dbx コマンドによるデバッグ

Forte Developer 6 update 2
(Sun WorkShop 6 update 2)

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303
U.S.A. 650-960-1300

Part No. 816-0885-01
2001 年 8 月 Revision A

原典： *Debugging a Program With dbx*
Part No: 806-7983-10
Revision A

製品名の変更について

Sun は新しい開発製品戦略の一環として、Sun の開発ツール群の製品名を Sun WorkShop™ から Forte™ Developer に変更いたしました。製品自体の内容に変更はなく、従来通りの高品質をお届けいたします。

これまでの Sun の主力製品である基本プログラミングツールに、Forte Fusion™ や Forte™ for Java™ といった Forte 開発ツールの得意とする、マルチプラットフォームおよびビジネスアプリケーション実装の機能を盛り込むことで、より広範囲できめ細かな製品ラインが完成されました。

WorkShop 5.0 で使用されていた名称と、Forte Developer 6 で使用される新しい名称の対応については、以下の表をご覧ください。

旧名称	新名称
Sun Visual WorkShop™ C++	Forte™ C++ Enterprise Edition 6
Sun Visual WorkShop™ C++ Personal Edition	Forte™ C++ Personal Edition 6
Sun Performance WorkShop™ Fortran	Forte™ for High Performance Computing 6
Sun Performance WorkShop™ Fortran Personal Edition	Forte™ Fortran Desktop Edition 6
Sun WorkShop Professional™ C	Forte™ C 6
Sun WorkShop™ University Edition	Forte™ Developer University Edition 6

製品名の変更に加えて、次の 2 つの製品について大きな変更があります。

- Forte for High Performance Computing には Sun Performance WorkShop Fortran に含まれていたすべてのツール、および C++ コンパイラが含まれます。したがって、High Performance Computing のユーザーは開発用に 1 つの製品だけを購入すれば済むことになります。
- Forte Fortran Desktop Edition は以前の Sun Performance WorkShop Personal Edition と同じです。ただし、この製品に含まれる Fortran コンパイラでは、自動並列化されたコード、および明示的な指令に基づいた並列コードは生成できません。この機能は Forte for High Performance Computing に含まれる Fortran コンパイラでは使用できます。

Sun の開発製品を引き続きご利用いただきましてありがとうございます。今後もみなさまのご要望にお応えする製品をお届けできるよう努力してまいります。

目次

はじめに

dbx は、対話型でソースレベルの、コマンド行ベースのデバッグツールです。このマニュアルは、Fortran、C、または C++ による開発経験を持ち、Solaris™ オペレーティング環境と UNIX® コマンドについてある程度の知識があり、dbx コマンドを使用してアプリケーションのデバッグを行いたいプログラマを対象にしています。このマニュアルでは、Sun WorkShop™「デバッグ」ウィンドウを使用して実行できる、同様のデバッグ操作についても参照することができます。

内容の紹介

このマニュアルは次の章と付録から構成されています。

第 1 章では、アプリケーションをデバッグするための dbx の使い方の基本を説明します。

第 2 章では、デバッグセッションの開始方法と停止方法、コンパイル時のオプション、およびデバッグ実行の全部または一部を保存して、それを後で再現する方法について説明します。

第 3 章では、dbx 環境変数を調整してデバッグ環境の特定の属性をカスタマイズする方法と、初期化ファイル .dbxrc を使用してセッションを通じて変更内容と調整内容を保存する方法について説明します。

第 4 章では、コードの表示、関数の表示、シンボルの検索、および変数、メンバー、型、クラスの参照について説明します。

第 5 章では、dbx でプログラムを実行、接続、続行、停止、および再実行する方法と、プログラムコードのステップ実行について説明します。

第 6 章では、ブレークポイントとトレースの設定、削除、一覧方法、およびウォッチポイントの設定方法などの一般的な操作について説明します。

第 7 章では、コールスタックの検証方法、およびコアファイルのデバッグ方法について説明します。

第 8 章では、データの評価方法、式の値や変数、データ構造などの表示方法、および式への値の割り当て方法について説明します。

第 9 章では、開発段階のアプリケーションにある実行時エラーを自動的に検出できる機能について説明します。

第 10 章では、Sun WorkShop でプログラムをデバッグするときに、データをグラフィック表示させる方法について説明します。

第 11 章では、dbx を終了せずにソースファイルを修正し、ファイルを再コンパイルして、プログラムの実行を続ける方法について説明します。

第 12 章では、dbx の `thread` コマンドを使用してスレッドに関する情報を検索する方法を説明します。

第 13 章では、子プロセスを作成するプロセスをデバッグするのに役立ついくつかの dbx 機能について説明します。

第 14 章では、dbx を使用してシグナルを処理する方法を説明します。

第 15 章では、dbx による C++ テンプレートのサポートについて説明します。また、C++ 例外を処理するために使用可能なコマンドと、dbx がこれらの例外をどのように処理するかについて説明します。

第 16 章では、Fortran で使用するいくつかの dbx 機能について説明します。

第 17 章では、イベント管理およびプロセス制御の各コマンドを機械命令レベルで使用する方法と指定アドレスのメモリーの内容を表示する方法、およびソース行を対応する機械命令とともに表示する方法について説明します。

第 18 章では、ksh-88 と dbx コマンド言語の間の違いについて説明します。

第 19 章では、動的にリンクされた共有ライブラリを使用するプログラムに対する dbx のデバッグサポートについて説明します。

付録 A では、プログラムを変更する dbx コマンド、および dbx のもとでプログラムを実行した場合の動作について説明します。

付録 B では、イベントの管理方法について説明します。また、デバッグ中のプログラムで特定のイベントが発生した場合に特定のアクションを実行するための、dbx の一般的な機能についても説明します。

付録 C では、すべての dbx コマンドの構文と機能について説明します。

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。

書体または記号	意味	例
`AaBbCc123`	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コーディング例。	`.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。 `machine_name% You have mail.`
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して表わします。	`machine_name% su` `Password:`
AaBbCc123 または ゴシック	コマンド行の可変部分。実際の名前または実際の値と置き換えてください。	`rm` *filename* と入力します。 `rm` ファイル名 と入力します。
『 』	参照する書名を示します。	『SPARCstorage Array ユーザーマニュアル』

書体または記号	意味	例
「」	参照する章、節、または、強調する語を示します。	第 6 章「データの管理」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ行幅を超える場合、バックスラッシュは、継続を示します。	`machinename% grep '^#define \` `  XV_VERSION_STRING'`
➤	階層メニューのサブメニューを選択することを示します。	作成:「返信」➤「送信者へ」

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)	#

サポートしているプラットフォーム

この Sun WorkShop™ リリースでは、Solaris™ SPARC™ プラットフォーム版と Solaris™ Intel プラットフォーム版をオペレーティング環境とするバージョン 2.6、7、および 8 をサポートしています。

Sun WorkShop の開発ツールとマニュアルページへのアクセス

Sun WorkShop 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ていません。SunWorkShop 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 SunWorkshop コンポーネントディレクトリを必要とします。SunWorkshop マニュアルページにアクセスするには、PATH 環境変数に SunWorkShop マニュアルページが必要です。

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、『Sun WorkShop 6 update 2 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 Sun WorkShop 6 update 2 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。 Sun WorkShop 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。

Sun WorkShop 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 Sun WorkShop 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. 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 Sun WorkShop 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。

Sun WorkShop のコンパイラとツールにアクセスするために PATH 環境変数を設定するには以下を実行します。

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

Sun WorkShop マニュアルページへのアクセス方法

Sun WorkShop 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. 次のように入力して、workshop マニュアルページを表示します。

```
% man workshop
```

2. 出力された場合、内容を確認します。

   workshop(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 Sun WorkShop 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
```

Sun WorkShop マニュアルへのアクセス

Sun WorkShop 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。

  Netscape™ Communicator 4.0 または互換性がある Netscape バージョンのブラウザで次のファイルにポイントします。

  ```
  /opt/SUNWspro/docs/ja/index.html
  ```

  製品ソフトウェアが /opt ディレクトリにインストールされていない場合は、システム上でこのディレクトリに相当するパスをシステム管理者に問い合わせてください。

- マニュアルは、docs.sun.com の Web サイトで入手できます。

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。

関連マニュアル

次の表では、docs.sun.com の Web サイトで利用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
Forte Developer 6 / Sun WorkShop 6 リリースマニュアル	Sun WorkShop 6 マニュアルの概要	Sun WorkShop 6で使用可能なマニュアルとそのアクセス方法について説明しています。
	Sun WorkShop の新機能	Sun WorkShop 6 の現在のリリースと以前のリリースでの新機能についての情報を記載しています。
	Sun WorkShop 6 リリースノート	インストールの詳細と Sun WorkShop 6 最終リリースの直前に判明した情報を記載しています。このマニュアルはコンポーネントごとの README ファイルにある情報を補足するものです。
Forte Developer 6 / Sun WorkShop 6	プログラムのパフォーマンス解析	新しい標本コレクタと標本アナラザの使い方について説明しています (上級者向けのプロファイリング事例と説明付き)。コマンド行解析ツール `er_print`、および UNIX プロファイルツール `prof`、 `gprof`、`tcov` についての情報も含んでいます。

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
	Sun WorkShop の概要	Sun WorkShop 統合プログラミング環境の基本的なプログラム開発機能について説明しています。
Forte C 6 / Sun WorkShop 6 Compilers C	C ユーザーズガイド	C コンパイラオプション、サン固有の機能 (プラグマ、 `lint` ツール、並列化、64 ビットオペレーティングシステムへの移行および ANSI/ISO 準拠 C) について説明しています。
Forte C++ 6 / Sun WorkShop 6 Compilers C++	C++ ユーザーズガイド	コマンド行オプションとコンパイラの使い方についての情報を記載しています。付録 C では、すべての dbx コマンドの構文と機能について説明します。
	C++ 移行ガイド	コードを本バージョンの Sun WorkShop C++ コンパイラに移行する方法について説明しています。
Forte™ for High Performance Computing 6 / Sun WorkShop 6 Compilers Fortran 77/95	Fortran ライブラリ・リファレンス	Fortran コンパイラによって提供されるライブラリルーチンの詳細について説明しています。
	Fortran プログラミングガイド	入出力、ライブラリ、プログラム分析、デバッグおよびパフォーマンスに関連する内容を記述しています。
	Fortran ユーザーズガイド	コマンド行オプションとコンパイラの使い方についての情報を記載しています。
	FORTRAN 77 言語リファレンス	Fortran 77 言語の包括的な参照情報を記載しています。

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
	Fortran 95 区間演算プログラミングリファレンス	Fortran 95 コンパイラによってサポートされる組み込み INTERVAL データについて説明しています。
数値計算ガイド	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。
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。

Sun のマニュアルの注文

製品マニュアルは docs.sun.com Web サイトまたは Fatbrain.com インターネットブックストアを通じて米国 Sun Microsystems,Inc. に直接注文できます。
Fatbrain.com の Sun Documentation Center へは次の URL でアクセスできます。

http://www.fatbrain.com/documentation/sun

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

dbx の概要

dbx は、対話型でソースレベルのコマンド行デバッグツールです。 dbx を使用すれば、プログラムを制御下に置いた状態で実行し、停止したプログラムの状態を調べることができます。このツールにより、プログラムの動的な実行を完璧に制御できるほか、パフォーマンスデータとメモリの使用状況の収集、メモリーアクセスの監視、およびメモリーリークの検出も行えます。

この章では、dbx によるアプリケーションのデバッグの基礎について説明します。 この章は、次の内容で構成されます。

- デバッグを目的としてコード をコンパイルする
- dbxを起動して プログラムを読み込む
- プログラム をdbxで実行する
- dbxを使用してプログラムをデバッグする
- dbx を終了する
- dbx オンラインヘルプにアクセスする

デバッグを目的としてコード をコンパイルする

dbx でソースレベルのデバッグを行えるようにプログラムを作成するには、-g オプションを付けてプログラムをコンパイルする必要があります。このオプションは、C、C++、Fortran 77、および Fortran 95 の各コンパイラで利用できます。詳細については、1 ページの「デバッグを目的としてコード をコンパイルする」を参照してください。

dbxを起動して プログラムを読み込む

dbxを 起動するには、シェルプロンプトでdbx を入力します。

```
$ dbx
```

dbx を起動してデバッグ対象プログラムを読み込むには、以下を入力します。

```
$ dbx program_name
```

dbx コマンドを使用すると、dbx を起動し、プロセス ID で指定した実行中プロセスに接続できます。

```
$ dbx - process_id
```

プロセスの ID がわからない場合、ps コマンドを使用して ID を決定し、dbx コマンドを使用してプロセスに接続します。次に例を示します。

```
$ ps -def | grep Freeway
   fred 25458   590  0 16:26:40 pts/8    0:00 grep Freeway
   fred 25458   590  0 16:26:40 pts/8    0:00 ./Freeway
$ dbx - 25448
- の読み込み中
ld.so.1 の読み込み中
libXm.so.4 の読み込み中
libgen.so.1 の読み込み中
libXt.so.4 の読み込み中
libX11.so.4 の読み込み中
libsocket.so.1 の読み込み中
libCrun.so.1 の読み込み中
libm.so.1 の読み込み中
libw.so.1 の読み込み中
libc.so.1 の読み込み中
libSM.so.6 の読み込み中
libICE.so.6 の読み込み中
libXext.so.0 の読み込み中
libnsl.so.1 の読み込み中
libdl.so.1 の読み込み中
libmp.so.2 の読み込み中
libc_psr.so.1 の読み込み中
ja.so.2 の読み込み中
methods_ja.so.2 の読み込み中
xlibi18n_ja.so.2 の読み込み中
xomLTRTTB.so.2 の読み込み中
ximp40.so.2 の読み込み中
プロセス 25448 に接続しました。
_libc_poll で停止しました 0xfef148b8 で
0xfef148b8: _libc_poll+0x0004:ta      0x8
現関数 :main
   48   XtAppMainLoop(app_context);
(dbx)
```

dbx コマンドと起動オプションの詳細については、296 ページの「dbx コマンド」、および dbx(1) マニュアルページを参照するか、dbx -h と入力してください。

すでに dbx を実行している場合、debug コマンドにより、デバッグ対象プログラムを読み込むか、デバッグしているプログラムを別のプログラムに切り替えることができます。

```
(dbx) debug program_name
```

すでに dbx を実行している場合、debug コマンドにより、dbx を実行中プロセスに接続することもできます。

```
(dbx) debug program_name process_ID
```

debug コマンドの詳細については、298 ページの「debug コマンド」を参照してください。

プログラム をdbxで実行する

dbx に最後に読み込んだプログラムを実行するには、run コマンドを使用します。引数を付けないで run コマンドを最初に入力すると、引数なしでプログラムが実行されます。引数を引き渡したりプログラムの入出力先を切り替えたりするには、次の構文を使用します。

```
run [ arguments ] [ < input_file ] [ > output_file ]
```

たとえば、

```
(dbx) run -h -p < input > output
実行中: a.out
(プロセス id 1234)
実行完了。終了コードは 0 です。
(dbx)
```

引数を付けないで run コマンド を繰り返し使用した場合、プログラムは前回の run コマンドの引数や入力先を使用します。rerun コマンドを使用すれば、オプションをリセットできます。run コマンドの詳細については、332 ページの「run コマンド」を参照してください。rerun コマンドの詳細については、330 ページの「rerun コマンド」を参照してください。

アプリケーションは、最後まで実行され、正常に終了するかもしれません。ブレークポイントが設定されている場合には、ブレークポイントでアプリケーションが停止するはずです。アプリケーションにバグが存在する場合は、メモリーフォルトまたはセグメント例外のため停止することがあります。

dbxを使用してプログラムをデバッグする

プログラムをデバッグする理由としては、以下が考えられます。

- クラッシュする場所と理由をつきとめるため クラッシュの原因をつきとめる方法としては、以下があります。
  - dbx でプログラムを実行する。dbx はクラッシュの発生場所をレポートします。
  - コアファイルを調べ、スタックトレースをチェックする (6 ページの「コアファイルを チェックする」、10 ページの「呼び出しスタックを確認する」参照)
- 以下の方法で、プログラムが不正な実行結果を出力する原因を判定します。
  - ブレークポイントを設定して実行を停止することにより、プログラムの状態をチェックし変数の値を調べる (7 ページの「ブレークポイントを設定する」 、11 ページの「変数を調べる」参照)
  - ソースコードを 1 行ずつステップ実行することによって、プログラムの状態がどのように変わっていくかを監視する (9 ページの「プログラムをステップ実行する」参照)
- メモリーリークやメモリー管理問題を見つける方法としては、以下があります。 実行時検査を行えば、メモリーアクセスエラーやメモリーリークエラーといった実行時エラーを確認できるとともに、メモリー使用状況を監視できる (12 ページの「メモリーアクセス問題とメモリーリークを検出する」参照)。

コアファイルを チェックする

プログラムがどこで クラッシュするかをつきとめるには、プログラムがクラッシュしたときのメモリーイメージであるコアファイルを調べるとよいでしょう。 where コマンドを使用すれば (355 ページの「where コマンド」参照)、コアをダンプしたときのプログラムの実行場所がわかります。

コアファイルを デバッグするには、以下を入力します。

```
$ dbx program_name core
```

あるいは

```
$ dbx - core
```

次の例では、プログラムがセグメント例外でクラッシュし、コアダンプが作成されています。 ユーザーは dbx を起動し、コアファイルを読み込みます。 次に、where コマンドを使用してスタックトレースを表示させます。これによって、ファイルfoo.cの9行目でクラッシュが発生したことがわかります。

```
$% dbx a.out core
a.out の読み込み中
core ファイルハンドラの読み込みに成功しました
ld.so.1 の読み込み中
libc.so.1 の読み込み中
libdl.so.1 の読み込み中
libc_psr.so.1 の読み込み中
プログラムはシグナル SEGV （フォルトのアドレスにマッピングしていません）
により停止しました
現関数 :main
    7       printf("string '%s' is %d characters long\n", msg,
strlen(msg));
(dbx) where
  [1] strlen(0x0, 0x0, 0xff334ab8, 0x7efefeff, 0x81010100,
0xff0000)、アドレス
0xff2b6d4c),
=>[2] main()、"core.c" の 7 行目
(dbx)
```

コアファイルのデバッグの詳細については、16 ページの「既存のコアファイルのデバッグ」を参照してください。 呼び出しスタックの詳しい使い方については、10 ページの「呼び出しスタックを確認する」を参照してください。

注 – プログラムが共有ライブラリと動的にリンクされている場合、できれば、コアファイルが作成されたオペレーティング環境でコアファイルをデバッグしてください。 別のオペレーティング環境で作成されたコアファイルをデバッグする方法については、17 ページの「一致しないコアファイルのデバッグ」を参照してください。

ブレークポイントを設定する

ブレークポイント とは、一時的にプログラムの実行を停止し、コントロールを dbx に渡す場所のことです。 バグが存在するのではないかと思われるプログラム領域にブレークポイントを設定します。 プログラムがクラッシュした場合、クラッシュが発生した箇所をつきとめ、その部分の直前のコードにブレークポイントを設定します。

プログラムがブレークポイントで停止したとき、プログラムの状態と変数の値を調べることができます。dbx では、さまざまな種類のブレークポイントを設定できます (第 6 章参照)。

最も単純なブレークポイントは、停止ブレークポイントです。 停止ブレークポイントを使用すれば、関数や手続きの中で停止 させることができます。 たとえば、main 関数が呼び出されたときに停止させる方法は次のとおりです。

```
(dbx) stop in main
(2) stop in main
```

stop in コマンドの詳細については、83 ページの「ソース行で when ブレークポイントを設定する」 と337 ページの「step コマンド」を参照してください。

また、特定のソースコード行で停止 するようにブレークポイントを設定することもできます。 たとえば、ソースファイルt.cの13行目で停止させる方法は次のとおりです。

```
(dbx) stop at t.c:13
(3) stop at "t.c":13
```

stop at コマンドの詳細については、83 ページの「ソース行で when ブレークポイントを設定する」 と337 ページの「step コマンド」を参照してください。

停止場所を確定するには、file コマンドで現在のファイルを設定し、list コマンドで停止場所とする関数を表示させます。 次に、stop at コマンドを使用してソース行にブレークポイントを設定します。

```
(dbx) file t.c
(dbx) list main
10    main(int argc, char *argv[])
11    {
12        char *msg = "hello world\n";
13        printit(msg);
14    }
(dbx) stop at 13
(4) stop at "t.c":13
```

ブレークポイントで停止したプログラムの実行を続行するには、cont コマンドを使用します (67 ページの「プログラムを継続する」 、288 ページの「cont コマンド」参照)。

現在のブレークポイントのリストを表示するには、status コマンドを使用します。

```
(dbx) status
(2) stop in main
(3) stop at "t.c":13
```

ここでプログラムを実行すれば、最初のブレークポイントでプログラムが停止します。

```
(dbx) run
...
main で停止しました 行番号 12 ファイル "t.c"
12        char *msg = "hello world\n";
```

プログラムをステップ実行する

ブレークポイントで停止した後、プログラムを1ソース行ずつステップ 実行すれば、あるべき正しい状態と実際の状態とを比較できます。 それには、step コマンドと next コマンドを使用します。 いずれのコマンドもプログラムのソース行を1行実行し、その行の実行が終了すると停止します。 この 2 つのコマンドは、関数呼び出しが含まれているソース行の取り扱い方が違います。 step コマンドは関数にステップインしますが、next コマンドは関数をステップオーバーします。step up コマンドは、現在実行している関数が、自身を呼び出した関数に制御を戻すまで実行され続けます。

注 – printfのようなライブラリ関数をはじめとする一部の関数は-g を使ってコンパイルされていないことがあります。dbx は、このような関数にはステップインできません。 このような場合、step と next は同じような動作を示します。

以下は、step コマンドと next コマンド、および 7 ページの「ブレークポイントを設定する」に設定されたブレークポイントの使用例です。

```
(dbx) stop at 13
(3) stop at "t.c":13
(dbx) run
実行中: a.out
(プロセス id 24999)
main で停止しました 行番号 13 ファイル "t.c"
   13           printit(msg);
(dbx) next
Hello world
main で停止しました 行番号 14 ファイル "t.c"
   14   }

(dbx) run
実行中: a.out
(プロセス id 25000)
main で停止しました 行番号 13 ファイル "t.c"
   13           printit(msg);
(dbx) step
printit で停止しました 行番号 3 ファイル "t.c"
    3           printf("%s\n", str);
(dbx) step up
Hello world
printit 戻り値 134524
main で停止しました 行番号 13 ファイル "t.c"
   13           printit(msg);
(dbx)
```

プログラムのステップ実行の詳細については、66 ページの「プログラムのステップ実行」を参照してください。 step コマンドと next コマンドの詳細については、337 ページの「step コマンド」 と 321 ページの「next コマンド」を参照してください。

呼び出しスタックを確認する

呼び出し スタックは、呼び出された後呼び出し側にまだ戻っていない、現在活動状態にあるルーチンすべてを示します。 呼び出しスタックには、呼び出された順序で関数とその引数が一覧表示されます。 プログラムフローのどこで実行が停止し、この地点までどのように実行が到達したのかが、スタックトレースに示されます。 スタックトレースは、プログラムの状態を最も簡潔に記述したものです。

スタックトレースを表示するには、where コマンドを使用します。

```
(dbx) stop in printf
(dbx) run
(dbx) where
  [1] printf(0x20d78, 0x20d7c, 0x0, 0x0, 0x0, 0x0),アドレス
0xff300b74
=>[2] printit(str = 0x20d7c "Hello World\n"), "t.c" の 3 行目
  [3] main(argc = 1, argv = 0xffbef0e4), "t.c" の 13 行目
```

-g オプションを使ってコンパイルされた関数の場合は引数の名前と型がわかっているので、正確な値が表示されます。 デバッグ情報を持たない関数の場合、16進数が引数として表示されます。 これらの数字に意味があるとは限りません。 たとえば、上記のスタックトレースのフレーム1は、$i0 から $i5のSPARC入力レジスタの内容を示しています。9 ページの「プログラムをステップ実行する」の例のprintfに引き渡された引数は2つだけなので、内容に意味があるレジスタは$i0 から $i1 までだけです。

-g オプションを使ってコンパイルされなかった関数の中でも停止することができます。 こういった関数の中で停止する場合、dbx は-g オプションを使ってコンパイルされた関数を持つフレームの中で最初のものをスタック内で検索し(上記の例では printit())、これに現在のスコープを設定します (46 ページの「スコープ」参照)。これは、矢印記号 (=>) によって示されます。

呼び出しスタックの詳細については、第 7 章を参照してください。

変数を調べる

プログラムの状態に関する十分な情報がスタックトレースに含まれているかもしれませんが、他の変数の値を調べる 必要が生じることも考えられます。 print コマンドは式を評価し、式の型に基づいて値を印刷します。 以下は、単純なC式の例です。

```
(dbx) print msg
msg = 0x20a84 "Hello world"
(dbx) print msg[0]
msg[0] = 'h'
(dbx) print *msg
*msg = 'h'
(dbx) print &msg
&msg = 0xefffe8b4
```

データ変更ブレークポイントを使用すれば、変数と式の値を追跡できます (77 ページの「データ変更ブレークポイントを設定する」参照)。 たとえば、変数countの値が変更されたときに実行を停止するには、以下を入力します。

```
(dbx) stop change count
```

メモリーアクセス問題とメモリーリークを検出する

実行時検査は、メモリーアクセス検査、およびメモリー使用状況とリーク検査の2部で構成されます。 アクセス検査は、デバッグ対象アプリケーションによるメモリーの使用がまちがっていないかどうかをチェックします。 メモリー使用状況とメモリーリークの検査では、未処理のヒープ空間すべてを記録し、必要に応じて、またはプログラム終了時に、利用できるデータ空間の走査および参照なしの空間の確認を行います。

メモリーアクセス検査、およびメモリー使用状況とメモリーリークの検査は、checkコマンドによって使用可能にします。 メモリーアクセス検査をオンにするには、以下を入力します。

```
(dbx) check -access
```

メモリー使用状況とメモリーリークの検査をオンにするには、以下を入力します。

```
(dbx) check -memuse
```

実行時検査をオンにしたら、プログラムを実行します。 プログラムは正常に動作しますが、それぞれのメモリーアクセスが発生する直前にその妥当性チェックが行われるため、動作速度は遅くなります。 無効なアクセスを検出すると、dbx はそのエラーの種類と場所を表示します。 現在のスタックトレースを取り出すにはwhereなどのdbxコマンド、変数を調べるにはprint を使用します。

実行時検査の詳細については、第 9 章を参照してください。

dbx を終了する

dbx セッションは、dbx を起動してから終了 するまで継続されます。dbx セッションのあいだ、任意の数のプログラムを連続してデバッグできます。

dbx セッションを終了するには、dbx プロンプトに quit と入力します。

```
(dbx) quit
```

dbx を起動し、*process_id* オプションで実行中プロセスに接続した場合、デバッグセッションを終了してもプロセスは存続します。dbx は、セッションの終了前に暗黙の detach を実行します。

dbx の終了の詳細については、25 ページの「デバッグセッションを終了する」を参照してください。

dbx オンラインヘルプにアクセスする

dbx には、help コマンドでアクセスできるヘルプファイルが含まれています。:

```
(dbx) help
```

dbx のオンラインヘルプは、Sun WorkShop オンラインヘルプにも含まれています。表示するには、Sun WorkShop メインウィンドウの「ヘルプ」>「ヘルプの目次」を選択します。

第2章

dbx の起動

この章では、dbx デバッグセッションを開始、実行、保存、復元、および終了する方法について説明します。この章の内容は次のとおりです。

この章の内容は次のとおりです。

- デバッグセッションを開始する
- 起動属性の設定
- 最適化コードのデバッグ
- デバッグセッションを終了する
- デバッグ実行の保存と復元

デバッグセッションを開始する

dbx の起動方法は、デバッグの対象、現在の作業ディレクトリ、dbx で必要な実行内容、dbx の習熟度、および dbx 環境変数を設定したかどうかによって異なります。

dbx セッションを開始する最も簡単な方法は、dbx コマンドをシェルプロンプトで入力する方法です。

```
$ dbx
```

シェルから dbx を起動し、デバッグするプログラムを読み込むには、次のように入力します。

```
$ dbx program_name
```

dbx コマンドおよび起動オプションについての詳細は、Sun WorkShop™ オンラインヘルプ、および dbx(1) マニュアルページの「dbx コマンドの使い方」の「dbx コマンド」および「dbx の起動」を参照してください。

Sun WorkShop の「デバッグ」ウィンドウを使用してプログラムのデバッグを開始すると、dbx は自動的に起動します (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「現在のプログラムのデバッグ」および「新しいプログラムのデバッグ」を参照してください)。

既存のコアファイルのデバッグ

コアダンプしたプログラムが共有ライブラリと動的にリンクしている場合、それが作成された同じオペレーティング環境でコアファイルをデバッグすることが重要です。dbx では、一致しないコアファイル (たとえば、バージョンまたはパッチレベルの異なる Solaris オペレーション環境で生成されたコアファイル) のデバッグに対しサポートが制限されます。

同じオペレーティング環境でのコアファイルのデバッグ

コアファイルをデバッグするには、次のように入力します。

```
$ dbx program_name core
```

コアファイルが現在のディレクトリに存在しない場合、パス名を指定できます (/tmp/core など)。

プログラムがコアをダンプしたときにどこで実行されていたかを確認するには、where コマンド (355 ページの「where コマンド」を参照) を使用してください

コアファイルをデバッグする場合、変数と式を評価して、プログラムがクラッシュした時点での値を確認することもできますが、関数呼び出しを行なった式を評価することはできません。ステップ実行したりブレークポイントを設定することはできません。

一致しないコアファイルのデバッグ

特定のシステム (コアホスト) で作成されたコアファイルを、デバッグのためにそのファイルを別のマシン (dbx ホスト) に読み込む場合があります。この場合、ライブラリに 関する 2 つの問題が発生します。

- コアホストのプログラムで使用される共有ライブラリが dbx ホストのライブラリと異なる場合があります。ライブラリに関して正しいスタックトレースを取得するには、dbx ホストでもオリジナルのライブラリを利用できなくてはなりません。
- dbx は、システム上の実行時リンカーとスレッドのライブラリについて実装詳細を分かりやすくするために、/usr/lib に配置されているライブラリを使用します。また、dbx が実行時リンカーのデータ構造とスレッドのデータ構造を理解できるように、コアホストからそれらのシステムライブラリを提供する必要性が出てくることもあります。

ユーザーライブラリとシステムライブラリは、パッチや主要な Solaris オペレーティング環境のアップグレードで変更できるため、収集したコアファイルで dbx を実行する前にパッチをインストールした場合など、この問題が同一ホストでも発生する可能性があります。

dbx は、一致しないコアファイルを読み込むと、次のエラーメッセージを 1 つ以上表示します。

```
dbx: コアファイル読み取りエラー: アドレス 0xff3d0914 は利用できません
dbx: 警告: could not initialize librtld_db.so.1 -- trying
libDP_rtld_db.so
dbx: %d のスレッド情報を取得できません 1 -- 一般的な libthread_db.so エラー
dbx: レジスタをフェッチしようとして失敗しました - スタックが破壊されました
dbx: (0xff363540) からのレジスタの読み取りに失敗しました -- デバッガは失敗しました
```

共有ライブラリ問題の回避

ライブラリ問題を回避し、一致しないコアファイルを dbx でデバッグするには、次の手順を実行します。

1. dbx 環境変数 core_lo_pathmap を on に設定します。
2. pathmap コマンドを使用して、コアファイルの正しいライブラリの配置場所を dbx に伝えます。

3. debug コマンドを使用して、プログラムとコアファイルを読み込みます。

たとえば、コアホストのルートパーティションが NFS を介してエクスポートされており、dbx ホストマシンの /net/core-host からアクセスできると仮定した場合、次のコマンドを使用して、プログラム prog とコアファイル prog.core をデバッグのために読み込みます。

```
(dbx) dbxenv core_lo_pathmap on
(dbx) pathmap /usr /net/core-host/usr
(dbx) pathmap /appstuff /net/core-host/appstuff
(dbx) debug prog prog.core
```

コアホストのルートパーティションをエクスポートしていない場合、手動でライブラリをコピーする必要があります。シンボリックリンクを再作成する必要はありません (たとえば、libc.so から libc.so.1 へのリンクを作成する必要はありません。libc.so.1 が利用可能であることだけを確認してください)。

注意点

ミスマッチコアファイルをデバッグする際に、次の点に注意してください。

- pathmap コマンドは '/' のパスマップを認識しないため、次のコマンドを使用できません。

  ```
  pathmap / /net/core-host
  ```

- pathmap コマンドの単一引数モードは、ロードオブジェクトのパス名を使用すると機能しません。そのため、2 つの引数をとる form-path to-path モードを使用してください。

- dbx ホストがコアホストと同一のバージョンまたはコアホストより最近のバージョンの Solaris オペレーティング環境を有している場合、コアファイルのデバッグが良好に機能する傾向にあります。ただし、これは必須ではありません。

- 必要となるシステムライブラリを次に示します。
  - ランタイムリンカーの場合：

    ```
    /usr/lib/librtld_db.so.1
    /usr/lib/sparcv9/librtld_db.so.1
    ```

  - スレッドライブラリの場合 (使用しているスレッドの実装に依存します)：

```
/usr/lib/libthread_db.so.1
/usr/lib/sparcv9/libthread_db.so.1
/usr/lib/lwp/libthread_db.so.1
/usr/lib/lwp/sparcv9/libthread_db.so.1
```

dbx が Solaris 7 または 8 オペレーティング環境で動作している場合、SPARC V9 バージョンが必要となります (コマンド `isalist` が sparcv9 を表示する場合)。これらのシステムライブラリは、ターゲットプログラムではなく dbx の一部として読み込まれて使用されるからです。

- スレッド化されたプログラムからコアファイルを調べていて、および `where` コマンドがスタックを表示しない場合、`lwp` コマンドを使用してみてください。次に例を示します。

```
(dbx) where
現スレッド : t@0
[1] 0x0(), at 0xffffffff
(dbx) lwps
o>l@1 シグナル SIGSEGV 現在の関数 _sigfillset()
(dbx) lwp l@1
(dbx) where
=>[1] _sigfillset(), "lo.c" の 2 行目
  [2] _liblwp_init(0xff36291c, 0xff2f9740, ...
  [3] _init(0x0, 0xff3e2658, 0x1, ...
...
```

スレッドスタックの欠如は、`thread_db.so.` に問題があることを示している場合があります。そのため、コアホストから正しい `libthread_db.so.1` ライブラリをコピーしてください。

詳細については、"16 ページの「既存のコアファイルのデバッグ」" を参照してください。

プロセス ID の使用

動作中のプロセスを dbx に接続できます。dbx コマンドに引数としてプロセス ID を指定します。

```
$ dbx program_name process_id
```

プログラムの名前を知らなくても、その ID を使用してプロセスに接続できます。

```
$ dbx - process_id
```

この場合、dbx はプログラムの名前を認識できないため、run コマンドの中でそのプロセスに引数を渡すことはできません。

詳細については、64 ページの「動作中のプロセスに dbx を接続する」を参照してください。

Sun WorkShop「デバッグ」ウィンドウの実行中のプロセスに dbx を接続するには、「デバッグ」➤「プロセスを接続」を選択します。詳細については、Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「実行中のプロセスへの接続」を参照してください。

dbx 起動時シーケンス

呼び出されると、dbx は、ディレクトリ *install-directory*/SUNWspro/lib (デフォルトの *install-directory* は /opt) で、インストール時の起動ファイルである .dbxrc を検索して読み込みます。

次に dbx は、現在のディレクトリ、そして $HOME で、起動ファイル .dbxrc を検索します。このファイルが見つからない場合は、現在のディレクトリ、そして $HOME で、起動ファイル .dbxinit を検索します。

一般に .dbxrc と .dbxinit ファイルのコンテンツは、1 つの大きな違いを除いて同じです。.dbxinit ファイルでは、alias コマンドは dalias として定義され、通常のデフォルト (Korn シェルに対する alias コマンド、kalias) ではありません。-s コマンド行オプションを使用することにより、異なる起動ファイルを明示的に指定することができます。詳細については、31 ページの「.dbxrc ファイルの使用」および Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「.dbxrc ファイルの作成」を参照してください。

起動ファイルには、任意の dbx コマンドが含まれ、一般に alias、dbxenv、pathmap、および Korn シェル関数定義が含まれます。ただし、特定のコマンドは、プログラムがロードされていること、またはプロセスが接続されていることを要求します。すべての起動ファイルは、プログラムまたはプロセスがロードされる前にロー

ドされます。さらに起動ファイルは、source または . (ピリオド) コマンドを使用することにより、その他のファイルのソースとなることもできます。起動ファイルを使用して、他の dbx オプションを設定することもできます。

dbx がプログラム情報をロードすると、Reading *filename* など一連のメッセージを印刷します。

プログラムが読み込みを終了すると、dbx は準備状態となり、プログラム (C、C++ については、main()、Fortan 77 、Fortan 9 については、 MAIN()) の「main」ブロックを表示します。一般に、ブレークポイントを設定（例: stop in main ）し C プログラムに対し run コマンドを実行します。

起動属性の設定

pathmap、dbxenv、alias コマンドを使用して、dbx セッションに対する起動プロパティを設定することができます。

デバッグ時ディレクトリへのコンパイル時ディレクトリのマッピング

デフォルトでは、dbx はプログラムがコンパイルされたディレクトリに、デバッグ中のプログラムに関連するソースファイルがないかを探します。ソースファイルまたはオブジェクトファイルがそのディレクトリにないか、または使用中のマシンが同じパス名を使用していない場合は、dbx にその場所を知らせる必要があります。

ソースファイルまたはオブジェクトファイルを移動した場合は、その新しい位置を検索パスに追加できます。pathmap コマンドは、ファイルシステムの現在のディレクトリと実行可能イメージ内の名前とのマッピングを作成します。このマッピングは、ソースパスとオブジェクトファイルパスに適用されます。

一般的なパスマップは、各自の .dbxrc ファイルに追加する必要があります。

ディレクトリ *from* から ディレクトリ *to* への新しいマッピングを確立するには、次のように入力します。

```
(dbx) pathmap [ -c ] from to
```

-c を使用すると、マッピングは現在の作業ディレクトリにも適用されます。

pathmap コマンドは、ホストによってベースパスの異なる、自動マウントされた明示的な NFS マウントファイルシステムを扱う場合にも役立ちます。現在の作業ディレクトリが自動マウントされたファイルシステムで不正確なオートマウンタが原因で起こる問題を解決する場合は、-c を使用してください。

/tmp_mnt と / のマッピングはデフォルトで存在します。

詳細については、323 ページの「pathmap コマンド」を参照してください。

dbx 環境変数の設定

dbxenv コマンドを使用すると、dbx カスタマイズ変数を表示または設定できます。dbxenv の値は、各自の .dbxrc ファイルに入れることによってカスタマイズします。変数を表示するには、次のように入力します。

```
(dbx) dbxenv
```

dbx 環境変数は設定することもできます。これらの変数の設定方法について詳しくは、第 3 章「dbx のカスタマイズ」を参照してください。

詳細については、22 ページの「dbx 環境変数の設定」および298 ページの「dbxenv コマンド」を参照してください。

ユーザー自身の dbx コマンドを作成

kalias または dalias コマンドを使用して、ユーザー自身の dbx コマンドを作成することができます。詳細については、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「kalias コマンド」および 295 ページの「dalias コマンド」を参照してください。

デバッグのため、プログラムをコンパイル

プログラムは -g または -g0 オプションでコンパイルし、dbx でデバッグする準備をする必要があります。

-g オプションは、コンパイル時にデバッグ情報を生成するよう、コンパイラに指示します。

たとえば、C++ を使用してコンパイルするには、次のように入力します。

```
% CC -g example_source.cc
```

C++ では、-g オプションは、デバッグをオンにし、関数のインライン化をオフにします。-g0 (ゼロ) オプションは、デバッグをオンにし、関数のインライン化には影響を与えません。-g0 オプションでインライン関数をデバッグすることはできません。-g0 オプションは、リンクタイムおよび dbx の起動時間を大幅に削減します (プログラムによるインライン関数の使用に依存します)。

dbx で使用するため、最適化コードをコンパイルするには、-O (大文字 O) と -g オプションの両方でソースコードをコンパイルします。

最適化コードのデバッグ

dbx ツールは、最適化コードのデバッグを部分的にサポートしています。サポートの範囲は、プログラムのコンパイル方法によって大幅に異なります。

最適化コードを分析する場合、次のことができます。

- 関数起動時に実行を停止する (`stop in` *function* コマンド)
- 引数を評価、表示、または変更する
- 大域変数または静的変数を評価、表示、変更する
- ある行から別の行へシングルステップする (`next` または `step` コマンド)

ただし、最適化されたコードを使用すると、dbx はローカル変数を評価、表示、または修正できなくなります。

最適化によりプログラムがコンパイルされ、同時に (-O -g オプションを使用して) デバッグが有効になると、dbx は制限されたモードで操作します。

どのような環境下でどのコンパイラがどの種類のシンボリック情報を発行したかについての詳細は、不安定なインターフェイスとみなされ、リリース移行時に変更される可能性があります。

ソース行についての情報が提供されます。ただし最適化プログラムについては、1 つのソース行に対するコードが複数の異なる場所で表示される場合があります。そのため、ソース行ごとにプログラムをステップすると、オプティマイザによってどのようにコードがスケジュールされたかに依存して、ソースファイルの周りで現在の行のジャンプが発生します。

末尾呼び出しを最適化すると、関数の最後の有効な操作が別の関数への呼び出しである場合、スタックフレームがなくなります。

通常、パラメータ、ローカル変数、およびグローバル変数のシンボリック情報は、最適化されたプログラムで利用できます。構造体、共用体、および C++ クラスの型情報とローカル変数、グローバル変数、およびパラメータの型と名前を利用できるはずです。プログラムにおけるこれらの項目の位置についての完全な情報は、最適化されたプログラムで入手できません。C++ コンパイラは、ローカル変数のシンボリック型情報を提供しません。ただし、C コンパイラは、それらの情報を提供します。

-g オプションを使用しないでコンパイルされたコード

ほとんどのデバッグサポートでは、プログラムを `-g` でコンパイルすることを要求していますが、dbx では、`-g` を使用しないでコンパイルされたコードに対し、次のレベルのサポートを提供しています。

- バックトレース (`dbx where` コマンド)
- 関数の呼び出し (ただし、パラメータチェックなし)
- 大域変数のチェック

ただし、dbx では、`-g` オプションでコンパイルされたコードを除いては、ソースコードを表示できません。これは、`strip -x` が適用されたコードについてもあてはまります。

dbx を完全にサポートするために -g オプションを必要とする共有ライブラリ

完全なサポートを提供するためには、共有ライブラリも `-g` オプションを使用してコンパイルする必要があります。`-g` によってコンパイルされていない共有ライブラリモジュールをいくつか使用してプログラムを作成した場合でも、そのプログラムをデバッグすることはできます。ただし、これらのライブラリモジュールに関する情報が生成されていないため、dbx の機能を完全に使用することはできません。

完全にストリップされたプログラム

dbx は、完全にストリップされた (制御データなどが取り除かれた) プログラムをデバッグすることができます。これらのプログラムには、プログラムをデバッグするために使用できる情報がいくつか含まれますが、外部から識別できる関数しか使用できません。一部の実行時検査は、ストリップされたプログラムまたはロードオブジェクトに対して動作します。メモリー使用状況検査およびアクセス検査は、`strip -x` でストリップされたコードに対して動作します。ただし、`strip` でストリップされたコードに対しては動作しません。

デバッグセッションを終了する

dbx の起動から終了までが 1 つの dbx セッションになります。1 つの dbx セッション中に、任意の数のプログラムを連続してデバッグできます。

dbx セッションを終了するには、dbx プロンプトで `quit` と入力します。

```
(dbx) quit
```

起動時にプロセス ID オプションを使用してデバッガを動作中のプロセスに接続した場合、デバッグセッションを終了しても、そのプロセスは終了しないで動作を続けます。すなわち、dbx はセッションを終了する前に自動的に `detach` コマンドを実行します。

プロセス実行の停止

Ctrl + C を使用すると、dbx を終了しないでいつでもプロセスの実行を停止できます。

dbx からのプロセスの切り離し

dbx をあるプロセスに接続した場合、`detach` コマンドを使用すると、そのプロセスおよび dbx セッションを終了せずに、そのプロセスを dbx から切り離すことができます。

プロセスを終了せずに dbx から切り離すには、次のように入力します。

```
(dbx) detach
```

詳細については、300 ページの「detach コマンド」を参照してください。

セッションを終了せずにプログラムを終了する

dbx の `kill` コマンドは、プロセスを終了するとともに、現在のプロセスのデバッグも終了します。ただし、`kill` コマンドは、dbx セッション自体を維持したまま、dbx で別のプログラムをデバッグできる状態にします。

プログラムを終了すると、dbx を終了しないで、デバッグ中のプログラムの残りを除去することができます。

dbx で実行中のプログラムを終了するには、次のように入力します。

```
(dbx) kill
```

詳細については、311 ページの「kill コマンド」を参照してください。

デバッグ実行の保存と復元

dbx には、デバッグ実行の全部または一部を保存して、それを後で再現するためのコマンドが 3 つあります。

- `save` [*-number*] [*filename*]
- `restore` [*filename*]
- `replay` [*-number*]

`save` コマンドの使用

`save` コマンドは、直前に実行された `run`、`rerun`、または `debug` コマンドから `save` コマンドまでに発行されたデバッグコマンドをすべてファイルに保存します。このデバッグセッションのセグメントは、デバッグ実行と呼ばれます。

save コマンドは、発行されたデバッグコマンドのリスト以外のものも保存します。実行開始時のプログラムの状態に関するデバッグ情報、つまり、ブレークポイント、表示リストなども保存されます。保存された実行を復元するとき、dbx は、保存ファイル内にあるこれらの情報を使用します。

デバッグ実行の一部、つまり、入力されたコマンドのうち指定する数だけ最後から除いたものを保存することもできます。次の例 A は、すべて保存された実行を示しています。例 B は、保存された同じ実行から、最後の 2 ステップを除いたものを示しています。

例 A：すべての実行の保存

```
debug
stop at line
run
next
next
stop at line
continue
next
next
step
next
save
```

例 B：実行の保存から最後の 2 ステップを除く

```
debug
stop at line
run
next
next
stop at line
continue
next
next
step
next
save-2
```

保存する実行の終了位置がわからない場合は、history コマンドを使用して、セッション開始以降に発行されたデバッグコマンドのリストを確認してください。

注 – デフォルトにより、save コマンドは特別な保存ファイルへ情報を書き込みます。デバッグ実行を後に復元可能なファイルへ保存する場合は、save コマンドでファイル名を指定することができます。28 ページの「一連のデバッグ実行をチェックポイントとして保存する」を参照してください。

---

save コマンドまでのデバッグ実行すべてを保存するには、次のように入力します。

```
(dbx) save
```

デバッグ実行の一部を保存するには、save *number* コマンドを使用します。*number* は、save コマンドの直前の、保存しないコマンドの数を示します。

```
(dbx) save -number
```

一連のデバッグ実行をチェックポイントとして保存する

ファイル名を指定しないでデバッグ実行を保存すると、情報は特殊な保存ファイルに書き込まれます。保存のたびに、dbx はこの保存ファイルを上書きします。しかし、ファイル名引数を save コマンドに指定すると、あるデバッグ実行をこのファイル名に保存後別のデバッグ実行を保存しても、前の内容を復元することができます。

一連の実行を保存すると、1 組のチェックポイントが与えられます。各チェックポイントは、セッションのさらに後から始まります。保存されたこれらの実行は任意に復元して続行し、さらに、以前の実行で保存されたプログラム位置と状態に dbx をリセットすることができます。

デバッグ実行を、デフォルトの保存ファイル以外のファイルに保存するには、次のように入力します。

