取り込みます。コンパイルとリンクを別々のコマンドで実行する場合は、リンク時の cc コマンドに必ず -xvector を使用してください。

-xvpara

(SPARC) ループが正しく並列化指定されていない場合に、#pragma MP 指令が指定されているループについて警告を出します。たとえば、オプティマイザがループの繰り返し中にデータの依存性を検出した場合に警告を出します。

-xvpara は、-xexplicitpar または -xparallel オプションと、#pragma MP を組み合わせて使用してください。詳細については、56 ページの「明示的な並列化およびプラグマ」を参照してください。

-Y*c*,<ディレクトリ>

c を検索するための新しいリ <ディレクトリ> 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
<ディレクトリ>/<ツール名> 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。

-YA,<ディレクトリ>

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,<ディレクトリ>

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。

-YP,<ディレクトリ>

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,<ディレクトリ>

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。

-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。詳細については、`lock_lint`(1) のマニュアルページを参照してください。

リンカーに渡されるオプション

`cc` は `-a`、`-e`、`-h`、`-r`、`-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。認識できないオプションは警告付きで `ld` に渡します。

付録 B

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。本章は、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。

記憶装置の割り当て

データ型とその表現方法について表 B-1 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、2 ギガバイト以下に制限すべきです。

表 B-1 データ型に対する記憶装置の割り当て

データ型	内部表現
`char` 型要素	8 ビット幅のシングルバイト。1 バイトで境界整列される。
`short` 型整数	ハーフワード (2 バイト、つまり 16 ビット)。2 バイトで境界整列される。
`int` 型	32 ビット (4 バイトまたは 1 ワード)、4 バイト境界で整列される。
`long` 型	v8 の 32 ビット (4 バイト、つまり 1 ワード)。4 バイトで境界整列される。 v9 では 64 ビット (4 バイト、つまり 1 ワード)。8 バイト境界で整列される。

表 B-1　データ型に対する記憶装置の割り当て (続き)

データ型	内部表現
`pointer` 型	v8 および Inel では 32 ビット (4 バイトまたは 1 ワード)、4 バイト境界で整列される。 v9 では 64 ビット (8 バイトまたは 2 ワード)、8 バイト境界で整列
`long long` 型[1]	*(SPARC)* 64 ビット (8 バイト、つまり 2 ワード)。 ダブルワードで境界整列される。 *(x86)* 64 ビット (8 バイト、つまり 2 ワード)。4 バイトで境界整列される。
`float` 型	32 ビット (4 バイト、つまり 1 ワード)。4 バイトで境界整列される。1 ビットの符号、8 ビットの指数部および 23 ビットの仮数部から成る。
`double` 型	64 ビット (8 バイト、つまり 2 ワード)。 *(SPARC)* 8 バイトで境界整列される。 *(x86)* 4 バイト境界に割り当てられる。 1 ビットの符号、11 ビットの指数部、52 ビットの仮数部から成る。
`long double` 型	v8 *(SPARC)* 128 ビット (16 バイト、つまり 4 ワード)。8 バイトで境界整列される。1 ビットの符号、15 ビットの指数部および 112 ビットの仮数部から成る。 v9 *(SPARC)* 128 ビット (16 バイト、つまり 4 ワード)。16 バイトで境界整列される。1 ビットの符号、15 ビットの指数部および 112 ビットの仮数部から成る。 *(x86)* 96 ビット (12 バイト、つまり 3 ワード)。4 バイトで境界整列される。1 ビットの符号、16 ビットの指数部および 64 ビットの仮数部から成る。16 ビットは使用されない。

1.`long long` は、`-Xc` モードで `-xc99=%none` を指定した場合は利用できません。

データ表現

使用しているアーキテクチャ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。以下の表に表現方法を示します。

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 B-2　`short` の表現 *(x86)*

ビット	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 B-3　`int` と `long` の表現

ビット	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 B-4　`long` の表現 (x86、SPARC v8、SPARC v9)