```
(dbx) save filename
```

保存された実行の復元

実行を保存したら、restore コマンドを使用して実行を復元できます。dbx は、保存ファイル内の情報を使用します。実行を復元すると、dbx は、まず内部状態をその実行の開始時の状態にリセットしてから、保存された実行内の各デバッグコマンドを再発行します。

注 – source コマンドは、ファイル内に保存された一連のコマンドを再発行しますが、dbx の状態をリセットはしません。これは、現在のプログラム位置からコマンドの一覧を再発行するだけです。

保存された実行の正確な復元に必要な条件

保存されたデバッグ実行を正確に復元するには、run タイプコマンドへの引数、手動入力、およびファイル入力などの、実行での入力すべてが正確に同じでなければなりません。

注 – セグメントを保存してから、restore を実行する前に run、rerun、または debug コマンドを発行すると、restore は 2 番目の引数を使用して、run、rerun、または debug コマンドを後で保存します。これらの引数が異なる場合、正確な復元が得られない可能性があります。

保存されたデバッグ実行を復元するには、次のように入力します。

```
(dbx) restore
```

デフォルトの保存ファイル以外のファイルに保存されたデバッグ実行を復元するには、次のように入力します。

```
(dbx) restore filename
```

replay を使用した保存と復元

replay コマンドは組合せのコマンドで、save -1 に続けて restore を発行するのと同じです。replay コマンドは負の *number* 引数をとります。これは、コマンドの save 部分に渡されるものです。デフォルトにより、*-number* の値は -1 になるため、replay は取り消しコマンドとして働き、直前に発行されたコマンドにいたるまで (ただしこのコマンドは除く) の前回の実行を復元します。

現在のデバッグ実行から、最後に発行されたデバッグコマンドを除くものを再現するには、次のように入力します。

```
(dbx) replay
```

現在のデバッグ実行を再現して、最後から 2 番目のコマンド以前で実行を停止するには、dbx の replay コマンドを使用します。ここで、*number* は、最後のデバッグコマンドから数えていくつ目のコマンドで停止するかその数を示します。

```
(dbx) replay -number
```

第3章

dbx のカスタマイズ

この章では、デバッグ環境の特定の属性をカスタマイズするために使用できる dbx 環境変数と、初期化ファイル .dbxrc を使用してカスタマイズの内容をセッション間で保存する方法について説明します。

この章の内容は次のとおりです。

- .dbxrc ファイルの使用
- dbx 環境変数と Korn シェル
- Sun WorkShop での dbx カスタマイズ
- dbx 環境変数の設定

.dbxrc ファイルの使用

dbx 初期化ファイル .dbxrc は、dbx を起動するたびに実行する dbx コマンドを保存します。通常このファイルには、デバッグ環境をカスタマイズするコマンドを記述しますが、任意の dbx コマンドを記述することもできます。デバッグ中に dbx をコマンド行からカスタマイズする場合、これらの設定値は、現在デバッグ中のセッションにしか適用されないことに注意してください。

注 – .dbxrc ファイルは、コードを実行するコマンドを含むことはできません。ただし、それらのコマンドをファイルに置き、dbx source コマンドを使用して、そのファイルでコマンドを実行することは可能です。

dbx はまず起動時に .dbxrc を検索します。検索順序は次のとおりです。

1. 現在のディレクトリ ./.dbxrc

2. ホームディレクトリ $HOME/.dbxrc

.dbxrc が見つからない場合、dbx は警告メッセージを出して、.dbxinit を検索します (dbx モード)。

検索順序は次のとおりです。

1. 現在のディレクトリ ./.dbxinit

2. ホームディレクトリ $HOME/.dbxinit

.dbxrc ファイルの作成

警告メッセージを抑制し、共通のカスタマイズおよびエイリアスを含む .dbxrc ファイルを作成するには、コマンド区画に次のように入力します。

```
help .dbxrc>$HOME/.dbxrc
```

テキストエディタを使用して、結果的にできたファイルをカスタマイズすることにより、実行したいエントリをコメント解除することができます。

初期化ファイル

次に .dbxrc ファイルの例を示します。

```
dbxenv case input_case_sensitive false
catch FPE
```

最初の行は、大文字/小文字区別の制御のデフォルト設定を変更するものです。

- dbxenv は、dbx 環境変数を設定するためのコマンドです (dbx 環境変数の完全なリストについては、22 ページの「dbx 環境変数の設定」を参照してください)。
- input_case_sensitive は、大文字/小文字の区別を制御するための dbx 環境変数です。
- false は、input_case_sensitive の設定値です。

次の行はデバッグコマンドの catch です。シグナル FPE を捕獲するように設定しています。

詳細については、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「.dbxrc ファイルの典型的な項目」、「.dbxrc ファイルの有用な別名」、「.dbxrc ファイルの有用な関数」を参照してください。

dbx 環境変数と Korn シェル

それぞれの dbx 環境変数は、ksh 変数としてアクセスすることもできます。ksh 変数の名前は、DBX_ を前に付けて、dbx 環境変数から引き出されます。たとえば、`dbxenv stack_verbose` と `echo $DBX_stack_verbose` の出力は同じです。

Sun WorkShop での dbx カスタマイズ

シェルのコマンド行からと同様、Sun WorkShop デバッグの dbx コマンドウィンドウで dbx を使用することにより、Sun WorkShop デバッグのカスタマイズ機能を活用して dbx を効率的に利用することができます。

デバッグオプションの設定

Sun WorkShop デバッグの dbx コマンドウィンドウで dbx を最初に使用する場合、「デバッグオプション」ダイアログで、dbx 環境変数を設定する必要があります。Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「デバッグオプションダイアログ」を参照してください。

「デバッグオプション」ダイアログを使用してデバッグオプションを設定すると、対応する環境変数への dbxenv コマンドが Sun WorkShop 構成ファイル .workshoprc に保存されます。「デバッグ」ウィンドウでプログラムのデバッグを開始すると、.workshoprc ファイルの設定と衝突する .dbxrc ファイルでの設定が優先されます。

統一されたオプションセットの維持

シェルのコマンド行および Sun WorkShop 内の両方から dbx を使用する場合、両方のモードに対して dbx をカスタマイズする、統一されたオプションセットを作成することができます。

統一されたオプションセットを作成するには、次の操作を行います。

1. 「デバッグ」ウィンドウで、「デバッグ」➤「デバッグ用オプション」を選択します。
2. 各オプションを設定して「了解」をクリックすると、現在の値がデフォルト値になります。
3. エディタウィンドウで、.dbxrc ファイルを開きます。
4. 最上部付近に、source $HOME/.workshoprc の行を入力します。
5. 関連する dbxenv コマンドをソース行の後の位置へ移動するか、それらを削除またはコメントアウトします。
6. ファイルを保存します。

「デバッグオプション」ダイアログでなされた変更は、Sun WorkShop の使用時およびシェルでの実行時の両方で、dbx に適用されます。

2 組のオプション維持

Sun WorkShop 内、およびシェルでコマンド行から dbx を実行するための異なるオプション設定を維持するには、.dbxrc ファイルの havegui 変数を使用して、dbxenv コマンドを条件付にすることができます。たとえば次のようになります。

```
if $havegui
  then
     dbxenv follow_fork_mode ask
     dbxenv stack_verbose on
  else
      dbxenv follow_fork_mode parent
      dbxenv stack_verbose off
```

カスタムボタンの保存

Sun WorkShop デバッグには、「カスタムボタン」ウィンドウ (および「デバッグ」ウィンドウおよびエディタウィンドウのツールバー) で、ボタンを追加、削除、編集するための「ボタン編集」ダイアログが提供されています。.dbxrc ファイルにボタ

ンを保存するために button コマンドを使用する必要はありません。「ボタン編集」ダイアログでは、.dbxrc ファイルへのボタンの追加、またはボタンの削除を行うことはできません。

「ボタン編集」ダイアログでは、.dbxrc ファイルに保存されたボタンを永久に削除することはできません。ボタンは、次のデバッグセッションで再表示されます。それらのボタンを削除するには、.dbxrc ファイルを編集してください。

dbx 環境変数の設定

dbxenv コマンドを使用して、dbx 環境変数を設定することにより、dbx セッションをカスタマイズすることができます。

特別な変数の値を表示するには、次のように入力します。

```
(dbx) dbxenv variable
```

すべての変数とその値を表示するには、次のように入力します。

```
(dbx) dbxenv
```

変数の値を設定するには、次のように入力します。

```
(dbx) dbxenv variable value
```

注 – 各変数には、DBX_trace_speed のように対応する ksh 環境変数があります。この変数を直接割り当てることも、dbxenv コマンドを使用することもできます。どちらでも結果は同じです。

以下に、設定可能なすべての dbx 環境変数を示します。

表 3-1　dbx 環境変数

dbx 環境変数	dbx 環境変数の機能
`allow_critical_exclusion on|off`	通常、`loadobject -x` は、dbx 機能に欠かせない特定の共有ライブラリの排除を許可しません。この変数を on に設定すると、その制限が解除されます。この変数が on の場合のみ、コアファイルをデバッグすることができます。デフォルト値は `off` です。
`aout_cache_size` *number*	a.out ロードオブジェクトのキャッシュサイズ。単一の dbx から順に *num* 個のプログラムをデバッグする場合は *num* に設定してください。*num* をゼロに設定しても、共有オブジェクトのキャッシュを行うことはできます。`locache_enable` の項目を参照してください。デフォルト値は 1 です。
`array_bounds_check on|off`	パラメータを on に設定すると、配列の上下限を検査します。デフォルト値は on です。
`core_lo_pathmap`	dbx が一致しないコアファイルの正しいライブラリを検索するためにパスマップ設定を使用するかどうかを制御します。デフォルト値は `off` です。
`delay_xs on|off`	`-xs` オプションでコンパイルされたモジュールのデバッグ情報を、dbx の起動時に読み込むか、あるいは遅延させるかを制御します。
`disassembler_version autodetect|v8|v9|v9vis`	SPARC プラットフォームでの SPARC V8、V9、またはビジュアル命令セットを持つ V9 のいずれかの逆アセンブラのバージョンを設定します。デフォルト値は `autodetect` で、a.out が実行されているマシンのタイプに従って、動的にモードを設定します。Intel プラットフォーム以外では、`autodetect` だけが有効です。 IA プラットフォーム：`autodetect` だけが有効です。

表 3-1　dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
fix_verbose on|off	fix 中のコンパイル行出力を制御します。デフォルト値は off です。
follow_fork_inherit on|off	子プロセスを生成した後、イベントを継承するかどうかを設定します。デフォルト値は off です。
follow_fork_mode parent|child|both|ask	現在のプロセスが fork、vfork、fork1 を実行しフォークした場合、どのプロセスを追跡するかを決定します。parent に設定すると、親を追跡し、child に設定すると、子を追跡します。both に設定すると、親プロセスをアクティブ状態にして子を追跡します。ask に設定すると、フォークが検出されるたびに、追跡するプロセスを尋ねます。デフォルトは parent です。
follow_fork_mode_inner unset|parent|child|both	フォークが検出された後、follow_fork_mode が ask に設定されていて、停止を選んだ時の設定です。この変数を設定すると、cont -follow を使用する必要はありません。
input_case_sensitive autodetect|true|false	autodetect に設定すると、デバッグ対象の言語に従って大文字/小文字の区別が自動的に選択されます。Fortran 77 および Fortran 95 ファイルの場合は false、そうでない場合は true です。true の場合は、変数と関数名では大文字/小文字が区別されます。変数と関数名以外では、大文字/小文字は区別されません。デフォルト値は autodetect です。

表 3-1　dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
language_mode autodetect|main|c| ansic|c++|objc|fortran| fortran90|native_java	式の構文解析と評価に使用される言語を指定します。autodetect では、デバッグ中のファイルの言語が設定されます。これは複数の言語を使用するプログラムをデバッグするときに便利です (デフォルトモード)。main では、プログラムのメインルーチンの言語が設定されます。これは同機種プログラムをデバッグするときに便利です。c、c++、ansic、objc、fortran、fortran90 および native_java では、それぞれ指定の言語に設定されます。
locache_enable on|off	ロードオブジェクトキャッシュを有効または無効にします。デフォルト値は on です。
mt_scalable on|off	有効の場合、dbx はリソースの使用方法において保守的となり、300 個以上の LWP を持つプロセスのデバッグが可能です。下方サイドは大幅に速度が減少します。デフォルト値は off です。
output_auto_flush on|off	呼び出しが行われるたびに、fflush() を自動的に呼び出します。デフォルト値は on です。
output_base 8|10|16 automatic	整数の定数を出力するためのデフォルト基数。デフォルト値は automatic です (ポインタは 16 進文字、その他すべては 10 進)。
output_dynamic_type on|off	on の場合、出力、表示、および検査のデフォルト出力を -d にします。デフォルト値は off です。
output_inherited_members on|off	on の場合、出力、表示、および検査のデフォルト出力を -r にします。デフォルト値は off です。
output_list_size *number*	list コマンドで出力する行のデフォルト数を指定します。デフォルト値は 10 です。
output_log_file_name *filename*	コマンドログファイルの名前。デフォルト値は /tmp/dbx.log *uniqueID* です。

表 3-1　dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
output_max_string_length *number*	char * で出力される文字数を設定します。デフォルト値は 512 です。
output_pretty_print on|off	出力、表示、および検査の出力デフォルトを -p に設定します。デフォルト値は off です。
output_short_file_name on|off	ファイル名を表示する時に短形式で表示します。デフォルト値は on です。
overload_function on|off	C++ の場合、on に設定すると、自動で多重定義された関数の解決を行います。デフォルト値は on です。
overload_operator on|off	C++ の場合、on に設定すると、自動で多重定義された演算子の解決を行います。デフォルト値は on です。
pop_auto_destruct on|off	on に設定すると、フレームをポップするときに、ローカルの適切なデストラクタを自動的に呼び出します。
proc_exclusive_attach on|off	on に設定すると、別のツールがすでに接続されている場合、dbx をプロセスへ接続しないようにします。警告：複数のツールが 1 つのプロセスに接続している状態でプロセスを制御しようとすると、混乱が生じるので注意してください。デフォルト値は on です。
rtc_auto_continue on|off	rtc_error_log_file_name にエラーを記録して続行します。デフォルト値は off です。
rtc_auto_suppress on|off	on に設定すると、特定の位置の RTC エラーが一回だけ報告されます。デフォルト値は off です。

表 3-1　dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
rtc_biu_at_exit on|off|verbose	check -memuse が明示的に、または check -all によって on になっている場合に使用されます。この値が on だと、簡易メモリー使用状況 (使用中ブロック) レポートがプログラムの終了時に作成されます。値が verbose の場合は、詳細メモリー使用状況レポートがプログラムの終了時に作成されます。値が off の場合、出力は生成されません。デフォルト値は on です。
rtc_error_limit *number*	報告される RTC エラーの数。デフォルト値は 1000 です。
rtc_error_log_file_name *filename*	rtc_auto_continue が設定されている場合に、RTC エラーが記録されるファイル名。デフォルト値は /tmp/dbx.errlog.*uniqueID* です。
rtc_error_stack on|off	on に設定すると、スタックトレースは、RTC 内部機構へ対応するフレームを示します。デフォルト値は off です。
rtc_inherit on|off	on に設定すると、デバッグプログラムから実行される子プロセスでランタイムチェックを有効にし、LD_PRELOAD が継承されます。デフォルト値は off です。
rtc_mel_at_exit on|off|verbose	リーク検査がオンの場合に使用されます。この値が on の場合は、簡易メモリーリークレポートがプログラムの終了時に作成されます。値が verbose の場合は、詳細メモリーリークレポートがプログラムの終了時に作成されます。off の場合は出力は生成されません。デフォルト値は on です。
run_autostart on|off	dbx で実行中でないプログラムで on の場合、step、next、stepi、および nexti を実行した場合、暗黙指定で run を実行し、言語依存のメインルーチンで停止します。on の場合、cont は必要に応じて run を暗黙指定します。デフォルト値は off です。

表 3-1　dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
`run_io stdio|pty`	ユーザープログラムの入出力が、dbx の stdio か、または特定の pty にリダイレクトされるかどうかを指定します。pty は、run_pty によって指定します。デフォルト値は stdio です。
`run_pty` *ptyname*	run_io が pty に設定されているときに使用する pty の名前を設定します。pty は GUI のラッパで使用されます。
`run_quick on|off`	on の場合、シンボリック情報は読み込まれません。シンボリック情報は、prog -readsysms を使用して要求に応じて読み込むことができます。それまで dbx は、デバッグ中のプログラムがストリップされているかのように動作します。デフォルト値は off です。
`run_savetty on|off`	dbx と デバッグ対象の間で、tty 設定、プロセスグループ、およびキーボード設定 (-kbd がコマンド行で使用されている場合) を多重化します。エディタやシェルをデバッグする際に便利です。dbx が SIGTTIN または SIGTTOU を取得しシェルに戻る場合は、on に設定します。速度を多少上げるには off に設定します。dbx がデバッグ対象に接続されるか WorkShop のもとで動作しているかということには無関係です。デフォルト値は on です。
`run_setpgrp on|off`	on の場合、プログラムが実行時に、フォークの直後に setpgrp(2) が呼び出されます。デフォルト値は off です。
`scope_global_enums on|off`	on の場合、列挙子の有効範囲はファイルスコープではなく大域スコープになります。デバッグ情報を処理する前に設定する必要があります (〜/.dbxrc)。デフォルト値は off です。
`scope_look_aside on|off`	ファイルの静的シンボルが、ファイルスコープにない場合でもそれを検出します。デフォルト値は on です。

表 3-1　dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
`session_log_file_name` *filename*	dbx がすべてのコマンドとその出力を記録するファイルの名前。出力はこのファイルに追加されます。デフォルト値は "" (セッション記録なし) です。
`satck_find_source on|off`	onに設定した場合、デバッグ中のプログラムが `-g` オプションなしでコンパイルされた指定の関数で停止したとき、dbx はソースを持つスタックフレームを検索し、自動的にポップアップを試みます。デフォルト値は on です。
`stack_max_size` *number*	where コマンドにデフォルトサイズを設定します。デフォルト値は 100 です。
`stack_verbose on|off`	where コマンドでの引数と行情報の出力を指定します。デフォルト値は on です。
`step_events on|off`	on に設定すると、ブレークポイントを許可する一方で、`step` および `next` コマンドを使用してコードをステップ実行できます。デフォルト値は `off` です。
`step_granularity statement|line`	ソース行ステップの細分性を制御します。`statement` に設定すると、次のコード `a(); b();` を、実行するための 2 つの next コマンドが必要です。`line` に設定すると、1 つの next コマンドでコードを実行します。複数行のマクロを処理する場合、行の細分化は特に有用です。デフォルト値は `statement` です。
`suppress_startup_message` *number*	リリースレベルを設定して、それより下のレベルでは起動メッセージが表示されないようにします。デフォルト値は 3.01 です。
`symbol_info_compression on|off`	各 `include` ファイルのデバッグ情報を一回だけ読み取ります。デフォルト値は on です。
`trace_speed` *number*	トレース実行の速度を設定します。値は、ステップ間の休止秒数になります。デフォルト値は `0.50` です。

第4章

コードの表示と別部分のコードへの移動

プログラムが停止するたびに dbx が表示するソースコードは、その停止位置に対応するコードです。また、プログラムが停止するたびに、dbx は現在の関数の値をプログラムが停止した関数の値に再設定します。プログラムの停止後、その停止場所以外の関数やファイルを一時的に表示することができます。

この章では、デバッグセッション中に dbx がどのようにコードを参照し、関数やシンボルを検索するかを説明します。また、コマンドを使用して、プログラムの停止位置とは別の場所のコードを一時的に表示したり、識別子、型、クラスの宣言を調べたりする方法も説明します。

この章は、次の各節から構成されています。

- コード位置へのマッピング
- 停止位置とは別の部分のコードを表示する
- スコープ決定演算子を使用してシンボルを修飾する
- シンボルを検索する
- 変数、メンバー、型、クラスを調べる
- 自動読み取り機能の使用

コード位置へのマッピング

dbx は、プログラムに関連するソースファイルおよびオブジェクトコードファイルの位置を知っていなければなりません。オブジェクトファイルのデフォルトディレクトリは、プログラムが最後にリンクされたときにオブジェクトファイルがあったディレクトリです。ソースファイルのデフォルトディレクトリは、最後のコンパイル時にそれらが存在したディレクトリです。ソースファイルまたはオブジェクトファイルを移

動したか、またはそれらを新しい位置にコピーした場合は、プログラムを再リンクするか、または新しい位置に変更してからデバッグを行うか、pathmap コマンドを使用します。

ソースファイルまたはオブジェクトファイルを移動した場合、その新しい位置を検索パスに追加できます。pathmap コマンドは、ファイルシステムの現在のディレクトリと実行可能イメージ内の名前のマッピングを作成します。このマッピングは、ソースパスとオブジェクトファイルパスに適用されます。

ディレクトリ *from* から ディレクトリ *to* への新しいマッピングを確立するには、次のように入力します。

```
(dbx) pathmap [-c] from to
```

-c を使用すると、このマッピングは、現在の作業ディレクトリにも適用されます。

pathmap コマンドは、ホストによってベースパスの異なる、自動マウントされた明示的な NFS マウントファイルシステムを扱う場合でも便利です。-c は、現在の作業ディレクトリが自動マウントされたファイルシステム上で不正確なオートマウンタが原因で起こる問題を解決する場合に使用してください。

/tmp_mnt と / のマッピングはデフォルトで存在します。

詳細については、323 ページの「pathmap コマンド」を参照してください。

スコープ

スコープとは、変数または関数の可視性について定義されたプログラムのサブセットです。あるシンボルの名前が特定の実行地点において可視となる場合、そのシンボルは「スコープ範囲内にある」ことになります。C 言語では、関数はグローバルまたはファイル固有のスコープを保持します。変数は、グローバル、ファイル固有、関数、またはブロックのスコープを保持します。

現在のスコープを変更する

dbx において、スコープは、シンボルの検索を開始する位置を示すこともあります。通常、スコープは現在の行になりますが、次のコマンドを使用することにより、実行地点を移動することなく現在のスコープを変更できます。

`func`

`file`

`up, down, frame`

`list` *procedure*

スコープ 検索規則の緩和

静的シンボルおよび C++ メンバー関数のスコープ検索規則を緩和するには、dbx 環境変数 `scope_look_aside`を on に設定します。

```
dbxenv scope_look_aside on
```

または、二重逆引用符接頭辞を使用します。

```
stop in ``func4  func4  は静的でスコープにない場合があります
```

dbx 環境変数 `scope_look_aside` が on に設定されている場合、dbx は次を検索します。

- その他のファイルで定義されている静的変数 (現在のスコープで見つからなかった場合)。`/usr/lib` に位置するライブラリのファイルは検索されません。
- クラス修飾子のない C++ メンバー関数
- その他のファイルの C++ インラインメンバー関数のインスタンス (メンバー関数が現在のファイルでインスタンス化されていない場合)

停止位置とは別の部分のコードを表示する

プログラムを実行していないときはいつでも、プログラム内の停止位置とは別の部分を表示できます。プログラムに含まれるすべての関数またはファイルを表示できます。現在のスコープはプログラムの停止位置に設定されます (46 ページの「スコープ」を参照してください)。この機能は、`stop at` ブレークポイントを設定し、停止

した時にソース行を決定する際に便利です。stop at ブレークポイントの詳細については、72 ページの「ソースコードの特定の行に stop ブレークポイントを設定する」および 73 ページの「関数に stop ブレークポイントを設定する」 を参照してください。

ファイルの内容を表示する

dbx がプログラムの一部として認識していれば、どのようなファイルでもその内容を表示できます (モジュールまたはファイルが -g オプションでコンパイルされていない場合でも可能です)。ファイルを表示した場合、現在の関数は変更されません。ファイルの内容を表示するためには、次のように入力します。

```
(dbx) file filename
```

file コマンドを引数を指定しないで使用すると、現在表示中のファイル名が表示されます。

```
(dbx) file
```

dbx は、行番号を指定しないと、最初の行からファイルを表示します。

```
(dbx) file filename ; list line_number
```

ソースコードの行で stop at ブレークポイントを設定する詳細については、72 ページの「ソースコードの特定の行に stop ブレークポイントを設定する」 を参照してください。

関数を表示する

func コマンドを使用すると、関数を表示できます。コマンド func に続けて、関数名を入力します。次に例を示します。

```
(dbx) func adjust_speed
```

func コマンドを引数なしで使用すると、現在表示中の関数が表示されます。

詳細については、307 ページの「func コマンド」を参照してください。

あいまいな関数名をリストから選択する (C++)

C++ の場合、あいまいな名前または多重定義されている関数名を指定してメンバー関数を表示しようとすると、多重定義されているというメッセージが表示され、指定された名前を持つ関数のリストが示されます。表示したい関数の番号を入力します。関数が属している特定クラスを知っている場合は、クラス名と関数名を入力できます。次に例を示します。

```
(dbx) func block::block
```

複数存在する場合の選択

同じスコープレベルから複数のシンボルにアクセスできる場合、dbx は、あいまいさについて報告するメッセージを出力します。

```
(dbx) func main
(dbx) which C::foo
識別子 'foo' が複数あります
以下の名前から 1 つ選択してください :
 0) Cancel
 1) 'a.out't.cc'C::foo(int)
 2) 'a.out't.cc'C::foo()
>1
'a.out't.cc'C::foo(int)
```

which コマンドのコンテキストでシンボル名のリストから特定のシンボルを選んでも、dbx またはプログラムの状態には影響しません。いずれにせよ、どのシンボルを選んでも名前を表示するだけです。

which コマンドは、dbx がどのシンボルを検索するかを前もって示すものです。あいまいな名前を指定して、多重定義されていると表示された場合は、該当する複数の名前のうちのどれを使用するかがまだ特定されていません。リストに表示されている名前から 1 つを選んでください。

詳細については、307 ページの「func コマンド」を参照してください。

ソースリストの出力

`list` コマンドは、ファイルまたは関数のソースリストを出力するために使用します。*filename* を指定した場合は *filename* の先頭を表示、*number* を指定した場合は現在のファイルの *number* 行目を表示、*function* を指定した場合はその関数を表示します。

詳細については、313 ページの「list コマンド」を参照してください。

呼び出しスタックの操作によってコードを表示する

プロセスが存在するときにコードを表示する方法としては、さらに「呼び出しスタックを操作する」方法があります。この方法では、スタック操作コマンドを使用して現在スタック上にある関数を表示します。その結果、現時点でアクティブなすべてのルーチンが表示されます

スタックを操作すると、現在の関数とファイルは、スタック関数を表示するたびに変更されます。停止位置は、スタックの「底」にあるものと考えられます。したがって、そこから離れるには `up` コマンドを使用します。つまり、`main` 関数などに向かって移動します。現在のフレーム方向へ移動するには、`down` コマンドを使用します。

コールスタックのウォーキングについての詳細は、90 ページの「スタックを移動してホームに戻る」を参照してください。

スコープ決定演算子を使用してシンボルを修飾する

`func` または `file` を使用する場合、スコープ決定演算子を使用して、ターゲットとして指定する関数の名前を修飾することができます。

`dbx` では、シンボルを修飾するためのスコープ決定演算子として、逆引用符演算子 (`` ` ``) と C++ のスコープ決定演算子 (`::`) 、ブロック局所演算子 (*:lineno*) を使用することができます。これらの演算子は別々に、ときには同時に使用します。

停止位置以外の部分のコードを表示するためにファイルや関数の名前を修飾するだけでなく、スコープ外の変数や式の出力や表示を行ったり、型やクラスの宣言を表示したり (what is コマンド) する場合にも、シンボルを修飾することが必要です。シンボルの修飾規則はすべての場合で同じです。この節で示す規則は、あらゆる種類のシンボル名の修飾に適用されます。

逆引用符演算子

逆引用符演算子 (`) は、大域スコープの変数あるいは関数を検索するために使用できます。

```
(dbx) print `item
```

プログラムでは、同じ関数名を 2 つの異なるファイル (またはコンパイルモジュール) で使用できます。この場合、dbx に対しても関数名を修飾して、表示する関数を認識させる必要があります。ファイル名に関連して関数名を修飾するには、汎用逆引用符 (`) スコープ決定演算子を使用してください。

```
(dbx) func `file_name`function_name
```

コロンを重ねたスコープ決定演算子 (C++)

次のような名前を持つ C++ のメンバー関数、トップレベル関数、またはグローバルスコープを伴う変数を修飾するときは、コロンを 2 つ重ねた演算子 (::) を使用します。

- 多重定義されている名前 (複数の異なる引数型で同じ名前が使用されている)
- あいまいな名前 (複数の異なるクラスで同じ名前が使用されている)

多重定義された関数名を修飾できます。多重定義された関数名を修飾しないと、dbx は多重定義表示リストを自動的に表示して、表示する関数を選択するよう要求します。関数のクラス名がわかっている場合は、それを二重コロンのスコープ決定演算子とともに使用して、名前を修飾できます。

```
(dbx) func class::function_name (args)
```

たとえば、hand がクラス名で draw が関数名の場合は、次のようになります。

```
(dbx) func hand::draw
```

ブロックローカル演算子

ブロックローカル演算子 (*:lineno*) は、逆引用符演算子と組み合わせて使用します。これは、必要なインスタンスを参照する式の行番号を識別します。

次の例では、:230 がブロックローカル演算子です。

```
(dbx) stop in `animate.o`change_glyph:230`item
```

リンカー名

dbx は、(C++ のようにさまざまな名前が混在するため)リンカー名ごとにシンボルを探すよう特別な構文を使用します。シンボル名の接頭辞として # 記号を付け、Korn シェルで $ 記号の前にエスケープ文字 \ を使用します。

例

```
(dbx) stop in #.mul
(dbx) whatis #\$FEcopyPc
(dbx) print `foo.c`#staticvar
```

スコープ決定パス

dbx 固有コマンドにターゲットとして指定されたシンボルが検索を必要とするものである場合、dbx はそのシンボルを次の順序で検索します。

1. 最初に現在の関数のスコープ内で検索を行います。プログラムが、入れ子になったブロックで停止した場合はそのブロック内で検索した後、その関数によって宣言されている外側のすべてのブロックのスコープ内で検索します。
2. C++ の場合のみ : 現在の関数クラスのクラスメンバーとその基底クラス。
3. C++ の場合のみ : 現在のネームスペース。

4. すぐ外側にある「コンパイル単位」: 一般に、現在の関数が含まれているファイル。

5. ロードオブジェクト[1]のスコープ。

6. 大域的スコープ。

7. 上記のすべてで該当するシンボルが見つからなかった場合は非公開変数、すなわちファイル内で「静的」な変数または関数と見なします。dbxenv による scope_look_aside の設定値によっては、コンパイル単位ごとにファイル静的シンボルを検索することもできます。

dbx はこの検索パスで最初に見つけたシンボルを使用します。変数が見つからなかった場合はエラーを報告します。

シンボルを検索する

同じ名前が多くの場所で使用されたり、プログラム内の異なる種類の構成要素を参照したりすることがあります。dbx コマンド whereis は、特定の名前を持つすべてのシンボルの完全修飾名 (すなわち位置) のリストを表示します。一方、dbx コマンド which は、特定の名前を dbx コマンドのターゲットとしたときに、実際に使用されるシンボルを示します。(356 ページの「which コマンド」を参照)

シンボルの出現を出力する

指定シンボルの出現すべてのリストを出力するには、whrereis *symbol* を使用します。ここで、*symbol* は任意のユーザー定義識別子にすることができます。次に例を示します。

```
(dbx) whereis table
前方:   `Blocks`block_draw.cc`table
関数:   `Blocks`block.cc'table::table(char*, int, int, const
          point&)
クラス: `Blocks`block.cc'class table
クラス: `Blocks`main.cc`class table
変数:   `libc.so.1'hsearch.c`table
```

1. ロードオブジェクトは「ロード可能オブジェクト」の略称であり、SVR4 ABI で定義されています。実行可能ファイル (a.out) および共用ライブラリ (*.so) はロードオブジェクトです。

この出力には、プログラムがシンボルを定義する読み込み可能オブジェクトの名前が、各オブジェクトの構成要素の種類 (クラス、関数、または変数) とともに示されます。

dbx シンボルテーブルの情報は必要に応じて読み取られるため、whereis コマンドは、すでに読み込まれているシンボルの出現についてしか出力しません。デバッグセッションが長くなると、出現のリストは大きくなります。

詳細については、356 ページの「whereis コマンド」を参照してください。

実際に使用されるシンボルを調べる

which コマンドにより、特定の名前を (完全に修飾しないで) デバッグコマンドのターゲットとして指定したときにどのシンボルが使用されるかを前もって調べることができます。

```
(dbx) func
wedge::wedge(char*, int, int, const point&, load_bearing_block*)
(dbx) which draw
`block_draw.cc`wedge::draw(unsigned long)
```

which コマンドに指定したシンボル名が局所的スコープにない場合、スコープ決定パスで検索が行われます。決定パスで最初に見つかった名前の完全修飾名が示されます。

決定パスに含まれる任意の場所で、同じスコープの該当するシンボルが複数見つかった場合、あいまいであることを示すメッセージが表示されます。

```
(dbx) which fid
識別子 'foo' が複数あります
以下の名前から 1 つ選択してください :
 0) Cancel
 1) `example`file1.c`fid
 2) `example`file2.c`fid
```

dbx は、あいまいなシンボル名をリストで示し、多重定義であることを表示します。which コマンドのコンテキストでシンボル名のリストから特定のシンボルを選んでも、dbx またはプログラムの状態には影響しません。ほとんどの場合、どのシンボルを選んでも名前が表示されるだけです。

which コマンドは、あるシンボル (この例の場合は block) をコマンド (たとえば、print コマンド) のターゲットにした場合に何が起こるかを前もって示すものです。あいまいな名前を指定して、多重定義が表示された場合は、該当する複数の名前のうちのどれを使用するかがまだ特定されていません。dbx は該当する名前を列挙し、ユーザーがそのうちの 1 つを選択するまで待機します。which コマンドの詳細については、356 ページの「which コマンド」を参照してください。

変数、メンバー、型、クラスを調べる

dbx コマンド whatis は、識別子、構造体、型、C++ のクラス、式の型の宣言または定義を出力します。検査できる識別子には、変数、関数、フィールド、配列、列挙定数が含まれます。

詳細については、352 ページの「whatis コマンド」を参照してください。

変数、メンバー、関数の定義を調べる

識別子の宣言を出力するには、次のように入力します。

```
(dbx) whatis identifier
```

識別名は、必要に応じてファイルおよび関数情報によって修飾します。

C++ については、whatis *identifier*　は、関数テンプレート例示をリストします。テンプレート定義は、whatis -t *identifier* を付けて表示されます。56 ページの「型およびクラスの定義を調べる」を参照してください。

メンバー関数を出力するには、次のように入力します。

```
(dbx) whatis block::draw
void block::draw(unsigned long pw);
(dbx) whatis table::draw
void table::draw(unsigned long pw);
(dbx) whatis block::pos
class point *block::pos();
(dbx) whatis table::pos
class point *block::pos();
```

データメンバーを出力するには、次のように入力します。

```
(dbx) whatis block::movable
int movable;
```

変数を指定すると、その変数の型が示されます。

```
(dbx) whatis the_table
class table *the_table;
```

フィールドを指定すると、そのフィールドの型が示されます。

```
(dbx) whatis the_table->draw
void table::draw(unsigned long pw);
```

メンバー関数で停止したときは、this ポインタを調べることができます。

```
(dbx) stop in brick::draw
(3) stop in brick::draw(void)
(dbx) cont
(dbx) where 1
=>[1] brick::draw(this = 0x39418)、"block_draw.cc" の 129 行目
(dbx) whatis this
class brick *this;
```

型およびクラスの定義を調べる

whatis コマンドの -t オプションは、型の定義を表示します。C++ については、whatis -t で表示されるリストは、テンプレート定義およびクラステンプレート例示を含みます。

型または C++ のクラスの宣言を出力するには次のようにします。

```
(dbx) whatis -t type_or_class_name
```

whatis コマンドには、継承されたメンバーを表示するための -r (再帰) オプションが用意されています。このオプションを指定すると、指定したクラス *class_name* の宣言とともに、そのクラスが基となるクラスから継承したメンバーが表示されます。

```
(dbx) whatis -t -r class_name
```

whatis -r による出力は、クラス階層と各クラスのサイズによって長くなることがあります。出力の先頭には、階層の最も上にあるクラスから継承されたメンバーのリストが示されます。メンバーのリストは、コメント行によって親クラスごとに分けられます。

ここに、2 つの例を示します。table クラスは、load_bearing_block クラスの子クラスの 1 つです。また、load_bearing_block クラスは、block の子クラスです。

-r を指定しないと、table クラスで宣言されているメンバーが示されます。

```
(dbx) whatis -t class table
class table : public load_bearing_block {
public:
    table::table(char *name, int w, int h, const class point
&pos);
    virtual char *table::type();
    virtual void table::draw(unsigned long pw);
};
```

次に、子クラスが継承するメンバーを表示するために whatis -r がその子クラスで使用された場合の結果を示します。

```
(dbx) whatis -t -r class table
class table : public load_bearing_block {
public:
  /* 基底 class table::load_bearing_block::block から */
  block::block();
  block::block(char *name, int w, int h, const class point &pos,
class
load_bearing_block *blk);
    virtual char *block::type();
    char *block::name();
    int block::is_movable();
//  protected: までのいくつかのメンバー省略
    char *nm;
    int movable;
    int width;
    int height;
    class point  position;
    class load_bearing_block *supported_by;
    Panel_item panel_item;
    /* 基底 class table::load_bearing_block から */
public:
    load_bearing_block::load_bearing_block();
    load_bearing_block::load_bearing_block(char *name, int w,
int h,const class point &pos, class load_bearing_block *blk);
    virtual int load_bearing_block::is_load_bearing();
    virtual class list *load_bearing_block::supported_blocks();
    void load_bearing_block::add_supported_block(class block
&b);
    void load_bearing_block::remove_supported_block(class block
&b);
    virtual void load_bearing_block::print_supported_blocks();
    virtual void load_bearing_block::clear_top();
    virtual void load_bearing_block::put_on(class block
&object);
    class point load_bearing_block::get_space(class block
&object);
    class point load_bearing_block::find_space(class block
&object);
    class point load_bearing_block::make_space(class block
&object);
```

```
(続き)
protected:
    class list *support_for;
    /* class table から */
public:
    table::table(char *name, int w, int h, const class point
&pos);
    virtual char *table::type();
    virtual void table::draw(unsigned long pw);
};
```

自動読み取り機能の使用

通常、デバッグしたいプログラムは全体を -g オプションを使用して、コンパイルする必要があります。プログラムのコンパイル方法によって、各プログラムおよび共有ライブラリモジュールについて生成されるデバッグ情報は、各プログラムおよび共有ライブラリモジュールのオブジェクトコードファイル (.o ファイル)、またはプログラム実行可能ファイルのいずれか、あるいはこの両方に保存されます。

-g -c コンパイラオプションによってコンパイルを行うと、各モジュールのデバッグ情報は、その .o ファイルに保存されます。dbx は、各モジュールのデバッグ情報を必要に応じて、セッション中に自動的に読み取ります。この必要に応じた読み取り機能は、自動読み取り機能と呼ばれます。自動読み取り機能は、dbx のデフォルト機能です。

自動読み取り機能によって、大きなプログラムを dbx に読み込むときに時間を大幅に節約することができます。自動読み取り機能は、プログラム .o ファイルが、dbx に認識された位置に続けて存在することによって機能しています。dbx の認識後に .o ファイルを移動しないで下さい。

注 – .o ファイル を .a ファイルに保存してから、アーカイブライブラリを使用してリンクすると、関連の .o ファイルを削除することができますが、.a ファイルは削除できません。

デフォルトにより、dbx は、コンパイル時に記録された絶対パスを使用して、プログラムがコンパイルされたときのディレクトリのファイル、およびリンクされた位置での .o ファイルを検索します。それらのファイルがない場合は、pathmap コマンドを使用してサーチパスを設定します。

オブジェクトファイルが作成されない場合、デバッグ情報は、実行可能ファイルに保存されます。つまり、.o ファイルを作成しないコンパイルの場合、コンパイラはすべてのデバッグ情報を実行可能ファイルに保存します。保存されたデバッグ情報は、-xs オプションでコンパイルされたアプリケーションと同じ方法で読み取られます。60 ページの「.o ファイルが存在しない場合のデバッグ」を参照してください。

.o ファイルが存在しない場合のデバッグ

-g -c オプションでコンパイルされたプログラムは、各モジュールのデバッグ情報を、そのモジュールの .o ファイルに保存します。自動読み取り機能を使用するには、プログラム の .o ファイルと共有ライブラリの .o ファイルが続けて存在している必要があります。

デバッグしたいモジュールのプログラムの .o ファイルまたは共有ライブラリの .o ファイルの維持が不可能な場合は、コンパイラの -xs オプションを使用して (-g オプションの後に付けます) プログラムをコンパイルします。-xs でコンパイルしたモジュールと、このオプションを使わずにコンパイルしたモジュールが混在してもかまいません。-xs オプションは、コンパイラのリンカーに、すべてのデバッグ情報を実行可能プログラムに配置するよう命令するオプションです。したがって、これらのモジュールをデバッグするのに、.o ファイルの存在は必要ありません。

dbx 4.0 では、-xs オプションでコンパイルされたモジュールのデバッグ情報は、dbx の起動時に読み込まれます。そのため、-xs でコンパイルされた大きなプログラムでは、dbx の起動に時間がかかることがあります。

dbx 5.0 では、-xs でコンパイルされたモジュールのデバッグ情報は、.o ファイルに保存されたデバッグ情報と同様に遅延して読み込まれます。しかし、dbx に、これらの情報を起動時に読み込むよう命令することができます。dbx の環境変数 delay_xs を使用すれば、-xs でコンパイルされたモジュールのデバッグ情報の読み込み遅延機能をオフにすることができます。この環境変数を設定するには、.dbxrc ファイルに次の行を追加します。

```
dbxenv delay_xs off
```

Sun WorkShop デバッグの「デバッグオプション」ダイアログを使用して、この変数を設定することもできます。Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「-xs を使ってコンパイルしたモジュールの読み込みを遅延する」を参照してください。

モジュールについてのデバッグ情報

`module` コマンドおよびそのオプションは、デバッグセッション中、プログラムモジュールを追跡するのに役立ちます。`module` コマンドを使用して、1 つまたはすべてのモジュールについてのデバッグ情報を読み込みます。通常 `dbx` は、必要に応じて、自動的にゆっくりとモジュールについてのデバッグ情報を読み込みます。

1 つのモジュール *name* についてのデバッグ情報を読み込むには、次のように入力します。

```
(dbx) module [-f] [-q] name
```

すべてのモジュールについてのデバッグ情報を読み込むには、次のように入力します。

```
(dbx) module [-f] [-q] -a
```

ここで、

`-a`	すべてのモジュールを指定します。
`-f`	ファイルが実行可能より新しい場合でも、デバッグ情報を強制的に読み込みます。
`-q`	静止モードを指定します。
`-v`	言語、ファイル名などを印刷する冗長モードを指定します。

現在のモジュール名を出力するには、次のように入力します。

```
(dbx) module
```

モジュールのリスト

modules コマンドは、モジュール名をリストすることにより、モジュールを追跡することができます。

すでに dbx に読み取られたデバッグ情報を含むモジュールの名前をリスト表示するには、次のように入力します。

```
(dbx) modules [-v] -read
```

すべてのプログラムモジュール名 (デバッグ情報付き、またはなし) をリスト表示するには、次のように入力します。

```
(dbx) modules [-v]
```

デバッグ情報付きのすべてのプログラムモジュール名をリスト表示するには、次のように入力します。

```
(dbx) modules [-v] -debug
```

ここで、

-v　　言語、ファイル名などを出力する冗長モードを指定します。

第5章

プログラムの実行制御

実行、ステップ、および続行に使用されるコマンド (run、rerun、next、step、および cont) は、**プロセス制御**コマンドと呼ばれます。付録 B で説明するイベント管理コマンドとともに使用すると、プログラムが dbx のもとで実行されるときに、その実行時の動作を管理できます。

この章は次の各節から構成されています。

- dbx でプログラムを実行する
- 動作中のプロセスに dbx を接続する
- プロセスから dbx を切り離す
- プログラムのステップ実行
- Control+C によってプロセスを停止する

dbx でプログラムを実行する

プログラムを初めて dbx に読み込むと、dbx はそのプログラムの「メイン」ブロック (C、C++、および Fortran 90 の場合は main、FORTRAN 77 の場合は MAIN) 上を移動します。dbx は続いて、ユーザーから出されるコマンドを待機します。ユーザーは、コード上を移動するか、イベント管理コマンドを使用できます。

プログラムを実行する前に、そのプログラムにブレークポイントを設定することもできます。準備ができたら、run コマンドを使用してプログラムを実行してください。

dbx で引数を指定しないでプログラムを実行するには、次のように入力します。

```
(dbx) run
```

任意でコマンド行の引数と入出力の切り替えを追加できます。この場合は、次のように入力します。

```
(dbx) run [arguments] [ < input_file] [ > output_file]
```

run コマンドの出力は、dbx を実行しているシェルに noclobber を設定した場合でも、既存ファイルを上書きします。

run コマンドそのものは、前の引数とリダイレクトを使用して、プログラムを実行します。詳細については、332 ページの「run コマンド」を参照してください。rerun コマンドは、元の引数とリダイレクトなしでプログラムを実行します。詳細については、330 ページの「rerun コマンド」を参照してください。

動作中のプロセスに dbx を接続する

すでに動作中のプログラムをデバッグしなければならないことがあります。動作中のプロセスにデバッグ機能を接続しなければならないのは、次のような場合です。

- 動作中のサーバーをデバッグしたいが、停止させたくない
- 動作中の GUI プログラムをデバッグしたいが、再起動したくない
- プログラムが無限ループに入っているかもしれないので、プログラムを停止させずにデバッグしたい

このような場合は、動作中のプログラムの プロセス ID (*process_id*) を引数として dbx コマンドに渡せば、そのプログラムに dbx を接続することができます。

デバッグを終了すると、detach コマンドが使用され、プロセスを終了することなく dbx の制御からプログラムを解放することができます。

動作中のプロセスに接続されているときに dbx を終了すると、dbx は終了前に自動的な切り離しを行います。

dbx とは関係なく実行されるプログラムへ dbx を接続するには、attach コマンドまたは debug コマンドを使用します。

すでに実行中のプロセスへ dbx を接続するには、次のように入力します。

```
(dbx) debug program_name process_id
または

(dbx) attach process_id
```

program_name を - (ダッシュ) で置換することができます。dbx は、 プロセス ID と関連するプログラムを自動的に検索し、ロードします。

詳細については、298 ページの「debug コマンド」および 282 ページの「attach コマンド」を参照してください。

dbx が実行中でない場合は、次のように入力して dbx を開始します。

```
% dbx program_name process_id
```

プログラムに dbx を接続すると、そのプログラムは実行を停止します。このプログラムは、dbx に読み込んだプログラムの場合と同様にして調べることができます。任意のイベント管理コマンドまたはプロセス制御コマンドを使用してデバッグできます。

特定の例外がある接続済みプロセスで実行時チェック機能を使用できます。135 ページの「接続されたプロセスへの RTC の使用」を参照してください。

「デバッグ」ウィンドウで「デバッグ」➤「プロセスを接続」を選択することにより、実行中のプロセスに dbx を接続することもできます。詳細については、Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「実行中のプロセスへの接続」を参照してください。

プロセスから dbx を切り離す

プログラムのデバッグが終了したら、detach コマンドを使用して dbx をプログラムから切り離してください。プログラムは dbx から独立して動作を再開します。

dbx の制御のもとで、プロセスを実行から切り離すには、次のように入力します。

```
(dbx) detach
```

詳細については、300 ページの「detach コマンド」を参照してください。

「デバッグ」ウィンドウで「実行」➤「プロセスを切り離す」を選択することにより、プロセスから dbx を切り離すこともできます。詳細については、Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「プロセスからの切り離し」を参照してください。

プログラムのステップ実行

dbx は、next、step というステップ実行のための基本コマンドに加え、ステップ実行の変形である step up と step to をサポートします。next と step はともに、プログラムにソースの 1 行を実行させ、停止します。

実行される行に関数呼び出しが含まれる場合、next コマンドにより、呼び出しは実行され、次の行で停止します (呼び出しを "ステップオーバー")。step コマンドは、呼び出された関数の最初の行で停止します (呼び出しへの "ステップ")。「デバッグ」ウィンドウで「実行」➤「ステップオーバー」を選択するか、ツールバーの「ステップオーバー」ボタンをクリックすることにより、関数をステップオーバーすることもできます。「実行」➤「ステップイン」を選択するか、ツールバーの「ステップイン」ボタンをクリックすることにより、関数をステップ実行することができます。

step up コマンドは、関数をステップ実行した後、呼び出し元の関数へプログラムを戻します。「デバッグ」ウィンドウで「実行」➤「ステップアウト」を選択するか、ツールバーの「ステップアウト」ボタンをクリックすることにより、関数をステップアウトすることもできます。

step to コマンドは、現在のソースファイルで指定されている関数にステップするか、関数が指定されていない場合は、現在のソース行のアセンブリコードにより最後に呼び出される関数にステップします。条件付の分岐により、関数の呼び出しが発生しないことがあります。また、現在のソース行で関数が呼び出されない場合もあります。このような場合、step to は現在のソース行をステップオーバーします。

詳細については、Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「プログラムのステップ実行」を参照してください。

シングルステップ

指定された数のコード行をシングルステップするには、実行したいコードの行数 [*n*] を付けた dbx コマンド、next または step を使用します。

```
(dbx) next n

または

(dbx) step n
```

step_granularity 環境変数は、ソース行ステップの細分化、つまり、コード行をステップ実行する必要のある next コマンドの数を決定します。詳細については、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「step_granularity 環境変数」を参照してください。

コマンドについての詳細は、321 ページの「next コマンド」および 337 ページの「step コマンド」を参照してください。

プログラムを継続する

プログラムを継続するには、cont コマンドを使用します。

```
(dbx) cont
```

「デバッグ」ウィンドウで「実行」➤「継続」を選択するか、ツールバーの「継続」ボタンをクリックすることにより、プログラムの実行を継続することもできます。

cont コマンドには、派生関数の cont at *line_number* があります。これを使用すると、現在のプログラム位置の行以外の行を指定して、プログラムの実行を再開することができます。これにより、再コンパイルすることなく、問題を起こすことがわかっている 1 行または複数行のコードをスキップできます。

指定の行でプログラムを継続するには、次のように入力します。

```
(dbx) cont at 124
```

行番号は、プログラムが停止しているファイルから計算される点に注意してください。指定した行番号は、関数のスコープ内になければなりません。

`cont at` *line_number* と `assign` とを組み合わせると、ある変数の値を正しく計算できない関数の呼び出しが含まれている行を実行しないようにすることができます。

特定の行からプログラムの実行を再開するには、次のようにします。

1. **`assign` を使用して変数に正しい値を代入します。**
2. **`cont at` *line-number* で、その値を正しく計算できない関数の呼び出しが含まれている行を飛ばします。**

プログラムが 123 行目で停止しているものとします。その行では、関数 `how_fast()` を呼び出しています。この関数は、変数 `speed` を正しく計算しません。`speed` の正しい値がわかっているため、`speed` に値を代入することができます。その後、`how_fast()` の呼び出しを飛ばして、プログラムの実行を 124 行目から継続します。

```
(dbx) assign speed = 180; cont at 124;
```

詳細については、288 ページの「cont コマンド」を参照してください。

このコマンドを `when` ブレークポイントコマンドとともに使用すると、プログラムは 123 行目の実行を試みるたびに `how_fast()` の呼び出しを飛ばします。

```
(dbx) when at 123 { assign speed = 180; cont at 124;}
```

when コマンドについての詳細は、次の節を参照してください。

- 72 ページの「ソースコードの特定の行に stop ブレークポイントを設定する」
- 75 ページの「異なるクラスのメンバー関数にブレークポイントを設定する」
- 75 ページの「同じクラスのメンバー関数にブレークポイントを設定する」
- 76 ページの「非メンバー関数に複数のブレークポイントを設定する」
- 354 ページの「when コマンド」

関数を呼び出す

プログラムが停止しているとき、dbx コマンド `call` を使用して関数を呼び出すことができます。`call` コマンドには、被呼び出し側関数に渡す必要のあるパラメータの値を指定することもできます。

関数 (手続き) を呼び出すには、関数の名前を入力し、その引数を指定します。たとえば、次のように入力します。

```
(dbx) call change_glyph(1,3)
```

引数 *parameters* は省略できますが、関数名 *function_name* の後には必ず括弧を入力してください。たとえば、次のように入力します。

```
(dbx) call type_vehicle()
```

`call` コマンドを使用して関数を明示的に呼び出したり、関数呼び出しを含む式を評価するか、`stop in glyph -if animate()` などの条件付修飾子を使用して、関数を暗示的に呼び出すことができます。

C++ 仮想関数は、`print` コマンドや `call` コマンド (326 ページの「print コマンド」または 283 ページの「call コマンド」参照) を使用するその他の関数、または関数呼び出しを実行するその他のコマンドと同様に呼び出すことができます。

関数が定義されているソースファイルが `-g` フラグでコンパイルされたものであるか、プロトタイプ宣言が現在のスコープで可視であれば、`dbx` は引数の数と型をチェックし、不一致があったときはエラーメッセージを出します。それ以外の場合、`dbx` は引数の数をチェックしません。

デフォルトでは、`call` コマンドが実行されるたびに、`dbx` は `fflush(stdout)` を自動的に呼び出し、入出力バッファに格納されているすべての情報を出力します。自動的なフラッシュをオフにするには、`dbxenv output_autoflush` を `off` に設定してください。

C++ の場合、`dbx` はデフォルト引数と関数の多重定義も処理します。可能であれば、C++ 多重定義関数の自動解析が行われます。関数を特定できない場合は (関数が `-g` でコンパイルされていない場合など)、多重定義名のリストが表示されます。

`call` を使用すると、`dbx` は `next` のように動作し、被呼び出し側から戻ります。しかし、プログラムが被呼び出し側関数でブレークポイントにあたると、`dbx` はそのブレークポイントでプログラムを停止し、メッセージを表示します。ここで `where` コマンドを実行すると、`dbx` コマンドのレベルを起点として呼び出しが行われたことが示されます。

実行を継続すると、呼び出しは正常に戻ります。強制終了、実行、再実行、デバッグを行おうとすると、dbx は入れ子になったインタプリタから回復しようとするので、コマンドが異常終了します。異常終了したコマンドは再発行することができます。また、pop -c コマンドを使用して、すべてのフレームを最後の呼び出しまでポップ (解放) することもできます。

Control+C によってプロセスを停止する

dbx で実行中のプロセスは、Control +C (^C) を使用して停止できます。^C によってプロセスを停止すると、dbx は ^C を無視しますが、子プロセスはそれを SIGINT と見なして停止します。このプロセスは、それがブレークポイントによって停止しているときと同じように検査することができます。

^C によってプログラムを停止した後に実行を再開するには、コマンド cont を使用します。実行を再開する場合、cont に修飾語 sig <シグナル名> は必要ありません。cont コマンドは、保留シグナルをキャンセルした後で子プロセスを再開します。

第6章

ブレークポイントとトレースの設定

dbx を使用すると、イベント発生時に、プロセスの停止、任意のコマンドの発行、または情報を表示することができます。イベントの最も簡単な例はブレークポイントです。その他のイベントの例として、障害、シグナル、システムコール、`dlopen()` の呼び出し、データ変更などがあります。

トレースは、変数の値の変更など、プログラム内のイベントに関する情報を表示します。トレースの動作はブレークポイントと異なりますが、トレースとブレークポイントは類似したイベントハンドラを共有します (258 ページの「イベントハンドラ」を参照してください)。

この章では、ブレークポイントとトレースを設定、クリア、およびリストする方法について説明します。ブレークポイントおよびトレースの設定に使用できるイベント仕様の完全な詳細については、260 ページの「イベント指定の設定」 を参照してください。

この章は、次の各節から構成されています。

- ブレークポイントを設定する
- ブレークポイントのフィルタの設定
- トレースの実行
- ソース行で when ブレークポイントを設定する
- 共用ライブラリでブレークポイント を設定する
- ブレークポイントをリストおよびクリアする
- ブレークポイントを有効および無効にする
- イベント効率

ブレークポイントを設定する

dbx では、ブレークポイントを設定するため、3 種類のコマンドを使用することができます。

- stop ブレークポイント – stop コマンドによって作成されたブレークポイントに到達すると、プログラムは停止します。停止したプログラムはほかの dbx コマンドを実行するまで再開されません。
- when ブレークポイント - プログラムは、when コマンドで作成されたブレークポイントに到達すると処理を停止し、1 つまたは複数のデバッグコマンドの実行後に処理を再開します。プログラムは、実行コマンドに stop が含まれていない限り処理を継続します。
- trace ブレークポイント - プログラムは、trace コマンドで作成されたブレークポイントに到達すると処理を停止し、イベント固有のトレース情報行を出力した後、処理を再開します。

stop、when、および trace コマンドはすべて、イベントの指定を引数として取ります。イベントの指定は、ブレークポイントのベースとなるイベントを説明しています。イベント指定の詳細については、260 ページの「イベント指定の設定」を参照してください。

マシンレベルのブレークポイントを設定するには、stopi、wheni、tracei コマンドを使用します (第 17 章を参照)。

ソースコードの特定の行に stop ブレークポイントを設定する

stop at コマンドを使用して、行番号にブレークポイントを設定します。ここで、*n* はソースコードの行番号、*filename* は任意のプログラムファイル名修飾子です。

```
(dbx) stop at filename: n
```

ここで、*n* はソースコードの行番号、*filename* はそのコードが含まれているソースファイルの名前 (省略可能) です。たとえば、次のようにします。

```
(dbx) stop at main.cc:3
```

指定された行が、ソースコードの実行可能行ではない場合、dbx は次の有効な実行可能行にブレークポイントを設定します。実行可能な行がない場合、dbx はエラーを出します。

停止する行を決定するには、file コマンドを使用して現在のファイルを指定し、list コマンドを使用して停止する関数をリストします。その後、stop at コマンドを使用して、ソース行にブレークポイントを設定します。

```
(dbx) file t.c
(dbx) list main
10    main(int argc, char *argv[])
11    {
12        char *msg = "hello world\n";
13        printit(msg);
14    }
(dbx) stop at 13
```

at an location イベントを指定する詳細については、242 ページの "at [filename:] lineno" を参照してください。

「ブレークポイント」ウィンドウ内でもブレークポイントを設定できます。詳細については、Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「ブレークポイント (行に設定)」を参照してください。

関数に stop ブレークポイントを設定する

dbx stop in コマンドを使用して、関数にブレークポイントを設定します。

```
(dbx) stop in function
```

指定関数中で停止するブレークポイントは、プロシージャまたは関数の最初のソース行の冒頭でプログラムの実行を中断します。

「デバッグ」ウィンドウで、関数にブレークポイントを設定することもできます。詳細については、Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「ブレークポイント (関数に設定)」を参照してください。

dbx は、以下の場合を除いては、ユーザーが参照している変数または関数を決定します。

- 名前のみで、オーバーロードした関数を参照する場合
- 先頭に ` が付く関数または変数を参照する場合

次の宣言を考えてみましょう。

```
int foo(double);
int foo(int);
int bar();
class x {
   int bar();
};
```

メンバーでない関数で停止する場合、次のように入力して、

```
stop in foo(int)
```

グローバル関数 foo(int) にブレークポイントを設定します。

メンバー関数にブレークポイントを設定するには、次のコマンドを使用します。

```
stop in x::bar()
```

次のように入力すると、

```
stop in foo
```

dbx は、ユーザーがグローバル関数 foo(int)、グローバル関数 foo(double) のどちらを意味しているのかを判断することができず、明確にするため、オーバーロードしたメニューを表示する場合があります。

次のように入力すると、

```
stop in `bar
```

dbx は、ユーザーがグローバル関数 bar()、メンバー関数 bar() のどちらを意味しているのかを判断することができないため、オーバーロードしたメニューを表示します。

in function イベントを指定する詳細については、260 ページの「in function」を参照してください。

C++ プログラムに複数のブレークポイントを設定する

異なるクラスのメンバー関数の呼び出し、特定のクラスのすべてのメンバー関数の呼び出し、または多重定義されたトップレベル関数の呼び出しに関連する問題が発生する可能性があります。このような場合に対処するために、inmember、inclass、infunction または inobject のキーワードのうちの 1 つを stop, when,または trace コマンドとともに使用することにより、1 回のコマンドで C++ コードに複数のブレークポイントを挿入できます。

異なるクラスのメンバー関数にブレークポイントを設定する

特定のメンバー関数のオブジェクト固有のもの (同じメンバー関数名でクラスの異なるもの) それぞれにブレークポイントを設定するには、stop inmember を使用します。

たとえば、関数 draw が複数の異なるクラスに定義されている場合は、それぞれの関数ごとにブレークポイントを設定します。

```
(dbx) stop inmember draw
```

inmember または inmethod イベントを指定する詳細については、261 ページの「inmember function inmethod function」 を参照してください。

「ブレークポイント」ウィンドウで、「指定メンバー中で」ブレークポイントを設定することもできます。詳細については、Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「指定メンバー中でブレークポイントの設定」を参照してください。

同じクラスのメンバー関数にブレークポイントを設定する

特定のクラスのすべてのメンバー関数にブレークポイントを設定するには、stop inclass コマンドを使用します。

ブレークポイントは、クラスで定義されたクラスメンバー関数にのみ挿入され、ベースクラスから継承した関数には挿入されません。

クラス shape のすべてのメンバー関数にブレークポイントを設定するには、次のように入力します。

```
(dbx) stop inclass shape
```

inclass イベントを指定する詳細については、261 ページの「inclass classname」を参照してください。

「ブレークポイント」ウィンドウで、「指定クラスの中で」ブレークポイントを設定することもできます。詳細については、Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「指定クラスの中でブレークポイントの設定」を参照してください。

stop inclass およびその他のブレークポイントを選択することにより、大量のブレークポイントが挿入される場合があるため、dbx 環境変数 step_events を必ず on に設定し、step および next コマンドの実行速度を上げるようにしてください (86 ページの「イベント効率」参照)。

非メンバー関数に複数のブレークポイントを設定する

多重定義された名前を持つ非メンバー関数 (同じ名前を持ち、引数の型または数の異なるもの) に複数のブレークポイントを設定するには、stop infunction コマンドを使用します。

たとえば、C++ プログラムで sort () という名前の関数が 2 種類定義されていて、一方が int 型の引数、もう一方が float 型の引数をとる場合に、両方の関数にブレークポイントを置くためには、次のように入力します。

```
(dbx) stop infunction sort {command;}
```

infunction イベントを指定する詳細については、260 ページの「in function」を参照してください。

「ブレークポイント」ウィンドウで、「指定関数中で」ブレークポイントを設定することもできます。詳細については、Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「ブレークポイント (関数に設定)」を参照してください。

オブジェクトにブレークポイントを設定する

In Object ブレークポイントを設定し、特定のオブジェクトインスタンスに適用する操作をチェックします。In Object ブレークポイントは、オブジェクトからの呼び出し時に、オブジェクトのクラスのすべての非静的メンバー関数 (継承された関数も含む) でプログラムを中断します。

オブジェクト foo にブレークポイントを設定するには、次のように入力します。

```
(dbx) stop inobject &foo
```

inobject イベントを指定する詳細については、262 ページの「inobject object-expression」を参照してください。

「ブレークポイント」ウィンドウで、「指定オブジェクト中で」ブレークポイントを設定することもできます。詳細については、Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「指定オブジェクト中でブレークポイントの設定」を参照してください。

データ変更ブレークポイントを設定する

dbx でデータ変更ブレークポイントを使用すると、変数値や式がいつ変更されたかをメモしておくことができます。

特定アドレスへのアクセス時にプログラムを停止する

特定のメモリーアドレスがアクセスされたときにプログラムを停止するには、次のように入力します。:

```
(dbx) stop access mode address-expression [, byte-size-expression]
```

mode は、メモリーのアクセス方法を指定します。この引数は、次の文字の 1 つまたはすべてで構成されます。

r	指定アドレスのメモリーが読み取られた .
w	メモリーに書き込まれた
x	メモリーの内容が実行された

mode は、次のいずれかの文字も含むことができます。

a　　アクセス後にプロセスを停止する (デフォルト)

b　　アクセス前にプロセスを停止する

両方の場合において、プログラムカウンタは違反命令をポイントします。「前」と「後」は副作用を指しています。

address-expression は、アドレス生成のための評価が可能な任意の式です。記号式を指定した場合、監視される領域のサイズは自動的に導出されます。そのサイズを無効にするには、*byte-size-expression* を指定します。また、記号ではない無型のアドレス式を使用することもできます。この場合、サイズの指定は必須です。

次の例では、メモリーアドレス `0x4762` が読み取られた後にプログラムが停止します。:

```
(dbx) stop access r 0x4762
```

次の例では、変数 speed に書き込みが行われる前にプログラムが停止します。:

```
(dbx) stop access wb &speed
```

`stop access` コマンドを使用する場合、次の点に注意してください。

- 変数に同じ値が書きこまれてもイベントが発生します。
- デフォルトにより、変数に書き込まれた命令の実行後にイベントが発生します。命令が実行される前にイベントを発生させるには、モードを b を指定します。

`access` イベントを指定する詳細については、262 ページの「access mode address-expression [, byte-size-expression]」を参照してください。

変数の変更時にプログラムを停止する

指定した変数の値が変更された場合にプログラム実行を停止するには、次のように入力します。:

```
(dbx) stop change variable
```

stop change コマンドを使用する場合、次の点に注意してください。

- dbx は、指定の変数の値に変更が発生した行の次の行でプログラムを停止します。
- *variable* が関数に対しローカルである場合、関数が初めて呼び出されて variable の記憶領域が割り当てられた時点で、変数に変更が生じたものとみなされます。パラメータについても同じことが言えます。
- マルチスレッドのアプリケーションに対し、このコマンドは機能しません。

change イベントを指定する詳細については、263 ページの「change variable」 を参照してください。

Sun WorkShop 「ブレークポイント」ウィンドウで 指定変数の値変更時ブレークポイントを設定することもできます。詳細については、Sun WorkShop オンラインヘルプを参照してください。

dbx は、自動シングルステップを実行し、各ステップで値をチェックすることにより、stop change を実装します。ライブラリが -g オプションでコンパイルされていない場合、ステップ実行においてライブラリの呼び出しが省略されます。そのため、制御が次のように流れていく場合、dbx はネストされた user_routine2 をトレースしません。トレースにおいて、ライブラリの呼び出しとネストされた user_routine2 の呼び出しが省略されるからです。

```
user_routine calls
   library_routine, which calls
     user_routine2, which changes variable
```

variable の値の変更は、user_routin2 が実行されている最中ではなく、ライブラリが呼び出しから戻った後に発生したように見えます。

dbx は、ブロックローカル変数 ({} でネストされている変数) の変更に対しブレークポイントを設定できません。「ネスト」されたブロックローカル変数でブレークポイントまたはトレースを設定しようとすると、その操作を実行できない旨を伝えるエラーメッセージが表示されます。

注 – change イベントよりも access イベントを使用した方が、迅速にデータ変更をチェックできます。access イベントは、自動的にプログラムをシングルステップする代わりに、はるかに迅速なページ保護スキーマを使用するからです。

条件付きでプログラムを停止する

条件文が真と評価された場合にプログラムを停止するには、次のように入力します。:

```
(dbx) stop cond condition
```

condition が発生すると、プログラムは処理を停止します。

`stop cond` コマンドを使用する場合、次の点に注意してください。

- `dbx` は、条件が真と評価された行の次の行でプログラムを停止します。
- このコマンドは、マルチスレッドのアプリケーションに対し機能しません。

condition イベントを指定する詳細については、263 ページの「cond condition-expression」を参照してください。

Sun WorkShop「ブレークポイント」ウィンドウで、指定条件成立時のカスタムブレークポイントを設定することもできます。詳細については、Sun WorkShop オンラインヘルプを参照してください。

ブレークポイントのフィルタの設定

`dbx` では、ほとんどのイベント管理コマンドが *event filter* 修飾子をオプションでサポートします。最も単純なフィルタは、プログラムがブレークポイントかトレースハンドラに到達した後、またはウォッチ条件の発生した後に、`dbx` に対してある特定の条件をテストするように指示します。

このフィルタの条件が真 (非 0) と評価された場合、イベントコマンドが適用され、プログラムはブレークポイントで停止します。条件が偽 (0) と評価された場合、`dbx` は、イベントが発生しなかったかのようにプログラムの実行を継続します。

フィルタを含む行または関数にブレークポイントを設定するには、オプションの `-if` *condition* 修飾文を `stop` コマンドまたは `trace` コマンドの末尾に追加します。

condition には、任意の有効な式を指定できます。コマンドの入力時に有効だった言語で書かれた、ブール値または整数値を返す関数呼び出しも有効な式に含まれます。

in や at など位置に基づくブレークポイントでは、スコープはブレークポイント位置のスコープになります。それ以外の場合、イベントではなくエントリ発生時のスコープになります。スコープを正確に指定するために逆引用符演算子 (51 ページの「逆引用符演算子」を参照) を使用しなければならないことがあります。

たとえば、次の 2 つのフィルタは異なります。

```
stop in foo -if a>5
stop cond a>5
```

前者は foo にブレークポイントが設定され、条件を検査します。後者は自動的に条件を検査します。

関数の戻り値をブレークポイントフィルタとして使用できます。次の例では、文字列 str の値が abcde の場合、プログラムが関数 foo() で停止します。:

```
(dbx) stop in foo -if !strcmp("abcde",str)
```

ブレークポイントフィルタの設定に変数スコープを使用できます。この例で、現在のスコープは in function foo() であり、local は main() で定義されたローカル変数です。

```
(dbx) stop access w &main`local -if pr(main`local) -in main
```

最初のうちは、条件付イベントコマンド (watch タイプのコマンド) の設定と、フィルタの使用とを混同してしまうかもしれません。概念的には、watch タイプのコマンドは、各行の実行前に検査される前提条件を作成します (watch のスコープ内で)。ただし、条件付トリガーのあるブレークポイントコマンドでも、それに接続するフィルタを持つことができます。

次に具体的な例を示します。:

```
(dbx) stop access w &speed -if speed==fast_enough
```

このコマンドは、変数 *speed* を監視するように dbx に指令します。*speed* に書き込みが行われると、-if フィルタが有効になります。dbx は *speed* の新しい値が fast_enough と等しいかどうかチェックします。等しくない場合、プログラムは実行を継続し、stop を「無視」します。

dbx 構文では、フィルタはブレークの「事後」、構文の最後で [-if *condition*] 文の形式で指定されます。

```
stop in function [-if condition]
```

イベント修飾子の詳細については、270 ページの「イベント指定のための修飾子」を参照してください。

トレースの実行

トレースは、プログラムの処理状況に関する情報を収集して表示します。プログラムが trace コマンドで作成されたブレークポイントに到達すると、プログラムの処理が停止され、イベント固有のトレース情報行が出力された後、処理が再開されます。

トレースは、ソースコードの各行を実行直前に表示します。極めて単純なプログラムを除くすべてのプログラムで、このトレースは大量の出力を生成します。

さらに便利なトレースは、フィルタを利用してプログラムのイベント情報を表示します。たとえば、関数の各呼び出し、特定の名前のすべてのメンバー関数、クラス内のすべての関数、または関数の各 exit をトレースできます。また、変数の変更もトレースできます。

トレースを設定する

コマンド行に trace コマンドを入力することにより、トレースを設定します。trace コマンドの基本構文は次の通りです。

```
trace event-specification [ modifier ]
```

トレースコマンドの完全な構文については、345 ページの「trace コマンド」を参照してください。

「ブレークポイント」ウィンドウで、トレースを設定することもできます。詳細については、Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「トレースの設定」を参照してください。

トレースで提供される情報は、トレースに関連する *event* の型に依存します (260 ページの「イベント指定の設定」を参照)。

トレース速度を制御する

トレースの出力が速すぎる場合がよくあります。dbx 環境変数 `trace_speed` を使用すると、各トレースの出力後の遅延を制御できます。デフォルトの遅延は 0.5 秒です。

トレース時の各行の実行間隔を設定するには、次のように入力します。

```
dbxenv trace_speed number
```

「デバッグオプション」ダイアログで、トレースの実行速度を設定することもできます。詳細については、Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「トレース速度の設定」を参照してください。

ファイルにトレース出力を転送する

`-file` *filename* オプションを使用すると、トレース出力をファイルに転送できます。たとえば、次のコマンドはトレース出力をファイル `trace1` に転送します。

```
(dbx) trace -file trace1
```

トレース出力を標準出力に戻すには、*filename* の代わりに -を使用します。トレース出力は常に *filename* に追加されます。これらの出力は、dbx がプロンプトを表示した際、およびアプリケーションの終了時に消去されます。dbx 接続後にプログラムの実行を再開するか新たに実行を開始すると、*filename* が常に開きます。

ソース行で when ブレークポイントを設定する

`when` ブレークポイントコマンドは `list` などその他の `dbx` コマンドを受け付けるため、ユーザーは独自のトレースを作成できます。.

```
(dbx) when at 123 { list $lineno;}
```

when コマンドは暗黙の cont コマンドとともに機能します。上の例では、現在の行のソースコードをリストした後、プログラムが実行を継続します。

when コマンドの完全な構文については、354 ページの「when コマンド」を参照してください。イベント修飾子の詳細については、270 ページの「イベント指定のための修飾子」を参照してください。

共用ライブラリでブレークポイント を設定する

dbx は、実行時リンカーにおけるプログムを呼び出すインターフェースを使用しているコード (dlopen()、dlclose ()、および関連関数を呼び出すコード) について完全なデバッグサポートを提供します。実行時リンカーは、プログラム実行の最中、共用ライブラリを結合および結合解除します。dlopen()/dlclose() のデバッグサポートが提供されているため、ユーザーは関数内部を操作したり、プログラムの起動時にリンクされたライブラリの場合と同様に、動的な共用ライブラリの関数にブレークポイントを設定することができます。

ただし、例外もあります。dbx は、dlopen() などで読み込まれていないロードオブジェクトにブレークポイントを配置できません。

- dlopen() で読み込まれる前のライブラリにブレークポイントを設定できません。
- ライブラリの最初の関数が呼び出されるまで、dlopen() で読み込まれるフィルタライブラリにブレークポイントを設定できません。

loadobjects コマンドを使用すると、事前読み込みリストにそれらのロードオブジェクトの名前を配置できます (316 ページの「loadobjectsコマンド」 を参照してください)。

dbx は、dlopen() で読み込まれたロードオブジェクトを確実に管理します。たとえば、新たに読み込まれたロードオブジェクトに設定されたブレークポイントは、次の run コマンドが実行されるまで維持されます。これは、ロードオブジェクトが dlclose() でロード解除された後、再度 dlopen() で読み込まれた場合も同様です。

ブレークポイントをリストおよびクリアする

dbx セッション中にブレークポイントやトレースポイントを複数設定することがよくあります。dbx には、それらのポイントを表示したりクリアしたりするためのコマンドが用意されています。

ブレークポイントとトレースポイントの表示

すべての有効なブレークポイントのリストを表示するには、status コマンドを使用します。ブレークポイントは ID 番号付きで表示され、この番号はほかのコマンドで使用できます。

C++ の多重ブレークポイントのところでも説明したように、dbx はキーワード inmember、inclass、infunction で設定された多重ブレークポイントを、1 つのステータス ID 番号を使用してまとめて報告します。

ステータス ID 番号を使用して特定のブレークポイントを削除

status コマンドを使用してブレークポイントをリスト表示した場合、dbx は、各ブレークポイントの作成時に割り当てられた ID 番号を表示します。 delete コマンドを使用することで、ID 番号によってブレークポイントを削除したり、キーワード all により、プログラム内のあらゆる場所に現在設定されているブレークポイントをすべて削除することができます。

ブレークポイントを ID 番号 *ID_number* によって削除するには、次のように入力します。

```
(dbx) delete 3 5
```

dbx に現在読み込まれているプログラムに設定されているすべてのブレークポイントを削除するには、次のように入力します。

```
(dbx) delete all
```

詳細については、300 ページの「delete コマンド」を参照してください。

ブレークポイントを有効および無効にする

ブレークポイントの設定に使用するイベント管理コマンド (stop、trace、when) は、イベントハンドラを作成します (258 ページの「イベントハンドラ」を参照してください)。これらの各コマンドは、ハンドラ ID (*hid*) として認識される番号を返します。ハンドラ ID は、ブレークポイントを有効または無効にする handler コマンド (309 ページの「handler コマンド」) の引数として利用できます。

Sun WorkShop「ブレークポイント」ウィンドウでブレークポイントを有効および無効にすることもできます (Sun WorkShop オンラインヘルプを参照してください)。

イベント効率

デバッグ中のプログラムの実行時間に関するオーバーヘッドの量はイベントの種類によって異なります。最も単純なブレークポイントのように、実際はオーバーヘッドが何もないイベントもあります。1 つのブレークポイントしかないイベントも、オーバーヘッドは最小です。

実際のブレークポイントがときには何百にもなることのある多重ブレークポイント (inclass など) は、コマンド発行時にのみオーバーヘッドがあります。これは、dbx が永続的ブレークポイントを使用するためです。永続的ブレークポイントは、プロセスに常に保持され、停止するたびに取り除かれたり、cont のたびに置かれたりすることはありません。

注 – step および next の場合、デフォルトでは、プロセスが再開される前にすべてのブレークポイントが取り除かれ、ステップが完了するとそれらは再び挿入されます。したがって、多くのブレークポイントを使用したり、多くのクラスで多重ブレークポイントを使用したりしているとき、step および next の速度は大幅に低下します。dbxenv 変数 step_events を使用して、step や next のたびにブレークポイントを取り除いたり、挿入し直したりするかどうかを制御することができます。

自動ステップ実行を利用するイベントは最も低速です。これは、各ソース行をステップ実行する単純な `trace step` コマンドの場合と同様にはっきりしています。一方、`stop change` *expression* や `trace cond` *variable* のようなイベントは、自動的にステップ実行するだけでなく、各ステップで式や変数を評価する必要があります。

これらのイベントは非常に低速ですが、イベントと修飾語 `-in` を使用した関数とを結び付けることで、効率が上がることがよくあります。たとえば、次のようにします。

```
trace next -in mumble
stop change clobbered_variable -in lookup
```

`trace -in main` を使用しないでください。これは、`main` によって呼び出された関数の中でも、トレースが有効になるためです。関数 `lookup()` が変数の値を頻繁に変更すると思われる場合には、この方法を使用してください。

第7章

呼び出しスタックの使用

この章では、dbx による呼び出しスタックの使用方法と、呼び出しスタックを処理するときの where、hide、および unhide コマンドの使用方法について説明します。

呼び出しスタックは、現在活動中のルーチン、すなわち、呼び出されているがそれぞれの呼び出し側に戻っていない関数群を表します。スタックフレームは、単一関数に割り当てられる呼び出しスタックのセクションです。

呼び出しスタックがメモリー上位 (上位アドレス) からメモリー下位に成長することから、*up* は呼び出し側 (最終的には main()) のフレームに向かうこと、そして *down* は呼び出された関数 (最終的には現在の関数) のフレームに向かうことを意味します。プログラムの現在位置 (ブレークポイント、ステップ実行の後、プログラムが異常終了してコアファイルが作成された、のいずれかの時点で実行されていたルーチン) はメモリー上位に存在しますが、main() のような呼び出し側ルーチンはメモリー下位に位置します。

この章は次の各節から構成されています。

- スタック上での現在位置の検索
- スタックを移動してホームに戻る
- スタックを上下に移動する
- 呼び出しスタックのポップ
- スタックフレームを隠す
- スタックトレースを表示して確認する

スタック上での現在位置の検索

where コマンドを使用すると、スタックでの現在位置を検索できます。

```
where [-f] [-h] [l] [-q] [-v] number_id
```

where コマンドは、クラッシュしてコアファイルを作成したプログラムの状態を知る場合にも役立ちます。プログラムがクラッシュしてコアファイルを作成した場合、そのコアファイルを dbx に読み込むことができます。(16 ページの「既存のコアファイルのデバッグ」、および Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「dbx によるコアファイルのデバッグ」を参照。)

where コマンドについての詳細は、355 ページの「where コマンド」を参照してください。

スタックを移動してホームに戻る

スタックを上下に移動して関数を表示すると、dbx はスタックの状態を表示し、矢印でその関数を示します。プログラムが停止している位置を「ホーム」と呼び、このホームを起点にし、up コマンド、down コマンド、frame コマンドを使用してスタックを上下に移動することができます。

dbx コマンドの up および down は、ともに引数として、スタック内で現在のフレームから移動するフレームの数を指定する値 (*number*) を受け付けます。*number* を指定しない場合、デフォルトは 1 です。-h オプションを指定すると、隠されたフレームもカウントされます。

スタックを上下に移動する

現在の関数以外の関数にあるローカル変数を調べることができます。

スタックの上方向への移動

呼び出しスタックを *number* で指定されたレベル分、上に (`main` に向かって) 移動するには、次のように入力します。

```
up [-h] [ number ]
```

number を指定しない場合、デフォルトは 1 レベルになります。詳細については、350 ページの「up コマンド」を参照してください。

スタックの下方向への移動

呼び出しスタックを *number* で指定されたレベル分、下に (現在の停止点に向かって) 移動するには、次のように入力します。

```
down [-h] [ number ]
```

number を指定しない場合、デフォルトは 1 レベルになります。詳細については、302 ページの「down コマンド」を参照してください。

「デバッグ」ウィンドウで呼び出しスタックの周りを移動することもできます (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「呼び出しスタック内の移動」を参照してください)。

特定フレームへの移動

`frame` コマンドは、`up` コマンドや `down` コマンドと同じような働きをします。このコマンドに `where` コマンドで得た番号を指定すると、その番号によって特定されるフレームに直接移動できます。このコマンドは、コマンド行でしか実行できません。

```
frame
frame -h
frame [-h] number
frame [-h] +number
frame [-h] -number
```

引数なしの `frame` コマンドは、現在のフレーム番号を出力します。*number* を指定すると、その番号によって示されるフレームに直接移動できます。"+" または "-" だけを指定すると、現在のフレームから 1 レベルだけ上 (+) または下 (-) に移動できます。また、正負の符号と *number* をともに指定すると、指定した数のレベルだけ上または下に移動できます。`-h` オプションをつけると、隠されたフレームもカウントされます。

pop コマンドを使用するか (92 ページの「呼び出しスタックのポップ」参照)、または「デバッグ」ウィンドウを使用して (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「現在のフレームへの呼び出しスタックのポップ」参照)、特定のフレームへ移動することもできます。

呼び出しスタックのポップ

呼び出しスタックから、停止した関数を削除し、呼び出し中の関数を新たに指定関数で停止する関数にすることができます。

呼び出しスタックの上下方向への移動とは異なり、スタックのポップは、プログラムの実行を変更します。スタックから停止した関数が削除されると、プログラムは以前の状態に戻ります。ただし、大域または静的変数、外部ファイル、共有メンバー、および同様のグローバル状態への変更は対象外です。

pop コマンドは、呼び出しスタックから 1 つまたは複数のフレームを削除します。たとえば、スタックから 5 つのフレームをポップするには、次のように入力します。

```
pop 5
```

指定のフレームへポップすることもできます。フレーム 5 へポップするには、次のように入力します。

```
pop -f 5
```

詳細については、325 ページの「pop コマンド」を参照してください。

「デバッグ」ウィンドウで、呼び出しスタックをポップすることもできます。詳細については、Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「一度に 1 つの関数ずつ呼び出しスタックをポップする」、「現在のフレームへの呼び出しスタックのポップ」、「呼び出しスタックからのデバッガフレームのポップ」を参照してください。

スタックフレームを隠す

hide コマンドを使用して、現在有効なスタックフレームフィルタをリスト表示します。

正則表現に一致するすべてのスタックフレームを隠すか、または削除するには、次のように入力します。

```
hide [ regular_expression ]
```

regular_expression は、関数名、またはロードオブジェクト名のいずれかを表し、ファイルの照合に sh または ksh の構文を使用します。

すべてのスタックフレームフィルタを削除するには、unhide を使用します。

```
unhide 0
```

hide コマンドは、番号とともにフィルタをリスト表示するため、このフィルタ番号を使用して unhide コマンドを使用することもできます。

```
unhide [ number | regular_expression ]
```

スタックトレースを表示して確認する

スタックトレースは、実行の停止した時のプログラムのフローと、その地点に到達するまでの過程を示します。プログラムの状態について最も正確な記述を提供します。

スタックトレースを表示するには、where コマンドを使用します。

-g オプションでコンパイルされた関数の場合、引数の名前と種類が既知であるため、正確な値が表示されます。デバッグ情報のない関数の場合、引数に対し 16 進数の数値が表示されます。これらの数値は必ずしも意味を持ちません。関数ポインタ 0 を介して関数が呼び出される場合、記号名の代わりに関数の値が下位 16 進数として示されます。

-g オプションでコンパイルされていない関数でトレースを停止できます。このような関数でトレースを停止すると、dbx はスタックを検索し、関数が -g オプションでコンパイルされている最初のフレームを探し、現在の適用範囲 (46 ページの「スコープ」 を参照) をそのフレームに設定します。このことは矢印記号 (=>) によって示されます。

次の例で、main() は -g オプションでコンパイルされているため、記号名と引数の値が表示されます。main() によって呼び出されたライブラリ関数は、-g でコンパイルされていないため、関数の記号名は表示されますが、引数については、$i0 から $i5 までの SPARC 入力レジスタの 16 進数の内容が示されます。

```
(dbx) where
  [1] _libc_poll(0xffbef3b0, 0x1, 0xffffffff, 0x0, 0x10,
0xffbef604), at 0xfef9437c
  [2] _select(0xffbef3b8, 0xffbef580, 0xffbef500, 0xffbef584,
0xffbef504, 0x4), at 0xfef4e3dc
  [3] _XtWaitForSomething(0x5a418, 0x0, 0x0, 0xf4240, 0x0, 0x1),
at 0xff0bdb6c
  [4] XtAppNextEvent(0x5a418, 0x2, 0x2, 0x0, 0xffbef708, 0x1), at
0xff0bd5ec
  [5] XtAppMainLoop(0x5a418, 0x0, 0x1, 0x5532d, 0x3, 0x1), at
0xff0bd424
=>[6] main(argc = 1, argv = 0xffbef83c), "main.cc" の 48 行目
```

次の例で、 プログラムはセグメント例外によりクラッシュしています。今回も main() だけが -g でコンパイルされているため、ライブラリ関数の引数が記号名ではなく 16 進数で表示されています。クラッシュの原因は、SPARC 入力レジスタ $i0 および $i1 において strlen() にヌルの引数が指定されたことにあると考えられます。

```
(dbx) run
実行中: Cdlib
(プロセス id 25805)

CD Library Statistics:0

 Titles:         11

 Total time:     9:29:30
 Average time:   0:51:46

シグナル SEGV (フォルトのアドレスにマッピングしていません) 関数 strlen
0xff2b6d4c で
0xff2b6d4c: strlen+0x0080:ld      [%o1], %o2
現関数 :main
  140       printf (" Longest CD:   %s  %d\n", 0x00, 0x00);
(dbx) where
  [1] strlen(0x0, 0x0, 0x0, 0x7efefeff, 0x81010100, 0xff33993c),
アドレス 0xff2b6d4c
  [2] _doprnt(0x117e1, 0x0, 0x0, 0x0, 0x2e, 0xff00), アドレス
0xff2b6d4c
  [2] _doprnt(0x117e1, 0x0, 0x0, 0x0, 0x2e, 0xff00)、アドレス
0xff2ff0c8
  [3] printf(0x117d0, 0xff33688c, 0xff33689c, 0xff33a1bc,
0xff332584, 0xff00)、アドレス 0xff300c30
=>[4] main(argc = 1, argv = 0xffbef08c), "Cdlib.c" の 140 行目
(dbx)
```

スタックトレースの 例については、10 ページの「呼び出しスタックを確認する」および 213 ページの「呼び出しのトレース」を参照してください。

第8章

データの評価と表示

dbx では、次の 2 通りの方法でデータをチェックすることができます。

- データの評価 (print) - 任意の式の値を検査します。
- データの表示 (display) - プログラムが停止するたびに式の値を検査し監視することができます。

この章は次の各節から構成されています。

- 変数と式の評価
- 式に値を代入する
- 配列を評価する

変数と式の評価

この節は、dbx を使用して変数および式を評価する方法について説明します。

実際に使用される変数を確認する

dbx がどの変数を評価するか確かでないときは、which コマンドを使用して dbx が使用する完全修飾名を調べてください 。

変数名が定義されているほかの関数やファイルを調べるには、whereis コマンドを使用します。

詳細については、356 ページの「which コマンド」および356 ページの「whereis コマンド」を参照してください。

現在の関数のスコープ外にある変数

現在の関数のスコープ外にある変数を評価 (監視) したい場合は、次のようにします。

- 関数の名前を修飾します。50 ページの「スコープ決定演算子を使用してシンボルを修飾する」を参照してください。

または

- 現在の関数を変更することにより、関数を表示します。47 ページの「停止位置とは別の部分のコードを表示する」を参照してください。

変数または式の値を出力する

式はすべて、現在の言語構文に従う必要がありますが、dbx がスコープおよび配列を処理するために導入したメタ構文は除きます。

変数または式を評価するには、次のように入力します。

```
print expression
```

詳細については、326 ページの「print コマンド」を参照してください。

注 – dbx は、C++ の `dynamic_cast` および `typeid` 演算子をサポートしています。これらの 2 つの演算子で式を評価すると、dbx は、コンパイラで提供された特定の rtti 関数へ呼び出しを行います。ソースが明示的に演算子を使用しない場合、これらの関数はコンパイラで生成されない場合があり、dbx は式を評価することができません。

C++ での表示

C++ では、オブジェクトポインタに 2 つの型があります。1 つは**静的な型**で、ソースコードに定義されています。もう 1 つは**動的な型**です。dbx は、動的な型のオブジェクトに関する情報を提供できる場合があります。

通常、オブジェクトに仮想関数テーブルの vtable が含まれる場合、dbx はこの vtable 内の情報を使用して、オブジェクトの型を正しく知ることができます。

print または display コマンドに -r (再帰的) オプションをつけて使用すると、dbx は、クラスによって直接定義されたデータメンバーすべてと、基底クラスから継承されたものを表示することができます。

これらのコマンドには、-d または +d オプションも使用できます。これは、dbxenv output_derived_type でデフォルト動作を切り替えることができます。

プロセスが何も実行されていないときに、-d フラグを使用するか、またはdbxenv output_dynamic_type を on に設定すると、プロセスがないときに動的情報にアクセスすることは不可能なため、プログラムは実行可能な状態ではないことを表すエラーメッセージが出されます。仮想継承から動的な型の検索を試みると、クラスポインタの不正なキャストを表すエラーメッセージが生成されます (仮想基底クラスから派生クラスへのキャストは C++ では無効です)。

C++ プログラムにおける無名引数を評価する

C++ では、次のように無名の引数を持つ関数を定義できます。

```
void tester(int)
{
};
main(int, char **)
{
   tester(1);
};
```

無名の引数はプログラム内のほかの場所では使用できませんが、dbx は無名引数を評価できる形式にコード化します。その形式は次のとおりです。

```
_ARG%n
```

ここで、dbx は %n に整数を割り当てます。

dbx によって割り当てられた引数名を入手するには、調べたい関数名を指定した whatis コマンドを実行します。

```
(dbx) whatis tester
void tester(int _ARG0);
(dbx) whatis main
int main(int _ARG1, char **_ARG2);
```

詳細については、352 ページの「whatis コマンド」を参照してください。

無名の関数引数を評価 (表示) するには、次のようにします。

```
(dbx) print _ARG1
_ARG1 = 4
```

ポインタを間接参照する

ポインタを間接参照すると、ポインタが指している内容に格納された値を参照できます。

ポインタを間接参照すると、dbx は評価結果を表示します。次の例は、ポインタを間接参照した場合です。

```
(dbx) print *t
*t = {
a = 4
}
```

「デバッグ」ウィンドウで、ポインタを間接参照することもできます。Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「ポインタの間接参照」を参照してください。

式を監視する

プログラムが停止するたびに式の値を監視することにより、特定の式または変数がいつどのように変化するかを効果的に知ることができます。display コマンドは、指定されている 1 つまたは複数の式または変数を監視するように dbx に命令します。監視は、undisplay コマンドによって取り消されるまで続けられます。

プログラムが停止するたびに変数または式の値を表示するには、次のようにします。

```
display expression, ...
```

一度に複数の変数を監視できます。オプションを指定しないで display コマンドを使用すると、監視対象のすべての式が表示されます。

詳細については、301 ページの「display コマンド」を参照してください。

「デバッグ」ウィンドウで、式の値を監視することもできます。Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「式の追加」を参照してください。

表示を取り消す (非表示)

監視している変数の値の表示は、undisplay コマンドで「表示」を取り消すまで続けられます。特定の式だけを表示しないようにすることも、現在監視しているすべての式の表示を中止することも可能です。

特定の変数または式の表示をオフにするには、次のようにします。

```
undisplay expression
```

現在監視しているすべての変数の表示をオフにするには、次のようにします。

```
undisplay 0
```

詳細については、348 ページの「undisplay コマンド」を参照してください。

「デバッグ」ウィンドウで、指定の式、またはすべての式の表示をオフにすることができます。Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「式の削除」を参照してください。

式に値を代入する

変数に値を代入するには、次のようにします。

```
assign variable = expression
```

「デバッグ」ウィンドウで、変数へ値を代入することもできます。Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「データ値の変更」を参照してください。

配列を評価する

配列の評価は、ほかの種類の変数を評価する場合と同じ方法で行います。

Fortran の配列の例 :

```
integer*4 arr(1:6, 4:7)
```

配列を評価するには、print コマンドを使用します。たとえば、次のようにします。

```
(dbx) print arr(2,4)
```

dbx コマンドの `print` を使用して、大型の配列の一部を評価することができます。配列を評価するには、次の操作を行います。

- 配列の断面化 - 多次元配列から任意の矩形ブロックまたは *n* 次元の領域を取り出して出力します。
- 配列の刻み - 指定された配列の断面 (配列全体のこともあります) から決まったパターンで特定の要素だけを取り出して出力します。

刻みは配列の断面化を行うときに必要に応じて指定することができます (刻みのデフォルト値は 1 で、その場合は各要素を出力します)。

配列の断面化

C、C++、Fortran では、`print` および `display` コマンドによって、配列の断面化を行うことができます。

C と C++ での配列の断面化の構文

配列の各次元を断面化するための print コマンドの完全な構文は次のとおりです。

`print` *array-expression [first-expression .. last-expression : stride-expression]*

arr-exp	配列またはポインタ型に評価されるべき式
first-exp	印刷される最初の要素。デフォルトは 0
last-exp	印刷される最後の要素。その上限にデフォルト設定される
stride-exp	刻み幅の長さ (スキップされる要素の数は *stride-expression*-1)。デフォルトは 1

最初、最後、および刻み幅の各式は、整数に評価されなければならない任意指定の式です。

例