ビット	内容
`24 - 31`	バイト 0 *(SPARC)* v8 バイト 4 *(SPARC)* v9 バイト 3 *(Intel)*
`16 - 23`	バイト 1 *(SPARC)* v8 バイト 5 *(SPARC)* v9 バイト 2 *(Intel)*

表 B-4　long の表現 (x86、SPARC v8、SPARC v9) (続き)

ビット	内容
8 - 15	バイト 2 *(SPARC)* v8 バイト 6 *(SPARC)* v9 バイト 1 *(Intel)*
0 - 7	バイト 3 *(SPARC)* v8 バイト 7 *(SPARC)* v9 バイト 0 *(Intel)*

表 B-5　long long[1] の表現

ビット	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)*

1.long long は -Xc モードでは使用できません。

浮動小数点表現

`float`、`double`、`long double` のデータ要素は、ISO IEEE 754-1985 規格に従って下の式のように表現されます。

$$(-1)^{s(e - bias)} \times 2 \quad j.f$$

- *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。その他の場合は暗黙的。
- *f* = 仮数部 (23 ビット)
- *u* = ビットが 0 または 1 を示す。

表 B-6　`float` の表現

ビット	名称
31	符号 (Sign)
23 - 30	指数部 (Exponent)
0 - 22	仮数部 (Fraction)

表 B-7　`double` の表現

ビット	名称
63	符号 (Sign)
52 - 62	指数部 (Exponent)
0 - 51	仮数部 (Fraction)

表 B-8　`long double` の表現 *(SPARC)*

ビット	名称
127	符号 (Sign)
112 - 126	指数部 (Exponent)
0 - 111	仮数部 (Fraction)

表 B-9　long double の表現 *(x86)*

ビット	名称
81 - 95	使用せず
79	符号 (Sign)
64 - 78	指数部 (Exponent)
63	先行ビット
0 - 62	仮数部 (Fraction)

詳細については、『数値計算ガイド』を参照してください。

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 B-10 float の表現

正規数 (0<e<255):	$(-1)^{Sign} 2^{(exponent-127)} 1.f$
非正規数 (e=0, f!=0):	$(-1)^{Sign} 2^{(-126)} 0.f$
ゼロ (e=0, f=0):	$(-1)^{Sign} 0.0$
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 B-11 `double` の表現

正規数 (0<e<2047):	$(-1)^{Sign}2^{(exponent-1023)}1.f$
非正規数 (e=0, f!=0):	$(-1)^{Sign}2^{(-1022)}0.f$
ゼロ (e=0, f=0):	$(-1)^{Sign}0.0$
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
無限大	s=u, e=2047(最大値); f=.0000～00 (すべてが 0)

表 B-12 `long double` の表現

正規数 (0<e<32767):	$(-1)^{Sign}2^{(exponent-16383)}1.f$
非正規数 (e=0, f!=0):	$(-1)^{Sign}2^{(-16382)}0.f$
ゼロ (e=0, f=0):	$(-1)^{Sign}0.0$
シグナルを発生する NaN	s=u, e=32767(最大値); f=1.0uuu～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=1.1uuu～uu
無限大	s=u, e=32767(最大値); f=1.0000～00 (すべてが 0)

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 B-13 重要な数の 16 進数表現 *(SPARC)*

値	`float` 型	`double` 型	`long double` 型
+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
+無限	7F800000	7FF0000000000000	7FFF0000000000000000000000000000
-無限	FF800000	FFF0000000000000	FFFF0000000000000000000000000000
NaN	7FBFFFFF	7FF7FFFFFFFFFFFF	7FFF7FFFFFFFFFFFFFFFFFFFFFFFFFFF

表 B-14 重要な数の 16 進数表現 *(x86)*

値	`float` 型	`double` 型	`long double` 型
+0	00000000	0000000000000000	00000000000000000000
-0	80000000	0000000080000000	80000000000000000000
+1.0	3F800000	000000003FF00000	3FFF8000000000000000
-1.0	BF800000	00000000BFF00000	BFFF8000000000000000
+2.0	40000000	0000000040000000	40008000000000000000
+3.0	40400000	0000000040080000	4000C000000000000000
+無限	7F800000	000000007FF00000	7FFF8000000000000000
-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。

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。

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最も速く変化します。

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の詳細については、319 ページの「記憶装置の割り当て」を参照してください。

表 B-15 配列の型と最大の大きさ

型	SPARC および x86 の最大要素数	SPARC V9 の最大要素数
`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
`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`[1]	536,870,911	288,230,376,151,711,743

1. `-Xc` モードで `-xc99=%none` を指定した場合は無効です。

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。

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 B-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 B-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	参照[1]	Inf	NaN
左側のオペランド:Inf	Inf	Inf	参照[1]	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 B-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 B-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 B-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
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。

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。

(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[1] 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 `%i0` と `%i1` に返されます。

1. `-Xc` モードで `-xc99=%none` を指定した場合は使用できません。

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。

関数から戻った後、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。上記のレジスタは、呼び出し側から見えます。

SPARC v9

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。

浮動小数点数の引数は、可能であれば浮動小数点数のレジスタで引き渡されます。

(x86)

関数は integer 型の結果をレジスタ %eax に返します。

long long の結果はレジスタ %edx と %eax に返されます。float、double、long double 型の結果はレジスタ %st(0) に返されます。

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、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。

関数から戻った後、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 の余分な引数を除く)。

付録 C

処理系定義された ISO/IEC C の動作

『情報システム用米国規格プログラミング言語 C (ANSI ISO/IEC 9899:1990[1])』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。

- 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 規格で対応する節番号を付けています。

ISO 規格との実装の比較

翻訳 (G.3.1)

括弧内の数は、ISO/IEC 9899/1990 規格の節番号に対応しています。

(5.1.1.3) 診断の認識

エラーメッセージは次の書式です。

ファイル名、`line` 行番号：メッセージ

警告メッセージは次の書式です。

ファイル名、`line` 行番号：警告メッセージ

- 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

1. 日本の対応規格は、JIS X 3010 - 1993 です。

環境 (G.3.2)

(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後は、`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。

(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。

識別子 (G.3.3)

(6.1.2) 外部リンケージのない識別子の先頭から (31 を越える) 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 を越える) 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。

文字 (G.3.4)

(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。

(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。

(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。

(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。

(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。

(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。

(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`、 `LANG` 環境変数のいずれかで指定されたものです。

(6.2.1.1.) 何もついていない `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 *(SPARC) (x86)*。

整数 (G.3.5)

(6.1.2.5) 整数の型の表現と値について

表 C-1　整数の表現と値

整数	ビット数	最小値	最大値
`char` *(SPARC) (x86)*	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` *(SPARC)* `v8`	32	-2147483648	2147483647
`long` *(SPARC)* `v9`	64	-9223372036854775808	9223372036854775807
`signed long` *(SPARC)* `v8`	32	-2147483648	2147483647
`signed long` *(SPARC)* `v9`	64	-9223372036854775808	9223372036854775807
`unsigned long` *(SPARC)* `v8`	32	0	4294967295
`unsigned long` *(SPARC)* `v9`	64	0	18446744073709551615
`long long`[1]	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。

(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。

(6.3) 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。

(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。

(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

右シフトの結果は符号付きの右シフトとなります。

浮動小数点 (G.3.6)

(6.1.2.5) 浮動小数点数の型の表現と値

表 C-2　浮動小数点数の値

`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 C-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 C-4　`long double` の値