```
(dbx) print arr[2..4]
arr[2..4] =
[2] = 2
[3] = 3
[4] = 4
(dbx) print arr[..2]
arr[0..2] =
[0] = 0
[1] = 1
[2] = 2

(dbx) print arr[2..6:2]
arr[2..6:2] =
[2] = 2
[4] = 4
[6] = 6
```

Fortran のための配列断面化構文

配列の各次元を断面化するための `print` コマンドの完全な構文は次のとおりです。

```
print array-expression (first-expression: last-expression : stride-expression)
```

array-expression	配列型に評価される式

first-expression	範囲内の最初の要素は、出力される最初の要素。下限にデフォルト設定
last-expression	範囲内の最後の要素。ただし刻み幅が 1 でない場合、出力される最後の要素とはならない。上限にデフォルト設定
stride-expression	刻み幅 。デフォルトは 1

最初、最後、および刻み幅の各式は、整数に評価されなければならない任意の式です。*n* 次元の断面については、カンマで各断面の定義を区切ります。

例

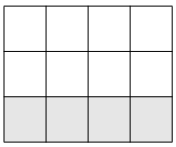
```
(dbx) print arr(2:6)
arr(2:6) =
(2) 2
(3) 3
(4) 4
(5) 5
(6) 6

(dbx) print arr(2:6:2)
arr(2:6:2) =
(2) 2
(4) 4
(6) 6
```

行と列を指定するには、次のように入力します。

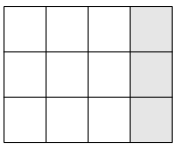
```
demo% F77 -g -silent ShoSli.f
demo% dbx a.out
a.out の読み込み中
(dbx) list 1,12
    1   INTEGER*4 a(3,4), col, row
    2   DO row = 1,3
    3       DO col = 1,4
    4         a(row,col) = (row*10) + col
    5       END DO
    6   END DO
    7   DO row = 1, 3
    8        WRITE(*,'(4I3)') (a(row,col),col=1,4)
    9   END DO
    10  END
(dbx) stop at 7
(1) stop at "ShoSli.f":7
(dbx) run
実行中 : a.out
MAIN で停止しました。行番号 7 ファイル "ShoSli.f"
    7    DO row = 1, 3
```

行 3 を印刷するには、次のように入力します。



```
(dbx) print a(3:3,1:4)
'ShoSli'MAIN'a(3:3, 1:4) =
        (3,1)   31
        (3,2)   32
        (3,3)   33
        (3,4)   34
(dbx)
```

列 4 を印刷するには、次のように入力します。



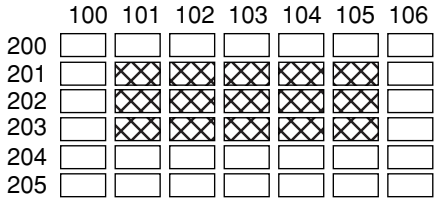
```
(dbx) print a(1:3,4:4)
'ShoSli'MAIN'a(1:3, 1:4) =
        (1,4)   14
        (2,4)   24
        (3,4)   34
(dbx)
```

配列の断面

2 次元の矩形配列の断面の例を示します。ここでは、刻み値が省略され、デフォルト値の 1 が使用されます。

```
print arr(201:203, 101:105)
```

このコマンドは、大型配列の要素のブロックを出力します。*stride-expression* が省略され、デフォルトの刻み値である 1 が使用されていることに注意してください。



最初の 2 つの式 (`201:203`) は、この 2 次元配列の第 1 次元 (3 行で構成される列) を指定します。配列の断面は行 `201` から始まり、行 `203` で終わります。次の 2 つの式 (`101:105`) は最初の組とコンマで区切られ、第 2 次元の配列の断面を定義します。配列の断面は列 `101` から始まり、列 `105` で終わります。

刻み幅

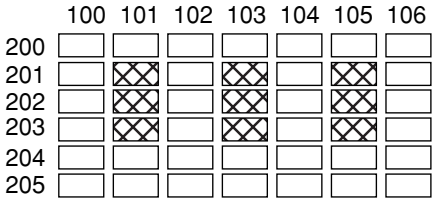
`print` コマンドで刻み幅を指定すると、配列の断面に含まれる特定の要素だけが評価されます。

配列の断面のための構文の 3 番目の式 (*stride-expression*) は、刻み幅の長さを指定します。*stride-expression* の値は印刷する要素を指定します。刻み幅のデフォルト値は 1 です。このとき、指定された配列の断面のすべての要素が評価されます。

ここに、上の例で使用したものと同じ配列があります。今度は、`print` コマンドの第 2 次元の配列の断面の定義に刻み幅の値として 2 を加えます。

```
print arr(201:203, 101:105:2)
```

刻み値として 2 を指定すると、各行を構成する要素が 1 つおきに出力されます。



print コマンドの配列の断面の定義を構成する式を省略すると、配列の宣言されたサイズに等しいデフォルト値が使用されます。このような簡易構文を使用した例を以下に示します。

1 次元配列の場合

`print arr`	デフォルトの境界で配列全体を出力します。
`print arr(:)`	デフォルトの境界とデフォルトの刻み (1) で、配列全体を出力します。
`print arr(::`*stride-expression*`)`	配列全体を *stride-expression* で指定された刻み幅で出力します。

2 次元配列の場合、次のコマンドは配列全体を出力します。

```
print arr
```

2 次元配列の第 2 次元を構成する要素を 2 つおきに出力します。

```
print arr (:,::3)
```

第9章

実行時検査

実行時検査 (RTC) を行うと、開発段階においてアプリケーションの実行時エラー (メモリーアクセスエラー、メモリーリークなど) を自動的に検出できます。メモリーの使用状況も監視できます。

この章は次の各節から構成されています。

- 概要
- RTC の使用
- メモリーアクセスエラーの検出 (SPARC のみ)
- メモリーリークの検査
- メモリー使用状況検査の使用
- エラーの抑止
- 子プロセスにおける RTC の実行
- 接続されたプロセスへの RTC の使用
- RTC での修正継続機能の使用
- 実行時検査アプリケーションプログラミング インタフェース
- バッチモードでの RTC の使用
- トラブルシューティングのヒント

注 – アクセス検査は SPARC でのみ利用可能です。

概要

RTC は、統合的なデバッグ機能であり、標本コレクタによるパフォーマンスデータの収集時を除けば、実行時にあらゆるデバッグ機能を利用できます。

次に、RTC の機能を簡単に説明します。

- メモリーアクセスエラーを検出する
- メモリーリークを検出する
- メモリー使用に関するデータを収集する
- すべての言語で動作する
- マルチスレッドコードで動作する
- 再コンパイル、再リンク、またはメークファイルの変更が不要である

`-g` フラグを付けてコンパイルすると、RTC エラーメッセージでのソース行番号の関連性が与えられます。RTC は、最適化 `-O` フラグによってコンパイルされたプログラムを検査することもできます。`-g` オプションによってコンパイルされていないプログラムについては、特殊な考慮事項があります。

RTC を実行するには、`dbx` コマンドを入力するか、「デバッグ」ウィンドウや「実行時検査」ウィンドウを利用します。Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「実行時検査の開始」を参照してください。

RTC を使用する場合

大量のエラーが一度に検出されないようにするには、RTC を開発サイクルの初期の段階で使用します。この段階では、プログラムの構成要素となる個々のモジュールを開発します。この各モジュールを実行する単位テストを作成し、RTC を各モジュールごとに 1 回ずつ使用して検査を行います。これにより、一度に処理するエラーの数が減ります。すべてのモジュールを統合して完全なプログラムにした場合、新しいエラーはほとんど検出されません。エラー数をゼロにした後でモジュールに変更を加えた場合にのみ、RTC を再度実行してください。

RTC の必要条件

RTC を使用するには、次の要件を満たす必要があります。

- Sun のコンパイラを使用してコンパイルされたプログラム
- `libc` を動的にリンクしている。
- `libc` の標準関数 `malloc`、`free`、`realloc` を利用するか、これらの関数を基にアロケータを使用します。RTC では、他のアロケータはアプリケーションプログラミングインタフェース (API) で操作します。139 ページの「実行時検査アプリケーションプログラミング インタフェース」を参照してください。

- 完全にストリップされていないプログラム。`strip -x` によってストリップされたプログラムは使用できます。

制限事項

実行時検査は、UltraSPARC™ プロセッサに基づいてないハードウェアにおいて、8M バイトより大きなプログラムのテキスト領域およびデータ領域を処理しません。詳細は、142 ページの「RTC の 8M バイト制限」を参照してください。

実行可能イメージに特殊ファイルを挿入すると、8Mバイトより大きなプログラムのテキスト領域とデータ領域を処理することができます。

RTC の使用

実行時検査を使用するには、使用したい検査の種類を指定します。

メモリー使用状況とメモリーリーク検査を有効化

メモリーの使用状況とメモリーリークの検査を有効にするには、次のように入力します。

```
(dbx) check -memuse
```

メモリーの使用状況とメモリーリークの検査は、「デバッグ」ウィンドウでも有効にできます。Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「すべてのメモリーリークの表示」を参照してください。

MUC か MLC がオンになっている場合、`showblock` コマンドを実行する、所定のアドレスにおけるヒープブロックに関する詳細情報を表示できます。この詳細情報では、ブロックの割り当て場所とサイズを知ることができます。詳細については、334 ページの「showblock コマンド」を参照してください。

メモリーアクセス検査を有効化

メモリーアクセス検査だけをオンにするには、次のように入力します。

```
(dbx) check -access
```

メモリーアクセスの検査も「デバッグ」ウィンドウで有効にできます。詳細については、Sun WorkShop オンラインヘルプの「「デバッグウィンドウの使い方」の「メモリーアクセス検査を有効にする」を参照してください。

すべての RTC を有効化

メモリーリーク、メモリー使用状況、およびメモリーアクセスの各検査をオンにするには、次のように入力します。

```
(dbx) check -all
```

詳細については、284 ページの「check コマンド」を参照してください。

RTC を無効化

RTC をすべて無効にするには、次のように入力します。

```
(dbx) uncheck -all
```

詳細については、347 ページの「uncheck コマンド」を参照してください。

プログラムを実行

目的のタイプの RTC を有効にしてテストするプログラムを実行します。この場合、ブレークポイントを設定してもしなくてもかまいません。

プログラムは普通に実行されますが、すべてのメモリーアクセスがその妥当性を確認するために検査されるので、実行速度は遅くなります。無効なアクセスがあると dbx によって検出され、そのタイプとエラーの発生場所が表示されます。制御はユーザー

に戻ります (dbx 環境変数 rct_auto_continue が on になっている場合を除きます (Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「rtc_auto_continue 環境変数」参照))。

次に、dbx コマンドを実行します。where コマンドでは現在のスタックトレースを呼び出すことができます。また print を実行すれば変数を確認できます。エラーが致命的でなければ、cont コマンドでプログラムの処理を続行します。プログラムは次のエラーまたはブレークポイントまで、どちらか先に検出されるところまで実行されます。詳細については、288 ページの「cont コマンド」を参照してください。

rtc_auto_continue が on に設定されている場合、RTC はそのままエラーを求めて自動的に続行されます。検出したエラーは、dbx 環境変数 rtc_error_log_name で指定したファイルにリダイレクトされます (Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「rtc_error_log _file_name 環境変数」を参照してください)。

RTC エラーの報告が不要な場合は、suppress コマンドを使用します。詳細については、341 ページの「suppress コマンド」を参照してください。

次の例は、hello.c と呼ばれるプログラムのメモリーアクセス検査とメモリー使用状況検査をオンにする方法を示しています。

```
% cat -n hello.c
     1 #include <stdio.h>
     2 #include <stdlib.h>
     3 #include <string.h>
     4
     5 char *hello1, *hello2;
     6
     7 void
     8 memory_use()
     9 {
    10      hello1 = (char *)malloc(32);
    11      strcpy(hello1, "hello world");
    12      hello2 = (char *)malloc(strlen(hello1)+1);
    13      strcpy(hello2, hello1);
    14 }
    15
    16 void
    17 memory_leak()
    18 {
    19      char *local;
    20      local = (char *)malloc(32);
    21      strcpy(local, "hello world");
    22 }
    23
```

(続き)

```
    24 void
    25 access_error()
    26 {
    27      int i,j;
    28
    29      i = j;
    30 }
    31
    32 int
    33 main()
    34 {
    35      memory_use();
    36      access_error();
    37      memory_leak();
    38      printf("%s\n", hello2);
    39      return 0;
    40 }

% cc -g -o hello hello.c
% dbx -C hello
hello のシンボル情報を読んでいます
rtld /usr/lib/ld.so.1 のシンボル情報を読んでいます
librtc.so のシンボル情報を読んでいます
libc.so.1 のシンボル情報を読んでいます
libdl.so.1 のシンボル情報を読んでいます
(dbx) check -access
アクセス検査 - ON
(dbx) check -memuse
メモリー使用状況検査 - ON
(dbx) run
実行中: hello
(プロセス id 18306)
実行時検査を有効にしています...終了
非初期化領域からの読み取り (rui):
4 バイト読み取り を アドレス 0xefffff068 で しようとしました
    それは 96 バイト 現スタックポインタより上 です
変数は 'j' です。
現関数 :access_error
   29           i = j;
(dbx) cont
```

```
(続き)

メモリーリーク検査中...

実際のリークの報告        (実際のリーク:          1   合計サイズ:       32 バイト)

合計      ブロック  リーク      割り当て呼出しスタック
サイズ      数       ブロック
                     アドレス
======   ======  ==========  ========================================
    32         1     0x21aa8  memory_leak < main


起こり得るリークの報告    (起こり得るリーク:        0   合計サイズ        0 バイト)
メモリー使用状況検査中...

ブロック使用量の報告      (ブロック使用量:        2   合計サイズ:       44 バイト)


  合計   割合  ブロック  平均     割り当て呼出しスタック
 サイズ   %       数     サイズ
=======  ====  ======  ======  ========================================
     32   72%       1      32  memory_use < main
     12   27%       1      12  memory_use < main

実行完了。終了コードは、0 です
```

関数 access_error() は、初期化される前の変数 j を読み取ります。RTC は、このアクセスエラーを非初期化領域からの読み取り (rui) として報告します。

関数 memory_leak() は、終了する前に local を解放 (free()) しません。memory_leak() が終了してしまうと、local がスコープ外になり、行 20 で確保したブロックがリークになります。

プログラムは、常にスコープ内にある大域変数 hello1 と hello2 を使用します。これらの変数はいずれも、使用中ブロック (biu) として報告される割り当て済みメモリーを動的に指します。

メモリーアクセスエラーの検出 (SPARC のみ)

RTC では、読み取り、書き込み、メモリー解放の各操作を監視することによって、プログラムがメモリーに正しくアクセスするかどうかを検査します。

プログラムは、さまざまな方法で間違ってメモリーを読み取ったり、メモリーに書き込んだりすることがあります。このようなエラーをメモリーアクセスエラーといいます。たとえば、ヒープブロックの `free()` 呼び出しを使用して、または関数がローカル変数にポインタを返したために、プログラムが参照するメモリーブロックの割り当てが解放されている可能性があります。アクセスエラーはプログラムでワイルドポインタの原因になり、間違った出力やセグメント不正など、プログラムの異常な動作を引き起こす可能性があります。メモリーアクセスエラーには、検出が非常に困難なものもあります。

RTC は、プログラムによって使用されているメモリーの各ブロックの情報を追跡するテーブルを管理します。プログラムがメモリー操作を行うと、RTC は関係するメモリーブロックの状態に対してその操作が有効かどうかを判断します。メモリーの状態として次のものがあります。

- 未割り当て (初期) 状態。メモリーは割り当てられていません。この状態のメモリーはプログラムが所有していないため、読み取り、書き込み、解放のすべての操作が無効です。
- 割り当て済み/未初期化。メモリーはプログラムに割り当てられていますが、初期化されていません。書き込み操作と解放操作は有効ですが、初期化されていないので読み取りは無効です。たとえば、関数に入るときに、スタック上にメモリーが割り当てられますが、初期化はされません。
- 読み取り専用。読み取りは有効ですが、書き込みと解放は無効です。
- 割り当て済み/初期化済み。割り当てられ、初期化されたメモリーに対しては、読み取り、書き込み、解放のすべての操作が有効です。

アクセスエラーを見つけるには、アクセス検査をオンにし、プログラムにアクセスエラーのリストを作成させるのが一番簡単な方法です。 RTC を使用してメモリーアクセスエラーを見つける方法は、コンパイラがプログラム中の構文エラーを見つける方法と似ています。いずれの場合でも、プログラム中のエラーが発生した位置と、その原因についてのメッセージとともにエラーのリストが生成され、リストの先頭から順に修正していかなければなりません。これは、あるエラーがほかのエラーと関連して

連結されたような作用があるためです。連結の最初のエラーが先頭の原因となり、そのエラーを修正することにより、そのエラーから派生した他の問題も解決されることがあります。たとえば、初期化されていないメモリーの読み取りにより、不正なポインタが作成されるとします。すると、これが原因となって不正な読み取りと書き込みのエラーが発生し、それがまた原因となってさらに別の問題が発生するというようなことになる場合があります。

メモリーアクセスエラーの報告

メモリーアクセスエラーを検出すると RTC は次の情報を出力します。

エラー	情報
種類	エラーの種類。
アクセス	試みられたアクセスの種類 (読み込みまたは書き込み)。
サイズ	試みられたアクセスのサイズ。
アドレス	試みられたアクセスのアドレス
詳細	アドレスについてのさらに詳しい情報。たとえば、アドレスがスタックの近くに存在する場合、現在のスタックポインタからの相対位置が与えられます。アドレスが複数存在する場合、一番近いブロックのアドレス、サイズ、相対位置が与えられます。
スタック	エラー時の呼び出しスタック (バッチモード)。
割り当て	*addr* がヒープにある場合、最も近いヒープブロックの割り当てトレースが与えられます。
場所	エラーが発生した位置。行が特定できる場合には、ファイル名、行番号、関数が示されます。行番号が分からないときは関数とアドレスが示されます。

代表的なアクセスエラーは次のとおりです。

```
非初期化領域からの読み取り (rui):
4 バイト読み取り を アドレス 0xefffee50 で しようとしました
    それは 96 バイト 現スタックポインタより上 です
変数は 'j' です。
現関数 :rui
  12            i = j;
```

メモリーアクセスエラー

RTC は、以下のメモリーアクセスエラーを検出します。

- `rui` (147 ページの「非初期化メモリーからの読み取り (rui) エラー」参照)
- `rua` (146 ページの「非割り当てメモリーからの読み取り (rua) エラー」参照)
- `wua` (147 ページの「非割り当てメモリーへの書き込み (wua) エラー」参照)
- `wro` (147 ページの「読み取り専用メモリーへの書き込み (wro) エラー」参照)
- `mar` (145 ページの「境界整列を誤った読み取り (mar) エラー」参照)
- `maw` (146 ページの「境界整列を誤った書き込み (maw) エラー」参照)
- `duf` (145 ページの「重複解放 (duf) エラー」参照)
- `baf` (144 ページの「不正解放 (baf) エラー」参照)
- `maf` (145 ページの「境界整列を誤った解放 (maf) エラー」参照)
- `oom` (146 ページの「メモリー不足 (oom) エラー」参照)

注 – RTC では、配列境界チェックは行いません。したがって、配列境界侵害はアクセスエラーにはなりません。

メモリーリークの検査

メモリーリークとは、プログラムで使用するために割り当てられているが、プログラムのデータ領域中のいずれも指していないポインタをもつ、動的に割り当てられたメモリーブロックを言います。そのようなブロックは、メモリーのどこに存在しているかプログラムにわからないため、プログラムに割り当てられていても使用することも解放することもできません。RTC はこのようなブロックを検知し、報告します。

メモリーリークは仮想メモリーの使用を増やし、一般的にメモリーの断片化を招きます。その結果、プログラムやシステム全体のパフォーマンスが低下する可能性があります。

メモリーリークは、通常、割り当てメモリーを解放しないで、割り当てブロックへのポインタを失うと発生します。メモリーリークの例を以下に示します。

```
void
foo()
{
    char *s;
    s = (char *) malloc(32);

    strcpy(s, "hello world");

   return; /* s が解放されていない。foo が戻るとき、malloc されたブロック
              を指しているポインタが存在しないため、ブロックはリークする*/
}
```

リークは、API の不正な使用が原因で起こる可能性があります。

```
void
printcwd()
{

    printf("cwd = %s\n", getcwd(NULL, MAXPATHLEN));

    return; /* libc の関数 getcwd() は、最初の引数が NULL の場合 malloc
               された領域へのポインタを返す。プログラムは、これを解放する必要
               がある。この場合、ブロックが解放されていないため、結果的にリー
               クになる。*/
}
```

メモリーリークを防ぐには、必要のないメモリーは必ず解放します。また、メモリーを確保するライブラリ関数を使用する場合は、メモリーを解放することを忘れないでください。

解放されていないブロックを「メモリーリーク」と呼ぶこともあります。ただし、この定義はあまり使用されません。プログラムが短時間で終了する場合でも、通常のプログラミングではメモリーを解放しないからです。プログラムにそのブロックに対するポインタがある場合、RTC はそのようなブロックはメモリーリークとして報告しません。

メモリーリーク検査の使用

RTC では、以下のメモリーリークエラーを検出します。

- maf (149 ページの「メモリーリーク (mel) エラー」参照)
- maf (148 ページの「レジスタ中のアドレス (air)」参照)
- maf (148 ページの「ブロック中のアドレス (aib)」参照)

注 – RTC のリーク検査を実行するには、標準の libc の malloc、free、realloc のいずれか、またはこれらの関数にもとづいた割り当て関数を使用する必要があります。

リークの可能性

RTC が「リークの可能性」として報告するエラーには 2 種類あります。1 つは、ブロックの先頭を指すポインタが検知されず、ブロックの内部を指しているポインタが見つかった場合です。これは、ブロック中のアドレス (aib) エラーとして報告されます。このようなブロック内部を指すポインタが見つかった場合は、プログラムに実際にメモリーリークが発生しています。ただし、プログラムによってはポインタに対して故意にそのような動作をさせている場合があり、これは当然メモリーリークではありません。RTC はこの違いを判別できないため、本当にリークが発生しているかどうかはユーザー自身の判断で行う必要があります。

もう 1 つのリークの種類は、レジスタ中のアドレス (air) エラーとして報告されるリークです。これは、ある領域を指すポインタがデータ空間中には存在せず、レジスタ内に存在する場合です。レジスタがブロックを不正に指していたり、古いメモリーポインタが残っている場合には、実際にメモリーリークが発生しています。ただし、コンパイラが最適化のために、ポインタをメモリーに書き込むことなく、レジスタのブロックに対して参照させることがありますが、この場合はメモリーリークではありません。プログラムが最適化され、showleaks コマンドでエラーが報告された場合のみ、リークでない可能性があります。詳細については、335 ページの「showleaks コマンド」を参照してください。

注 – RTC リーク検査では、標準 libc malloc/free/realloc 関数またはアロケータをこれらの関数に基いて使用する必要があります。ほかのアロケータについては、137 ページの「RTC での修正継続機能の使用」を参照してください。

リークの検査

メモリーリーク検査がオンの場合、メモリーリークの走査は、テスト中のプログラムが終了する直前に自動的に実行されます。検出されたリークはすべて報告されます。プログラムを、kill コマンドによって強制的に終了してはなりません。次に、典型的なメモリーリークエラーによるメッセージを示します。

```
メモリーリーク (mel):
大きさ 1024 バイト のリークのあるブロックをアドレス 0x21718 に発見
割り当て時のスタックの状態
    [1] foo() 行番号 63 "test.c"
    [2] main() 行番号 47 "test.c"
```

「デバッグ」ウィンドウで、コールスタック場所のハイパーテキストリンクをクリックすると、エディタウィンドウのソースコード行に移動できます。

プログラムには通常 main (FORTRAN 77 では MAIN) 手続きが存在します。プログラムは exit(3) が呼び出されるか、main から返った時点で終了します。いずれの場合でも、main のすべての局所変数はプログラムが停止するまでスコープから出ず、それらを指す特定のヒープブロックはすべてメモリーリークとして報告されます。

main() に割り当てられているヒープブロックはプログラムでは解放しないのが一般的です。これらのヒープブロックはプログラムが停止するまでスコープ内に残り、プログラムの停止後オペレーティングシステムによって自動的に解放されるためです。main() に割り当てられたブロックがメモリーリークとして報告されないようにするには、main() が終了する直前にブレークポイントを設定しておきます。プログラムがそこで停止したとき、RTC の showleaks コマンドを実行すれば、main() とそこで呼び出されるすべての手続きで参照されなくなったヒープブロックのすべてが表示されます。

詳細については、335 ページの「showleaks コマンド」を参照してください。

または、「デバッグ」ウィンドウでもメモリーリークを検査できます。Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「すべてのメモリーリークの表示」を参照してください。

メモリーリークの報告を理解する

リーク検査を有効にすると、プログラムの終了時にリークレポートが自動的に生成されます。`kill` コマンドでプログラムを終了した場合を除き、リークの可能性がすべて報告されます。レポートの詳細レベルは、`dbx` 環境変数 `rtc_mel_at_exit` (Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「rtc_mel_at_exit 環境変数」参照) か、「デバッグオプション」ダイアログの「自動リーク報告」オプション (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「自動リーク報告の生成」参照) で制御します。デフォルトで、非冗長リークレポートが生成されます。

レポートは、リークのサイズによってソートされます。実際のメモリーリークが最初に報告され、次に可能性のあるリークが報告されます。詳細レポートには、スタックトレース情報の詳細が示されます。行番号とソースファイルが使用可能であれば、これらも必ず含まれます。

次のメモリーリークエラー情報が、2 種類の報告のどちらにも含まれます。

リークしたブロックが割り当てられた場所
リークしたブロックのアドレス
リークしたブロックのサイズ
割り当て時の呼び出しスタック。`check -frames` によって制約される

「実行時検査」ウィンドウに表示される非冗長レポートでは、エラー情報がテーブルにカプセル化されますが、冗長レポートでは、エラーごとに独立したエラーメッセージが表示されます。どちらもソースコードにおけるエラー位置までのハイパーテキストリンクを備えています。

次に、対応する簡易メモリーリークレポートを示します。

```
実際のリークの報告       (実際のリーク:       3  合計サイズ:  2427 バイト)

合計      ブロック  リーク       割り当て呼出しスタック
サイズ    数        ブロック
                    アドレス
======  ======   ========   ====================================
  1852       2       -       true_leak < true_leak
   575       1     0x22150   true_leak < main

起こり得るリークの報告   (起こり得るリーク:     1  合計サイズ     8 バイト)

合計      ブロック  リーク       割り当て呼出しスタック
サイズ    数        ブロック
                    アドレス
======  ======  ==========  ====================================
     8       1    0x219b0   in_block < main
```

次に、典型的な詳細リークレポートを示します。

```
実際のリークの報告       (実際のリーク:       3  合計サイズ:  2427 バイト)

メモリーリーク (mel):
大きさ 1 バイト のリークのあるブロックをアドレス 0x20f18 に発見
割り当て時のスタックの状態:
    [1] true_leak() 行番号 220 "leaks.c"
    [2] true_leak() 行番号 224 "leaks.c"


起こり得るリークの報告   (起こり得るリーク:     1  合計サイズ     8 バイト)

メモリーリークの可能性 -- ブロック中のアドレス (aib):
大きさ 4 バイト のリークのあるブロックをアドレス 0x20ef8 に発見
割り当て時のスタックの状態:
    [1] in_block() 行番号 177 "leaks.c"
    [2] main() 行番号 100 "leaks.c"
```

リークレポートの生成

showleaks コマンドを使用すると、いつでもリークレポートを要求することができます。このコマンドは、前回の showleaks コマンド以降の新しいメモリーリークを報告するものです。詳細については、335 ページの「showleaks コマンド」を参照してください。

リークレポート

リークレポートの数が多くなるのを避けるため、RTC は同じ場所で割り当てられたリークを自動的に 1 つにまとめて報告します。1 つにまとめるか、それぞれ各リークごとに報告するかは、一致フレーム数引数によって決まります。この引数は、check -leaks コマンドを実行する際は -match *m* オプション、showleaks コマンドを実行する際は -m オプションで指定します。呼び出しスタックが 2 つ以上のリークを割り当てる際に *m* 個のフレームと一致した場合は、リークは 1 つにまとめて報告されます。

以下の 3 つの呼び出しシーケンスを考えてみます。

ブロック 1	ブロック 2	ブロック 3
[1] malloc	[1] malloc	[1] malloc
[2] d() at 0x20000	[2] d() at 0x20000	[2] d() at 0x20000
[3] c() at 0x30000	[3] c() at 0x30000	[3] c() at 0x31000
[4] b() at 0x40000	[4] b() at 0x41000	[4] b() at 0x40000
[5] a() at 0x50000	[5] a() at 0x50000	[5] a() at 0x50000

これらのブロックがすべてメモリーリークを起こす場合、*m* の値によって、これらのリークを別々に報告するか、1 つのリークが繰り返されたものとして報告するかが決まります。*m* が 2 のとき、ブロック 1 とブロック 2 のリークは 1 つのリークが繰り返されたものとして報告されます。これは、malloc() の上にある 2 つのフレームが共通しているためです。ブロック 3 のリークは、c() のトレースがほかのブロックと一致しないので別々に報告されます。*m* が 2 より大きい場合、RTC はすべてのリークを別々に報告します (malloc はリークレポートでは表示されません)。

一般に、*m* の値が小さければリークのレポートもまとめられ、*m* の値が大きければまとめられたリークレポートが減り、別々のリークレポートが生成されます。

メモリーリークの修正

RTC からメモリーリーク報告を受けた場合にメモリーリークを修正する方法についてのガイドラインを以下に示します。

- リークの修正で最も重要なことは、リークがどこで発生したかを判断することです。作成されるリーク報告は、リークが発生したブロックの割り当てトレースを示します。リークが発生したブロックは、ここから割り当てられたことになります。次に、プログラムの実行フローを見て、どのようにそのブロックを使用したかを調べます。
- ポインタが失われた箇所が明らかな場合は簡単ですが、それ以外の場合は `showleaks` コマンドを使用してリークの検索範囲を狭くすることができます。`showleaks` コマンドは、デフォルトでは前回このコマンドを実行した後に検出されたリークのみを報告するため、`showleaks` を繰り返し実行することにより、ブロックがリークを起こした可能性のある範囲が狭まります。

詳細については、335 ページの「showleaks コマンド」を参照してください。

メモリー使用状況検査の使用

メモリ使用状況検査は、使用中のヒープメモリーすべてを確認することができます。この情報によって、プログラムのどこでメモリーが割り当てられたか、またはどのプログラムセクションが大半の動的メモリーを使用しているかを知ることができます。この情報は、プログラムの動的メモリー消費を削減するためにも有効であり、パフォーマンスの向上に役立ちます。

メモリー使用状況検査は、パフォーマンス向上または仮想メモリーの使用制御に役立ちます。プログラムが終了したら、メモリー使用状況レポートを生成できます。メモリー使用情報は、メモリーの使用状況を表示させるコマンドを使用して、プログラムの実行中に随時入手することもできます。詳細については、335 ページの「showmemuse コマンド」を参照してください。

メモリー使用状況検査をオンにすると、リーク検査もオンになります。プログラム終了時のリークレポートに加えて、使用中ブロック (`biu`) レポートも得ることができます。デフォルトでは、使用中ブロックの簡易レポートがプログラムの終了時に生成されます。これは、`dbxenv` 変数 `rtc_biu_at_exit` によって指定できます。(Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「rtc_biu_at_exit 環境変

数」参照) か、「デバッグオプション」ダイアログの「自動リーク報告」オプション (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「自動リーク報告の生成」参照) で制御します。

次に、典型的な簡易メモリー使用状況レポートを示します。

```
ブロック使用量の報告    (ブロック使用量: 5    合計サイズ:    40 バイト)

 合計      割合  ブロック   平均       割り当て呼び出しスタック
 サイズ     %     数    サイズ
======= ==== ====== ======  ======================================
     16  40%      2       8  nonleak < nonleak
      8  20%      1       8  nonleak < main
      8  20%      1       8  cyclic_leaks < main
      8  20%      1       8  cyclic_leaks < main
```

次に、対応する詳細メモリー使用状況レポートを示します。

```
ブロック使用量の報告    (ブロック使用量: 5    合計サイズ:    40 バイト)

ブロック使用状況 (biu):
サイズ 8 バイト のブロックをアドレス 0x25498 で見つけました (合計 40.00%)
割り当て時のスタックの状態:
      [1] nonleaks() 行番号 182 "memuse.c"
      [2] nonleaks() 行番号 185 "memuse.c"

ブロック使用状況(biu):
サイズ 8 バイト のブロックをアドレス 0x21898 で見つけました (合計 20.00%)
割り当て時のスタックの状態:
      [1] nonleaks() 行番号 182 "memuse.c"
      [2] main() 行番号 74 "main.c"

ブロック使用状況 (biu):
サイズ 8 バイト のブロックをアドレス 0x21958 で見つけました (合計 20.00%)
割り当て時のスタックの状態:
      [1] cycle_leaks() 行番号 154 "memuse.c"
      [2] main() 行番号 118 "main.c"

ブロック使用状況(biu):
サイズ 8 バイト のブロックをアドレス 0x21978 で見つけました (合計 20.00%)
割り当て時のスタックの状態:
      [1] cycle_leaks() 行番号 115 "memuse.c"
      [2] main() 行番号 118 "main.c"
```

showmemuse コマンドを使用すると、メモリー使用状況レポートをいつでも要求できます。

エラーの抑止

RTC はエラーレポートの数や種類を限定するよう、エラーの抑制機能を備えています。エラーが発生してもそれが抑制されている場合は、エラーは無視され、報告されずにプログラムは継続します。

エラーは suppress コマンド (341 ページの「suppress コマンド」参照) で抑止できます。最新のエラー報告を抑止するには、「実行時検査」ウィンドウの「最後に報告されたリークの抑止」ボタン (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「最後に報告されたエラーの抑止」参照) を使用します。

エラー抑止を取り消すには、unsuppress コマンド (350 ページの「unsuppress コマンド」参照) を使用します。

抑止機能は同じデバッグ節内の run コマンドの実行期間中は有効ですが、debug コマンドを実行すると無効になります。

抑止のタイプ

次の抑制機能があります。

スコープと種類による抑制

どのエラーを抑止するかを指定する必要があります。以下のように、プログラムのどの部分に抑制を適用するかを指定できます。

大域	スコープが指定されていないと全体のスコープが対象になり、すべてのプログラムに適用されます。
ロードオブジェクト	共有ライブラリなど、すべてのロードオブジェクトが対象になります。
ファイル	特定のファイルのすべての関数が対象になります。

関数	特定の関数が対象になります。
行	特定のソース行が対象になります。
アドレス	特定のアドレスが対象になります。

最新エラーの抑止

デフォルトで RTC を実行すると、最新のエラーで同じエラーが繰り返し報告されることがなくなります。この機能は、dbx 環境変数 rtc_auto_suppress で制御します。rtc_auto_suppress が on のとき (デフォルト)、特定個所の特定エラーは最初の発生時にだけ報告され、その後同じエラーが同じ場所で発生しても報告が繰り返されることはありません。最新エラーを抑止すると、繰り返し実行するループに 1 つのエラーがあっても、それが何度も報告されることがなく、便利です。

エラー報告回数の制限

dbx 環境変数 rtc_error_limt では、報告されるエラーの回数を制御します。エラー制限は、アクセスエラーとリークエラーに別々に設定します。たとえば、エラー制限を 5 に設定すると、プログラムの終了時のリークレポートと、showleaks コマンドの実行ごとに、アクセスエラーとリークエラーがそれぞれ最高で 5 回報告されます。デフォルトは 1000 です。

エラー抑止の例

次の例では、main.cc はファイル名、foo と bar は関数を示し、a.out は実行可能ファイルの名前を示します。

割り当てが関数 foo で起こったメモリーリークは報告しません。

```
suppress mel in foo
```

libc.so.1 から割り当てられた使用中のブロック報告を抑止します。

```
suppress biu in libc.so.1
```

a.out の非初期化機能からの読み取りを抑止します。

```
suppress rui in a.out
```

ファイル main.cc の非割り当てメモリーからの読み取りを報告しません。

```
suppress rua in main.cc
```

main.cc の行番号 10 での重複解放を抑止します。

```
suppress duf at main.cc:10
```

関数 bar のすべてのエラー報告を抑止します。

```
suppress all in bar
```

詳細については、341 ページの「suppress コマンド」を参照してください。

デフォルトの抑止

RTC では、-g オプション (記号) を指定してコンパイルを行わなくてもすべてのエラーを検出できます。しかし、非初期化メモリーからの読み取りなど、正確さを保証するのに 記号 (-g) 情報が必要な特定のエラーもあります。このため、a.out の rui や 共有ライブラリの rui, aib, air など特定のエラーは、記号情報が取得できない場合は、デフォルトで抑制されます。この動作は、suppress や unsuppress コマンドの -d オプションを使用することで変更できます。

たとえば、以下を実行すると、RTC は記号情報が存在しない (-g オプションを指定しないでコンパイルした) コードについて「非初期化メモリーからの読み取り (rui)」を抑制しません。

```
unsuppress -d rui
```

詳細については、350 ページの「unsuppress コマンド」を参照してください。

抑止によるエラーの制御

プログラムが大きい場合、エラーの数もそれに従って多くなることが予想されます。このような場合は、suppress コマンドを使用することにより、エラーレポートの数を管理しやすい大きさまで抑制し、一度で修正するエラーを制限します。抑制するエラーの数を徐々に減らしながら、この動作を繰り返してください。

たとえば、一度で検出するエラーをタイプによって制限できます。一般的によくあるエラーのタイプは rui、rua、wua に関連したもので、この順序で検出されます。rui エラーはそれほど致命的なエラーではなく、このエラーが検出されてもたいていの場合プログラムは問題なく実行終了します。それに比べて rua と wua エラーは不正なメモリーアドレスにアクセスし、ある種のコーディングエラーを引き起こすため、問題は深刻です。

まず rui と rua エラーを抑制し、wua エラーをすべて修正した後、もう一度プログラムを実行します。次に rui エラーだけを抑制し、rua エラーをすべて修正した後、もう一度プログラムを実行します。さらにエラーの抑制をせずに、すべての rui エラーを修正します。最後にプログラムを実行し、エラーがすべて修正されたことを確認してください。

最新のエラー報告を抑止するには、「suppress -last」を実行します (または「実行時検査」ウィンドウの「最後に報告されたエラーの抑止」ボタンをクリックします)。

子プロセスにおける RTC の実行

子プロセスで RTC を実行するには、dbx 環境変数 rtc_inherit を on に設定します。デフォルトでは off になります (Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「rtc_inherit 環境変数」参照)。

親で RTC が有効になっていて、dbx 環境変数 follow_fork_mode が child に設定されているときに dbx を実行すると子プロセスの RTC を実行できます (Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「follow_fork_mode 環境変数」参照)。

分岐が発生すると、dbx は子に RTC を自動的に実行します。プログラムが exec （）を呼び出すと、exec （）を呼び出すプログラムの RTC 設定がそのプログラムに渡ります。

特定の時間に RTC の制御下におくことができるプロセスは 1 つだけです。次に例を示します。

```
% cat -n program1.c
     1 #include <sys/types.h>
     2 #include <unistd.h>
     3 #include <stdio.h>
     4
     5 int
     6 main()
     7 {
     8      pid_t child_pid;
     9      int parent_i, parent_j;
    10
    11      parent_i = parent_j;
    12
    13      child_pid = fork();
    14
    15      if (child_pid == -1) {
    16          printf("parent: Fork failed\n");
    17          return 1;
    18      } else if (child_pid == 0) {
    19          int child_i, child_j;
    20
    21          printf("child: In child\n");
    22          child_i = child_j;
    23          if (execl("./program2", NULL) == -1) {
    24              printf("child: exec of program2 failed\n");
    25              exit(1);
    26          }
    27      } else {
    28         printf("parent: child's pid = %d\n", child_pid);
    29      }
    30      return 0;
    31 }
%
```

```
% cat -n program2.c
     1
     2 #include <stdio.h>
     3
     4 main()
     5 {
     6       int program2_i, program2_j;
     7
     8       printf ("program2: pid = %d\n", getpid());
     9       program2_i = program2_j;
    10
    11       malloc(8);
    12
    13       return 0;
    14 }
%
```

RTC は親プロセス、program1 の最初のエラーを報告します。

```
% cc -g -o program1 program1.c
% cc -g -o program2 program2.c
% dbx -C program1
program1 のシンボル情報を読んでいます
rtld /usr/lib/ld.so.1 のシンボル情報を読んでます
librtc.so のシンボル情報を読んでいます
libc.so.1 のシンボル情報を読んでいます
libdl.so.1 のシンボル情報を読んでいます
(dbx) check -all
アクセス検査 - ON
メモリー使用状況検査 - ON
(dbx) dbxenv follow_fork_mode child
(dbx) run
実行中: program1
(プロセス id 3885)
実行時検査を有効にしています...終了
非初期化領域からの読み取り (rui):
4 バイト読み取り を アドレス 0xefffff110 で しようとしました
    それは 104 バイト 現スタックポインタより上 です
変数は 'parent_j' です。
現関数 :main
   11   parent_i = parent_j;
```

follow_fork_mode が child に設定されているため、フォークが起こると、エラー検査が親プロセスから子プロセスに切り替えられます。

RTC は子プロセスのエラーを報告します。

program2 の実行が起こると、RTC 設定値は program2 から継承されるため、アクセスおよびメモリー使用状況の検査がそのプロセスに対して有効になります。

```
(dbx) cont
dbx: 警告: フォークしました。親プロセス内での実行時検査機能は休止します
プロセス 3885 から切り離し中
parent: child's pid = 3885
プロセス 10711 に接続しました。
_libc_fork で停止しました アドレス 0xef6b6040
0xef6b6040: _fork+0x0008:bgeu    _fork+0x30
現関数 :main
   13       child_pid = fork();
(dbx) cont
child: In child
非初期化領域からの読み取り (rui):
4 バイト読み取り を アドレス 0xeffff108 で しようとしました
    それは 96 バイト 現スタックポインタより上 です
変数は 'child_j' です。
現関数 :main
   22           child_i = child_j;
(dbx) cont
dbx: プロセス 10711 は exec("./program2") をするところです
dbx: プログラム "./program2" が今 exec されました
dbx: オリジナルプログラムに戻るには "debug $oprog" を使用します
program2 のシンボル情報を読んでいます
すでに読んでいるので、ld.so.1 を飛ばします
すでに読んでいるので、librtc.so を飛ばします
すでに読んでいるので、libc.so.1 を飛ばします
すでに読んでいるので、libdl.so.1 を飛ばします
実行時検査を有効にしています...終了
main で停止しました 行番号 8  ファイル "program2.c"
    8       printf("program2: pid = %d\n", getpid());
(dbx) cont
program2: pid = 13886
```

RTC は、実行されたプログラム、program2 のアクセスエラーを報告します。

RTC は、RTC 制御下にある間に終了したプロセス、program2 に関するメモリー使用状況レポートとメモリーリークレポートを出力します。

```
非初期化領域からの読み取り (rui):
4 バイト読み取り を アドレス 0xeffff13c で しようとしました
    それは 100 バイト 現スタックポインタより上 です
変数は 'program2_j' です。
現関数 :main
    9       program2_i = program2_j;
(dbx) cont
メモリーリーク検査中...
実際のリークの報告 (実際のリーク:  1  合計サイズ:   8 バイト)

合計        ブロック   リーク        割り当て呼出しスタック
サイズ      数         ブロック
                       アドレス
======  ======  ==========  ============================
     8       1     0x20c50  main

起こり得るリークの報告 (起こり得るリーク: 0 合計サイズ: 0 バイト)

実行完了。終了コードは、0 です
```

接続されたプロセスへの RTC の使用

実行時検査は、影響を受けるメモリーがすでに割り当てられている場合に RUI が検出できなかった例外を伴う接続済みプロセスで機能します。ただし、実行時検査を開始する際、librtc.so を事前に読み込んでおく必要があります。接続先のプロセスが 64 ビット SPARC V9 プロセスである場合、sparcv9 librtc.so を使用します。製品が /opt にインストールされている場合、librtc は次の場所にあります。

sparc v9 の場合、/opt/SUNWspro/lib/v9/librtc.so

その他のすべての SPARC プラットフォームの場合、
/opt/SUNWspro/lib/librtc.so

librtc.so を事前に読み込むには、次のように入力します。

```
% setenv LD_PRELOAD path-to-librtc/librtc.so
```

librtc.so を常時読み込んだ状態にせず、必要なときにだけ読み込まれるように LD_PRELOAD を設定してください。次に例を示します。

```
% setenv LD_PRELOAD...
% アプリケーションの実行
% unsetenv LD_PRELOAD
```

プロセスに接続したら、RTC を有効にすることができます。

接続したいプログラムがフォークされるか、または別のプログラムによって実行された場合は、LD_PRELOAD をフォークを行うメインプログラムに設定する必要があります。LD_PRELOAD の設定値は fork/exec を通して継承されます。

RTC での修正継続機能の使用

RTC を修正継続機能とともに使用すると、プログラミングエラーを簡単に分離して修正することができます。修正継続機能を組み合わせて使用すると、デバッグに要する時間を大幅に削減することができます。次に例を示します。

```
% cat -n bug.c
     1  #include <stdio.h>
     2  char *s = NULL;
     3
     4  void
     5  problem()
     6  {
     7          *s = 'c';
     8  }
     9
    10  main()
    11  {
    12          problem();
    13          return 0;
    14  }
    15
% cat -n bug-fixed.c
     1  #include stdio.h
     2  cahr *s = NULL;
     3
     4  void
     5  problem()
     6  {
     7
     8          s = (cahr *)malloc(1);
     9          *s = 'c';
    10  }
    11
    12  main()
```

```
    13  {
    14          problem();
    15          return 0;
    16  }

% cc -g bug.c
% dbx -C a.out
a.out のシンボル情報を読んでいます
rtld /usr/lib/ld.so.1 のシンボル情報を読んでます
librtc.so のシンボル情報を読んでいます
libc.so.1 のシンボル情報を読んでいます
libdl.so.1 のシンボル情報を読んでいます
(dbx) check -access
アクセス検査 - ON
(dbx) run
実行中: a.out
(プロセス id 15052)
実行時検査を有効にしています...終了
非割り当て領域への書き込み (wua):
1 バイト書き込み を NULL ポインタを通して しようとしました。
現関数 :problem
    7   *s = 'c';
(dbx) pop
main で停止しました 行番号 12  ファイル "bug.c"
   12   problem();
(dbx) cp bug-fixed.c bug.c
(dbx) fix
修正中 "bug.c" ......
 pc は "bug.c":14  に移動しました
main で停止しました 行番号 14  ファイル "bug.c"
   14   problem();
(dbx) cont