`long double`	
ビット数	128 *(SPARC)* 80 *(x86)*
最小値	3.3621031431120935062626778173217526 03E-4932 *(SPARC)* 3.362103143112093506262 7E-4932 *(x86)*
最大値	1.1897314953572317650857593266280070 16E+4932 *(SPARC)* 1.189731495357231765021 3E4932 *(x86)*
イプシロン	1.9259299443872358530559779425849273 19E-34 *(SPARC)* 1.0842021724855044340075E-19 *(x86)*

(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。

(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。

配列とポインタ (G.3.7)

(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。

-Xarch=v9 では、unsigned long です。

(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。

(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 において定義された int 型です。

-Xarch=v9 では、long 型です。

レジスタ (G.3.8)

(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。

構造体、共用体、列挙型、およびビットフィールド (G.3.9)

(6.3.2.3) 共用体のオブジェクトのメンバーは他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。

(6.5.2.1) 構造体のメンバーのパディングと整列条件

表 C-5　構造体メンバーのパディングと整列

型	整合の境界	バイト境界
`char`	バイト	1
`short`	ハーフワード	2
`int`	ワード	4
`long` *(SPARC)* `v8`	ワード	4
`long` *(SPARC)* `v9`	ダブルワード	8
`float` *(SPARC)*	ワード	4
`double` *(SPARC)*	ダブルワード *(SPARC)* ワード *(x86)*	8 *(SPARC)* 4 *(x86)*
`long double` *(SPARC)* `v8`	ダブルワード *(SPARC)* ワード *(x86)*	8 *(SPARC)* 4 *(x86)*
`long double` *(SPARC)* `v9`	クワドワード	16
`pointer` *(SPARC)* `v8`	ワード	4
`pointer` *(SPARC)* `v9`	クワドワード	8
`long long`[1]	ダブルワード *(SPARC)* ワード *(x86)*	8 *(SPARC)* 4 *(x86)*

1. `-Xc` モードでは無効です。

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、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。

(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。

(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。

(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ることはありません。

(6.5.2.2) 列挙型の値を表現するための整数型

int 型です。

修飾子 (G.3.10)

(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。

宣言子 (G.3.11)

(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。

文 (G.3.12)

(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。

プリプロセッサ指令 (G.3.13)

(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他の式のものと同一の数値を持ちます。

(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 *(SPARC) (x86)*。

(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限り、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中で検索します。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の中を検索します。

< > 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においてのみ開始されます。

(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

インクルード命令の引用符付きのファイル名はサポートされます。

(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。

(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。詳細については、16 ページの「プラグマ」を参照してください。

- `align` <整数> (<変数>[,<変数>])
- `does_not_read_global_data` (<関数>[,<関数>])
- `does_not_return` (<関数>[,<関数>])
- `does_not_write_global_data` (<関数>[,<関数>])
- `error_messages` (on|off|default, <タグ>... <タグ>)
- `fini` (<関数 1>[, <関数 2>..., <関数 *n*>])
- `ident` (<文字列>)
- `init` (<関数 1>[, <関数 2>..., <関数 *n*>])
- `inline` (<関数>[,<関数>])
- `int_to_unsigned` (<関数>)
- `MP serial_loop`
- `MP serial_loop_nested`
- `MP taskloop`
- `no_inline` (<関数>[,<関数>])
- `nomemorydepend`
- `no_side_effect` (<関数>[,<関数>])
- `opt_level` (<関数>[,<関数>])
- `pack`(*n*)
- `pipeloop`(*n*)
- `rarely_called` (<関数>[,<関数>])
- `redefine_extname` <旧外部参照名> <新外部参照名>
- `returns_new_memory` (<関数>[,<関数>])
- `unknown_control_flow` (<名前>[,<名前>]...)
- `unroll` (<展開関数>)
- `weak` (<シンボル 1> [=< シンボル 2>])

(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。

ライブラリ関数 (G.3.14)

(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。

(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: <文>. file <ファイル名>, line <番号>
```

- <文> は表明に失敗した文です
- <ファイル名> は障害を持ったファイルの名前です
- <番号> は障害が発生した行の番号です

(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 C-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

(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 C-7　ドメインエラーの場合の戻り値 (1 / 2)

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
DOMAIN	pow(0,neg)	0.0	-HUGE_VAL

表 C-7　ドメインエラーの場合の戻り値 (2 / 2)

エラー	数値演算関数	コンパイラモード	
		-Xs,-Xt	**-Xa,-Xc**
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

(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。

(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。

(7.7.1.1) signal 関数に対するシグナルの設定

表 C-8 に signal 関数が認識する各シグナルの意味を示します。

表 C-8　signal シグナルの意味

シグナル	No.	デフォルト	イベント
SIGHUP	1	終了	ハングアップ
SIGINT	2	終了	割り込み
SIGQUIT	3	コア	終了
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)

表 C-8　signal シグナルの意味 (続き)

シグナル	No.	デフォルト	イベント
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)
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

表 C-8　signal シグナルの意味 (続き)

シグナル	No.	デフォルト	イベント
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
SIGWAITINGT	32	無視	プロセスの LWP がブロックされた

(7.7.1.1) singal 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。

(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のか。

signal(sig、SIG_DFL); 相当のものは、常に実行されます。

(7.7.1.1) 'signal' 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。

(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。

(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。

(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。

(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。

(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限り、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。

(7.9.3) 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。ライン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) 出力の各行の待ち行列を作ります。

(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。

(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。

(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。

(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。

(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。

(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。

(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。

(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。

ロケール固有の動作 (G.4)

(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。

(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以下のようなロケール固有の性質があります。

(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、文字セットに拡張はありません。

(5.2.2) 印刷方向

常に左から右に印刷されます。

(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、10 進小数点はピリオド (.) です。

(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

345 ページの表 C-6 を参照してください。

(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。

(7.12.3.5) 時間と日付の書式

ロケール依存です。C ロケールでの月の名前は次のとおりです。

表 C-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 C-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 D

C99 でサポートされている機能

この付録では、ISO/IEC 9899:1999、Programming Language - C の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、273 ページの「-xc99[=o]」 を参照してください。

注 - コンパイラは、デフォルトでは以下で説明している 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` にあります) 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。

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、以下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", 1 行目: 型の組み合わせが不適切です
```

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%
```

_Pragma

_Pragma (*string-literal*) という形式の単項演算子の式は、以下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。以下の例で説明します。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
```

以下は、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%
```

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);
}
```

配列宣言子で使用可能な static およびその他の型修飾子

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のにも 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。

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));
```

その後で、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、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;
```

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。

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", 1 行目: 警告: 明示的な型が与えられていません
  "test.c", 3 行目: 警告: 明示的な型が与えられていません
example%
```

暗黙の int および暗黙の関数宣言の禁止

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", 3 行目: 警告: 暗黙的な関数宣言: printf
example%
```

for ループ文での宣言

6.8.5 繰り返し文

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)。

C99 のキーワード

6.4.1 キーワード

C99 の規格では、以下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

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。他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、54 ページの「制限付きポインタ」を参照してください。

__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。

引数の個数が可変するマクロ

6.10.3 マクロ置換

C コンパイラで、以下の形式の #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);
```

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));
```