実行完了。終了コードは、0 です
(dbx) quit
'a.out' の下記のモジュールは変更されました (修正済み):
bug.c
プログラムを再構築することをお忘れなく。
```

修正と継続についての詳細は、第 11 章を参照してください。

実行時検査アプリケーションプログラミングインタフェース

リーク検出およびアクセスの両方の検査では、共有ライブラリ `libc.so` 内の標準ヒープ管理ルーチンを使用する必要があります。これは、RTC がプログラム内のすべての割り当てと開放を追跡できるためです。アプリケーションの多くは、独自のメモリー管理ルーチンを `malloc/free` にかぶせて作成したり、または最初から作成します。独自のアロケータ (専用アロケータと呼ばれる) を使用すると、RTC はそれらを自動的に追跡できません。したがって、それらの不正な使用によるリークエラーとメモリーアクセスエラーを知ることができません。

ただし、RTC には専用アロケータを使用するための API があります。この API を使用すると、専用アロケータを、標準ヒープアロケータと同様に扱うことができます。API 自体はヘッダーファイル `rtc_api.h` に入っており、WorkShop の一部として出されます。マニュアルページの `rtc_api(3x)` には、RTC API 入口の詳細が記載されています。

専用アロケータがプログラムヒープを使用しない場合の RTC アクセスエラーレポートには小さな違いがいくつかあります。エラーレポートに、割り当て項目は含まれません。

バッチモードでの RTC の使用

`bcheck(1)` は、`dbx` の RTC 機能の便利なバッチインタフェースです。これは、`dbx` のもとでプログラムを実行し、デフォルトにより RTC エラー出力をデフォルトファイルの `program.errs` に入れます。

`bcheck` は、メモリーリーク検査、メモリーアクセス検査、メモリー使用状況検査のいずれか、またはこのすべてを実行できます。デフォルトでは、リーク検査だけが実行されます。この使用方法の詳細については、`bcheck(1)` のマニュアルページを参照してください。

bcheck 構文

bcheck の構文は次のとおりです。

```
bcheck [-access | -all | -leaks | -memuse] [-o logfile] [-q]
[-s script] program [args]
```

-o *logfile* オプションを使用すると、ログファイルに別の名前を指定することができます。-s *script* オプションはプログラムの実行前に *script* を実行します。ファイル *script* に含まれる dbx コマンドを読み取ることができます。*script* ファイルには通常、suppress や dbxenv などのコマンドが含まれていて、bcheck によるエラー出力を調整します。

-g オプションは、bcheck を完全な静止状態にして、プログラムと同じ状況になります。これは、スクリプトまたはメークファイルで bcheck を使用したい場合に便利です。

bcheck 構文

hello に対してリーク検査だけを実行します。

```
bcheck hello
```

mach に引数 5 を付けてアクセス検査だけを実行します。

```
bcheck -access mach 5
```

cc に対してメモリー使用状況検査だけを静止状態で実行し、通常の終了状況で終了します。

```
bcheck -memuse -q cc -c prog.c
```

プログラムは、実行時エラーがバッチモードで検出されても停止しません。すべてのエラー出力がエラーログファイル *logfile* にリダイレクトされます。しかしプログラムは、ブレークポイントを検出するか、またはプログラムが割り込みを受けると停止します。

バッチモードでは、完全なスタックバックトレースが生成されて、エラーログファイルにリダイレクトされます。スタックフレームの数は、dbxenv 変数 stack_max_size によって指定できます。

ファイル *logfile* がすでに存在する場合、bcheck はそのファイルの内容を消去してから、そこに出力をリダイレクトします。

dbx からバッチモードを直接有効化

バッチモードに似たモードは、dbx 環境変数 rtc_auto_continue と rtc_error_log_file_name から直接有効にできます (Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「rtc_auto_continue 環境変数」と「rtc_error_log_file_name 環境変数」参照)。

rtc_auto_continue が on に設定されていると、RTC はそのままエラーを求めて自動的に実行されます。検出したエラーは、dbx 環境変数 rtc_error_log_name で指定したファイルにリダイレクトされます (Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「rtc_error_log_file_name 環境変数」参照)。デフォルトログファイル名は、/tmp/dbx.errorlog.*uniqueid* です。すべてのエラーを端末にリダイレクトするには、rtc_error_log_file_name 環境変数を /dev/tty に設定します。

rtc_auto_continue はデフォルトで off です。

トラブルシューティングのヒント

プログラム中でエラー検査がオンになっていて、プログラムが実行中の場合、次のエラーが検出されることがあります。

```
librtc.so と dbx とのバージョンが合いません。; エラー検査を休止状態にしました
```

これは、RTC を接続されたプロセスに使用していて、LD_PRELOAD を、各自の Sun WorkShop dbx に添付されたもの以外の librtc.so バージョンに設定した場合に起こる可能性があります。これを修正するには、LD_PRELOAD の設定値を変更してください。

```
パッチエリアが遠すぎます (8MB の制限); アクセス検査を休止状態にしました
```

RTC は、アクセス検査を有効にするためにロードオブジェクトに十分に近いパッチスペースを検出できませんでした。この章に後述される「rtc_patch_area」の項を参照してください。

RTC の 8M バイト制限

以下に説明する 8M バイトの制限は、UltraSPARC (TM) プロセッサに基づくハードウェアに適用されません。これらのハードウェアでは、dbx は、分岐を使用する代わりにトラップハンドラを呼び出すことができます。トラップハンドラに制御を移行すると、実行速度が最大 10 倍遅くなりますが、8M バイトの制限に悩まされることはなくなります。ハードウェアが UltraSPARC プロセッサに基づいている限り、トラップは、必要に応じて自動的に使用されます。ハードウェアをチェックするには、システムコマンド isalist を実行し、実行結果に文字列 sparcv8plus が含まれていることを確認します。

アクセス検査を実行するために、dbx の RTC 機能は各ロードおよびストア命令を、パッチ領域への分岐命令と置き換えます。この分岐命令の有効範囲は 8Mバイトです。これは、デバッグされたプログラムが、置き換えられた特定のロード/ストア命令の 8Mバイトのアドレス空間をすべて使いきってしまった場合、パッチ領域を保存する場所がなくなることを意味します。

RTC がメモリーへのすべてのロードおよびストアにまったく割り込めない場合、RTC は正確な情報を提供できないので完全に無効になります。リークの検査は影響を受けません。

この制約にぶつかった場合、dbx は何らかの処置を施します。その結果、問題が修正できれば続行しますが、問題が修正できない場合は、エラーメッセージを表示しアクセス検査を終了します。

8M バイトの制限値に達したら、以下の対策をとってください。

1. 64 ビット SPARC V9 の代わりに 32 ビット SPARC V8 を使用します。

   -xarch=v9 オプションでコンパイルされたアプリケーションで 8M バイト問題が発生するときは、32 ビットバージョンのアプリケーションでメモリーテストをしてください。64 ビットアドレスには長いパッチ命令シーケンスが必要であり、32 ビットアドレスを使用すれば 8M バイトの制限を緩和できるからです。これでも問題が解決しない場合は、32 ビットプログラムと 64 ビットプログラムいずれの場合にも、以下の対策をとってください。

2. パッチ領域オブジェクトファイルを追加します。

   rtc_patch_area シェルスクリプトを使用し、大きな実行可能ファイルや共有ライブラリの中間にリンクできる特別な .o ファイルを作成すれば、パッチ領域を拡大できます。rtc_patch_area(1) マニュアルページを参照してください。

   dbx の実行時に 8M バイト制限に達すると、大きすぎる読み込みオブジェクト (メインプログラムや共有ライブラリ) が報告され、その読み込みプロジェクトに必要なパッチ領域値が出力されます。

   最適な結果を得るには、実行可能ファイルや共有ライブラリ全体に特別なパッチオブジェクトファイルを均等に分散させ、デフォルトサイズ (8M バイト) かそれよりも小さいサイズを使用します。dbx が必要とする必要値の 10 % から 20 % の範囲を超えてパッチ領域を追加しないでください。たとえば、dbx が .out に 31M バイトを要求する場合は、rtc_patch_area スクリプトで作成したそれぞれのサイズが 8M バイトのオブジェクトファイルを 4 つ追加し、実行可能ファイル内でそれらをほぼ均等に分割します。

   dbx の実行時に、実行可能ファイルに明示的なパッチ領域が見つかると、パッチ領域になっているアドレス範囲が出力されるので、リンク回線に正しく指定することができます。

3. 読み込みオブジェクトが大きい場合は、小さい読み込みオブジェクトに分割します。

   実行ファイルや大きなライブラリ内のオブジェクトファイルを小さいオブジェクトファイルグループに分割します。それらを小さいパーツにリンクします。大きいファイルが実行可能ファイルの場合、小さい実行可能ファイルと共有ライブラリに分割します。大きいファイルが共有ライブラリの場合、複数の小さいライブラリのセットに再編します。

   この方法では、dbx により、異なる共有オブジェクト間でパッチコード用の領域を探すことができます。

4. パッド .so ファイルを追加します。

   この処置は、プロセスの起動後に接続する場合にのみ必要です。

   実行時リンカーによるライブラリの配置間隔が狭すぎてライブラリ間にパッチ領域を作成できない場合があります。RTC を on にして dbx が実行可能ファイルを起動すると、dbx は実行時リンカーに対して共用ライブラリ間に新たなギャップを挿入するよう指示しますが、実行時検査を有効にして dbx で起動されていないプロセスに接続しても、ライブラリ間が狭すぎて対応できません。

   この場合 (そしてプログラムを dbx で起動できない場合) は、rtc_patch_area スクリプトで共有ライブラリを作成し、他の共有ライブラリ間でプログラムにリンクしてください。詳細については、rtc_patch_area(1) マニュアルページを参照してください。

RTC エラー

RTC で報告されるエラーは、通常はアクセスエラーとリークの 2 種類があります。

アクセスエラー

アクセス検査がオンのとき、RTC による検出と報告の対象になるのは次のタイプのエラーです。

不正解放 (baf) エラー

意味 : 割り当てられたことのないメモリーを解放しようとした。

考えられる原因 : free() または realloc() にヒープデータ以外のポインタを渡した。

例 :

```
char a[4];
char *b = &a[0];


free(b);            /* 不正解放 (baf) */
```

重複解放 (duf) エラー

意味 : すでに解放されているヒープブロックを解放しようとした。

考えられる原因 : 同じポインタを使用して free() を 2 回以上呼び出した。C++ では、同じポインタに対して “delete” 演算子を 2 回以上使用した。

例 :

```
char *a = (char *)malloc(1);
free(a);
free(a);                /* 重複解放 (duf) */
```

境界整列を誤った解放 (maf) エラー

意味 : 境界合わせされていないヒープブロックを解放しようとした。

考えられる原因 : free() または realloc() に正しく境界合わせされていないポインタを渡した。malloc によって返されたポインタを変更した。

例 :

```
char *ptr = (char *)malloc(4);
ptr++;
free(ptr);                /* 境界整列を誤った解放 (maf) */
```

境界整列を誤った読み取り (mar) エラー

意味 : 適切に境界合わせされていないアドレスからデータを読み取ろうとした。

考えられる原因 : ハーフワード、ワード、ダブルワードの境界に合わせられていないアドレスから、それぞれ 2 バイト、4 バイト、8 バイトを読み取った。

例 :

```
char *s = “hello world”;
int *i = (int *)&s[1];
int j;

j = *i;             /* 境界整列を誤った読み取り (mar) */
```

境界整列を誤った書き込み (maw) エラー

意味 : 適切に境界合わせされていないアドレスにデータを書き込もうとした。

考えられる原因 : ハーフワード、ワード、ダブルワードの境界に合わせられていないアドレスに、それぞれ 2 バイト、4 バイト、8 バイトを書き込んだ。

例 :

```
char *s = "hello world";
int *i = (int *)&s[1];


*i = 0;             /* 境界整列を誤った書き込み (maw) */
```

メモリー不足 (oom) エラー

意味 : 利用可能な物理メモリーより多くのメモリーを割り当てようとした。

考えられる原因 : プログラムがこれ以上システムからメモリーを入手できない。oom エラーは、malloc() からの戻り値が NULL かどうか検査していない (プログラミングでよく起きる誤り) ために発生する問題の追跡に役立ちます。

例 :

```
      char *ptr = (char *)malloc(0x7fffffff);
      /* メモリー不足 (oom), ptr == NULL */
```

非割り当てメモリーからの読み取り (rua) エラー

意味 : 存在しないメモリー、割り当てられていないメモリー、マップされていないメモリーからデータを読み取ろうとした。

考えられる原因 : ストレイポインタ (不正な値を持つポインタ)、ヒーブブロック境界のオーバーフロー、すでに解放されたヒープブロックへのアクセス。

例 :

```
char c, *a = (char *)malloc(1);
c = a[1];  /* 非割り当てメモリーからの読み取り (rua) */
```

非初期化メモリーからの読み取り (rui) エラー

意味 : 初期化されていないメモリーからデータを読み取ろうとした。

考えられる原因 : 初期化されていない局所データまたはヒープデータの読み取り。

例 :

```
foo()
{   int i,  j;
j = i;  /* 非初期化メモリーからの読み取り (rui) */
}
```

読み取り専用メモリーへの書き込み (wro) エラー

意味 : 読み取り専用メモリーにデータを書き込もうとした。

考えられる原因 : テキストアドレスへの書き込み、読み取り専用データセクション (.rodata) への書き込み、読み取り専用として mmap されているページへの書き込み。

例 :

```
foo()
{  int *foop = (int *) foo;
*foop = 0;/* 読み取り専用メモリーへの書き込み (wro) */
}
```

非割り当てメモリーへの書き込み (wua) エラー

意味 : 存在しないメモリー、割り当てられていないメモリー、マップされていないメモリーにデータを書き込もうとした。

考えられる原因 : ストレイポインタ、ヒープブロック境界のオーバーフロー、すでに解放されたヒープブロックへのアクセス。

例 :

```
char *a = (char *)malloc(1);
a[1] = '\0';/* 非割り当てメモリーへの書き込み (wua) */
```

メモリーリークエラー

リーク検査をオンにしておくと、RTC では次のエラーが報告されます。

ブロック中のアドレス (aib)

意味: メモリーリークの可能性がある。割り当てたブロックの先頭に対する参照はないが、そのブロック内のアドレスに対する参照が少なくとも 1 つある。

考えられる原因: そのブロックの先頭を示す唯一のポインタが増分された。

例:

```
char *ptr;
main()
{
   ptr = (char *)malloc(4);
   ptr++;         /* Address in Block */
}
```

レジスタ中のアドレス (air)

意味: メモリーリークの可能性がある。割り当てられたブロックが解放されておらず、そのブロックに対する参照がプログラムのどこにもないが、レジスタには参照がある。

考えられる原因: コンパイラがプログラム変数をメモリーではなくレジスタにだけ保存している場合にこのエラーになる。最適化をオンにしてコンパイラを実行すると、ローカル変数や関数パラメタにこのような状況がよく発生する。最適化をオンにしていないのにこのエラーが発生する場合は、メモリーリークが疑われる。ブロックを解放する前に、割り当てられたブロックに対する唯一のポインタが範囲外を指定するとメモリーリークになる。

例:

```
if (i == 0) {
       char *ptr = (char *)malloc(4);
       /* ptr is going out of scope */
}
  /* Memory Leak or Address in Register */
```

メモリーリーク (mel) エラー

意味 : 割り当てられたブロックが解放されておらず、そのブロックへの参照がプログラム内のどこにも存在しない。

考えられる原因 : プログラムが使用されなくなったブロックを解放しなかった。

例 :

```
char *ptr;
    ptr = (char *)malloc(1);
    ptr = 0;
/* メモリーリーク (mel) */
```

第10章

データの視覚化

Sun Workshop でプログラムをデバッグするときに、データをグラフィック表示させる必要がある場合は、データの視覚化を行います。

データの視覚化機能はデバッグに最適です。サイズが大きく複雑なデータセットでも容易にチェックや確認ができ、結果をシミュレーションしたり、計算結果を対話的に操作できます。「データグラフ」ウィンドウでは、プログラムデータを「見る」ことができ、グラフィカルにデータを解析できます。グラフの印刷やファイルへの出力が可能です。

この章は、次の各節から構成されています。

- 適切な配列式の指定
- 配列表示の自動更新
- 表示の変更
- 視覚化されたデータの分析
- Fortran のサンプルプログラム
- C のサンプルプログラム

適切な配列式の指定

データを表示させるには、配列とその表示方法を指定する必要があります。「デバッグ」ウィンドウで「式」フィールドに配列名を入力して、「データグラフ」ウィンドウを呼び出します。Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「配列の評価」を参照してください。複雑な (Fortran) 配列型を除く、すべてのスカラ配列型がサポートされています。

一次元の配列は、x 軸にインデックス、y 軸に配列値を示したグラフ (ベクトルグラフ) になります。二次元の配列のデフォルトのグラフ表示 (エリアグラフ) は、 x 軸で第 1 次元のインデックス、y 軸で 第 2 次元のインデックス、z 軸で配列値を示したグラフになります。*n* 次元の配列をグラフ表示できますが、これらの次元の中で変更できるのは、 2 つまでです。

データセット全体を調べる必要はありません。次の例で示すように、配列の断面を指定するだけで済みます。図 10-1 と図 10-2 は、この章の最後にある Fortran のサンプルプログラムの `bf` 配列を示しています。

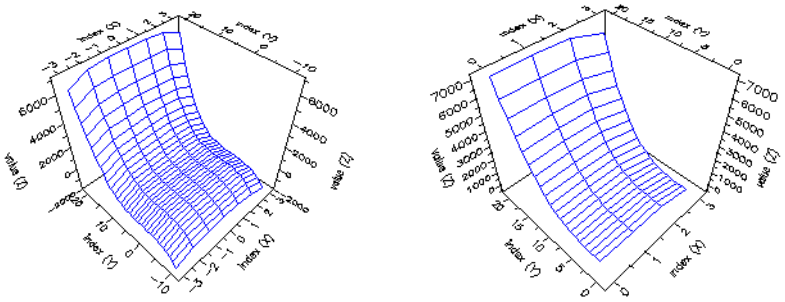


図 10-1 `bf` 配列名のみのグラフ (左)、`bf (0:3,1:20)` のグラフ (右)

次の 2 つの図は、指定された値の範囲の配列を示したものです。

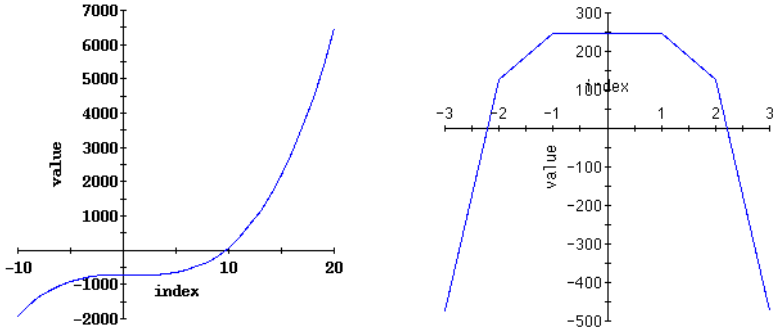


図 10-2 配列 `bf : bf (3,:)` (左)、`bf (:,7)` (右)

配列グラフの作成

ソースコード内の式を選択してグラフ化すれば、アプリケーションの実行時にさまざまなポイントを比較できます。グラフでは、プログラムのどこに問題があるかを調べることができます。「データグラフ」ウィンドウでは、配列をグラフ化し、同じデータをさまざまな視点から検討することができます (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「データグラフウィンドウ」参照)。

Fortran の複雑な配列以外すべてサポートしています。

配列グラフ作成の準備

配列のグラフを作成するには、次のように操作します。

1. 次のいずれかの操作で、「デバッグ」ウィンドウにプログラムを読み込みます。

   - 「デバッグ」➤「新規プログラム」と選択してプログラムを読み込みます (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「新しいプログラムのデバッグ」参照)。
   - プログラムが現在の Sun WorkShop セッションにすでに読み込まれている場合は、「デバッグ」メニューのプログラムリストからそのプログラムを選択します (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「現在のプログラムのデバッグ」参照)。

2. プログラムに少なくともブレークポイントを 1 つ設定します。

   この場合、ブレークポイントはプログラムの最後に 1 つ設定するか、任意の場所に 1 つ以上設定します (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「ブレークポイント (行に設定)」参照)。

3. プログラムを実行します。

   (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「プログラムの開始」参照)。

   ブレークポイントでプログラムが停止したら、調べる配列を選択します。

さまざまな配列グラフの作成

以上でグラフ作成の準備ができました。Sun WorkShop ではさまざまな方法でグラフを作成できます。

- 「デバッグ」ウィンドウの「式」フィールドに配列名を入力し、「データ」➤「式をグラフ表示」を選択するか、テキストエディタで配列を選択し、「デバッグ」ウィンドウで「データ」➤「選択項目をグラフ表示」を選択します。
- 「データ表示」タブか別の「データ表示」ウィンドウの「データ」メニューか、同じ内容のポップアップメニュー (右クリックで開く) から「グラフ」コマンドを選択します。「データ表示」タブやウィンドウの配列をグラフにできる場合、「グラフ」コマンドが有効になります。
- 「データグラフ」ウィンドウから、「グラフ」➤「新規」を選択し、「データグラフ: 新規」ウィンドウの「式」フィールドに配列名を入力し、「グラフ」をクリックします。

  「現在表示中のグラフを置き換え」ラジオボタンをクリックし、「グラフ」をクリックすると、現在のグラフの代わりに新しいグラフが表示されます。または新しい「データグラフ」ウィンドウが開きます。
- 「dbx コマンド」ウィンドウでは、`vitem` コマンドにより、「dbx」コマンド行から「データグラフ」を直接表示できます (351 ページの「vitem コマンド」参照)。次のように入力します。

```
(dbx) vitem -new array-expression
```

ここで、

`array-expression` は、表示する配列式を表します。

配列表示の自動更新

配列式の値は、プログラムの実行につれて変化します。配列式の新しい値をプログラムの指定個所に表示するか、一定の時間間隔で表示するかを、「デバッグオプション」ダイアログの「データグラファ」タブの「更新」セクションで選択できます (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「データグラファのデフォルト設定」参照)。

プログラムがブレークポイントで停止するごとに配列の値を更新させる場合は、「更新時期」の「プログラム停止」オプションをオンにする必要があります。このようにすると、プログラムが進行していく段階で、各停止位置ごとの配列値を見ることができます。このオプションは、各ブレークポイントでデータの変更を表示させる場合に使用します。この機能は、デフォルトではオフに設定されています。

実行時間が長いプログラムの場合は、「更新時期」の「一定間隔で」オプションを選択すると、時間の経過とともに変化する配列値の変化を見ることができます。一定間隔での更新を使用すると、タイマーが一定 (n) の間隔ごとに自動的に設定されます。この間隔はデフォルトでは、1 秒に設定されています。デフォルトの設定を変更する場合は、「グラフ」➤「デフォルトオプション」を選択し、「デバッグオプション」ダイアログボックス内の更新時間間隔を変更します (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「更新された配列値の表示」参照)。

注 – タイマーに設定された一定の間隔ごとに、Sun Workshop はグラフ表示の更新を行いますが、配列がこの特定の時間に範囲外にある場合は、更新は行われません。

また、タイマーは、データ収集、メモリーリークまたはアクセス検査時にも使用されるため、標本コレクタまたは実行時検査の実行中には、一定間隔での更新は設定できません。

表示の変更

データがグラフィック表示されたら、「データグラフ」ウィンドウ内のコントロールを使って表示画面の調整やカスタマイズが行えます。この節では、ユーザーが試してみるることができるグラフ表示の例をいくつか示します。

どのようなタイプの配列でもそれをグラフ化するときに、「データグラフ」ウィンドウを開くと、「拡大」と「縮小」オプションが表示されています。エリアグラフの場合は、このウィンドウには「軸回転」と「遠近率」フィールドが組み込まれます。これらのオプションを使用すると、その軸を回転させたり、奥行きを増減することによってグラフの表示を変更できます。グラフを簡単に回転させるには、グラフ上にカーソルを置いてマウスの右ボタンを押し、そのままグラフをドラッグします。また、各軸の回転の角度を、「軸回転」フィールドで指定することもできます。

これ以外のオプションを表示させる場合は、「オプション表示」ボタンをクリックします。オプションがすでに表示されている場合は、「Hide」をクリックしてその他のオプションを非表示にしてください。

図 10-3 は、同じ配列を 2 種類の表示画面で示したものです。左の図では、配列は、「ワイヤーメッシュ」構造の「表面」グラフタイプで表示されています。右の図では、配列は、「範囲線」によって輪郭を描く「等高線」グラフタイプで表示されています。

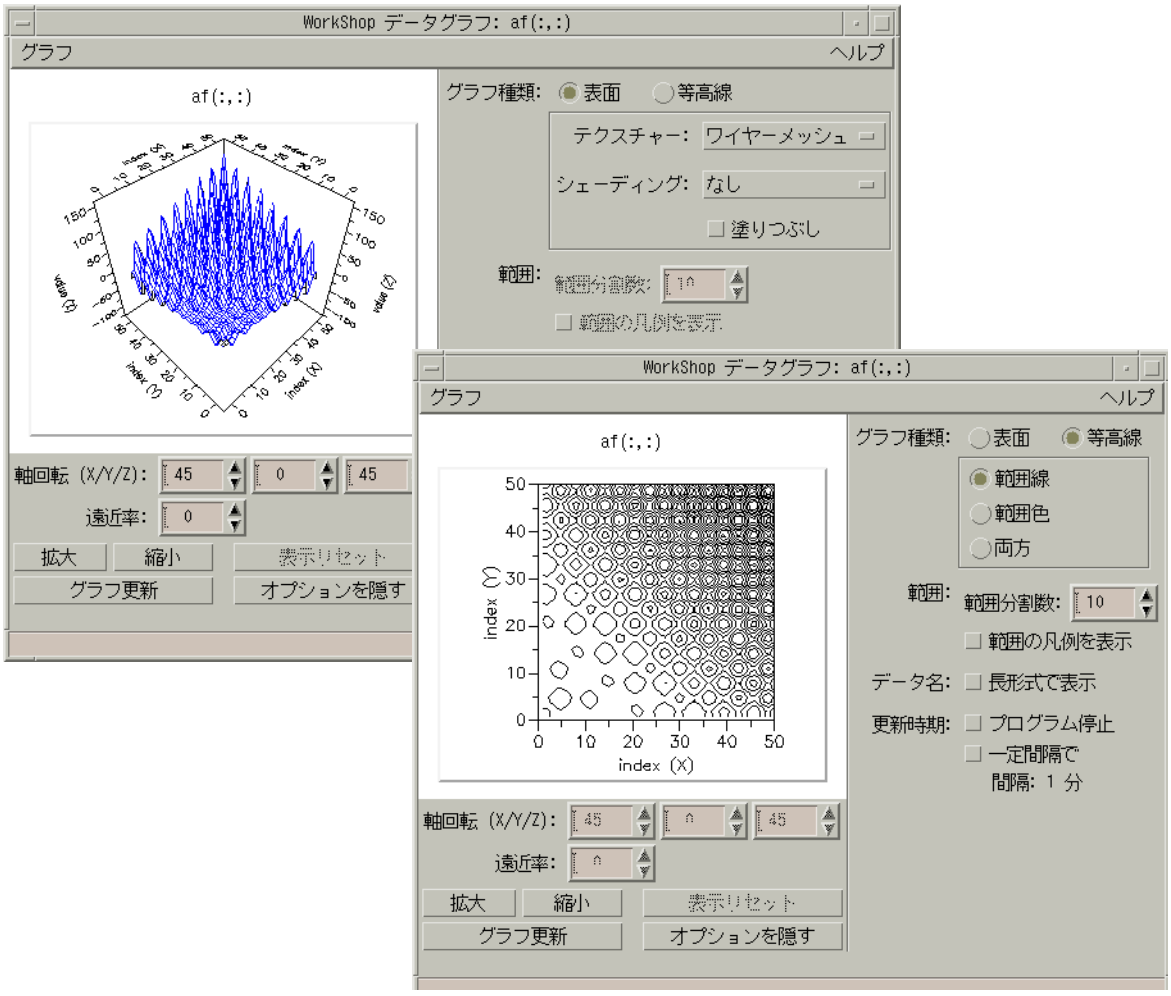


図 10-3 同じ配列の2種類の表示画面

「等高線」グラフタイプを選択すると、「範囲」オプションで、どのエリアのデータ値が変化したかが解ります。

「表面」グラフタイプの表示オプションには、「テクスチャー」、「シェーディング」、「塗りつぶし」があります。

「表面」グラフのテクスチャーの選択肢には、図 10-4 に示す「ワイヤーメッシュ」と「範囲線」があります。

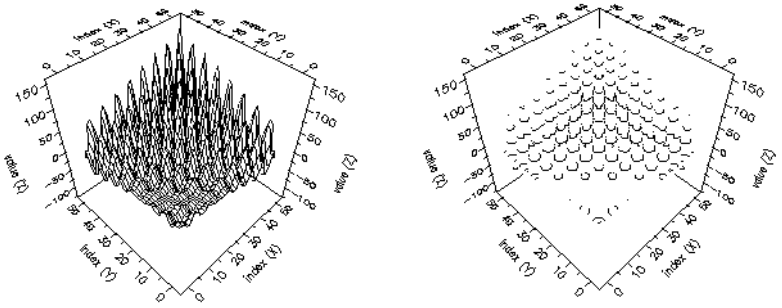


図 10-4　ワイヤーメッシュ (左) と 範囲線 (右) の 「表面」グラフ

「表面」グラフのシェーディング選択肢には、図 10-5 に示す「光源」と「範囲色」があります。

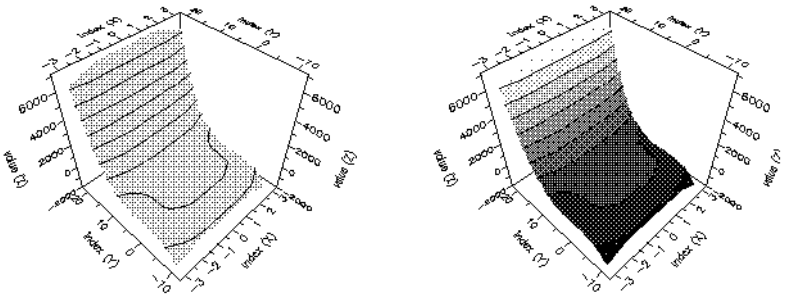


図 10-5　「光源」(左) と「範囲色」(右) の 「表面」グラフ

図 10-6 のように、グラフの領域に陰影を付けたり、表面を塗りつぶしたグラフを作成するには、「塗りつぶし」をオンにします。

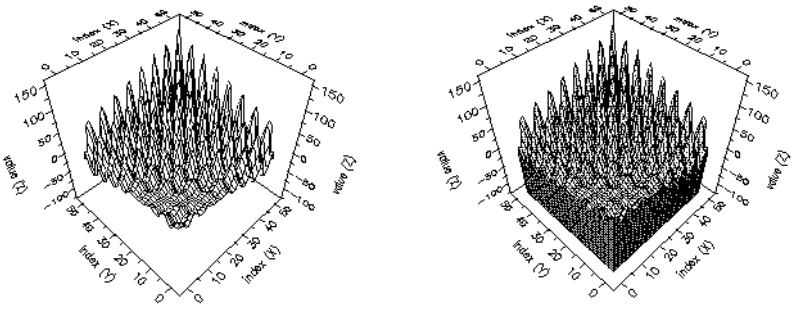


図 10-6 陰影付き (左) と表面を塗りつぶした (右)「表面」グラフ

データ範囲線で面グラフを表示するにはグラフタイプに「等高線」を選択します。図 10-7 のように、「等高線」グラフには、グラフをラインとカラーのいずれか、またはその両方で表示するオプションがあります。

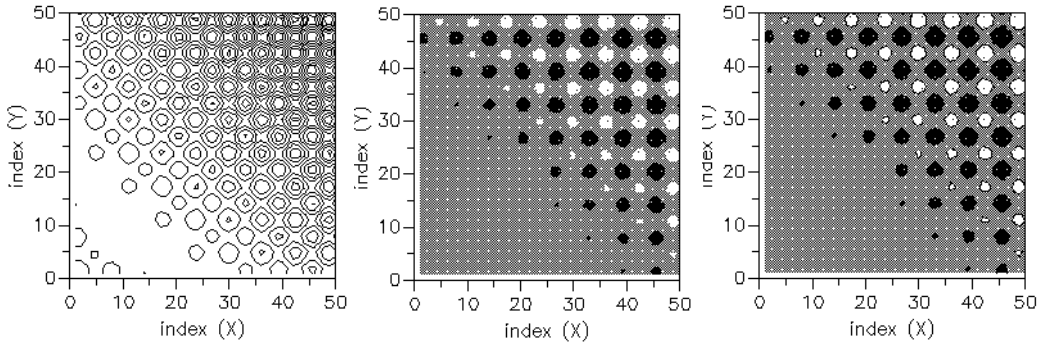


図 10-7 ライン (左)、カラー (中央)、両方 (右) による「等高線」グラフ

表示するデータの値の範囲を変更するには、「範囲: 分割数」テキストボックスの値を変更します。

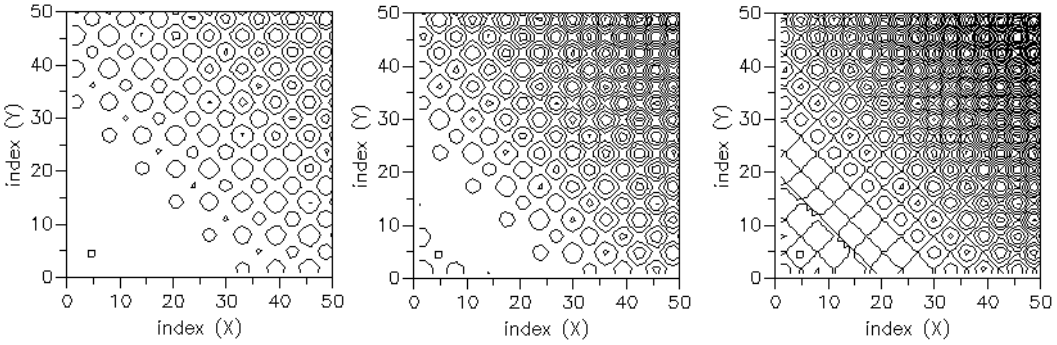


図 10-8 「範囲: 分割数」が (左から右に) 5、10、15 の「等高線」グラフ

分割数を大きくすると、カラーマップに悪影響が出ることがあります。

範囲の間隔の凡例を表示するには、「範囲の凡例を表示」ボタンをクリックします。

視覚化されたデータの分析

視覚化されたデータの更新方法は、更新内容によって異なります。たとえば、その必要に応じて、ブレークポイントで更新することも指定した時間間隔で更新することもできます。変更内容を表示したり、最終的な結果を分析したりすることができます。この節では、さまざまなシチュエーションのシナリオを想定しています。

Sun Workshop には、`dg_fexamp` (Fortran) と `dg_cexamp` (C) の 2 つのサンプルプログラムが付属しています。データの視覚化を説明する以下のシナリオの中で、この 2 つのサンプルプログラムを使用します。

また、これらのプログラムは練習用にも使用できます。このサンプルプログラムは、*install-dir*`/SUNWspro/WS6/examples/gatecrasher` にあります。デフォルトの *install-dir* は `/opt` です。このプログラムを使用する場合は、このディレクトリに移動して、`make` と入力すると、実行可能プログラムが作成されます。

シナリオ 1: 同じデータを違う視点で表示させて比較する

同じデータを何回でもグラフ化できるため、グラフタイプを変えたり、データのセグメントを変えたりして比較できます。

1. C または Fortran のサンプルプログラムをロードします。
2. プログラムの最後にブレークポイントを設定します。

   (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「ブレークポイント (行に設定)」参照)。
3. プログラムを起動し、ブレークポイントまで実行します。
4. 「デバッグ」ウィンドウの「式」フィールドに `bf` と入力します。
5. 「データ」➤「式をグラフ表示」を選択します。

   `bf` 配列の両方の次元の表面グラフが表示されます。
6. もう一度「データ」➤「式をグラフ表示」を選択し、もう 1 つ同じグラフを表示します。
7. 「データグラフ」ウィンドウのどれかで、「オプション表示」をクリックし、グラフタイプで「等高線」ラジオボタンを選択します。

   こうして、同じデータを異なるグラフタイプで表示して比較します。
8. サンプルプログラムが Fortran の場合は `bf (1,:)`、C の場合は `bf[1][..]` と「式」テキストボックスに入力します。
9. 「データ」➤「式をグラフ表示」を選択し、`bf` 配列におさめられたデータセクションのグラフを表示します。

   データを以上のさまざまなグラフに表示して比較できます。

シナリオ 2: データのグラフを自動的に更新する

「デバッグオプション」ダイアログの「データグラファ」タブの「更新: プログラムが停止したとき」オプションをオンにすると、グラフの更新を自動制御できます。これは、プログラム実行中のデータの変化を比較するための機能です。

1. C または Fortran のサンプルプログラムを読み込みます。
2. `bf` 関数の外側のループの最後にブレークポイントを設定します。

   (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「ブレークポイント (行に設定)」参照)。

3. プログラムを起動し、ブレークポイントまで実行します。

   (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「プログラムの開始」参照)。

4. 「デバッグ」ウィンドウの「式」テキストボックスに `bf` と入力します。

5. 「データ」➤「式をグラフ表示」を選択します。

   最初のループ結果が表示されると、`bf` 配列の値がグラフで表示されます。

6. 「オプション表示」をクリックし、「更新時期: プログラム停止」チェックボックスを選択します。

7. 「実行」➤「継続」を選択し、さらにこのプログラムを数ループ実行します。

   プログラムがブレークポイントで停止するたびに前回のループで設定された値でグラフが更新します。

   ブレークポイントごとに最新のデータによるグラフを表示する場合、この自動更新機能を使用すれば、手間が省けます。

シナリオ3: プログラムの異なる個所のデータグラフを比較する

グラフは、手動でも更新できます。

1. C または Fortran のサンプルプログラムを読み込みます。

2. `af` 関数の外側のループの最後にブレークポイントを設定します。

   (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「ブレークポイント (行に設定)」参照)。

3. プログラムを起動し、ブレークポイントまで実行します。

   (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「プログラムの開始」参照)。

4. 「デバッグ」ウィンドウの「式」テキストボックスに `af` と入力します。

5. 「データ」➤「式をグラフ表示」を選択します。

   最初のループ結果が表示されると、`af` 配列の値がグラフで表示されます。このグラフでは自動更新がオフになっていることを確認してください (デフォルトの設定)。

6. 「実行」➤「継続」を選択し、このプログラムをもう 1 ループ実行します。

7. 「データ」➤「式をグラフ表示」を選択し、af 配列のグラフをもう 1 つ表示します。

   このグラフのデータ値は、外側のループの 2 番めの実行時に設定された値です。

   これで、af 配列の 2 つのループ実行に使用されたデータを比較することができます。自動更新をオフにしたグラフは、自動、または手動で連続的に更新されるグラフの基準グラフとして使用できます。

シナリオ 4: 同じプログラムを別々に実行した結果のデータグラフを比較する

同じプログラムを複数回実行させる場合、次にそのプログラムを実行させるまで前のデータグラフが画面に残っています。前にプログラムを実行したときのデータから作成したグラフは、手動で更新するか、または自動更新機能がオンになっていない限り上書きされません。

1. C または Fortran のサンプルプログラムをロードします。

2. プログラムの最後にブレークポイントを設定します。

   (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「ブレークポイント (行に設定)」参照)。

3. プログラムを起動し、ブレークポイントまで実行します。

   (Sun WorkShop オンラインヘルプの「デバッグウィンドウの使い方」の「プログラムの開始」参照)。

4. 「デバッグ」ウィンドウの「式」テキストボックスに vec と入力します。

5. 「データ」➤「式をグラフ表示」を選択します。

   (正弦波曲線で) vec 配列のグラフが表示されます。

6. これでプログラムを編集できます (sin を cos に置き換えるなど)。修正継続機能 (第 11 章参照)でプログラムを再コンパイルし、処理を続けます (「修正」ツールバーボタンをクリックします)。

7. プログラムを再起動します。

   自動更新機能がオフになっているため、プログラムがブレークポイントに達しても前回のグラフは更新されません。

8. 「データ」➤「式をグラフ表示」を選択します (「式」テキストボックスの値は引き続き vec です)。

   前回のグラフの横に、現在の vec 値のグラフが表示されます。

   これで、プログラムを 2 回実行してできたグラフを比較できます。前回プログラムを実行して作成されたグラフは、「グラフ更新」ボタンで手動で更新するか、自動更新機能をオンにしている場合にだけ変更されます。

Fortran のサンプルプログラム

```
real x,y,z,ct
real vec(100)
real af(50,50)
real bf(-3:3,-10:20)
real cf(50, 100, 200)

do x = 1,100
            ct = ct + 0.1
            vec(x) = sin(ct)
enddo

do x = 1,50
           do y = 1,50
              af(x,y) = (sin(x)+sin(y))*(20-abs(x+y))
           enddo
 enddo

do x = -3,3
          do y = -10,20
            bf(x,y) = y*(y-1)*(y-1.1)-10*x*x*(x*x-1)
          enddo
enddo

do x = 1,50
          do y = 1,100
            do z = 1,200
              cf(x,y,z) = 3*x*y*z - x*x*x - y*y*y - z*z*z
            enddo
          enddo
enddo

end
```

C のサンプルプログラム

```
#include <math.h>
main()
{
    int x,y,z;
    float ct=0;
    float vec[100];
    float af[50][50];
    float bf[10][20];
    float cf[50][100][200];

    for (x=0; x<100; x++)
    {
          ct = ct + 0.1;
          vec[x] = sin(ct);
    }
    for (x=0; x<50; x++)
    {
          for (y=0; y<50; y++)
          {
          af[x][y] = (sin(x)+sin(y))*(20-abs(x+y));
          }
    }
    for (x=0; x<10; x++)
    {
          for (y=0; y<20; y++)
          {
            bf[x][y] = y*(y-1)*(y-1.1)-10*x*x*(x*x-1);
          }
    }
    for (x=0; x<50; x++)
    {
          for (y=0; y<100; y++)
          {
          for (z=0; z<200; z++)
          {
                  cf[x][y][z] = 3*x*y*z - x*x*x - y*y*y - z*z*z ;
          }
          }
    }
}
```

第11章

修正継続機能 (fix と continue)

`fix` を使用すると、デバッグプロセスを停止しないで、編集されたソースコードを簡単に再コンパイルすることができます。

この章は次の各節から構成されています。

- 修正継続機能の使用
- プログラムの修正
- 修正後の変数の変更
- ヘッダファイルの変更
- C++ テンプレート定義の修正

修正継続機能の使用

`fix` と `continue` の各機能を使用すると、ソースファイルを修正して再コンパイルし、プログラム全体を作成しなおすことなく実行を続けることができます。`.o` ファイルを更新して、それらをデバッグ中のプログラムに組み込むことにより、再リンクの必要がなくなります。

この機能を使用する利点は次のとおりです。

- プログラムをリンクしなおす必要がない。
- プログラムを `dbx` に再読み込みする必要がない。
- 修正した位置からプログラムの実行を再開できる。

注 – 構築が進行中の場合は、Sun WorkShop の `fix` を使用しないでください。

---

fix と continue の働き

fix コマンドを使用するには、エディタウィンドウでソースを編集する必要があります (コードの変更方法については、168 ページの「fix と continue によるソースの変更」参照)。変更結果を保存して fix と入力します。*fix* コマンドについては、305 ページの「fix コマンド」を参照してください。

fix が実行されると、dbx は適切なコンパイラオプションでコンパイラを呼び出します。変更後のファイルがコンパイルされ、一時共有オブジェクト (.so) ファイルが作成されます。古いファイルと新しいファイルとを比較することによって、修正の安全性を検査する意味上のテストが行われます。

実行時リンカーを使用して新しいオブジェクトファイルが動作中のプロセスにリンクされ、プログラムカウンタが古い関数から新しい関数の同じ行に移動します (その関数が修正中のスタックの 1 番上にある場合)。さらに、古いファイルのブレークポイントがすべて新しいファイルに移動します。

対象となるファイルがデバッグ情報つきでコンパイルされているかどうかに関わらず、fix と cont を実行できます。ただし、デバッグ情報なしでコンパイルされているファイルの場合には多少の機能制限があります。305 ページの「fix コマンド」の -g オプションの解説を参照してください。

共有オブジェクト (.so) ファイルの修正は可能ですが、その場合、そのファイルを特別なモードでオープンする必要があります。dlopen 関数の呼び出しで、RTLD_NOW|RTLD_GLOBAL または RTLD_LAZY|RTLD_GLOBAL のどちらかを使用します。

fix と continue によるソースの変更

fix と continue を使用すると、ソースを次の方法で変更できます。

- 関数の各行を追加、削除、変更する。
- 関数を追加または削除する。
- 大域変数および静的変数を追加または削除する。

古いファイルから新しいファイルに関数をマップすると問題が起きることがあります。ソースファイルの編集時にこのような問題の発生を防ぐには、次のことを守ってください。

- 関数の名前を変更しない。

- 関数に渡す引数の型を追加、削除、または変更しない。
- スタック上で現在アクティブな関数の局所変数の型を追加、削除、または変更しない。
- テンプレートの宣言やテンプレートインスタンスを変更しない。C++ テンプレート関数定義の本体でのみ修正可能です。

上記の変更を行う場合は、fix と continue で処理するよりプログラム全体を作りなおす方が簡単です。

プログラムの修正

変更後にソースファイルを再リンクするとき `fix` コマンドを使用すればプログラム全体を再コンパイルしなくて済みます。引き続きプログラムの実行を続けることができます。

ファイルを修正するには、次の手順に従ってください。

1. **変更をソースファイルに保存します。**

   この手順を忘れても、Sun WorkShop では自動的に変更が保存されます。

2. **`dbx` プロンプトで `fix` と入力します。**

   修正は無制限に行うことができますが、1 つの行でいくつかの修正を行った場合は、プログラムを作成しなおすことを考えてください。`fix` コマンドは、メモリー内のプログラムのイメージを変更しますが、ディスク上のイメージは変更しません。また修正を行うと、メモリーのイメージは、ディスク上のイメージと同期しなくなります。

   `fix` コマンド は、実行可能ファイル内での変更ではなく、`.o` ファイルとメモリーイメージの変更だけを行います。プログラムのデバッグを終了したら、プログラムを作成しなおして、変更内容を実行可能ファイルにマージする必要があります。デバッグを終了すると、プログラムを作成しなおすように指示するメッセージが出されます。

   `-a` 以外のオプションを指定し、ファイル名引数なしで `fix` コマンドを実行すると、現在変更を行なったソースファイルだけが修正されます。

fix を実行すると、コンパイル時にカレントであったファイルの現在の作業ディレクトリが検索されてからコンパイル行が実行されます。したがってコンパイル時とデバッグ時とでファイルシステム構造が変化すると正しいディレクトリが見つからなくなることがあります。これを防ぐには、pathmp コマンドを使用します。これは 1 つのパス名から別のパス名までのマッピングを作成するコマンドです。マッピングはソースパスとオブジェクトファイルパスに適用されます。

修正後の続行

プログラムの実行を継続するには、cont コマンドを使用します (288 ページの「cont コマンド」参照)。

プログラムの実行を再開するには、変更による影響を判断するための以下の条件に注意してください。

実行された関数への変更

すでに実行された関数に変更を加えた場合、その変更内容は次のことが起こるまで無効です。

- プログラムが再び実行される
- その関数が次に呼び出される

変数への単純な変更以上のことを修正した場合は、fix コマンドに続けて run コマンドを使用してください。run コマンドを使用すると、プログラムの再リンクが行われないため処理が速くなります。

呼び出されていない関数への変更

呼び出されていない関数に変更を加えた場合、変更内容は、その関数が呼び出されたときに有効になります。

現在実行中の関数への変更

現在実行中の関数に変更を加えた場合、fix コマンドの影響は、変更内容が停止した関数のどの場所に関連しているかによって異なります。

- 実行済みのコードを変更しても、そのコードは再実行されません。コードを実行するには、現在の関数をスタックからポップし (325 ページの「pop コマンド」と「デバッグウィンドウの使い方」の「現在のフレームへの呼び出しスタックのポップ」参照)、変更した関数を呼び出した位置から処理を続けます。取り消すことのできない副作用 (ファイルのオープンなど) が発生しないか、コードの内容をよく理解しておく必要があります。
- 変更内容がまだ実行されていないコードにある場合は、新しいコードが実行されます。

現在スタック上にある関数への変更

停止された関数ではなく、現在スタック上にある関数に変更を加えた場合、変更されたコードは、その関数の現在の呼び出しでは使用されません。停止した関数から戻ると、スタック上の古いバージョンの関数が実行されます。

この問題を解決する方法はいくつかあります。

- 変更したすべての関数がスタックから削除されるまでpop コマンドを実行します。コードを実行して問題が発生しないか確認します。
- `cont at` *linenum* コマンドを使用して、別の行から実行を続ける。
- データ構造を手作業で修正してから (`assign` コマンドを使用)、実行を続ける。
- `run` コマンドを使用してプログラムを再び実行する。

スタック上の修正された関数にブレークポイントがある場合、このブレークポイントは、新しいバージョンの関数に移動します。古いバージョンが実行される場合、プログラムはこれらの関数で停止しません。

修正後の変数の変更

大域変数への変更は、pop コマンドでも `fix` コマンドでも取り消されません。大域変数に正しい値を手作業で再び割り当てるには、`assign` コマンドを使用してください。281 ページの「assign コマンド」参照)。

以下の例は、修正継続機能を使用して簡単なバグを修正する方法を示しています。6 行目で NULL ポインタを逆参照しようとしたときに、セグメンテーションエラーが発生します。