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
```

静的関数用の inline 指示子

6.7.4 関数指示子

C99 では、関数指示子 inline が追加されました。inline は、内部リンクのある関数で完全に機能します。外部リンクで定義された関数の場合は、inline 関数指示子はインライン定義だけを作成します。関数の外部定義は作成されません。したがって、外部リンクのあるインライン関数へのポインタは、変換単位ごとに一意になり、評価結果は同一になりません。

// を使用したコードのコンパイル

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、文字列定数、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。

付録 E

パフォーマンスチューニング (*SPARC*)

この付録では SPARC プラットフォー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。

制限

処理速度を最適化すると、ほとんどのアプリケーションのパフォーマンスも向上します。しかし C ライブラリの中には、処理速度を最適化することができないものがあります。次にその例を挙げます。

- 整数算術ルーチン

  現在の SPARC V8 プロセッサでは、整数の乗算や除算の命令をサポートしています。しかし標準の C ライブラリルーチンがそれらの命令を使用すると、プログラムが SPARC V7 プロセッサ上で実行されている場合、カーネルエミュレーションの負荷のために処理速度が遅くなるか、また最悪の場合にはまったく実行できないことも予想されます。したがって、整数の乗算や除算の命令は標準の C ライブラリルーチンでは使用できません。

- ダブルワードのメモリーアクセス

  SPARC のダブルワードのロード命令やストア命令 (`ldd` および `std`) を使用すると、`memmove()` や `bcopy()` といったブロックのコピーや移動のルーチンの処理速度を飛躍的に上げることができます。しかし `memmove()` や `bcopy()` は、フレームバッファのような 64 ビットアクセスをサポートしていないメモリーにマップされたデバイスには適していません。したがって、`ldd` や `std` は標準 C ライブラリルーチンでは使用できません。

- メモリー割り当てのアルゴリズム

  C ライブラリルーチンの `malloc()` と `free()` は、処理速度や使用領域、およびコーディング時のエラーの起こしやすさ等を考慮した結果の妥協案として UNIX で実装されたものです。協調システム (buddy system) アルゴリズムにもとづくメモリーアロケータは、標準ライブラリより処理速度が速いものがほとんどですが、そのかわりに使用する領域も増えてしまいがちです。

`libfast.a` ライブラリ

ライブラリ `libfast.a` は標準 C ライブラリ機能バージョンの処理速度を上げたものです。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。

1. `libfast.a` を使用する場合

   - アプリケーションの 1 つのバイナリを SPARC V7 と V8 の両方のプラットフォームで実行しなければならないのに整数の乗算や除算の性能が重要である場合。

     重要なルーチン: `.mul`、`.div`、`.rem`、`.umul`、`.udiv`、`.urem`

   - メモリー割り当てのパフォーマンスが重要で、通常最も多く割り当てられるメモリーのサイズが 2 の階乗に近い場合。

     重要なルーチン: `malloc()`、`free()`、`realloc()`

   - ブロックの移動またはフィルのルーチンのパフォーマンスが重要である場合。

     重要なルーチン: `bcopy()`、`bzero()`、`memcpy()`、`memmove()`、`memset()`

2. `libfast.a` を使用してはいけない場合

   - アプリケーションが、64 ビットのメモリー操作をサポートしていない入出力デバイスへのユーザーモードのメモリーマップされたアクセスを必要とする場合。
   - アプリケーションがマルチスレッド対応である場合。

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 F

K&R Sun C と Sun ISO C の違い

この付録では、従来の K&R Sun C と Sun ISO C の違いを説明します。

詳細については、1 ページの「準拠規格」を参照してください。

K&R Sun C と Sun ISO C との間の非互換性

表 F-1　K&R Sun C と Sun ISO C との非互換性

項目	Sun C (K$R)	Sun ISO C
`main()` の `envp` 引数	`main()` の 3 番目の引数として `envp` を使用できる。	3 番目の引数として使用できるが、この使用法は厳密には ISO C 規格に準拠しない。
キーワード	識別子 `const`、`volatile`、`signed` を普通の識別子として扱う。	`const`、`volatile`、`signed` はキーワードである。
ブロック内の `extern` と `static` 関数宣言	これらの関数宣言をファイルスコープに拡張する。	ISO 規格は、ブロックスコープ関数宣言がファイルスコープに拡張されることを保証しない。
識別子	識別子でドル記号 (`$`) 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`

表 F-1　K&R Sun C と Sun ISO C との非互換性

項目	Sun C (K$R)	Sun ISO C
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。
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/short` 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` または `union` のメンバーの名前空間	メンバー選択演算子を使用する `struct`、`union`、および算術型は、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	`lvalue` としてのキャストをサポートする。 例 : `(char *)ip = &char;`	この機能はサポートしない。

暗黙の `int` 宣言	明示的な型指定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。
	例 :
	`num; /* num は暗黙の int */`
	`int num2; /* num2 は明示的に宣言された int */`
		`num;` 宣言 (明示的な型指定子 `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。 例 : `int;`	タグを除いて、空の宣言を使用できない。

表 F-1　K&R Sun C と Sun ISO C との非互換性

項目	Sun C (K$R)	Sun ISO C

型定義の型指定子	typedef 宣言で unsigned、short、long などの型指定子を使用できる。
	例 :
	`typedef short small;`
	`unsigned small x;`
		typedef 宣言は型指定子で変更できない。

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他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。他の型は未定義。

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 型指定子は、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、uion 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 指令の後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。

表 F-1　K&R Sun C と Sun ISO C との非互換性

項目	Sun C (K$R)	Sun ISO C
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 PATE(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。例: `#define PASTE(A,B) A##B` さらに、Sun ISO C プリプロセッサは Sun C の方法を認識しない。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。

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)`
	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)`
		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。例:
		`struct {int a[3]; int b;} \`
		`w[] = {{1},2};`
		は、次を生成する。
		
		`sizeof(w) = 32`
		`w[0].a = 1, 0, 0`
		`w[0].b = 0`
		`w[1].a = 2, 0, 0`
		`w[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。

表 F-1　K&R Sun C と Sun ISO C との非互換性

項目	Sun C (K$R)	Sun ISO C

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'`

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他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。例:
	`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。

表 F-1　K&R Sun C と Sun ISO C との非互換性

項目	Sun C (K$R)	Sun ISO C
名前空間	struct、union、enum のタグ、struct、union、enum のメンバー、および、その他すべてのうち 3 つだけを識別する。	ラベル名、タグ (キーワード struct、union、enum の後に続く名前)、struct、union、enum のメンバー、および、通常の識別子のうち 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。	
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。 例: `wchar_t wc = L'x';`	この構文をサポートする。
'\a' と '\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。 例: `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。

表 F-1　K&R Sun C と Sun ISO C との非互換性

項目	Sun C (K$R)	Sun ISO C
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。
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__	

キーワード

次の表は、ISO C 規格、Sun ISO C コンパイラ、および Sun 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 F-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[1]
return	short	signed	sizeof
static	struct	switch	typedef
union	unsigned	void	volatile
while			

1. `-xc99=%all` でのみ定義。

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、Sun C のキーワードのリストを示します。

表 F-3　Sun 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	

付録G

OpenMP の実装固有情報

この付録では、*OpenMP C and C++ Application Program Interface Version 1.0 - October 1998* (`www.openmp.org` から入手可能) の実装固有の詳細情報について説明します。

- `OMP_SCHEDULE` 環境変数を明示的に定義していない場合は、この実装は `schedule(runtime)` によるループに静的スケジューリングを使用します。
- 明示的に定義されたスケジュール句が存在しない場合、デフォルトは静的スケジューリングになります。
- `opm_set_num_threads` 関数または `OMP_NUM_THREADS` 環境変数を使用してチームのスレッド数を明示的に指定しない場合、デフォルトは 1 になります。
- `omp_set_dynamic` 関数または `OMP_DYNAMIC` 環境変数を使用して、スレッドの動的な調整を有効にするかどうかを明示的に指定しない場合、デフォルトにより、動的な調整が有効になります。
- 入れ子にされた並列化機能はサポートされておらず、デフォルトで無効になっています。

索引

D

E

F

G

I

J

M

N

O

P

R

S

T

U

V

W

X

Z

あ

け

こ

さ

し

す

せ

そ

た

ち

て