```
(dbx) list 1,$
    1   #include<stdio.h>
    2
    3   char *from = "ships";
    4   void copy(char *to)
    5   {
    6   while ((*to++ = *from++) != '\0');
    7   *to = '\0';
    8   }
    9
   10   main()
   11   {
   12   char buf[100];
   13
   14   copy(0);
   15   printf("%s\n", buf);
   16   return 0;
   17   }
(dbx) run
実行中: testfix
(プロセス id 11667)
シグナル SEGV (フォルトのアドレスにマッピングしていません) 関数 copy 行番号 6  ファイル "testfix.c"
    6   while ((*to++ = *from++) != '\0');
```

edit でファイルの編集をします。14 行目を 0 ではなく buf をコピー (copy) するように変更し、fix を実行します。

```
(dbx) edit
# エディタが起動します。
   14  copy(0);# 変更前
   14  copy(buf);# 変更後
# エディタを保存して終了します。
(dbx) fix
修正中  "testfix.c" ......
 pc は "testfix.c":6  に移動しました
copy で停止しました 行番号 6  ファイル "testfix.c"
    6       while ((*to++ = *from++) != '\0');
```

ここでプログラムを続行しても、NULL ポインタがスタックをプッシュしているためセグメント例外が返されます。pop コマンドを使用して、スタックフレームを 1 つ上がってください。

```
(dbx) pop
main で停止しました 行番号 14  ファイル "testfix.c"
   14   copy(buf);
```

ここでプログラムを続行すると、プログラムは実行されますが、大域変数 from がすでに増分されているため正しい値が出力されません。assign コマンドを使用して大域変数を復元し、続行してください。プログラムは次のように正しい値を表示します。assign コマンドを使用しないと、プログラムは ships と表示すべきところを hips と表示します。

```
(dbx) assign from = from-1
(dbx) cont
ships
```

ヘッダファイルの変更

場合によってはソースファイルだけでなくヘッダ (.h) ファイルも変更することがあります。変更したヘッダファイルをインクルードしている、プログラム内のすべてのソースファイルから、それらのヘッダファイルをアクセスするには、そのヘッダファイルをインクルードしているすべてのソースファイルのリストを引数として fix コマンドに渡す必要があります。ソースファイルのリストを指定しなければ、主要ソースファイルだけが再コンパイルされ、変更したヘッダファイルは主要ソースファイルにしかインクルードされず、プログラムの他のソースには変更前のヘッダファイルがインクルードされたままになります。

C++ テンプレート定義の修正

C++ テンプレート定義は直接修正できないので、これらのファイルはテンプレートインスタンスで修正します。テンプレート定義ファイルを変更しなかった場合に日付チェックを上書きするには、`-f` オプションを使用します。dbx によるテンプレート定義 `.o` ファイルの検索範囲は、デフォルトのリポジトリディレクトリ `SunWS_cache` です。dbx の `fix` コマンドは `-ptr` コンパイラスイッチをサポートしていません。

第12章

マルチスレッドアプリケーションのデバッグ

dbx では Solaris スレッドや POSIX スレッドを使用するマルチスレッドアプリケーションをデバッグできます。dbx には、各スレッドのスタックトレースの確認、全スレッドの再実行、特定のスレッドに対する step や next の実行、スレッド間の移動をする機能があります。

dbx は、libthread.so が使用されているかどうかを検出することによって、マルチスレッドプログラムかどうかを認識します。プログラムは、-lthread または -mt を使用してコンパイルすることによって明示的に、あるいは -lpthread を使用してコンパイルすることによって暗黙的に libthread.so を使用します。

この章では、dbx の thread コマンドを使用して、スレッドに関する情報を入手したり、デバッグを行う方法について説明します。

この章は次の各節から構成されています。

- マルチスレッドデバッグについて
- LWP 情報について

マルチスレッドデバッグについて

dbx は、マルチスレッドプログラムを検出すると、libthread_db.so の dlopen を試行します。これは、/usr/lib にあるスレッドデバッグ用の特別なシステムライブラリです。

dbx は同期的に動作します。つまり、スレッドか軽量プロセス (LWP) のいずれかが停止すると、ほかのスレッドおよび LWP もすべて自動的に停止します。この動作は、「世界停止」モデルと呼ばれる場合があります。

注 – マルチスレッドプログラミングと LWP については、『Solaris マルチスレッドのプログラミング』を」参照してください。

スレッド情報

dbx では、次のスレッド情報を入手できます。

```
(dbx) threads
    t@1 a l@1  ?()  実行中：現在の関数 main()
    t@2      ?() 0xef751450 でスリープ：現在の関数 in_swtch()
    t@3 b l@2  ?() 実行中：現在の関数 sigwait()
    t@4     consumer() 0x22bb0 でスリープ：現在の関数
_lwp_sema_wait()
  *>t@5 b l@4 consumer()  ブレークポイント：現在の関数
Queue_dequeue()
    t@6 b l@5 producer()      実行中：現在の関数 _thread_start()
(dbx)
```

情報の各行の内容は次のとおりです。

- * (アスタリスク) は、ユーザーの注意を必要とするイベントがこのスレッドで起こったことを示します。通常は、ブレークポイントに付けられます。

  アスタリスクの代わりに「o」が示される場合は、dbx 内部イベントが発生しています。

- > (矢印) は現在のスレッドを示します。
- t@*num* はスレッド ID であり、特定のスレッドを指します。*number* は、thr_create によって返された thread_t の値になります。
- b l@*num* はそのスレッドが指定の LWP に結合されていることを表し、a l@*num* はそのスレッドがアクティブであることを表します。すなわちそのスレッドはオペレーティングシステムにて実行可能です。
- thr_create に渡されたスレッドの開始関数。?() は開始関数が不明であることを示します。
- スレッドの状態。(スレッドの状態については、表 12-1 参照)。

■ スレッドが現在実行している関数。

表 12-1 スレッドの状態と LWP の状態

スレッドの状態と LWP の状態	解説
中断	スレッドは明示的に中断されています。
実行可能	スレッドは実行可能であり、コンピューティング可能なリソースとして LWP を待機しています。
ゾンビ	結合されてないスレッド (`thr_exit()`) がある場合、`thr_join()` で再結合するまでゾンビ状態になります。`THE_DETACHED` は、スレッド作成時に指定するフラグです (`thr_create()`)。非結合のスレッドは、再実行されるまでゾンビ状態です。
syncobj でスリープ中	スレッドは所定の同期オブジェクトでブロックされています。`libthread` と `libthread_db` によるサポートレベルにより、*syncobj* が伝える情報は単純な 16 進アドレスになったり、より詳細な内容になります。
アクティブ	LWP でスレッドがアクティブですが、`dbx` は LWP をアクセスできません。
未知	`dbx` では状態を判定できません。
lwpstate	結合スレッドやアクティブスレッドの状態に、LWP の状態が関連付けられています。
実行中	LWP が実行中でしたが、他の LWP と同期して停止しました。
システムコール *num*	所定のシステムコール番号の入口で LWP が停止しました。
システムコール *num* 戻り	所定のシステムコール番号の出口で LWP が停止しました。
ジョブコントロール	ジョブコントロールにより、LWP が停止しました。
LWP 中断	LWP がカーネルでブロックされています。
シングル中断	LWP により、1 ステップが終了しました。
ブレークポイント	LWP がブレークポイントに達しました。
障害 *num*	LWP に所定の障害番号が発生しました。
シグナル *name*	LWP に所定のシグナルが発生しました。
プロセス sync	この LWP が所属するプロセスの実行が開始しました。
LWP 終了	LWP は終了プロセス中です。

別のスレッドのコンテキストの表示

表示コンテキストを別のスレッドに切り替えるには、thread コマンドを使用します。この構文は次のとおりです。

```
thread [-info] [-hide] [-unhide] [-suspend] [-resume] thread_id
```

現在のスレッドを表示するには、次のように入力します。

```
thread
```

スレッド *thread_id tid* に切り替えるには、次のように入力します。

```
thread thread_id
```

thread コマンドの詳細については、319ページの「thread コマンド」を参照してください。

スレッドリストの表示

スレッドリストを表示するには、thread コマンドを使用します。構文は次のとおりです。

```
threads [-all] [-mode [all|filter] [auto|manual]]
```

既知のスレッドすべてのリストを表示するには、次のように入力します。

```
threads
```

通常は表示されないスレッド (ゾンビ) などを表示するには、次のように入力します。

```
threads -all
```

スレッドリストについては、176 ページの「スレッド情報」を参照してください。

threads コマンドの詳細については、345 ページの「threads コマンド」を参照してください。

実行の再開

プログラムの実行を再開するには、cont コマンドを使用します。現在、スレッドは同期ブレークポイントを使用して、すべてのスレッドが実行を再開するようにしています。

LWP 情報について

通常は LWP を意識する必要はありません。ただし、スレッドレベルでの問い合わせが完全にできない場合には、lwps コマンドを使用して、LWP に関する情報を入手できます。

```
(dbx) lwps
    l@1 実行中：現在の関数 main()
    l@2 実行中：現在の関数 sigwait()
    l@3 実行中：現在の関数 _lwp_sema_wait()
  *>l@4 ブレークポイント：現在の関数 Queue_dequeue()
    l@5 実行中：現在の関数 _thread_start()
(dbx)
```

LWP リストの各行の内容は、次のとおりです。

- * (アスタリスク) は、ユーザーの注意を要するイベントが この LWP で起こったことを示します。
- 矢印は現在の LWP を表します。
- l@*num* は特定の LWP を示します。
- 179 ページの「LWP 情報について」では詳しい LWP の状態を説明しています。
- *func_name()* は、LWP が現在実行している関数を示します。

第13章

子プロセスのデバッグ

この章では、子プロセスのデバッグ方法を説明します。dbx は、fork(2) および exec(2) を介して子を作成するプロセスのデバッグに役立つ機能をいくつか備えています。

この章は次の各節から構成されています。

- 単純な接続の方法
- exec 機能後のプロセス追跡
- fork 機能後のプロセス追跡
- イベントとの対話

単純な接続の方法

子プロセスがすでに作成されている場合は、次のいずれかの方法でそのプロセスに接続できます。

dbx 起動時、シェルから次のように入力します。

```
$ dbx program_name process_id
```

コマンド行からは次のように入力します。

```
dbx) debug program_name process_id
```

どちらの場合も*program_name* を "-" に置き換えることができます。そうすると、dbx は指定されたプロセス ID (*process_id*) に対応する実行可能ファイルを自動的に見つけ出します。
"-" を使用すると、それ以後 run コマンドおよび rerun コマンドは機能しません。これは、dbx が実行可能ファイルの絶対パス名を知らないためです。

Sun WorkShop の「デバッグ」ウィンドウからは、実行中の子プロセスにも結合できます (Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「実行中のプロセスの接続」を参照してください)。

exec 機能後のプロセス追跡

子プロセスが新しいプログラムを exec(2) 関数を用いて実行すると、そのプロセス ID は変わりませんが、プロセスイメージは変化します。dbx は exec() の呼び出しを自動的に検知し、新しく実行されたプログラムを自動的に再読み込みします。

実行可能ファイルの元の名前は、$oprog に保存されます。この名前に復帰するには、debug $oprog を使用します。

fork 機能後のプロセス追跡

子プロセスが、関数 vfork()、fork(1)、または fork(2) を呼び出すと、プロセス ID が変化しますが、プロセスイメージは変化しません。dbxenv 環境変数 follow_fork_mode の設定値にしたがって、dbx は次のように動作します。

parent (親プロセス)	従来の動作です。dbx は fork を無視し、親プロセスを追跡します。
child (子プロセス)	dbx は、新しいプロセス ID で、分岐先の子に自動的に切り替わります。
both (両方)	このモードは、Sun WorkShop から dbx を使用する場合しか利用できません。

ask (質問)　　dbx が fork を検出するたびにプロンプトが表示され、parent、child、both のどのモードを使用するか問い合わせてきます。stop を選択すると、プログラムの状態を調べてから、cont を使用して実行を続けることができます。プロンプトに従って次の処理を選択します。

イベントとの対話

exec() 関数や fork() 関数では、ブレークポイントや他のイベントが、すべて削除されます。しかし、dbx 環境変数 follow_fork_inherit を on に設定するか、-perm eventspec 修飾子でイベントを持続イベントにすれば、ブレークポイントや他のイベントは削除されません。イベント仕様修飾子の使用方法の詳細については、付録 B を参照してください。

第14章

シグナルの処理

この章では、dbx を使用してシステムシグナルを処理する方法を説明します。dbx は、catch というブレークポイントコマンドをサポートします。catch コマンドは、catch リストに登録されているシステムシグナルのいずれかが検出された場合にプログラムを停止するよう dbx に指示します。

また、dbx コマンド cont、step、next は、オプション -sig *signal_name* をサポートします。このオプションを使用すると、実行を再開したプログラムに対し、cont -sig コマンドで指定したシグナルを受信した場合の動作をさせることができます。

この章は次の各節から構成されています。

- シグナルイベントについて
- システムシグナルを捕獲する
- プログラム内でシグナルを送信する
- シグナルの自動処理

シグナルイベントについて

デバッグ中のプロセスにシグナルが送信されると、そのシグナルはカーネルによって dbx に送られます。通常、このことはプロンプトによって示されますが、そこでは次の 2 つの操作から 1 つを選択してください。

■ プログラムを再開するときにそのシグナルを「取り消し」ます。これは、cont コマンドのデフォルトの動作です。これにより、次の図 14-1のような SIGINT (Control-C) を使用した割り込みと再開が容易になります。

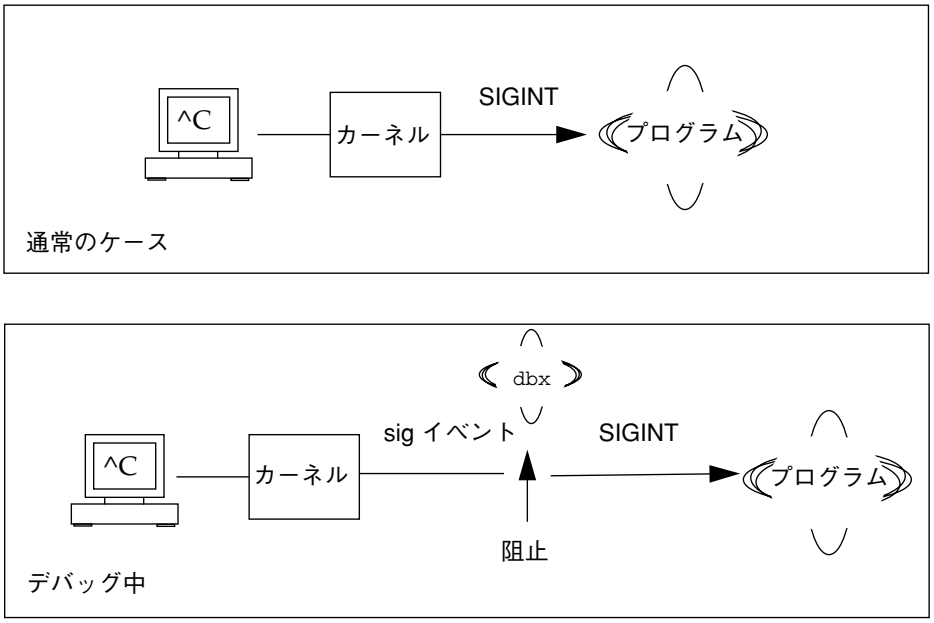


図 14-1 SIGINT シグナルの阻止と取り消し

■ 次のコマンドを使用して、シグナル *sig* をプロセスに「転送」します。

```
cont -sig sig
```

さらに、特定のシグナルを頻繁に受信する場合、そのシグナルを dbx が自動的に転送するように設定できます。次のように入力します。

```
ignore sig # "ignore"
```

以上の操作をしてもシグナルはプロセスに送信されます。シグナルがデフォルト設定で、このように自動送信されるようになっているからです (310 ページの「ignore コマンド」参照)。

システムシグナルを捕獲する

デフォルトのシグナル捕獲リスト (catch リスト) には、33 種類の検出可能なシグナルのうちの 22 種類が含まれています (これらの数はオペレーティングシステムとそのバージョンによって異なります)。デフォルトの catch リストは、リストにシグナルを追加したり削除したりすることによって変更できます。

現在捕獲されているシグナルのリストを調べるには、シグナルの引数を指定せずに、次のように入力します。

```
(dbx) catch
```

プログラムで検出された場合でも、現在無視されているシグナルのリスト (ignore リスト) を調べるには、シグナル名の引数を指定せずに、次のように入力します。

```
(dbx) ignore
```

デフォルトの `catch` リストと `ignore` リストを変更する

どのシグナルでプログラムを停止するかは、2 つのリストの間でシグナル名を移動することによって制御します。シグナル名を移動するには、一方のリストに現在表示されているシグナル名を、もう一方のリストに引数として渡します。

たとえば、`QUIT` シグナルと `ABRT` シグナルを catch リストから ignore リストに移動するには、次のように入力します。

```
(dbx) ignore QUIT ABRT
```

`FPE` シグナルをトラップする

浮動小数点の計算が必要なコードを扱っている場合には、プログラム内で発生した例外をデバッグしなければならないことがよくあります。オーバーフローやゼロ除算などの浮動小数点例外が発生すると、例外を起こした演算の結果としてシステムが「適

正な」答を返します。適正な答が返されることで、プログラムは正常に実行を続けることができます (Solaris 2.x コンピュータは、IEEE 標準のバイナリ浮動小数点演算定義の、例外に対する「適正 (reasonable) な」答を実装しています)。

浮動小数点例外に対して適正な答を返すため、例外によって自動的に SIGFPE シグナルが生成されることはありません。例外の場合 (ゼロで整数を割ると整数がオーバーフローする場合など) は、デフォルトでは SIGFPE シグナルをトリガーします。

例外の原因を見つけ出すためには、例外によって SIGFPE シグナルが生成されるように、トラップハンドラをプログラム内で設定する必要があります (トラップハンドラの例については、マニュアルの ieee_handler(3m) コマンドを参照)。

トラップを有効にするには、次のコマンド等を利用します。

- ieee_handler
- fdsetmask (fdsetmask(3c) マニュアルページ参照)
- -ftrap コンパイラフラグ (FORTRAN 77 と Fortran 95 については、マニュアルページ f77(1) と f95(1) を参照)

ieee_handler コマンドを使用してトラップハンドラを設定すると、ハードウェア浮動小数点状態レジスタ内のトラップ許可マスクがセットされます。このトラップ許可マスクにより、実行中に例外が発生すると SIGFPE シグナルが生成されます。

トラップハンドラ付きのプログラムをコンパイルした後、そのプログラムを dbx に読み込んでください。ここで、SIGFPE シグナルが捕獲されるようにするには、dbx のシグナル捕獲リスト (catch リスト) に FPE を追加する必要があります。

```
(dbx) catch FPE
```

FPE はデフォルトでは ignore リストに含まれています。

FPE を catch リストに追加後、dbx でプログラムを実行します。トラップしている例外が発生すると SIGFPE シグナルが生成され、dbx はプログラムを停止します。ここで、呼び出しスタックを (dbx コマンド where を使用して) トレースすることにより、プログラムの何行目で例外が発生したかを調べることができます。355 ページの「where コマンド」参照)。

例外処理の原因追求

例外処理の原因を調べるには、`regs -f` コマンドを実行して浮動小数点状態レジスタ (FSR) を表示します。このレジスタで、発生した例外処理 (`aexc`) フィールドと現在の例外処理 (`cexc`) フィールドの内容を確認します。このフィールドには次のような浮動小数点例外条件が格納されています。

- 無効なオペランド
- オーバフロー
- アンダフロー
- ゼロによる除算
- 不正確な結果

浮動小数点状態レジスタの詳細については、『SPARC アーキテクチャマニュアル バージョン 8』(V9 の場合はバージョン 9) を参照してください。

プログラム内でシグナルを送信する

`dbx` コマンド `cont` は、オプション `-sig` *signal* をサポートします。このオプションを使用すると、実行を再開したプログラムに対し、**指定した**システムシグナル*signal* を受信した場合の動作をさせることができます。

たとえば、プログラムに `SIGINT` (`^C`) の割り込みハンドラが含まれている場合、`^C` を入力することによって、アプリケーションを停止し、`dbx` に制御を返すことができます。ここで、プログラムの実行を継続するときにオプションなしの `cont` コマンドを使用すると、割り込みハンドラは実行されません。割り込みハンドラを実行するためには、プログラムに `SIGINT` シグナルを送信する必要があります。次のコマンドを使用します。

```
(dbx) cont -sig int
```

`stop`、`next`、`detach` コマンドも、`-sig` オプションを指定できます。

シグナルの自動処理

イベント管理コマンドでは、シグナルをイベントとして処理することもできます。次の 2 つのコマンドの結果は同じになります。

```
(dbx) stop sig signal
(dbx) catch signal
```

プログラミング済みのアクションを関連付ける必要がある場合、シグナルイベントがあると便利です。

```
(dbx) when sig SIGCLD {echo Got $sig $signame;}
```

この場合は、まず SIGCLD を ignore リストに必ず移動してください。

```
(dbx) ignore SIGCLD
```

第 15 章

C++ のデバッグ

この章では、dbx による C++ の例外の処理方法と C++ テンプレートのデバッグについて説明します。これらの作業を実行するために使用するコマンドの要約とコード例も示します。

この章は次の各節から構成されています。

- C++ での dbx の使用
- dbx での例外処理
- C++ テンプレートでのデバッグ

C++ プログラムのコンパイルの詳細については、1 ページの「デバッグを目的としてコード をコンパイルする」を参照してください。

C++ での dbx の使用

この章では C++ デバッグの 2 つの特殊な点を中心に説明しますが、dbx を使用すると、C++ プログラムのデバッグに次の機能を利用することができます。

- クラスと型定義の検索 (56 ページの「型およびクラスの定義を調べる」参照)
- 継承されたデータメンバーの出力または表示 (98 ページの「C++ での表示」参照)
- オブジェクトポインタに関する動的情報の検索 (98 ページの「C++ での表示」参照)
- 仮想関数のデバッグ (68 ページの「関数を呼び出す」参照)
- 実行時型情報の使用 (98 ページの「変数または式の値を出力する」参照)
- クラスのすべてのメンバー関数に対するブレークポイントの設定 (75 ページの「同じクラスのメンバー関数にブレークポイントを設定する」」参照)

- 多重定義されたすべてのメンバー関数に対するブレークポイントの設定 (75 ページの「異なるクラスのメンバー関数にブレークポイントを設定する」参照)
- 多重定義されたすべての非メンバー関数に対するブレークポイントの設定 (76 ページの「非メンバー関数に複数のブレークポイントを設定する」参照)
- 特定オブジェクトのすべてのメンバー関数に対するブレークポイントの設定 (73 ページの「関数に stop ブレークポイントを設定する」参照)
- 多重定義された関数/データメンバーの処理 (77 ページの「オブジェクトにブレークポイントを設定する」参照)

これらの詳細については、索引の C++ の下の項目を参照してください。

dbx での例外処理

プログラムは例外が発生すると実行を停止します。例外は、ゼロによる除算や配列のオーバーフローといったプログラムの障害を知らせるものです。ブロックを設定して、コードのどこかほかの場所で起こった式による例外を捕獲できます。

プログラムのデバッグ中、dbx を使用すると次のことが可能になります。

- スタックを解放する前に処理されていない例外を捕獲する
- 予期できない例外を捕獲する
- スタックを解放する前に、特定の例外が処理されたかどうかに関係なく捕獲する
- 特定の例外がプログラム内の特定の位置で起こった場合、それが捕獲される場所を決める

例外処理の発生個所で `step` コマンドを実行すると、スタックの解放時に実行された最初のデストラクタの先頭に制御が戻ります。`step` を実行して、スタックの解放時に実行されたデストラクタを終了すると、制御は次のデストラクタの先頭に移ります。こうしてすべてのデストラクタが終了した後に `step` コマンドを実行すると、例外処理の原因を扱う捕獲ブロックに制御が移ります。

例外処理コマンド

`exception [-d | +d]` コマンド

`exception` コマンドでは、デバッグ時にいつでも例外処理の型を確認できます。オプションなしで `exception` コマンドを実行するときに表示される型は、`dbx` 環境変数 `output_dynamic_type` の設定で制御できます。

- この変数を `on` に設定すると、派生型が表示されます。
- この変数を `off` (デフォルト) に設定すると、静的な型が表示されます。

`-d` オプションや `+d` オプションを指定すると、環境変数の設定が無効になります。

- `-d` を設定すると、派生型が表示されます。
- `+d` を設定すると、静的な型が表示されます。

詳細については、283ページの「exception コマンド」を参照してください。

`intercept [-a|-x|`*typename*`]` コマンド

スタックを解放する前に、特定の型の例外を阻止または捕獲できます。`intercept` コマンドを引数を付けずに使用すると、阻止される型がリストで示されます。`-a` を使用すると、すべての例外が阻止されます。阻止リストに型を追加するには `typename` を使用します。`-x` を使用すると、特定の型を阻止から除外することができます。

たとえば、`int` を除くすべての型を阻止するには、次のように入力します。

```
(dbx) intercept -a
(dbx) intercept -x int
```

詳細については、311 ページの「intercept コマンド」を参照してください。

`unintercept [-a|-x |`*typename*`]` コマンド

`unintercept` コマンドは、阻止リストから例外の型を削除するために使用します。引数を付けずにこのコマンドを使用すると、阻止されている型のリストが示されます (`intercept` コマンドに同じ)。`-a` を使用すると、リストから阻止された型すべてが削除されます。*typename* を使用すると、阻止リストから 1 つの型を削除することができます。`-x` は、特定の型を阻止から除外することをやめるために使用します。

詳細については、349 ページの「unintercept コマンド」を参照してください。

whocatches *typename* コマンド

whocatches コマンドは、*typename* の例外が実行の現時点で送出された場合に、どこで捕獲されるかを報告するものです。このコマンドは、例外がスタックのトップフレームから送出された場合に何が起こるかを検出する場合に使用します。

typename を捕獲した元の送出の行番号、関数名、およびフレーム数が表示されます。捕獲ポイントがスルーを行っている関数と同じ関数内にあると、このコマンドは、「型にはハンドルがありません」というメッセージを表示します。

詳細については、357 ページの「whocatches コマンド」を参照してください。

例外処理の例

次の例は、例外を含むサンプルプログラムを使用して、dbx で例外処理がどのように実行されるかを示しています。型 int の例外が、関数 bar で送出されて、次の捕獲ブロックで捕獲されています。

```
1  #include <stdio.h>
2
3  class c {
4      int x;
5    public:
6      c(int i) { x = i; }
7      ~c() {
8      printf("destructor for c(%d)\n", x);
9   }
10  };
11
12  void bar() {
13      c c1(3);
14      throw(99);
15  }
16
17  int main() {
18      try {
19          c c2(5);
20          bar();
21          return 0;
22      }
23      catch (int i) {
24          printf("caught exception %d\n", i);
25      }
26  }
```

サンプルプログラムからの次のトランスクリプトは、dbx の例外処理機能を示しています。

```
(dbx) intercept
-unhandled -unexpected
(dbx) intercept int
<dbx> intercept
-unhandled -unexpected int
(dbx) stop in bar
(2) stop in bar()
(dbx)run
実行中: a.out
(process id 304)
bar で停止しました 行番号 13 ファイル "foo.cc"
   13       c c1(3);
(dbx) whocatches int
int が行番号 24 で捕獲されました、関数 main (フレーム番号 2)
(dbx) whocatches c
dbx: class c の実行時型情報がありません (送出も捕獲もされていない)
(dbx) cont
例外の型 int が行番号 24 で捕獲されました、関数 main (フレーム番号 4)
_exdbg_notify_of_throw で停止しました アドレス 0xef731494
0xef731494: _exdbg_notify_of_throw        :       jmp     %o7
+ 0x8
現関数 :bar
   14        throw(99);
(dbx) step
c::~c で停止しました 行番号 8  ファイル "foo.cc"
    8         printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(3)
c::~c で停止しました 行番号 9  ファイル "foo.cc"
    9       }
(dbx) step
c::~c で停止しました 行番号 8  ファイル "foo.cc"
    8        printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(5)
c::~c で停止しました 行番号 9  ファイル "foo.cc"
    9       )
(dbx) step
main で停止しました 行番号 24  ファイル "foo.cc"
   24           printf("caught exception %d\n", i);
(dbx) step
caught exception 99
main で停止しました 行番号 26  ファイル "foo.cc"
   26   }
```

C++ テンプレートでのデバッグ

dbx は C++ テンプレートをサポートしています。クラスおよび関数テンプレートを含むプログラムを dbx に読み込み、クラスや関数に対して使用する任意の dbx コマンドをテンプレートに対して次のように呼び出すことができます。

- クラスまたは関数テンプレートのインスタンス化にブレークポイントを設定する(202 ページの「stop inclass classname コマンド」、202 ページの「stop infunction name コマンド」、202 ページの「stop in function コマンド」、参照)
- すべてのクラスおよび関数テンプレートのインスタンス化のリストを出力する(199 ページの「whereis name コマンド」参照)
- テンプレートおよびインスタンスの定義を表示する(200 ページの「whatis name コマンド」参照)
- メンバーテンプレート関数と関数テンプレートのインスタンス化を呼び出す(203 ページの「call function_name (parameters) コマンド」参照)
- 関数テンプレートのインスタンス化の値を出力する(203 ページの「print コマンド」参照)
- 関数テンプレートのインスタンス化のソースコードを表示する(204 ページの「list コマンド」参照)

テンプレートの例

次のコード例は、クラステンプレート `Array` とそのインスタンス化、および関数テン

プレート square とそのインスタンス化を示しています。

```
 1  template<class C> void square(C num, C *result)
 2  {
 3      *result = num * num;
 4  }
 5
 6  template<class T> class Array
 7  {
 8  public:
 9      int getlength(void)
10      {
11          return length;
12      }
13
14      T & operator[](int i)
15      {
16          return array[i];
17      }
18
19      Array(int l)
20      {
21          length = l;
22          array = new T[length];
23      }
24
25      ~Array(void)
26      {
27          delete [] array;
28      }
29
30  private:
31      int length;
32      T *array;
33  };
34
35  int main(void)
36  {
37      int i, j = 3;
38      square(j, &i);
39
40      double d, e = 4.1;
41      square(e, &d);
42
```

```
43      Array<int> iarray(5);
44      for (i = 0; i < iarray.getlength(); ++i)
45      {
46          iarray[i] = i;
47      }
48
49      Array<double> darray(5);
50      for (i = 0; i < darray.getlength(); ++i)
51      {
52          darray[i] = i * 2.1;
53      }
54
55      return 0;
56  }
```

この例の内容は次のとおりです。

- `Array` はクラステンプレート
- `square` は関数テンプレート
- `Array<int>` はクラステンプレートインスタンス化 (テンプレートクラス)
- `Array<int>::getlength` はテンプレートクラスのメンバー関数
- `square(int, int*)` と `square(double, double*)` は関数テンプレートのインスタンス化 (テンプレート関数)

C++ テンプレートのコマンド

以下に示すコマンドは、テンプレートおよびインスタンス化されたテンプレートに使用します。クラスまたは型定義がわかったら、値の出力、ソースリストの表示、またはブレークポイントの設定を行うことができます。

`whereis` *name* コマンド

`whereis` コマンドは、関数テンプレートまたはクラステンプレートの、インスタンス化された関数やクラスの出現すべてのリストを出力するために使用します。

クラステンプレートの場合は、次のように入力します。

```
(dbx) whereis Array
メンバー関数: 'Array<int>::Array(int)
メンバー関数: 'Array<double>::Array(int)
クラステンプレートインスタンス: 'Array<int>
クラステンプレートインスタンス: 'Array<double>
クラステンプレート: 'a.out'template_doc_2.cc'Array
```

関数テンプレートの場合は、次のように入力します。

```
(dbx) whereis square
関数テンプレートインスタンス: 'square<int>(__type_0,__type_0*)
関数テンプレートインスタンス: 'square<double>(__type_0,__type_0*)
関数テンプレート:      'a.out'square
```

_type_0 パラメータは、0 番めのパラメータを表します。_type_1 パラメータは、次のパラメータを表します。

詳細については、356 ページの「whereis コマンド」を参照してください。

whatis *name* コマンド

関数テンプレートおよびクラステンプレートと、インスタンス化された関数やクラスの定義を出力するために使用します。

クラステンプレートの場合は、次のように入力します。

```
(dbx) whatis Array
template<class T> class Array ;
完全なテンプレート宣言を得るために次を実行してください:
                                        'whatis -t Array<int>';
```

クラステンプレートとの構造については次のように実行します。

```
(dbx) whatis Array
複数の識別子 'Array'.
次のどれかを選択してください :
 0) Cancel
 1) Array<int>::Array(int)
 2) Array<double>::Array(int>
> 1
Array<int>::Array(int 1);
```

関数テンプレートの場合は、次のように入力します。

```
(dbx) whatis square
複数の識別子 'square'.
次のどれかを選択してください:
 0) Cancel
 1) square<int(__type_0,__type_0*)
 2) square<double>(__type_0,__type_0*)
> 2
void square<double>(double num, double *result);
```

クラステンプレートのインスタンス化の場合は、次のように入力します。

```
(dbx) whatis -t Array<double>
class Array<double> {
public:
   int Array<double>::getlength();
   double &Array<double>::operator[](int i);
   Array<double>::Array<double>(int l);
   Array<double>::~Array<double>();
private:
   int length;
   int *array;
};
```

関数テンプレートのインスタンス化の場合は、次のように入力します。

```
(dbx) whatis square(int, int*)
void square(int num, int *result);
```

詳細については、356 ページの「whereis コマンド」とを参照してください。

stop inclass *classname* コマンド

テンプレートクラスのすべてのメンバー関数を停止するには、次のように入力します。

```
(dbx)stop inclass Array
(2) stop inclass Array
```

stop inclass コマンドを使用して、特定のテンプレートクラスのメンバー関数すべてにブレークポイントを設定します。

```
(dbx) stop inclass Array<int>
(2) stop inclass Array<int>
```

詳細については、339 ページの「stop コマンド」と 261 ページの「inclass classname」を参照してください。

stop infunction *name* コマンド

stop infunction コマンドを利用して、指定した関数テンプレートのインスタンスにブレークポイントを設定します。

```
(dbx) stop infunction square
(9) stop infunction square
```

詳細については、339 ページの「stop コマンド」と 261 ページの「infuction function」を参照してください。

stop in *function* コマンド

stop in コマンドを使用して、あるテンプレートクラスのメンバー関数、またはテンプレート関数にブレークポイントを設定します。

クラスインスタンス化のメンバーの場合は、次のとおりです:

```
(dbx) stop in Array<int>::Array<int>(int l)
(2) stop in Array<int>::Array<int>(int)
```

```
(dbx) stop in square(double, double*)
(6) stop in square(double, double*)
```

詳細については、339 ページの「stop コマンド」と 261 ページの「infuction function」を参照してください。

call *function_name* (*parameters*) コマンド

スコープ内で停止した場合に、関数インスタンス化やクラステンプレートのメンバー関数を明示的に呼び出すには、call コマンドを使用します。dbx で正しいインスタンスを選択できない場合、表示されるメニューを利用します。

```
(dbx) call square(j,i)
```

詳細については、283 ページの「call コマンド」を参照してください。

print コマンド

print コマンドを使用して、インスタンス化された関数またはクラステンプレートメンバー関数を評価します。

```
(dbx) print iarray.getlength()
iarray.getlength() = 5
```

print を使用して this ポインタを評価します。

```
(dbx) whatis this
class Array<int> *this;
(dbx) print *this
*this = {
    length = 5
    array   = 0x21608
}
```

詳細については、326 ページの「print コマンド」を参照してください。

list コマンド

list コマンドを使用して、指定のインスタンス化された関数のソースリストを出力します。

```
(dbx) list square(int, int*)
```

詳細については、313 ページの「list コマンド」を参照してください。

第16章

dbx を使用した Fortran のデバッグ

この章では、Fortran で使用されることが多いいくつかの dbx 機能を紹介します。dbx を使用して Fortran コードをデバッグするときの助けになる、dbx に対する要求の例も示してあります。

この章は次の各節から構成されています。

- Fortran のデバッグ
- セグメント不正のデバッグ
- 例外の検出
- 呼び出しのトレース
- 配列の操作
- 組み込み関数
- 複合式
- 論理演算子
- Fortran 95 構造型の表示
- Fortran 95 構造型へのポインタ

Fortran のデバッグ

次のアドバイスと概要は、Fortran プログラムをデバッグするときに役立ちます。

カレントプロシージャとカレントファイル

デバッグセッション中、dbx は、1 つのプロシージャと 1 つのソースファイルをカレントとして定義します。ブレークポイントの設定要求と変数の出力または設定要求は、カレントの関数とファイルに関連付けて解釈されます。したがって、`stop at 5` は、カレントファイルがどれであるかによって、3 つの異なるブレークポイントのうち 1 つを設定します。

大文字

プログラムのいずれかの識別子に大文字が含まれる場合、dbx はそれらを認識します。いくつかの旧バージョンの場合のように、大文字/小文字を区別するコマンド、または区別しないコマンドを指定する必要はありません。

FORTRAN 77、Fortran 95 と dbx は、大文字/小文字を区別するモードまたは区別しないモードのいずれかに統一する必要があります。

- 大文字/小文字を区別しないモードでコンパイルとデバッグを行うには、`-U` オプションを付けずにこれらの処理を行います。その場合、dbx `input_case_sensitive` 環境変数のデフォルト値は `false` になります。

ソースに `LAST` という変数がある場合、dbx では、`print LAST` コマンドおよび `print last` コマンドはいずれも要求どおりに動作します。FORTRAN 77、Fortran 95 と dbx は、`LAST` と `last` を要求通り同じものとして扱います。

- 大文字/小文字を区別するモードでコンパイルとデバッグを行うには、`-U` オプションを付けます。その場合、`dbx input_case_sensitive` 環境変数のデフォルト値は `true` になります。

ソースに `LAST` という変数と `last` という変数がある場合、dbx では、`print LAST` コマンドは動作しますが、`print last` コマンドは動作しません。FORTRAN 77、Fortran 95 と dbx はいずれも、`LAST` と `last` を要求どおりに区別します。

注 – dbx input_case_sensitive 環境属性の環境変数を false に設定しても、dbx ではファイル名またはディレクトリ名について、大文字/小文字を常に区別します。

最適化プログラム

最適化プログラムをデバッグするには、次のことを行ってください。

- -g を付けて、-O*n* を付けずにメインプログラムをコンパイルする。
- 適切な -O*n* を付けて、プログラムのルーチンを 1 つおきにコンパイルする。
- dbx のもとで実行を開始する。
- デバッグしたいルーチンに対して fix -g *any*.f を使用する。ただし、-O*n* は付けない。
- コンパイルされたルーチンに対して cont コマンドを使用する。

デバッグのメインプログラムは次のとおりです。

デバッグの主プログラム

a1.f

```
      PARAMETER ( n=2 )
      REAL twobytwo(2,2) / 4 *-1 /
      CALL mkidentity( twobytwo, n )
      PRINT *, determinant( twobytwo )
      END
```

デバッグのサブルーチン

a2.f

```
      SUBROUTINE mkidentity ( array, m )
      REAL array(m,m)
      DO 90 i = 1, m
      DO 20 j = 1, m
          IF ( i .EQ. j ) THEN
          array(i,j) = 1.
          ELSE
          array(i,j) = 0.
          END IF
20    CONTINUE
90    CONTINUE
      RETURN
      END
```

a3.f

```
      REAL FUNCTION determinant ( a )
      REAL a(2,2)
      determinant = a(1,1) * a(2,2) - a(1,2) / a(2,1)
      RETURN
      END
```

dbx のサンプルセッション

以下の例では、上記のサンプルプログラム my_program を使用しています。

1. -g オプションでコンパイルとリンクをします。

   この処理は、まとめて 1 回または 2 回に分けて実行することができます。

   -g フラグ付きコンパイルとリンクを 1 度にまとめて行います。

```
demo% f95 -o my_program -g a1.f a2.f a3.f
```

   コンパイルとリンクを分けて行います。

```
demo% f95 -c -g a1.f a2.f a3.f
demo% f95 -o my_program a1.o a2.o a3.o
```

2. 実行可能ファイル my_program について dbx を起動します。

```
demo% dbx my_program
Reading symbolic information…
```

3. **stop in** *subnam* と入力して、最初の実行可能文の前にブレークポイントを設定する。*subnam* は、サブルーチン、関数、ブロックデータサブプログラムを示します。

   main プログラム中の最初の実行可能文で停止します。

```
(dbx) stop in MAIN
(2) stop in MAIN
```

   通常 MAIN は大文字ですが、*subnam* は大文字でも小文字でもかまいません。

4. run コマンドを入力して、dbx からプログラムを実行します。dbx の起動時に指定された実行可能ファイルの中で、プログラムが実行されます。

```
(dbx) run
実行中: my_program
MAIN で停止しました 行番号 3  ファイル "a1.f"
    3   call mkidentity( twobytwo, n )
```

ブレークポイントに到達すると、dbx はどこで停止したかを示すメッセージを表示します。上の例では、a1.f ファイルの行番号 3 で停止しています。

5. print コマンドを使用して、値を出力します。

n の値を出力します。

```
 (dbx) print n
 n = 2
```

マトリックス twobytwo を出力します。

```
(dbx) print twobytwo
twobytwo =
    (1,1)       -1.0
    (2,1)       -1.0
    (1,2)       -1.0
    (2,2)       -1.0
```

マトリックス array を出力します。

```
(dbx) print array
dbx: "array" が現在のスコープに定義されていません。
(dbx)
```

ここで array は定義されていないため、出力は失敗します (mkidentity 内でのみ有効)。

6. next コマンドを使用して、次の行に実行を進めます。

次の行に実行を進めます。

```
(dbx) next
MAIN で停止しました 行番号 4  ファイル "a1.f"
    4  print *, determinant( twobytwo )
(dbx) print twobytwo
twobytwo =
    (1,1)        1.0
    (2,1)        0.0
    (1,2)        0.0
    (2,2)        1.0
(dbx) quit
demo%
```

next コマンドは現在のソース行を実行し、次のソース行で停止します。これは副プログラムの呼び出しを 1 つの文として数えます。

next コマンドと step コマンドを比較します。step コマンドは、ソースの次の行または副プログラムの次のステップを実行します。通常、次の実行可能ソース文がサブルーチンまたは関数呼び出しの場合、各コマンドは次の処理を行います。

- step コマンドは、副プログラムのソースの最初の文にブレークポイントを設定します。
- next コマンドは、呼び出し元のプログラム中で、呼び出しの後の最初の文にブレークポイントを設定します。

7. quit コマンドを入力して、dbx を終了します。

```
(dbx)quit
demo%
```

セグメント不正のデバッグ

プログラムでセグメント不正 (SIGSEGV) が発生するのは、プログラムが使用可能なメモリー範囲外のメモリーアドレスを参照したことを示します。

セグメント不正の主な原因を以下に示します。

- 配列インデックスが宣言された範囲外にある。
- 配列インデックス名のつづりが間違っている。
- 呼び出し元のルーチンでは引数に REAL を使用しているが、呼び出し先のルーチンでは INTEGER が使われている。
- 配列インデックスの計算が間違っている。
- 呼び出し元ルーチンの引数が足りない。
- ポインタを定義しないで使用している。

dbx により問題を見つける方法

問題のあるソース行を見つけるには、dbx を使用してセグメント例外が発生したソースコード行を検出します。

プログラムを使ってセグメント例外を生成します。

```
demo% cat WhereSEGV.f
    INTEGER a(5)
    j = 2000000
    DO 9 i = 1,5
    a(j) = (i * 10)
9   CONTINUE
    PRINT *, a
    END
demo%
```

dbx を使用してセグメント例外が発生した行番号を検出します。

```
demo% f95 -g -silent WhereSEGV.f
demo% a.out
セグメント例外
demo% dbx a.out
a.out の記号情報の読み込み中
シグナル SEGV でプログラムが停止しました (セグメント侵害)
(dbx) run
実行中:
シグナル SEGV (障害アドレスにマッピングがありません)
    ファイル "WhereSEGV.f" の行番号 4 の MAIN で
    4                    a(j) = (i * 10)
(dbx)
```

例外の検出

プログラムが例外を受け取る原因は数多く考えられます。問題を見つける方法の 1 つとして、ソースプログラムで例外が発生した行番号を検出して調べる方法があります。

-ftrap=common によってコンパイルすると、すべての例外に対してトラップが強制的に行われます。

例外が発生した箇所を検索します。

```
demo% cat wh.f
                call joe(r, s)
                print *, r/s
                end
                subroutine joe(r,s)
                r = 12.
                s = 0.
                return
                end
demo% f95 -g -o wh -ftrap=common wh.f
demo% dbx wh
wh の記号情報を読み込み中
(dbx) catch FPE
(dbx) run
実行中:
(プロセス ID 17970)
ファイル "wh.f" の行番号 2 の MAIN にシグナル FPE (ゼロで除算した浮動小
数点)
   2                    print *, r/s
(dbx)
```

呼び出しのトレース

プログラムがコアダンプで終了したため、終了するまでの呼び出しシーケンスが必要な場合があるとします。このシーケンスをスタックトレースといいます。

where コマンドは、プログラムフローの実行が停止した位置、およびどのようにその位置に達したかを表示します。これを呼び出し先ルーチンのスタックトレースといいます。

ShowTrace.f は、呼び出しシーケンスでコアダンプを数レベル深くする、つまりスタックトレースを示すために考えられたプログラムです。

実行が停止した時点から呼び出しシーケンスを表示します。

順序が逆になっていることに注意してください。

MAIN が calc を呼び出し、calc が calcb を呼び出しています。

23 行目で実行が停止しました。

calcb が calc の 9 行目で呼び出されました。
calc が MAIN の 3 行目で呼び出さ れました。

```
demo% f77 -silent -g ShowTrace.f
demo% a.out
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation Fault (core dumped)
demo% dbx a.out
a.out のシンボル情報を読み込んでいます。
...
(dbx) run
実行中: a.out
(process id 1089)
シグナル SEGV (フォルトのアドレスにマッピングしていません) 関数
calcb 行番号 23  ファイル "ShowTrace.f"
   23                   v(j) = (i * 10)
(dbx) where -V
=>[1] calcb(v = ARRAY , m = 2) 行番号 23 ファイル
"ShowTrace.f"
  [2] calc(a = ARRAY , m = 2, d = 0) 行番号 9 ファイル
"ShowTrace.f"
  [3] MAIN() 行番号 3 ファイル "ShowTrace.f"
(dbx)
```

配列の操作

dbx が配列を認識し、配列を出力します。

```
demo% dbx a.out
a.out の読み込み中
ld.so.1 の読み込み中
 ......
(dbx) list
    2               DO 90 I = 1,4
    3                       DO 20 J = 1,4
    4                               IARR(I,J) = (I*10) + J
    5       20              CONTINUE
    6       90      CONTINUE
    7               END
(dbx) stop at 7
(2) stop at "Arraysdbx.f":7
(dbx) run
実行中: a.out
(プロセス id 23495)
mb.so.1 の読み込み中
wcwidth.so.1 の読み込み中
MAIN で停止しました 行番号 7  ファイル "Arraysdbx.f"
    7               END
(dbx) print IARR
iarr =
    (1,1)       11
    (2,1)       21
    (3,1)       31
    (4,1)       41
    (1,2)       12
    (2,2)       22
    (3,2)       32
    (4,2)       42
    (1,3)       13
    (2,3)       23
    (3,3)       33
    (4,3)       43
    (1,4)       14
    (2,4)       24
    (3,4)       34
    (4,4)       44
```

```
続き

(dbx) print IARR(2,3)
iarr(2, 3) = 23
(dbx) quit
```

Fortran の配列のスライスについては、103 ページの「Fortran のための配列断面化構文」を参照してください。

Fortran 95 割り当て可能配列

次の例は、dbx で割り当て済み配列を処理する方法を示しています。

Alloc.f95

```
demo% f95 -g Alloc.f95
demo% dbx a.out
(dbx) list 1,99
    1   PROGRAM TestAllocate
    2   INTEGER n, status
    3   INTEGER, ALLOCATABLE :: buffer(:)
    4           PRINT *, 'Size?'
    5           READ *, n
    6           ALLOCATE( buffer(n), STAT=status )
    7          IF ( status /= 0 ) STOP 'cannot allocate buffer'
    8           buffer(n) = n
    9           PRINT *, buffer(n)
   10           DEALLOCATE( buffer, STAT=status)
   11   END
```

```
(dbx) stop at 6
(2) "alloc.f95":6で停止
(dbx) stop at 9
(3) "alloc.f95":9で停止
(dbx) run
実行中: a.out
(プロセス id 10749)
 サイズは ?
1000
MAIN で停止しました 行番号 6  ファイル "alloc.f95"
    6           ALLOCATE (buffer(n), STAT=status)
(dbx) whatis buffer
INTEGER*4 , allocatable::buffer(:)
(dbx) next
MAIN で停止しました 行番号 7  ファイル "alloc.f95"
    7           IF (status /= 0) STOP 'cannot allocate buffer'
(dbx) whatis buffer
INTEGER*4  buffer(1:1000)
(dbx) cont
MAIN で停止しました 行番号 9  ファイル "alloc.f95"
    9           PRINT *,buffer(n)
(dbx) print n
n = 1000
(dbx) print buffer(n)
buffer(n) = 1000
```

行番号 6 に未知のサイズ

行番号 9 に既知のサイズ

バッファ (1000) に 1000 を格納

```
(dbx) stop at 6
(2) "alloc.f95":6で停止
(dbx) stop at 9
(3) "alloc.f95":9で停止
(dbx) run
実行中: a.out
(プロセス id 10749)
 サイズは ?
1000
MAIN で停止しました 行番号 6  ファイル "alloc.f95"
    6           ALLOCATE (buffer(n), STAT=status)
(dbx) whatis buffer
INTEGER*4 , allocatable::buffer(:)
(dbx) next
MAIN で停止しました 行番号 7  ファイル "alloc.f95"
    7           IF (status /= 0) STOP 'cannot allocate buffer'
(dbx) whatis buffer
INTEGER*4  buffer(1:1000)
(dbx) cont
MAIN で停止しました 行番号 9  ファイル "alloc.f95"
    9           PRINT *,buffer(n)
(dbx) print n
n = 1000
(dbx) print buffer(n)
buffer(n) = 1000
```

行番号 6 に未知のサイズ

行番号 9 に既知のサイズ

バッファ (1000) に 1000 を格納

組み込み関数

dbx は、Fortran の組み込み関数（SPARC™ プラットフォームのみ）を認識します。

dbx での組み込み関数を示します。

```
demo% cat ShowIntrinsic.f
    INTEGER i
    i = -2
    END
(dbx) stop in MAIN
(2) stop in MAIN
(dbx) run
実行中: shi
(プロセス id 18019)
MAIN で停止しました 行番号 2  ファイル "shi.f"
    2         i = -2
(dbx) whatis abs
Generic intrinsic function: "abs"
(dbx) print i
i = 0
(dbx) step
MAIN で停止しました 行番号 3  ファイル "Shi.f"
    3             end
(dbx) print i
i = -2
(dbx) print abs(1)
abs(i) = 2
(dbx)
```

複合式

dbx は、Fortran 複合式も認識します。

dbx での複合式を示します。

```
demo% cat ShowComplex.f
      COMPLEX z
      z = (2.0, 3.0)
      END
demo% f95 -g -silent ShowComplex.f
demo% dbx a.out
a.out の読み込み中
ld.so.1 の読み込み中
  ......
(dbx) stop in MAIN
(2) stop in MAIN
(dbx) run
実行中: a.out
(プロセス id 23567)
mb.so.1 の読み込み中
wcwidth.so.1 の読み込み中
MAIN で停止しました 行番号 2  ファイル "ShowComplex.f"
    2         z = (2.0, 3.0)
(dbx) whatis z
complex*8  z
(dbx) print z
z = (0.0,0.0)
(dbx) next
MAIN で停止しました 行番号 3  ファイル "ShowComplex.f"
    3         END
(dbx) print z
z = (2.0,3.0)
(dbx) print z+(1.0, 1.0)
z+(1,1) = (3.0,4.0)
(dbx) quit
```

論理演算子

dbx は、Fortran の論理演算子を配置し、出力することができます。

dbx での論理演算子を示します。

```
demo% cat ShowLogical1.f
      LOGICAL a, b, y, z
      a = .true.
      b = .false.
      y = .true.
      z = .false.
      END
demo% f95 -g ShowLogical1.f
demo% dbx a.out
(dbx) list 1,9
    1         LOGICAL a, b, y, z
    2         a = .true.
    3         b = .false.
    4         y = .true.
    5         z = .false.
    6         END
(dbx) stop at 5
(2) stop at "ShowLogical.f":5
(dbx) run
実行中: a.out
(プロセス ID 15394)
MAIN で停止しました 行番号 5  ファイル "ShowLogical.f"
    5         z = .false.
(dbx) whatis y
logical*4  y
(dbx) print a .or. y
a.OR.y = true
(dbx) assign z = a .or. y
(dbx) print z
z = false
(dbx) quit
demo%
```

Fortran 95 構造型の表示

構造体、Fortran 95 構造型を dbx で表示できます。

```
demo% f95 -g DebStruct.f95
demo% dbx a.out
(dbx) list 1,99
    1   Program Struct  ! Debug a Structure
    2       TYPE product
    3           INTEGER         id
    4           CHARACTER*16    name
    5           CHARACTER*8     model
    6           REAL            cost
    7           REAL            price
    8       END TYPE product
    9
   10       TYPE(product) :: prod1
   11
   12       prod1%id = 82
   13       prod1%name = "Coffe  Cup"
   14       prod1%model = "XL"
   15       prod1%cost = 24.0
   16       prod1%price = 104.0
   17       WRITE (*, *) prod1%name
   18   END
(dbx) stop at 17
(2) stop at "DebStruct.f95":17
(dbx) run
```

```
実行中: a.out
(プロセス id 23677)
MAIN で停止しました 行番号 17  ファイル "DebStruct.f95"
   17       WRITE (*, *) prod1%name
(dbx) whatis prod1
product  prod1
(dbx) whatis -t product
type product
    INTEGER*4 id
    character*16 name
    character*8 model
    REAL*4 cost
    REAL*4 price
end type product
(dbx) n
 Coffe  Cup
MAIN で停止しました 行番号 18  ファイル "DebStruct.f95"
   18   END
(dbx) print prod1
prod1 = (
    id    = 82
    name  = "Coffe  Cup      "
    model = "XL       "
    cost  = 24.0
    price = 104.0
)
```

Fortran 95 構造型へのポインタ

構造体、Fortran 95 構造型およびポインタを dbx で表示できます。

DebStruc.f95

```
demo% f95 -o debstr -g DebStruct.f95
demo% dbx debstr
debstr の読み込み中
ld.so.1 の読み込み中
 ......
(dbx) list 1,$
    1 Program DebStruPtr  ! Debug a Structure & pointer
    2       TYPE product
    3           INTEGER         id
    4           CHARACTER*16    name
    5           CHARACTER*8     model
    6           REAL            cost
    7           REAL            price
    8       END TYPE product
    9
   10       TYPE(product), TARGET :: prod1, prod2
   11       TYPE(product), POINTER :: curr, prior
   12
   13       curr => prod2
```

構造型を宣言します。

prod1 および prod2 ターゲットを宣言します。(#10)

curr および prior ポインタを宣言します。(#11)

curr が prod2 を指すようにします。(#13)

prior が prod1 を指すようにします。(#14)

prior を初期化します。(#15)

curr を prior に設定します。(#20)

curr および prior から名前を出力します。(#21)

```
   14       prior => prod1
   15       prior%id = 82
   16       prior%name = "Coffe  Cup"
   17       prior%model = "XL"
   18       prior%cost = 24.0
   19       prior%price = 104.0
   20       curr = prior
   21       WRITE (*, *) curr%name, " ", prior%name
   22   END PROGRAM DebStruPtr
   23
(dbx) stop at 21
(2) stop at "DebStruct.f90":21
(dbx) run
実行中: debstr
(dbx) run
実行中: debstr
(プロセス id 23726)
MAIN で停止しました 行番号 21  ファイル "DebStruct.f95"
   21       WRITE (*, *) curr%name, " ", prior%name
(dbx) print prod1
prod1 = (
    id    = 82
    name  = "Coffe  Cup      "
    model = "XL      "
    cost  = 24.0
    price = 104.0
)
```

上記において dbx は、構造型のすべての要素を表示します。

構造体を使用して、Fortran 95 構造型の項目について照会できます。

変数について尋ねます。

型 (-t) について尋ねます。

```
(dbx) whatis prod1
product  prod1
(dbx) whatis -t product
type product
    INTEGER*4 id
    character*16 name
    character*8 model
    REAL*4 cost
    REAL*4 price
end type product
```

ポインタを出力するには、次のようにします。

dbx は、アドレスであるポインタの内容を表示します。このアドレスは、実行のたびに異なる場合があります。

```
(dbx) print prior
prior = (
    id    = 82
    name  = "Coffe  Cup      "
    model = "XL       "
    cost  = 24.0
    price = 104.0
)
```

第17章

機械命令レベルでのデバッグ

この章は、イベント管理コマンドやプロセス制御コマンドを機械命令レベルで使用する方法と、特定のアドレスにおけるメモリーの内容を表示する方法、対応する機械命令とともにソース行を表示する方法を説明します。コマンド next、step、stop、trace のそれぞれに、対応する機械命令レベルのコマンド nexti、stepi、stopi、tracei が用意されています。regs コマンドは、機械語レジスタを出力するために使用できます。また、print コマンドは、個々のレジスタを出力するために使用できます。

この章は次の各節から構成されています。

- メモリーの内容を調べる
- 機械命令レベルでのステップ実行とトレース
- 機械命令レベルでブレークポイントを設定する
- adb コマンドの使用
- regs コマンドの使用

メモリーの内容を調べる

アドレスと examine または x コマンドを使用して、メモリーロケーションの内容を調べたり、各アドレスでアセンブリ言語命令を出力したりすることができます。アセンブリ言語の**デバッガ**である adb(1) から派生したコマンドを使用して、以下の項目について問い合わせることができます。

- アドレス — "=" (等号) を使用。
- あるアドレスに格納されている内容 — "/" (スラッシュ) を使用。

dis、listi コマンドを使用して、アセンブリ命令とメモリーの内容を調べることができます。(231 ページの「dis コマンドの使用」」と232 ページの「listi コマンドの使用」参照)。

examine または x コマンドの使用

examine コマンドまたはその別名 x を使用すると、メモリーの内容やアドレスを表示することができます。

あるメモリーの内容を表示するには、書式 *fmt* の *count* 項目の *address* で表される次の構文を使用します。デフォルトの *addr* は、前に表示された最後のアドレスの次のアドレスになります。デフォルト *count* は 1 です。デフォルト *fmt* は、前の examine または x コマンドで使用されたものと同じです。

examine コマンドの構文は次のとおりです。

```
examine [address] [/ [count] [format]]
```

address1 から *address2* までのメモリー内容を書式 *fmt* で表示するには、次のように入力します。

```
examine address1, address2 [/ [format]]
```

アドレスの内容ではなくアドレスを指定の書式で表示するには、次のように入力します。

```
examine address = [format]
```

examine によって最後に表示されたアドレスの次のアドレスに格納された値を出力するには、次のように入力します。

```
examine +/ i
```

式の値を出力するには、式をアドレスとして入力します。

```
examine address=format
examine address=
```

アドレス (address)

address はアドレスの絶対値、またはアドレスとして使用できる任意の式です。+ (プラス記号) はデフォルトのアドレスの次のアドレスを表します。

たとえば、次のアドレスは有効です。

`0xff99`	絶対アドレス
`main`	関数のアドレス
`main+20`	関数アドレス + オフセット
`&errno`	変数のアドレス
`str`	文字列を指すポインタ変数

メモリーを表示するためのアドレス表現は、名前の前にアンパサンド `&` をつけて指定します。関数名はアンパサンドなしで使用できます。`&main` は `main` と同じです。レジスタは、名前の前にドル記号 `$` を付けることによって表します。

書式 (format)

format は、`dbx` がアドレスの問い合わせ結果を表示するときの書式です。生成される出力は、現在の表示書式 *format* によって異なります。表示書式を変更する場合は、異なる *format* コードを使用してください。

各 `dbx` セッションの初めに設定されるデフォルトの書式は `x` です。このとき、16 進表記のアドレスと値が 1 ワード (32 ビット) で表示されます。次の表は、表示書式の一覧です。

`i`	アセンブラ命令として表示
`d`	10 進表記の16 ビット (2 バイト) で表示
`D`	10 進表記の32 ビット (4 バイト) で表示

`o`	8 進表記の16 ビット (2 バイト) で表示
`O`	8 進表記の32 ビット (4 バイト) で表示
`x`	16 進表記の16 ビット (2 バイト) で表示
`X`	16 進表記の32 ビット (4 バイト) で表示 (デフォルトの書式)
`b`	8 進表記のバイトで表示
`c`	1 バイトの文字で表示
`w`	ワイド文字で表示
`s`	NULL バイトで終わる文字列で表示
`W`	ワイド文字列で表示
`f`	単精度浮動小数点数として表示
`F,g`	倍精度浮動小数点数として表示
`E`	拡張精度浮動小数点数として表示
`ld, lD`	10 進数として 32 ビット (4 バイト) で表示 (`D` と同じ)
`lo, lO`	8 進数として 32 ビット (4 バイト) で表示 (`O` と同じ)
`lx, LX`	16 進数として 32 ビット (4 バイト) で表示 (`X` と同じ)
`Ld, LD`	10 進数として 64 ビット (8 バイト) で表示
`Lo, LO`	8 進数として 64 ビット (8 バイト) で表示
`Lx, LX`	16 進数として 64 ビット (8 バイト) で表示

繰り返し (count)

count は、10 進法での反復カウントを示します。増分サイズは、メモリーの表示書式によって異なります。

アドレスの使用例

次の例は、*count* および *format* の各オプションを付けてアドレスを使用して、現在の停止点から始まる 5 つの連続する分解された命令を表示する方法を示しています。

SPARC の場合は次のとおりです。

```
(dbx) stepi
stopped in main at 0x108bc
0x000108bc: main+0x000c: st    %l0, [%fp - 0x14]
(dbx) x 0x108bc/5i
0x000108bc: main+0x000c: st    %l0, [%fp - 0x14]
0x000108c0: main+0x0010: mov   0x1,%l0
0x000108c4: main+0x0014: or    %l0,%g0, %o0
0x000108c8: main+0x0018: call  0x00020b90 [unresolved PLT 8:
malloc]
0x000108cc: main+0x001c: nop
```

x86 の場合は次のとおりです。

```
(dbx) x &main/5i
0x08048988: main       :  pushl  %ebp
0x08048989: main+0x0001:  movl   %esp,%ebp
0x0804898b: main+0x0003:  subl   $0x28,%esp
0x0804898e: main+0x0006:  movl   0x8048ac0,%eax
0x08048993: main+0x000b:  movl   %eax,-8(%ebp)
```

dis コマンドの使用

このコマンドは、表示書式を i として指定した examine コマンドと同じです。

dis コマンドの構文は次のようになります。

```
dis [address] [address1, address2] [/count]
```

dis コマンドの動作は次のとおりです。

- 引数なしで実行すると、+ で始まる 10 の命令を表示します。
- 引数 *address* だけを指定して実行すると、*address* で始まる 10 の命令を逆アセンブルします。
- 引数 *address 1* と *address 2* を指定して実行すると、*address 1* から *address 2* までの命令を逆アセンブルします。
- *count* だけを指定して実行すると、+ で始まる count 命令を表示します。

`listi` コマンドの使用

対応するアセンブリ命令とともにソース行を表示するには `listi` コマンドを使用します。これは `list -i` と同じです。50 ページの「ソースリストの出力」の `list -i` についての説明を参照してください。

SPARC の場合は次のとおりです。

```
(dbx) listi 13, 14
   13       i = atoi(argv[1]);
0x0001083c: main+0x0014:  ld      [%fp + 0x48], %l0
0x00010840: main+0x0018:  add     %l0, 0x4, %l0
0x00010844: main+0x001c:  ld      [%l0], %l0
0x00010848: main+0x0020:  or      %l0, %g0, %o0
0x0001084c: main+0x0024:  call    0x000209e8 [unresolved PLT 7: atoi]
0x00010850: main+0x0028:  nop
0x00010854: main+0x002c:  or      %o0, %g0, %l0
0x00010858: main+0x0030:  st      %l0, [%fp - 0x8]
   14       j = foo(i);
0x0001085c: main+0x0034:  ld      [%fp - 0x8], %l0
0x00010860: main+0x0038:  or      %l0, %g0, %o0
0x00010864: main+0x003c:  call    foo
0x00010868: main+0x0040:  nop
0x0001086c: main+0x0044:  or      %o0, %g0, %l0
0x00010870: main+0x0048:  st      %l0, [%fp - 0xc]
```

x86 の場合は次のとおりです。

```
(dbx) listi 13, 14
   13       i = atoi(argv[1]);
0x080488fd: main+0x000d:  movl   12(%ebp),%eax
0x08048900: main+0x0010:  movl   4(%eax),%eax
0x08048903: main+0x0013:  pushl  %eax
0x08048904: main+0x0014:  call   atoi <0x8048798>
0x08048909: main+0x0019:  addl   $4,%esp
0x0804890c: main+0x001c:  movl   %eax,-8(%ebp)
   14       j = foo(i);
0x0804890f: main+0x001f:  movl   -8(%ebp),%eax
0x08048912: main+0x0022:  pushl  %eax
0x08048913: main+0x0023:  call   foo <0x80488c0>
0x08048918: main+0x0028:  addl   $4,%esp
0x0804891b: main+0x002b:  movl   %eax,-12(%ebp)
```

機械命令レベルでのステップ実行とトレース

機械命令レベルの各コマンドは、対応するソースレベルのコマンドと同じように動作します。ただし、動作の単位はソース行ではなく、単一の命令です。

機械命令レベルでステップ実行する

ある機械命令から次の機械命令に 1 つだけステップ実行するには、`nexti` コマンドまたは `stepi` コマンドを使用します。

`nexti` コマンドと `stepi` コマンドは、それぞれに対応するソースコードレベルのコマンドと同じように動作します。すなわち、`nexti` コマンドは '*over*' 関数を実行し、`stepi` は次の命令が呼び出した関数をステップ実行します (呼び出された関数の最初の命令で停止します)。コマンドの書式も同じです。詳細については、321 ページの「next コマンド」と337 ページの「step コマンド」を参照してください。

`nexti` と `stepi` の出力は、対応するソースレベルのコマンドの場合と次の 2 つの違いがあります。

- ソースコードの行番号の代わりに、プログラムが停止した**アドレス**が出力に含まれる。
- ソースコード行の代わりに、デフォルトの出力に**逆アセンブルされた命令**が示される。

例を示します。

```
(dbx) func
hand::ungrasp
(dbx) nexti
ungrasp +0x18:  call support
(dbx)
```

詳細については、321 ページの「next コマンド」と339 ページの「stepi コマンド」を参照してください。

機械命令レベルでトレースする

機械命令レベルでのトレースは、ソースコードレベルでのトレースと同じように行われます。ただし、tracei コマンドを使用する場合は例外で、実行中のアドレスまたはトレース対象の変数の値がチェックされた場合にだけ、単一の命令が実行されます。tracei コマンドは、stepi のような動作を自動的に行います。すなわち、プログラムは 1 度に 1 つの命令だけ進み、関数呼び出しに入ります。

tracei コマンドを使用すると、各命令が実行され、アドレスの実行またはトレース中の変数または式の値を dbx が調べている間、プログラムは一瞬停止します。このように tracei コマンドの場合、実行速度がかなり低下します。

トレースとそのイベント使用および修飾子については、82 ページの「トレースの実行」と、346 ページの「tracei コマンド」を参照してください。

構文は次のとおりです。

```
tracei event-specification [modifier]
```

共通に使用される tracei 書式は次のとおりです。

`tracei step`	各命令をトレース
`tracei next`	各命令をトレースするが、呼び出しを飛び越します。
`tracei at` *address*	指定のコードアドレスをトレース

詳細については、346 ページの「tracei コマンド」を参照してください。

SPARC の場合は次のとおりです。

```
(dbx) tracei next -in main
(dbx) cont
0x00010814: main+0x0004:  clr     %l0
0x00010818: main+0x0008:  st      %l0, [%fp - 0x8]
0x0001081c: main+0x000c:  call    foo
0x00010820: main+0x0010:  nop
0x00010824: main+0x0014:  clr     %l0
....
....
(dbx) (dbx) tracei step -in foo -if glob == 0
(dbx) cont
0x000107dc: foo+0x0004:  mov     0x2, %l1
0x000107e0: foo+0x0008:  sethi   %hi(0x20800), %l0
0x000107e4: foo+0x000c:  or      %l0, 0x1f4, %l0     ! glob
0x000107e8: foo+0x0010:  st      %l1, [%l0]
0x000107ec: foo+0x0014:  ba      foo+0x1c
....
....
```

機械命令レベルでブレークポイントを設定する

機械命令レベルでブレークポイントを設定するには、stopi コマンドを使用します。stopi は次の構文を使用して *event_specification* を受け入れます。

```
stopi event-specification [modifier]
```

一般的に使用される stopi コマンドの書式は次のとおりです。

```
stopi [at address] [if cond]
stopi in function [-if cond]
```

詳細については、339 ページの「stepi コマンド」を参照してください。

あるアドレスにブレークポイントを設定する

特定のアドレスにブレークポイントを設定するには、コマンドペインで次のように入力します。

```
(dbx) stopi at address
```

例 :

```
(dbx) nexti
hand::ungrasp で停止しました 0x12638 で
(dbx) stopi at &hand::ungrasp
(3) stopi at &hand::ungrasp
(dbx)
```

adb コマンドの使用

adb(1) 構文で adb コマンドを入力できます。また、すべてのコマンドを adb 構文として解釈する adb モードに変更することもできます。ほとんどの adb コマンドがサポートされています。

詳細については、281 ページの「adbコマンド」を参照してください。

regs コマンドの使用

regs コマンドを使用すると、すべてのレジスタの値を表示することができます。

次に、regs コマンドの構文を示します。

```
regs [-f] [-F]
```

-f には浮動小数点レジスタ (単精度)、-F には浮動小数点レジスタ (倍精度) が含まれます。これは、SPARC だけのオプションです。

詳細については、329 ページの「regs コマンド」を参照してください。

SPARC の場合は次のようになります。

```
dbx[13] regs -F
現スレッド: t@1
現フレーム: [1]
g0-g3    0x00000000 0x0011d000 0x00000000 0x00000000
g4-g7    0x00000000 0x00000000 0x00000000 0x00020c38
o0-o3    0x00000003 0x00000014 0xef7562b4 0xefffff420
o4-o7    0xef752f80 0x00000003 0xefffff3d8 0x000109b8
l0-l3    0x00000014 0x0000000a 0x0000000a 0x00010a88
l4-l7    0xefffff438 0x00000001 0x00000007 0xef74df54
i0-i3    0x00000001 0xefffff4a4 0xefffff4ac 0x00020c00
i4-i7    0x00000001 0x00000000 0xefffff440 0x000108c4
y        0x00000000
psr      0x40400086
pc       0x000109c0:main+0x4    mov     0x5, %l0
npc      0x000109c4:main+0x8    st      %l0, [%fp - 0x8]
f0f1     +0.00000000000000e+00
f2f3     +0.00000000000000e+00
f4f5     +0.00000000000000e+00
f6f7     +0.00000000000000e+00
...
```

プラットフォーム固有のレジスタ

次の表は、式で使用できる SPARC および x86 のプラットフォームに固有のレジスタ名を示しています。

SPARC レジスタ情報

SPARC システムのレジスタ情報は次のとおりです。

レジスタ	説明
`$g0-$g7`	「大域」レジスタ
`$o0-$o7`	「出力」レジスタ
`$l0-$17`	「局所」レジスタ
`$i0-$i7`	「入力」レジスタ
`$fp`	フレームポインタ (レジスタ $i6 と等価)
`$sp`	スタックポインタ (レジスタ $o6 と等価)

レジスタ	説明
$y	Y レジスタ
$psr	プロセッサ状態レジスタ
$wim	ウィンドウ無効マスクレジスタ
$tbr	トラップベースレジスタ
$pc	プログラムカウンタ
$npc	次のプログラムカウンタ
$f0-$f31	FPU “f” レジスタ
$fsr	FPU 状態レジスタ
$fq	FPU キュー

$f0f1 $f2f3...$f30f31 のような浮動小数点レジスタのペアは、C の “doule” 型とみなされます。通常、$fN レジスタは C の “float” 型とみなされます。これらのペアは、$d0...$d30 とも表します。

次の追加レジスタは、SPARC V9 および V8+ ハードウェアで使用できます。

```
$g0g1 through $g6g7
$o0o1 through $o6o7
$xfsr $tstate $gsr
$f32f33 $f34f35 through $f62f63 ($d32 ... $$d62)
```

SPARC のレジスタとアドレッシングの詳細については、『SPARC アーキテクチャマニュアル バージョン 8』(トッパン刊) および『Sun-4 Assembly Language Reference Manual』を参照してください。

Intel レジスタ情報

Intel システムのレジスタ情報は次のとおりです。

レジスタ	説明
$gs	代替データセグメントレジスタ
$fs	代替データセグメントレジスタ
$es	代替データセグメントレジスタ

レジスタ	説明
$ds	データセグメントレジスタ
$edi	デスティネーションインデックスレジスタ
$esi	ソースインデックスレジスタ
$ebp	フレームポインタ
$esp	スタックポインタ
$ebx	汎用レジスタ
$edx	汎用レジスタ
$ecx	汎用レジスタ
$eax	汎用レジスタ
$trapno	例外ベクトル番号
$err	例外を示すエラーコード
$eip	命令ポインタ
$cs	コードセグメントレジスタ
$eflags	フラグ
$es	代替データセグメントレジスタ
$uesp	ユーザースタックポインタ
$ss	スタックセグメントレジスタ

一般的に使用されるレジスタには、マシンに依存しない名前が別名として指定されます。

レジスタ	説明
$sp	スタックポインタ ($uesp と同じ)。
$pc	プログラムカウンタ ($eip と同じ)。
$fp	フレームポインタ ($ebp と同じ)。

80386 用の下位 16 ビットのレジスタは次のとおりです。

レジスタ	説明
$ax	汎用レジスタ
$cx	汎用レジスタ
$dx	汎用レジスタ
$bx	汎用レジスタ
$si	ソースインデックスレジスタ
$di	デスティネーションインデックスレジスタ
$ip	命令ポインタ (下位 16 ビット)
$flags	フラグ (下位 16 ビット)

上記のうち最初の 4 つの 80386 用 16 ビットレジスタは、8 ビットずつに分割できます。

レジスタ	説明	
$al	レジスタの下位 (右) 部分	$ax
$ah	レジスタの上位 (左) 部分	$ax
$cl	レジスタの下位 (右) 部分	$cx
$ch	レジスタの上位 (左) 部分	$cx
$dl	レジスタの下位 (右) 部分	$dx
$dh	レジスタの上位 (左) 部分	$dx
$bl	レジスタの下位 (右) 部分	$bx
$bh	レジスタの上位 (左) 部分	$bx

80387 用レジスタは次のとおりです。

レジスタ	説明
$fctrl	コントロールレジスタ
$fstat	状態レジスタ
$ftag	タグレジスタ

レジスタ	説明
`$fip`	命令ポインタオフセット
`$fcs`	コードセグメントセレクタ
`$fopoff`	オペランドポインタオフセット
`$fopsel`	オペランドポインタセレクタ
`$st0 - $st7`	データレジスタ

第18章

dbx の Korn シェル機能

dbx コマンド言語は Korn シェル (ksh 88) の構文にもとづいており、入出力リダイレクション、ループ、組み込み算術演算、ヒストリ、コマンド行編集 (コマンド行モードのみで、dbx からは利用不可能) といった機能を持っています。

dbx 初期化ファイルが起動時に見つからない場合、dbx はksh モードを想定します。

この章は次の各節から構成されています。

- 実装されていない ksh-88 の機能
- ksh-88 から拡張された機能
- 名前が変更されたコマンド

実装されていない ksh-88 の機能

ksh-88 の次の機能は dbx では実装されていません。

- `set -A`*name* による配列 *name* への値の代入
- `set -o` の以下のオプション : `allexport bgnice gmacs markdirs noclobber nolog privileged protected viraw`
- `typeset` の以下の属性 : `-l -u -L -R -H`
- バッククォート (`` `...` ``) によるコマンドの置き換え (代わりに `$(...)` を使用)
- 複合コマンド `[[`<式>`]]` による式の評価
- `@(`<パターン>`[|`<パターン>`]...)` による拡張パターン照合
- コプロセス (バックグラウンドで動作し、プログラム交信するコマンドまたはパイプライン)

ksh-88 から拡張された機能

dbx では、次の機能が追加されました。

- 言語式 `$[ p -> flags ]`
- `typeset -q` (ユーザー定義関数のための特殊な引用を可能にする)
- csh のような `history` および `alias` の引数
- `set +o path` (パス検索を無効にする)
- `0xabcd` (8 進数および 16 進数を示す C の構文)
- `bind` による `emacs` モードバインディングの変更
- `set -o hashall`
- `set -o ignore suspend`
- `print -e` および `read -e` (`-r` (raw) の逆の働きをする)
- dbx コマンドが組み込まれている

名前が変更されたコマンド

ksh コマンドとの衝突を避けるために dbx コマンドの一部の名前が変更されています。

- dbx の `print` コマンドはそのままで、ksh の `print` コマンドが `kprint` という名前に変更されました。
- ksh の `kill` コマンドが dbx の `kill` コマンドにマージされました。
- `alias` コマンドは、dbx 互換モードでないかぎり ksh のエイリアスとして機能します。
- *addr/fmt* は現在 `examine` *addr/fmt* です。
- */pattern* は現在 `search` *pattern* です。
- ?*pattern* は現在 `bsearch` *pattern* です。

第19章

共有ライブラリのデバッグ

dbx は動的にリンクされた共有ライブラリのデバッグを完全にサポートしています。ただし、これらのライブラリが -g オプションを使用してインストールされていることが前提になります。

この章は、次の各節から構成されています。

- 動的リンカー
- 読み込み済みの共有オブジェクトのデバッグサポート
- 修正と継続
- 動的にリンクしたライブラリにブレークポイントを設定する

動的リンカー

動的リンカーは “rtld”、”実行時 ld”、または “ld.so” とも呼ばれ、実行中のアプリケーションに共有オブジェクト (ロードオブジェクト) を組み込むように準備します。rtld が稼働状態になるのは主に次の 2 つの場合です。

- プログラムの起動時 – プログラムの起動時、rtld はまずリンク時に指定されたすべての共有オブジェクトを動的に読み込みます。ldd (1) を使用すれば、プログラムによって読み込まれる共有オブジェクトを調べることができます。これらは「あらかじめ読み込まれた」共有オブジェクトで、一般に libc.so、libC.so、libX.so などがあります。

- アプリケーションから呼び出しがあった場合 – アプリケーションでは、関数呼び出し dlopen(3) と dlclose(3) を使用して共有オブジェクトやプレーンな実行可能ファイルの読み込みや読み込みの取り消しを行います。共有オブジェクト (.so) や通常の実行可能ファイル (a.out) のことを、dbx では「ロードオブジェクト」といいます。

dbx における *loadobject* は共有オブジェクト (.so) または実行可能ファイル (a.out) を表します。

リンクマップ

動的リンカーは、読み込んだすべてのオブジェクトのリストを、*link map* というリストで管理します。このリストは、デバッグするプログラムのメモリーに保存され、librctld_db.so で間接的にアクセスできます。これはデバッガ用に用意された特別なシステムライブラリです。

起動手順と .init セクション

.init セクションは、共有オブジェクトの読み込み時に実行される、その共有オブジェクトのコードの一部分です。たとえば、.init セクションは、C++ 実行時システムがすべての静的初期化関数を呼び出すときに使用します。

動的リンカーは最初にすべての共有オブジェクトにマップインし、それらのオブジェクトをリンクマップに登録します。その後、動的リンカーはリンクマップに含まれる各オブジェクトの .init セクションを順に実行します。syncrtld イベントは、これら 2 つの動作の間に発生します。

プロシージャ・リンケージ・テーブル

PLT は、共有オブジェクトの境界間の呼び出しを容易にするために rtld によって使用される構造体です。たとえば、printf の呼び出しはこの間接テーブルによって行います。その方法の詳細については、SVR4 ABI に関する汎用リファレンスマニュアルおよびプロセッサ固有のリファレンスマニュアルを参照してください。

複数の PLT 間で step コマンドと next コマンドを操作するために、dbx は各ロードオブジェクトの PLT テーブルを追跡する必要があります。テーブル情報は rtld ハンドシェークと同時に入手されます。

読み込み済みの共有オブジェクトのデバッグサポート

読み込み済みの共有オブジェクトにブレークポイントを設定するには、dbx にそのルーチンのアドレスを渡す必要があります。dbx がルーチンのアドレスを受け取るには、共有オブジェクトベースアドレスが必要です。たとえば以下のような簡単な処理でも dbx で行うには特別な配慮が必要です。

```
stop in printf
run
```

新しいプログラムが読み込まれるたびに 、dbx は rtld がリンクマップの作成を完了した場所までプログラムを自動的に実行します。dbx はリンクマップを読み取り、ベースアドレスを格納します。その後、プロセスは終了し、メッセージとプロンプトが表示されます。dbx は、これらの動作中にメッセージを表示しません。

この時点で、ベース読み込みアドレスとともに libc.so のシンボルテーブルを調べることができるため、printf のアドレスがわかります。

rtld によってリンクマップが作成されるのを待機し、リンクマップの先頭にアクセスするまでの dbx の動作を「rtld ハンドシェーク」と呼びます。rtld がリンクマップを作成し、dbx がすべてのシンボルテーブルを読み取ると、イベント syncrtld が発生します。

修正と継続

dlopen() で読み込んだ共有オブジェクトに fix と continue を使用する場合、開き方を変更しないと fix と continue が正しく機能しません。モード RTLD_NOW | RTLD_GLOBAL または RTLD_LAZY | RTLD_GLOBAL を使用します。

動的にリンクしたライブラリにブレークポイントを設定する

dbx は dlopen() または dlclose() の発生を自動的に検出し、読み込まれたオブジェクトの記号テーブルを読み込みます。dlopen() で共有オブジェクトを読み込むと、そのオブジェクトにブレークポイントを設定できます。またプログラムのその他の任意の場所で行う場合と同様にデバッグも可能です。

共有オブジェクトを dlclose() で読み込み解除しても、dbx はそのオブジェクトに設定されていたブレークポイントを記憶しているので、たとえアプリケーションを再実行しても、共有オブジェクトが dlopen() で再び読み込まれれば再びそのブレークポイントを設定しなおします。(dbx のバージョン 5.0 より以前のものは、代わりにブレークポイントに '(defunct)' とマーキングします。これはユーザーが削除して置き換える必要があります。)

ただし、dlopen() で共有オブジェクトが読み込まれるのを待たなくても共有オブジェクトにブレークポイントを設定したり、その関数やソースコードを検索することはできます。デバッグするプログラムが dlopen() で読み込む共有オブジェクトの名前がわかっていれば、dbx をアレンジしてその記号テーブルを、次のコマンドで dbx にあらかじめ読み込んでおくことができます。

```
loadobjects -p /usr/java1.1/lib/libjava_g.so
```

これで、dlopen() で読み込む前でも、この読み込みオブジェクト内でモジュールと関数を検索してその中にブレークポイントを設定できます。読み込みが済んだら、dbx はブレークポイントを自動的に設定します。

動的にリンクしたライブラリにブレークポイントを設定する場合、以下の制約があります。

- dlopen() で読み込んだ「フィルタ」ライブラリには、その中の最初の関数が呼び出されるまでブレークポイントは設定できません。
- dlopen() でライブラリを読み込むと、初期化ルーチン _init() が呼び出されます。このルーチンがライブラリ内の他のルーチンを呼び出すこともあります。この初期化が終了するまで、.dbx は読み込んだライブラリにブレークポイントを設定できません。具体的には、dbx は、dlopen で読み込んだライブラリ内の _init() では停止できません。

付録 A

プログラム状態の変更

ここでは、dbx を使用しないでプログラムを実行する場合と比べながら、dbx で実行する際のプログラムまたはプログラムの動作を変更する dbx の使用法とコマンドについて説明します。プログラムに変更を加えるコマンドがどれかを理解する必要があります。

この付録は、次の各節から構成されています。

- dbx 下でプログラムを実行することの影響
- プログラムの状態を変更するコマンドの使用

dbx のもとでプログラムを実行することの影響

アプリケーションは、dbx のもとで実行される場合、本来と動作が異なることがあります。dbx は被デバッグプログラムに対する影響を最小限に抑えようとはしますが、次の点に注意する必要があります。

- -C オプション付きで起動しないでください。また、RTC は無効にしてください。RTC のライブラリの `librtc.so` をプログラムに読み込むと、プログラムの動作が変わる可能性があります。
- dbx 初期化スクリプトで環境変数が設定されていることを忘れないでください。スタックベースは、dbx のもとで実行する場合、異なるアドレスから始まります。これは、各自の環境と `argv[]` の内容によっても異なり、ローカル変数の割り当てが若干異なります。これらが初期化されていないと、異なる乱数を受け取ります。この問題は実行時検査によって検出できます。

- プログラムは、使用前に malloc() されたメモリーを初期化しません。これは、前述の状態と似ています。この問題は、実行時検査によって検出できます。
- dbx は LWP 作成イベントと dlopen イベントを捕獲しなければならず、これによって、タイミングに左右されやすいマルチスレッドアプリケーションが影響を受ける可能性があります。
- dbx は、シグナルに対するコンテキスト切り替えを実行するため、タイミングに影響を受けるシグナルを多用する場合、動作が異なってしまうおそれがあります。
- プログラムは、mmap() が、マップされたセグメントについて常に同じベースアドレスを返すことを期待します。dbx のもとで実行すると、アドレス空間が混乱して、
  mmap() は、dbx を使用しないでプログラムを実行したときと同じアドレスを返せなくなります。プログラムでこのことが問題になるかどうかを判断するには、mmap() の使用場所をすべて調べて、返される値がハードコードされたアドレスではなく、プログラムによって実際に使用されることを確認してください。
- プログラムがマルチスレッド化されている場合、データの競合が存在するか、またはスレッドスケジュールに依存する可能性があります。dbx のもとで実行すると、スレッドスケジュールが混乱して、プログラムが通常の順序とは異なる順序でスレッドを実行するおそれがあります。このような状態を検出するには、lock_lint を使用してください。

あるいは、adb または truss を使用して実行した場合に同じ問題が起こるか確認してください。

dbx によって強いられる混乱を最小限に抑えるには、アプリケーションが自然な環境で実行されているときに dbx を接続するようにしてください。

プログラムの状態を変更するコマンドの使用

assign コマンド

assign コマンドは、*expresions* の値を *variable* に割り当てます。dbx 内で使用すると *var* の値が永久に変更されます。

```
assign variable = expression
```

pop コマンド

dbx の pop コマンドは、スタックから 1 つまたは複数のフレームをポップ (解放) します。

pop	カレントフレームをポップ
pop *number*	*number* 個のフレームをポップ
pop -f *number*	指定のフレーム数までフレームをポップ

ポップされた呼び出しはすべて、再開時に再び実行されて、プログラムに望ましくない変更が加えられる可能性があります。pop は、ポップされた関数にローカルなオブジェクトのデストラクタも呼び出します。

詳細については、325 ページの「pop コマンド」を参照してください。

call コマンド

call コマンドを dbx で使用すると、ある手続きが呼び出されて、その手続きは指定通りに実行されます。

```
call proc([params])
```

この手続きは、プログラムの一部を変更する可能性があります。dbx は、プログラムソースに呼び出しを組み込んだ場合と同様に、実際に呼び出しを行います。

詳細については、283 ページの「call コマンド」を参照してください。

print コマンド

式の値を印刷するには、次のように入力します。

```
print expression, ...
```

式に関数呼び出しがある場合は、call コマンドと同じ考慮事項が適用されます。C++ では、多重定義演算子による予期しない副作用にも注意する必要があります。

詳細については、326 ページの「print コマンド」を参照してください。

when コマンド

when コマンドの一般的な構文は次のとおりです。

```
when event-specification [modifier] {command ... ;}
```

イベントが発生すると、*commands* が実行されます。

ある行または手続きに到達すると、コマンドが実行されます。どのコマンドを出したかによって、プログラムの状態が変わる可能性があります。

詳細については、354 ページの「when コマンド」を参照してください。

fix コマンド

fix を使用すると、プログラムに対して、実行中の変更を加えることができます。

```
fix
```

これは非常に便利なツールですが、fix は変更されたソースファイルを再コンパイルして、変更された関数をアプリケーションに動的にリンクすることに注意してください。

第 11 章「修正継続機能 (fix と continue)」を参照して、fix と continue の制限事項を必ず確認してください。

詳細については、305 ページの「fix コマンド」を参照してください。

cont at コマンド

この cont at コマンドは、プログラムが実行される順序を変更します。実行を *line* で指定した行で続けられます。プログラムがマルチスレッド化されている場合は ID が必要です。

```
cont at line id
```

これにより、プログラムの結果が変更される可能性があります。

付録 B

イベント管理

イベント管理は、デバッグ中のプログラムで特定のイベントが発生したときに特定のアクションを実行する、dbx の一般的な機能です。dbx を使用すると、イベント発生時に、プロセスの停止、任意のコマンドの発行、または情報を表示することができます。イベントの最も簡単な例はブレークポイントです (第 6 章を参照してください)。その他のイベントの例として、障害、信号、システムコール、`dlopen()` の呼び出し、およびデータの変更などがあります (77 ページの「データ変更ブレークポイントを設定する」 を参照してください)。

この章は、次の各節から構成されています。

- イベントハンドラ
- イベントハンドラの作成
- イベントハンドラを操作するコマンド
- イベントカウンタ
- イベント指定の設定
- イベント指定修飾子
- 解析とあいまいさに関する注意
- 事前定義済み変数
- イベントハンドラの設定例

イベントハンドラ

イベント管理は「**ハンドラ**」の概念に基づくもので、この名前はハードウェアの割り込みハンドラからきたものです。通常、ハンドラは各イベント管理コマンドによって作成されます。これらのコマンドは、**イベント指定**と関連する一連のアクションで構成されます (260 ページの「イベント指定の設定」参照)。イベント指定は、ハンドラを発生させるイベントを指定します。

イベントが発生し、ハンドラが引き起こされると、イベント指定に含まれる任意の修飾子に従って、ハンドラはイベントを評価します (270 ページの「イベント指定のための修飾子」参照)。修飾子によって課された条件にイベントが適合すると、ハンドラの関連アクションが実行されます (つまり、ハンドラが起動します)。

プログラムイベントを dbx アクションに対応付ける例は、特定の行にブレークポイントを設定するものです。

ハンドラを作成する最も一般的な形は、when コマンドを使用するものです。

```
when event-specification {action; ... }
```

この章の例は、when を使用した表現でコマンド (stop、step、ignore など) を記述する方法を示します。これらの例は、when とその配下にある**ハンドラ**機構の柔軟性を示すものですが、常に同じ働きをするとは限りません。

イベントハンドラの作成

when、stop、trace コマンドを使用して、イベントハンドラを作成します (詳細については、354 ページの「when コマンド」、339 ページの「stop コマンド」、および 345 ページの「trace コマンド」を参照してください)。

共通の when 構文は、stop を使用して簡単に表現できます。

```
when event-specification { stop -update; whereami; }
```

event-specification は、イベント管理コマンド `stop`、`when`、`trace` にて使用され、関心のあるイベントを指定します。(260 ページの「イベント指定の設定」参照)。

`trace` コマンドのほとんどは、`when` コマンド、`ksh` 機能、イベント変数を使用して手動で作成することができます。これは、スタイル化されたトレーシング出力を希望する場合、特に有益です。

コマンドが実行される度に、ハンドラ id (*hid*) 番号を返します。事前定義変数 `$newhandlerid` を介してこの番号にアクセスすることができます。

イベントハンドラを操作するコマンド

次のコマンドを使用して、イベントハンドラを操作することができます。各コマンドの詳細については、それぞれの節を参照してください。

- `status` - ハンドラを表示 (336 ページの「status コマンド」」参照)。
- `delete` - 一時ハンドラを含むすべてのハンドラを削除します (300 ページの「delete コマンド」参照)。
- `clear` - ブレークポイントの位置にもとづいてハンドラを削除します (288 ページの「clear コマンド」参照)。
- `handler - enable` - ハンドラを有効にします (309 ページの「handler コマンド」参照)。
- `handler - disable` - ハンドラを無効にします。
- `cancel` - 信号を取り消し、プロセスを継続させます 283 ページの「cancel コマンド」参照)。

イベントカウンタ

イベントハンドラはカウンタを備えており、制限値と実際のカウンタを保持します。イベントが発生するたびにカウンタをインクリメント (1 つ増加) し、その値が制限値に達すると、ハンドラに対応するアクションを起動してカウンタをゼロにリセットします。デフォルトの制限値は 1 です。プロセスが再実行されるたびに、すべてのイベントカウンタがリセットされます。

`stop`、`when`、`trace` コマンドを持つ `-count` 修飾子を使用して、カウント制限を設定することができます (271 ページの「-count n -count infinity」参照)。このほか、`handler` コマンドを使用して、個々のイベントハンドラを操作できます。

```
handler [ -count | -reset ] hid new-count new-count-limit
```

イベント指定の設定

イベント指定子は、`stop`、`when`、`trace` コマンドがイベントタイプやパラメータを表すために使用します。その書式は、イベントタイプを表すキーワードと省略可能なパラメータで構成されます。詳細については、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「イベント指定」を参照してください。

ブレークポイントイベント仕様

次に、ブレークポイントイベントに対するイベント仕様を説明します。ブレークポイントとは、アクションが発生する位置であり、その位置でプログラムは実行を停止します。

`in` *function*

関数が入力され、最初の行が実行される直前。この行は、ローカル変数を初期化する行になることがあります。C++ のコンストラクターの場合、すべてのベースクラスのコンストラクターの実行後に実行されます。`-instr` 修飾子が使用される場合 (271

ページの「-instr」参照) は、関数の最初の命令が実行される直前です。func 仕様は、仮パラメータを含むことができるため、多重定義関数名、またはテンプレートインスタンスの指定に役立ちます。たとえば、次のように使用できます。

```
stop in mumble(int, float, struct Node *)
```

注 – in *function* と -in *function* 修飾子とを混同しないでください。

---

at [*filename:*]*lineno*

指定の行が実行される直前。*filename* を指定した場合は、指定ファイルの指定の行が実行される直前。ファイル名には、ソースファイル名またはオブジェクトファイル名を指定します。引用符は不要ですが、ファイル名に特殊文字が含まれる場合は、必要な場合もあります。指定の行がテンプレートコードに含まれる場合、ブレークポイントは、そのテンプレートのすべてのインスタンス上に置かれます。

infuction *function*

function と名付けられたすべての多重定義関数、およびテンプレートインスタンスのすべてに対し、in *function* と同じ働きをします。

inmember *function*
inmethod *function*

すべてのクラスの *function* と名付けられたメンバー関数に対し、in *function* と同じ働きをします。

inclass *classname*

classname のベースではなく、*classname* のメンバーであるすべてのメンバー関数に対し、in *function* と同じ働きをします。

`inobject` *object-expression*

object-expression に指定されているアドレスのオブジェクトを呼び出したメンバー関数が呼び出されているとき。`stop inobject` *ox* は次のコードとほとんど同じ働きをしますが、`inclass` と異なり、動的な *ox* のベースが含まれます。

データ変更イベント仕様

次に、ウォッチポイントイベントに対するイベント仕様について説明します。詳細については、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「ウォッチポイント仕様」を参照してください。

`access` *mode address-expression* [*, byte-size-expression*]

address-expression で指定されたメモリーがアクセスされたとき。

mode には、メモリーがアクセスされたことを示す、次のいずれか (またはすべて) の文字を指定します。

`r` 指定アドレスのメモリーが読み取られた
`w` メモリーが書き込まれた
`x` メモリーが実行された

さらに *mode* には、次のいずれかの文字も指定することができます。

`a` アクセス後にプロセスを停止する (デフォルト)
`b` アクセス前にプロセスを停止する

いずれの場合も、プログラムカウンタは副作用アクションの前後で違反している命令をポイントします。

address-expression は、その評価によりアドレスを生成できる任意の式です。シンボル式を使用すると、監視される領域のサイズが自動的に推定されます。このサイズは、*byte-size-expression* を指定することにより、上書きされます。シンボルを使用しない、型を持たないアドレス式を使用することもできますが、その場合はサイズを指定する必要があります。たとえば、次のようにします。

```
stop access w 0x5678, sizeof(Complex)
```

access コマンドには、2 つの一致する範囲が重複しない、という制限があります。

注 – access イベント仕様は、modify イベント仕様の代替です。どちらの構文も、Solaris 2.6、Solaris 7、Solaris 8 で動作します。ただし、そのうち Solaris 2.6 を除くオペレーティング環境では、access に modify と同じ制限が課せられ、wa モード以外は使用できません。

---

change *variable*

variable の値は変更されました。change イベントは、次のコードとほとんど同じ働きをします。

```
when step { if [ $last_value !=$[variable]] then
                  stop
            else
                  last_value=$[variable]
          }
```

cond *condition-expression*

condition-expression によって示される条件が真と評価されます。*condition-expression* には任意の式を使用できますが、整数型に評価されなければなりません。cond イベントは、次のコードとほとんど同じ働きをします。

```
stop step -if conditional_expression
```

システムイベント仕様

次に、システムイベントに対するイベント仕様について説明します。

`dlopen [ lib-path ]|dlclose [ lib-path ]`

これらのイベントは、`dlopen()` または `dlclose()` の呼び出しが正常終了した後に発生します。`dlopen()` または `dlclose()` の呼び出しにより、複数のライブラリが読み込まれることがあります。これらのライブラリのリストは、事前定義済み変数 `$dllist` で常に入手できます。`$dllist` の中の最初のシェルの単語は実際には "+" または "-" で、それぞれライブラリが追加されているか、削除されているかを示します。

lib-path は、該当する共有ライブラリの名前です。これを指定した場合、そのライブラリが読み込まれたり、読み込みが取り消されたりした場合にだけイベントが起動します。その場合、`$dlobj` にライブラリの名前が格納されます。また、`$dllist` も利用できます。

lib-path が / で始まる場合は、パス名全体が比較されます。それ以外の場合は、パス名のベースだけが比較されます。

lib-path を指定しない場合、イベントは任意の `dl` 動作があるときに必ず起動します。`$dlobj` は空になりますが、`$dllist` は有効です。

`falut` *fault*

`fault` イベントは、指定の障害に遭遇したとき、発生します。障害は、アーキテクチャ依存です。dbx に対して知られる次の一連の障害は、proc(4) マニュアルページで定義されています。

変数	定義
FLTILL	不正命令
FLTPRIV	特権つき命令
FLTBPT*	ブレークポイント命令
FLTTRACE*	トレーストラップ (ステップ実行)
FLTWATCH	ウォッチポイントトラップ
FLTACCESS	メモリーアクセス (境界合わせなど)
FLTBOUNDS*	メモリー境界 (無効なアドレス)
FLTIOVF	整数オーバーフロー
FLTIZDIV	整数ゼロ除算

変数	定義
FLTPE	浮動小数点例外
FLTSTACK	修復不可能なスタックフォルト
FLTPAGE	修復可能なページフォルト

注 – BPT、TRACE、BOUNDS は、ブレークポイントとステップ実行を実現するため、dbx で使用されます。これらを操作すると、dbx の動作に影響を及ぼす場合があります。

これらの障害は、/sys/fault.h から抜粋されています。*fault* には上記の名前を大文字または小文字で指定できるほか、実際のコードも指定できます。また、コードの名前には、接頭辞 FLT- をつけることがあります。

lwp_exit

lwp_exit イベントは、lwp が終了したとき、発生します。$lwp には、終了した LWP (軽量プロセス) の id が含まれます。

sig *sig*

sig *sig* イベントは、デバッグ中のプログラムに信号が初めて送られたとき、発生します。*sig* は、10 進数、または大文字、小文字の信号名のいずれかです。接頭辞は任意です。このイベントは、catch および ignore コマンドからは完全に独立しています。ただし、catch コマンドは次のように実現することができます。

```
function simple_catch {
    when sig $1 {
        stop;
        echo Stopped due to $sigstr $sig
        whereami
    }
}
```

注 – `sig` イベントを受け取った時点では、プロセスはまだそれを見ることができません。指定の信号を持つプロセスを継続する場合のみ、その信号が転送されます。

`sig` *sig sub-code*

指定の *sub-code* を持つ指定の信号が child に初めて送られたとき、`sig` *sig sub-code* イベントが発生します。信号同様、*sub-code* は 10 進数として、大文字または小文字で入力することができます。接頭辞は任意です。

`sysin` *code*|*name*

指定されたシステムコールが起動された直後で、プロセスがカーネルモードに入ったとき。

dbx の認識するシステムコールは `procfs`(4) の認識するものに限られます。これらのシステムコールはカーネルでトラップされ、`/usr/include/sys/syscall.h` に列挙されます。

これは、ABI の言うところのシステムコールとは違います。ABI のシステムコールの一部は部分的にユーザーモードで実装され、非 ABI のカーネルトラップを使用します。ただし、一般的なシステムコールのほとんど (シグナル関係は除く) は `syscall.h` と ABI で共通です。

`sysout` *code*|*name*

指定されたシステムコールが終了し、プロセスがユーザーモードに戻る直前。

`sysin`|`sysout`

引数がないときは、すべてのシステムコールがトレースされます。ここで、`modify` イベントや RTC (実行時検査) などの特定の `dbx` は、子プロセスにその目的でシステムコールを引き起こすことがあることに注意してください。トレースした場合にそのシステムコールの内容が示されることがあります。

実行進行状況イベント仕様

次に、実行進行状況に関するイベントのイベント仕様について説明します。

`next`

`next` イベントは、関数がステップされないことを除いては、`step` イベントと同様です。

`returns`

このイベントは、現在表示されている関数の戻りのブレークポイントです。表示されている関数を使用するのは、いくつかの `up` を行なった後に `returns` イベント指定を使用できるようにするためです。通常の戻りイベントは常に一時イベント (`-temp`) で、動作中のプロセスが存在する場合にだけ作成できます。

`returns` *func*

これは一時イベントではありません。特定の関数がその呼び出し場所にリターンするたびに発生します。戻り値は示されませんが、SPARC プラットフォームでは `$o0`、Intel プラットフォームでは `$eax` を使用して、必須戻り値を調べることができます。

SPARC	`$o0`
Intel	`$eax`

このイベントは、次のコードとほとんど同じ働きをします。

```
when in func { stop returns; }
```

`step`

`step` イベントは、ソース行の先頭の命令が実行されると発生します。たとえば、次のようにシンプルに表現することができます。

```
when step { echo $lineno: $line; }; cont
```

step イベントを有効にするということは、次に cont コマンドが使用されるときに自動的にステップ実行できるように dbx に命令することと同じです。step (および next) イベントは一般的なステップコマンド終了時に発生しません。むしろ step コマンドは、ほとんど次のように指定します。

```
alias step="when step -temp { whereami; stop; }; cont"
```

その他のイベント仕様

次に、その他のタイプのイベントに対するイベント仕様を説明します。

attach

dbx がプロセスを正常に接続した直後。

detach

dbx がプロセスを切り離す直前。

lastrites

デバッグ対象のプロセスが終了する直前。このイベントが発生するのは次の 3 つの場合です。

- システムコール _exit (2) が呼び出し中 (これは、明示的に呼び出されたとき、または
  main() のリターン時に発生します)。
- 終了シグナルが送信されようとするとき。
- dbx コマンド kill によってプロセスが強制終了されつつあるとき。

これは、プロセスの状態を調べる最後の機会です。このイベントの後にプログラムの実行を再開すると、プロセスは終了します。

proc_gone

dbx がデバッグ中のプロセスと関連しなくなるとき。事前定義済み変数 $reason に、signal、exit、kill、または detach のいずれかが設定されます。

prog_new

follow exec の結果、新規のプログラムがロードされると、prog_new イベントが発生します。

注 – このイベントのハンドラは常に存在しています。

---

stop

プロセスが停止したとき。特に stop ハンドラによりユーザーがプロンプトを受け取るときのようにプロセスが停止すると、このイベントが起動します。次に例を示します。

```
display x
when stop {print x;}
```

sync

デバッグ対象のプロセスが exec() で実行された直後。a.out で指定されたメモリーはすべて有効で存在しますが、あらかじめ読み込まれるべき共有ライブラリはまだ読み込まれていません。たとえば printf は dbx に認識されていますが、まだメモリーにはマップされていません。

stop コマンドにこのイベントを指定しても期待した結果は得られません。when コマンドに指定してください。

syncrtld

このイベントは、sync (被デバッグ側が共有ライブラリをまだ処理していない場合は attach) の後に発生します。すなわち、動的リンカーの起動時コードが実行され、あらかじめ読み込まれている共有ライブラリすべてのシンボルテーブルが読み込まれた後、ただし、.init セクション内のコードがすべて実行される前に発生します。

stop コマンドにこのイベントを指定しても期待した結果は得られません。when コマンドに指定してください。

throw

処理されない、または予期されない例外がアプリケーションによって投げ出されると、throw イベントが発生します。

`throw` *type*

例外 *type* が `throw` イベントで指定されると、そのタイプの例外のみが `throw` イベントを発生させます。

`throw -unhandled`

`-unhandled` は、投げ出されたが、それに対するハンドラがない例外を示す、特別な例外タイプです。

`throw -unexpected`

`-unexpected` は、それを投げ出した関数の例外仕様を満たさない例外を示す、特別な例外タイプです。

`timer` *seconds*

デバッグ中のプログラムが *seconds* 間実行されると、`timer` イベントが発生します。このイベントで使用されるタイマーは、`collector` コマンドで共有されます。解像度はミリ秒であるため、秒の浮動小数点値 (0.001 など) が使用可能です。

イベント指定のための修飾子

イベント指定のため修飾子は、ハンドラの追加属性を設定します。最も一般的な種類はイベントフィルタです。修飾子はイベント指定のキーワードの後に指定しなければなりません。修飾語はすべて '-' で始まります (その前にブランクが置かれます)。各修飾子の構成は次のとおりです。

`-if` *cond*

event-spec で指定されたイベントが発生したとき、条件 *cond* が評価されます。イベントは、条件が非ゼロと評価された場合にだけ発生すると考えられます。

`-if` が、`in` または `at` などの単独のソース位置に基づくイベントで使用された場合、*cond* はその位置に対応するスコープで評価されます。そうでない場合は、必要なスコープによって正しく修飾する必要があります。

`-in` *func*

ハンドラは、指定した関数 *func*、または *func* から呼び出された関数によって制御されている間だけ有効になります。関数へ入った回数は、再帰呼び出しに正しく対応するため「参照による計数」が行われます。

`-disable`

無効な状態にしてイベントを作成します。

`-count` *n*
`-count infinity`

`-count` *n* および `-count infinity` 修飾子は、0 からのハンドラカウントを持ちます (260 ページの「イベントカウンタ」参照)。イベントが発生する度、*n* に達するまでカウントはインクリメントします。一度それが生じると、ハンドラはファイアし、カウンタはゼロにリセットされます。

プログラムが実行または再実行されると、すべてのイベントのカウントがリセットされます。より具体的に言えば、カウントは `sync` イベントが発生するとリセットされます。

`-temp`

一時ハンドラを作成します。イベントが発生すると、一時イベントは削除されます。デフォルトではハンドラは、一時イベントではありません。ハンドラが計数ハンドラ (`-count` が指定されたイベント) の場合はゼロに達すると自動的に破棄されます。

一時ハンドラをすべて削除するには `delete -temp` を実行します。

`-instr`

イベントを命令レベルで動作させます。これにより、ほとんどの 'i' で始まるコマンドは不要となります。この修飾子は、イベントハンドラの 2 つの面を修飾します。

- 出力されるどのメッセージもソースレベルの情報ではなく、アセンブリレベルを示す。
- イベントの細分性が命令レベルになる。たとえば `step -instr` は、命令レベルのステップ実行を意味する。

-thread *thread_id*

イベントを引き起こしたスレッドが *thread_id* と一致する場合に限り、アクションが実行されます。プログラムの実行を繰り返すうちに特定スレッドの *threa_id* が変わってしまうことがあります。

-lwp *lwp_id*

イベントを引き起こしたスレッドが *lwp_id* と一致する場合に限り、アクションが実行されます。プログラムの実行を繰り返すうちに特定スレッドの *lwp_id* が変わってしまうことがあります。

-hidden

ハンドラが正規の `status` コマンドに示されないようにします。隠されたハンドラを表示するには、`status -h` を使用してください。

-perm

通常、すべてのハンドラは、新しいプログラムが読み込まれると廃棄されます。この修飾子を使用すると、ハンドラはデバッグが終わっても保存されます。`delete` コマンド単独では、永続ハンドラは削除されません。永続ハンドラを削除するには、`delete -p` を使用してください。

解析とあいまいさに関する注意

イベント指定と修飾子のための構文の特徴は次のとおりです。

- キーワード駆動型である。
- 主に、空白によって区切られた「単語」に分割される点など、すべて `ksh` の規約にもとづいている。

下位互換性のため、式の中には空白を含むことができます。そのため、式の内容があいまいになることがあります。たとえば、次の 2 つのコマンドがあるとします。

```
when a -temp
when a-temp
```

上の例では、アプリケーションで *temp* という名前の変数が使用されていても、dbx は -temp を修飾子としてイベント指定を解釈します。下の例では、a-temp がまとめて言語固有の式解析プログラムに渡され、*a* および *temp* という変数が存在しなければ、エラーになります。オプションを括弧で囲むことにより、解析を強制できます。

事前定義済み変数

読み取り専用の ksh 事前定義済み変数がいくつか用意されています。以下に示す変数は常に有効です。

変数	定義
$ins	現在の命令の逆アセンブル
$lineno	現在の行番号 (10 進数)
$vlineno	現在アクセスしている行番号 (10 進数)
$line	現在の行の内容
$func	現在の関数の名前
$vfunc	現在「表示している」関数
$class	$func が所属するクラスの名前
$vclass	$vfunc が所属するクラスの名前
$file	現在のファイルの名前
$vfile	現在表示しているファイルの名前
$loadobj	現在のロードオブジェクトの名前
$vloadobj	現在表示している現在のロードオブジェクトの名前
$scope	逆引用符表記での現在の PC のスコープ
$vscope	現在表示している逆引用符表記での PC のスコープ
$funcaddr	$func のアドレス (16 進数)
$caller	$func を呼び出している関数の名前

変数	定義
$dllist	dlopen イベントまたは dlclose イベントの後、dlopen または dlclose された直後のロードオブジェクトのリストが格納されます。$dllist の中の先頭の単語は実際には "+" または "-" です。これは、dlopen と dlclose のどちらが発生したかを示します。
$newhandlerid	最後に作成されたハンドラの ID
$firedhandlers	停止の原因となった最近のハンドラ ID のリストです。リストにあるハンドラには、status コマンドの出力時に「*」が付きます。
$proc	現在デバッグ中のプロセスの ID
$lwp	現在の LWP の ID
$thread	現在のスレッドの ID
$prog	デバッグ中のプログラムの絶対パス名
$oprog	$prog の古い値または元の値。これは、exec() に続いて、デバッグしていたものに戻る場合に便利です。
$exitcode	プログラムの最後の実行状態を終了します。この値は、プロセスが実際には終了していない場合、空文字列になります。

たとえば、whereami は次のように実装できます。

```
function whereami {
  echo Stopped in $func at line $lineno in file $(basename $file)
  echo "$lineno\t$line"
}
```

when コマンドに対して有効な変数

次の変数は、when コマンドの本体内でのみ有効です。

$handlerid

本体の実行中、$handlerid にはそれが属する when コマンドの ID が格納されます。次のコマンドは同じ結果になります。

```
when X -temp { do_stuff; }
when X  { do_stuff; delete $handlerid; }
```

$booting

イベントがブートプロセス中に起こると、true (真) に設定されます。新しいプログラムは、デバッグされるたびに、まず共有ライブラリのリストと位置を確認できるよう、ユーザーに通知されないまま実行されます。プロセスはその後終了します。ブートはこのようなシーケンスで行われます。

ブートが起こっても、イベントはすべて使用可能です。この変数は、デバッグ中に起こる sync および syncrtld のイベントと、通常の実行中に起こるこれらのイベントを区別するときに使用してください。

イベント別の有効変数

以下の表で指定されている、特定のイベントの場合のみ有効な変数があります。

表 19-1 sig イベントに固有の変数

変数	説明
$sig	イベントを発生させたシグナル番号
$sigstr	$sig の名前
$sigcode	適用可能な場合、$sig のサブコード
$sigcodestr	$sigcode の名前
$sigsender	必要であれば、シグナルの送信者のプロセス ID

表 19-2 exit イベントに固有の変数

変数	説明
$exitcode	_exit(2) または exit(3) に渡された引数の値、または main の戻り値

表 19-3 dlopen および dlclose イベントに固有の変数

変数	説明
$dlobj	dlopen または dlclose されたロードオブジェクトのパス名

表 19-4 sysin および sysout イベントに固有の変数

変数	説明
$syscode	システムコールの番号
$sysname	システムコールの名前

表 19-5 proc_gone イベントに固有の変数

変数	説明
$reason	シグナル、終了、強制終了、または切り離しのいずれか。

イベントハンドラの設定例

次に、イベントハンドラの設定例をあげます。

配列メンバーへのストアに対するブレークポイントを設定する

array[99] でブレークポイントを設定するには、次のように入力します。

```
(dbx) stop access w &array[99]
(2) stop access w &array[99], 4
(dbx) run
実行中: watch.x
(プロセス id 9247)
ウォッチポイント &array[99] (0x20b68[4]) 行番号 12  ファイル "watch.c"
   12           array[i] = i;
```

単純なトレースを実行する

単純なトレースの例 :

```
(dbx) when step { echo at line $lineno; }
```

関数の中だけイベントを有効にする (in *func*)

たとえば、

```
(dbx) trace step -in foo
```

は、次のようなスクリプトと等価です。

```
    # ハンドラの使用不可能状態を作成する
    when step -disable { echo Stepped to $line; }
    t=$newhandlerid    # ハンドラ ID を憶えておく
    when in foo {
     # foo を入力するとトレースハンドラ -enable "$t" が
     # 使用可能になり、foo から返るとトレースが
     # 使用不可能になります。
     when returns { handler -disable "$t"; };
    }
```

実行された行の数を調べる

小規模なプログラムで何行実行されたかを調べます。

```
(dbx) stop step -count infinity # ステップ実行し、count=inf (関数が
無限大) になったところで停止する
(2) stop step -count 0/infinity
(dbx) run
...
(dbx) status
(2) stop step -count 133/infinity
```

ここでは、プログラムを停止させているのではなく、明らかにプログラムが終了しています。133 は実行された行数です。ただし、このプロセスは非常に低速です。この方法が有効なのは、何度も呼び出される関数にブレークポイントを設定している場合です。

実行された命令の数をソース行で調べる

特定の行で実行された命令の数を数えます 。:

```
(dbx) ...                              # 調べたい行まで移動する
(dbx) stop step -instr -count infinity
(dbx) step ...
(dbx) status
(3) stop step -count 48/infinity # 48 48 個の命令が実行された
```

ステップ実行している行で関数呼び出しが行われる場合、最終的にそれらの呼び出しもカウントされます。step イベントの代わりに next イベントを使用すれば、そのような呼び出しはカウントされません。

イベント発生後にブレークポイントを有効にする

別のイベントが発生した場合のみ、ブレークポイントを有効にします。たとえば、プログラムで関数 hash が 1300 番目のシンボル検索以後に正しく動作しなくなるとします。次のように入力します。

```
(dbx) when in lookup -count 1300 {
    stop in hash
    hash_bpt=$newhandlerid
    when proc_gone -temp { delete $hash_bpt; }
}
```

注 – $newhandlerid が、実行された直後の stop in コマンドを参照している点に注意してください。

---

replay 時にアプリケーションファイルをリセットする

アプリケーションが処理するファイルを replay 中にリセットする必要がある場合、プログラムを実行するたびに自動的にリセットを行うハンドラを書くことができます。

```
(dbx)  when sync { sh regen ./database; }
(dbx)  run < ./database...# この間にデータベースファイルが壊れた場合
(dbx)  save
...                              # run が自動的に行われ、sync イベントが
(dbx)  restore                 # 発生し、regen が実行される。
```

プログラムの状態を調べる

プログラムの実行中にその状態をすばやく調べます。

```
(dbx) ignore sigint
(dbx) when sig sigint { where; cancel; }
```

プログラムを停止しないでそのスタックトレースを調べるためには、ここで ^C を押します。

コレクタはこれ以外のことも実行できますが、基本的にコレクタの手動標本収集モードが実行する機能は、このように状態を調べます。ここではすでに ^C を使用したため、プログラムに割り込むには SIGQUIT (^\) を使用します。

浮動小数点例外を捕捉する

特定の浮動小数点例外を捕捉します。ここでは、IEEE オーバーフローだけを捕捉しています。

```
(dbx) ignore FPE                    # デフォルトのハンドラをオフにする
(dbx) help signals | grep FPE  # サブコードの名前を思い出せない
...
(dbx) stop sig fpe FPE_FLTUND
...
```

付録C

コマンドリファレンス

この付録では、dbxコマンドの構文と機能について詳しく説明します。

adbコマンド

adb コマンドは、adb 形式のコマンドを実行したりadbモードを設定したりします。

構文

adb *adb*コマンド	adb 形式のコマンドを実行します。
adb	adb モードを設定します。abdモードを終了するには、$q を使用します。

assign コマンド

assign コマンドは、新しい値をプログラム変数に代入します。

構文

assign *variable* = *expression*

ここで、

expression は、*variable* に代入される値です。

attach コマンド

attach コマンドは実行中プロセスに dbx を接続し、実行を停止してプログラムをデバッグ制御下に入れます。

構文

attach *process_id*	プロセスID *process_id* を持つプログラムのデバッグを開始します。 dbx が、/proc を使用してプログラムを見つけます。.
attach -p *process_id program_name*	プロセスID *process_id* を持つ *program* のデバッグを開始します。
attach *program_name process_id*	プロセスID *process_id* を持つ *program* のデバッグを開始します。*program* として-を使用できます。dbx が /procを使用してプログラムを見つけます。
attach -r ...	-r オプションをつけて dbx を使用すると、display、trace、 when、stop のコマンドがすべて保持されます。 -r オプションを使用しなかった場合は、暗黙の delete all とundisplay 0 が実行されます。

ここで、

process_id は、実行中プロセスのプロセスIDです。

program_name は、実行中プログラムのパス名です。

bsearch コマンド

bsearch コマンドは、現在のソースファイルにおいて逆方向検索を行います。

構文

bsearch *string*	現在のファイルの中で、*string* を逆方向で検索します。
bsearch	最後の検索文字列を使用して検索を繰り返します。

ここで、

string は、文字列です。

call コマンド

call コマンドは、手続きを呼び出します。

構文

call *procedure* ([*parameters*])

ここで、

procedure は、手続きの名前です。

parameters は、手続きのパラメータです。

call コマンドによって関数を呼び出すこともできます。戻り値を調べるには、print コマンドを使用します (326 ページの「print コマンド」参照)。

呼び出された関数がブレークポイントに達することがあります。 呼び出しを続行するにはcontコマンドを使用し (288 ページの「cont コマンド」参照)、中止するにはpop -c を使用します (325 ページの「pop コマンド」参照)。 後者は、呼び出された関数がセグメント例外を引き起こしたときにも利用できます。

cancel コマンド

cancel コマンドは、現在のシグナルを取り消します。 このコマンドは、主としてwhen コマンドの本体内で使用します (354 ページの「when コマンド」参照)。

通常、シグナルが取り消されるのは、dbx がシグナルのため停止した場合です。whenコマンドがシグナルイベントに接続されている場合、そのシグナルが自動的に取り消されることはありません。cancel コマンドを使用すれば、シグナルを明示的に取り消せます。

構文

cancel

catch コマンド

catch コマンドは、指定のシグナルを捕獲します。

シグナルを捕獲すると、プロセスがそのシグナルを受信したときに dbx がプログラムを停止します。 その時点でプログラムを続行しても、シグナルがプログラムによって処理されることはありません。

構文

`catch`	捕獲するシグナルのリストを出力します。
`catch` *number number...*	*number* の番号のシグナルを捕獲します。
`catch` *signal signal...*	*signal* という名前のシグナルを捕獲します。 SIGKILL を捕獲したり無視したりすることはできません。
`catch $(ignore)`	すべてのシグナルを捕獲します。

ここで、

number は、シグナルの番号です。

signal は、シグナルの名前です。

check コマンド

check コマンドは、メモリーへのアクセス、メモリーリーク、メモリー使用状況をチェックし、実行時検査 (RTC) の現在状態を出力します。 このコマンドによる実行時検査機能は、debug コマンドによって初期状態にリセットされます。

構文

`check -access`

アクセス検査を起動します。 RTCは、次のエラーを報告します。

baf	不正解放
duf	重複解放
maf	境界整列を誤った解放
mar	境界整列を誤った読み取り
maw	境界整列を誤った書き込み
oom	メモリー不足
rua	非割り当てメモリーからの読み取り
rui	非初期化メモリーからの読み取り
wro	読み取り専用メモリーへの書き込み
wua	非割り当てメモリーへの書き込み

デフォルトの場合、各アクセスエラーが検出されるとプロセスが停止されます。このデフォルト動作を変更するには、dbx 環境変数 `rtc_auto_continue` を使用します。 `on` が設定されている場合、アクセスエラーはファイルに記録されます (ファイル名はdbx 環境変数 `rtc_error_log_file_name` によって制御します)。 298 ページの「dbxenv コマンド」を参照してください。

デフォルトの場合、それぞれのアクセスエラーが報告されるのは、最初に発生したときだけです。 この動作を変更するには、dbx 環境変数 `rtc_auto_suppress` を使用します (この変数のデフォルト値は `on` です)。 298 ページの「dbxenv コマンド」を参照してください。

`check -leaks [-frames` *n*`] [-match` *m*`]`

リーク検査をオンにします。 RTC は、次のエラーを報告します。

aib	メモリーリークの可能性 - 唯一のポインタがブロック中央を指す。
air	メモリーリークの可能性 - ブロックを指すポインタがレジスタ内にのみ存在する。
mel	メモリーリーク - ブロックを指すポインタがない。

リーク検査がオンの場合、プログラムが存在していれば自動リークレポートが作成されます。 このとき、可能性のあるリークを含むすべてのリークが報告されます。 デフォルトの場合、簡易レポートが作成されます (`dbx` 環境変数 `rtc_mel_at_exit` によって制御します)。 ただし、リークレポートをいつでも要求することができます (335 ページの「showleaks コマンド」参照)。

`-frames` *n* は、リーク報告時に最大 *n* 個のスタックフレームが表示されることを意味します。 `-match` *m* は、複数のリークをまとめます。2 個以上のリークに対する割り当て時の呼び出しスタックが *n* 個のフレームに一致するとき、これらのリークは 1 つのリークレポートにまとめて報告されます。

n のデフォルト値は、8または *m* の値です (どちらか大きい方)。 *n* の最大値は 16 です。*m* のデフォルト値は 2 です。

`check -memuse [-frames` *n*`] [-match` *m*`]`

メモリー使用状況 (memuse) 検査をオンにします。check-memuse は、check-leaks も示します。 プログラム終了時のリークレポートだけではなく、使用中ブロック (biu) レポートも作成されます。 デフォルトの場合、簡易使用中レポートが生成されます (`dbx` 環境変数 `rtc_biu_at_exit`によって制御します)。プログラム実行中、プログラムのなかでメモリーが割り当てられた場所をいつでも調べることができます (335 ページの「showmemuse コマンド」参照)。

`-frames` *n* は、メモリーの使用状況とリークを報告するときに最大*n* 個のスタックフレームが表示されることを意味します。 `-match` *m* は、複数のリークをまとめます。2個以上のリークに対する割り当て時の呼び出しスタックが *n* 個のフレームに一致するとき、これらのリークは1つのリークレポートにまとめて報告されます。

n のデフォルト値は、8 または*m*の値です (どちらか大きい方)。 *n* の最大値は 16 です。*m* のデフォルト値は 2 です。`check -leaks` も参照してください。

`check -all [-frames n] [-match m]`

`check -access` または `check -memuse [-frames n] [-match m]` と同じ。

dbx 環境変数 `rtc_biu_at_exit` の値は`check -all`によって変更されないので、デフォルトの場合、終了時にメモリー使用状況レポートは生成されません。`rtc_biu_at_exit` 環境変数については、296 ページの「dbx コマンド」を参照してください。

`check [`*functions*`] [`*files*`] [`*loadobjects*`]`

functions、*files*、 *loadobjects*における `check -all` 、`suppress all`、または `unsuppress all` と同じ。

ここで、

functions は、1個または複数の関数名です。

files は、1個または複数のファイル名です。

loadobjects は、1個または複数のロードオブジェクト名です。

これを使用することにより、特定の場所を対象として実行時検査を行えます。

注 – RTCですべてのエラーを検出する際、`-g` を付けてプログラムをコンパイルする必要はありません。 ただし、特定のエラー (ほとんどは非初期化メモリーから読み取られるもの) の正確さを保証するには、シンボリック (`-g`) 情報が必要となることがあります。 このため、一部のエラー (`a.out` の`rui` と共有ライブラリの `rui + aib + air`) は、シンボリック情報を利用できないときには抑止されます。 この動作は、`suppress` と `unsuppress` によって変更できます。

---

cont コマンド

cont コマンドは、プロセスの実行を続行します。

構文

`cont`	実行を続行します。 MT プロセスではすべてのスレッドが再開されます。 Control-C を使用すると、プログラムの実行が停止します。
`cont ... -sig` *signal*	シグナル *signal* によって実行を続行します。
`cont ...` *id*	*id* は、続行対象のスレッドまたはLWPを指定します。
`cont at` *line* `[` *id* `]`	行 *line* から実行を続行します。アプリケーションがマルチスレッドである場合には、*id* が必要となります。
`cont ... -follow parent|child|both`	dbx `follow_fork_mode` 環境変数が ask に設定されているときに `stop` を選択した場合、どのプロセスを追跡するかをこのオプションによって選択します。Sun WorkShop Debugging で利用できるのは both だけです。

clear コマンド

clear コマンドは、ブレークポイントをクリアします。

引数 `inclass`、`inmethod`、または `infunction` をつけた `stop`、 `trace`、または `when` コマンドを使用して作成したイベントハンドラは、ブレークポイントセットを作成します。 `clear` コマンドで指定した*line*がこれらのブレークポイントのどれかに一致した場合、そのブレークポイントだけがクリアされます。 特定のセットに属するブレークポイントをこの方法でクリアした後、そのブレークポイントを再び使用可能にすることはできません。 ただし、関連するイベントハンドラをいったん使用不可にした後使用可能にすると、すべてのブレークポイントが再設定されます。

構文

`clear`	現在の停止点にあるブレークポイントをすべてクリアします。
`clear` *line*	*line* にあるブレークポイントすべてをクリアします。
`clear` *filename*`:`*line*	*filename* の *line* にあるブレークポイントをすべてクリアします。

ここで、

line は、ソースコード行の番号です。

filename は、ソースコードファイルの名前です。

collector コマンド

`collector` コマンドは、パフォーマンスアナライザによって分析するパフォーマンスデータを収集します。

構文

`collector` *command_list*	1個または複数の collector コマンドを指定します。
`address_space` *options*	アドレス空間設定値を指定します(291 ページの「collector address_space コマンド」参照)。
disable	データ収集を停止して現在の実験をクローズします(291 ページの「collector disable コマンド」参照)。
enable	コレクタを使用可能にして新規の実験をオープンします (291 ページの「collector enable コマンド」参照)。
hwprofile	ハードウェアカウンタプロファイル設定値を指定します (291 ページの「collector hwprofile コマンド」参照)。
pause	パフォーマンスデータの収集は停止しますが、実験はオープン状態のままとします (292 ページの「collector pause コマンド」参照)。
`profile` *options*	呼び出しスタックプロファイルデータを収集するための設定値を指定します (292 ページの「collector profile コマンド」参照)。
resume	一時停止後、パフォーマンスデータの収集を開始します (293 ページの「collector resume コマンド」参照)。
`sample` *options*	標本設定値を指定します(293 ページの「collector sample コマンド」参照)。
`show` *options*	現在のコレクタ設定値を表示します (293 ページの「collector show コマンド」参照)。
`status`	現在の実験に関するステータスを照会します (294 ページの「collector status コマンド」参照)。
`store` *options*	ファイルの制御と設定値を実験します (294 ページの「collector store コマンド」参照)。
`synctrace` *options*	スレッド同期待ちトレースデータの設定値を指定します (295 ページの「collector synctrace コマンド」参照)。

ここで、

options は、各コマンドで指定できる設定値です。

データの収集を開始するには、`collector enable` と入力します。

データ収集を停止するには、`collector disable` と入力します。

collector address_space コマンド

collector address_space コマンドは、アドレス空間データを収集するかどうかを指定します。

構文

collector address_space on|off	on を指定すると、アドレス空間データが収集されます。デフォルトの場合、アドレス空間データは収集されません。

collector disable コマンド

collector disable コマンドは、データ収集を停止して現在の実験をクローズします。

構文

```
collector disable
```

collector enable コマンド

collector enable コマンドは、コレクタを使用可能にして新規の実験をオープンします。

構文

```
collector ensable
```

collector hwprofile コマンド

collector hwprofile コマンドは、ハードウェアカウンタオーバーフロープロファイルデータ収集のオプションを指定します。

構文

`collector hwprofile on|off`	デフォルトの場合、ハードウェアカウンタオーバーフロープロファイルデータは収集されません。 このデータを収集するには、`on` を指定します。
`collector hwprofile list`	利用できるカウンタのリストを出力します。
`collector hwprofile counter` *`name interval name2`*	ハードウェアカウンタ名と間隔を指定します。

ここで、

name は、ハードウェアカウンタの名前です。

interval は、ミリ秒単位による収集間隔です。

name2 は、第 2 ハードウェアカウンタの名前です。

ハードウェアカウンタはシステム固有であるため、どのようなカウンタを利用できるかはご使用のシステムによって異なります。 多くのシステムでは、ハードウェアカウンタオーバーフロープロファイル機能をサポートしていません。 こういったマシンの場合、この機能は使用不可になっています。

`collector pause` コマンド

`collector pause` コマンドはデータ収集を停止しますが、現在の実験はオープン状態のままとします。 `collector resume` コマンドを使用すれば、データ収集を再開できます (293 ページの「collector resume コマンド」参照)。

構文

```
collector pause
```

`collector profile` コマンド

collector profile コマンドは、プロファイルデータ収集のオプションを指定します。

構文

`collector profile on|off`	プロファイルデータ収集モードを指定します。
`collector profile timer` *milliseconds*	プロファイルタイマー時間を指定します。

collector resume コマンド

`collector resume` コマンドは、`collector pause` コマンドによる一時停止の後、データ収集を再開します (292 ページの「collector pause コマンド」参照)。

構文

```
collector resume
```

collector sample コマンド

`collector sample` コマンドは、標本モードと標本間隔を指定します。

構文

`collector sample periodic|manual`	標本モードを指定します。
`collector sample period` *seconds*	標本間隔を*seconds.* 単位で指定します。

ここで、

seconds は、標本間隔の長さです。

collector show コマンド

`collector show` コマンドは、1個または複数のオプションカテゴリの設定値を表示します。

構文

`collector show all`	すべての設定値を表示します。
`collector show profile`	呼び出しスタックプロファイル設定値を表示します。
`collector show synctrace`	スレッド同期待ちトレース設定値を表示します。
`collector show sample`	サンプル設定値を表示します。
`collector showstore`	ストア設定値を表示します。
`collector show address_space`	アドレス空間設定値を表示します。

`collector status` コマンド

`collector status` コマンドは、現在の実験のステータスについて照会します。

構文

```
collector status
```

`collector store` コマンド

`collector store` コマンドは、実験が保存されているディレクトリとファイルの名前を指定します。

構文

`collector store directory` *pathname*	実験が保存されているディレクトリを指定します。
`collector store filename` *filename*	実験ファイル名を指定します。
`collector store group` *string*	実験グループ名を指定します。

ここで、

pathname は、実験を保存するディレクトリのパス名です。

filename は、実験ファイルの名前です。

string は、実験グループの名前です。

collector synctrace コマンド

collector synctrace コマンドは、同期待ちトレースデータの収集オプションを指定します。

構文

collector synctrace on|off	デフォルトの場合、スレッド同期待ちトレースデータは収集されません。 このデータを収集するには、on を指定します。
collector threshold *microseconds*	しきい値をマイクロ秒単位で指定します。 デフォルト値は 100 です。
collector threshold calibrate	しきい値は、自動的に算出されます。

ここで、

microseconds は、この値未満であるときに同期待ちイベントが破棄されるしきい値です。

dalias コマンド

dalias コマンドは、dbx形式の (csh形式) 別名を定義します。

構文

dalias	(dbx alias) 現在定義されている別名をすべて一覧表示します。
dalias *name*	別名 *name* の定義がある場合には、それを表示します。
dalias *name definition*	*name* を *definition* の別名として定義します。*definition* には、空白を含めることができます。セミコロンまたは改行によって定義を終端させます。

ここで、

name は、別名の名前です。

definition は、別名の定義です。

dbx は、別名に通常使用される次の csh 履歴置換メタ構文を受け付けます。

!:<n>

!-<n>

!^

!$

!*

通常、! の前にはバックスラッシュを付ける必要があります。 たとえば、

```
dalias goto "stop at \!:1; cont; clear"
```

詳細については、csh(1) マニュアルページを参照してください。

dbx コマンド

dbx コマンドは、dbx を起動します。

構文

`dbx` *options program_name*	*program_name* をデバッグします。
`dbx` *options program_name core*	コアファイル *core* によって *program_name* をデバッグします。
`dbx` *options program_name process_id*	プロセスID *process_id* を持つ *program_name* をデバッグします。
`dbx` *options* `-` *process_id*	プロセスID *process_id* をデバッグします。dbx は、/proc によってプログラムを見つけます。
`dbx` *options* `-` *core*	コアファイル *core* を使用してデバッグします。298 ページの「debug コマンド」も参照してください。
`dbx` *options* `-r` *program_name arguments*	引数 *arguments* を付けて *program_name* を実行します。異常終了した場合は *program_name* のデバッグを開始します。そうでない場合はそのまま終了します。

ここで、

options は、次のオプションです。

`-c` *commands*	入力を指示するプロンプトを表示する前に *commands* を実行します。
`-C`	実行時検査ライブラリをあらかじめ読み込みます (284 ページの「check コマンド」参照)。
`-d`	`-s`を付けて使用した場合、読み取った *file* を削除します。
`-e`	入力コマンドを表示します。
`-f`	コアファイルが一致しない場合でも、コアファイルの読み込みを強制します。
`-h`	dbx の使用法ヘルプ情報を出力します。
`-I` *dir*	*dir* を `pathmap` セットに追加します (323 ページの「pathmap コマンド」参照)。
`-k`	キーボード変換状態を保存し復元します。
`-q`	読み取りスタブに関するメッセージを抑止します。
`-r`	プログラムを実行し、プログラムが正常終了した場合は終了させます。
`-R`	`dbx.`に関する README ファイルを表示します。
`-s` *file*	`.dbxrc` や `.dbxinit`の代わりに *file* を使用します。
`-S`	サイト固有 init ファイルの読み取りを抑止します。
`-V`	dbxのバージョンを出力します。
`-w` *n*	`where`コマンドで *n* 個のフレームをとばします。

`--` オプションリストの終わりを指定します。プログラム名がダッシュで始まるときに使用します。

program_name は、デバッグ対象プログラムの名前です。

process_id は、実行中プロセスのプロセスIDです。

arguments は、*program_name* に引き渡す引数です。

dbxenv コマンド

`dbxenv` コマンドは、`dbx` 環境変数の表示や設定を行います。

構文

`dbxenv` `dbx` 環境変数の現在の設定値を表示します。

`dbxenv` *environment_variable* *environment_variable* に *setting*.を設定します。
setting

ここで、

environment_variable は、`dbx` 環境変数です。

setting は、その変数の有効な設定値です。

debug コマンド

`debug` コマンドは、デバッグ対象プログラムの表示や変更を行います。

構文

`debug`	デバッグ対象プログラムの名前と引数を出力します。
`debug` *program_name*	プロセスやコアなしで *program_name* のデバッグを開始します。
`debug -c` *core program_name*	コアファイル*core*による *program_name* のデバッグを開始します。
`debug -p` *process_id program_name*	プロセス ID*core* を持つ *program_name* のデバッグを開始します。
`debug` *program_name core*	コアファイル*core*による *program_name* のデバッグを開始します。*program_name* として**-**を使用できます。dbxは、コアファイルから実行可能ファイルの名前を取り出そうとします。 詳細については、16 ページの「既存のコアファイルのデバッグ」 を参照してください。
`debug` *program_name process_id*	プロセスID *process_id* を持つ *program_name* のデバッグを開始します。*program_name* として**-**を使用できます。`dbx` が `/proc` を使用してプログラムを見つけます。
`debug -f ...`	コアファイルが一致しない場合でも、コアファイルの読み込みを強制します。
`debug -r ...`	`-r` オプションをつけて `dbx` を使用すると、`display`、`trace`、 `when`、`stop` のコマンドがすべて保持されます。 `-r` オプションを使用しなかった場合は、暗黙の `delete all` と`undisplay 0` が実行されます。
`debug -clone ...`	`-clone` オプションは新たな `dbx` プロセスの実行を開始するので、複数のプロセスを同時にデバッグできます。Sun WorkShop Debugging で使用する場合にのみ有効です。
`debug -clone`	何もデバッグしない `dbx` プロセスを新たに開始します。Sun WorkShop Debugging で使用する場合にのみ有効です。
`debug [`*options*`] --` *program_name*	*program_name* がダッシュで始まる場合でも、*program_name* のデバッグを開始します。

ここで、

core は、コアファイルの名前です。

process_id は、実行中プロセスのプロセスIDです。

program_name は、プログラムのパス名です。

debug でプログラムを読み込むと、リーク検査とアクセス検査はオフになります。check コマンドを使用すれば、これらの検査を使用可能にできます (284 ページの「check コマンド」参照)。

delete コマンド

delete コマンドは、ブレークポイントなどのイベントを削除します。

構文

`delete [-h] handler_id ...`	指定の *handler_id* を持つ trace コマンド、when コマンド、または stop コマンドを削除します。 隠しハンドラを削除するには、-h オプションを使用する必要があります。
`delete [-h] 0 | all | -all`	常時隠しハンドラを除き、trace コマンド、 when コマンド、stop コマンドをすべて削除します。 -h を指定すると、隠しハンドラも削除されます。
`delete -temp`	一時ハンドラをすべて削除します。
`delete $firedhandlers`	最後の停止を引き起こしたハンドラすべてを削除します。

ここで、

handler_id は、ハンドラの識別子です。

detach コマンド

detach コマンドは、dbx の制御からターゲットプロセスを解放します。

構文

`detach`	ターゲットから dbx を切り離し、保留状態のシグナルがある場合はそれらのシグナルを取り消します。
`detach -sig signal`	指定の *signal* を転送している間、切り離します。

ここで、

signal は、シグナルの名前です。

dis コマンド

dis コマンドは、マシン命令を逆アセンブルします。

構文

`dis` *address* `[/` *count*`]`	アドレス *address* を始点とし、*count* 命令 (デフォルトは 10) を逆アセンブルします。
`dis` *address1*`,` *address2*	*address1* から *address2* までの命令を逆アセンブルします。
`dis`	+ の値を始点とし、10 個の命令を逆アセンブルします (303 ページの「examine コマンド」参照)。
`dis /`*count*	+ を始点とし、*count* 個の命令を逆アセンブルします。

ここで、

address は、逆アセンブルを開始するアドレスです。

address1 は、逆アセンブルを開始するアドレスです。

address2 は、逆アセンブルを停止するアドレスです。

count は、逆アセンブル対象命令の数です。

display コマンド

display コマンドは、すべての停止点で式を評価し表示します。

構文

`display`	表示されている式のリストを表示します。
`display` *expression*`, ...`	すべての停止点で式 *expression* の値を表示します。
`display [-r|+r|-d|+d|-p|+p|-f`*format*`|-F`*format*`|--]` *expression*`,...$`*newline*	これらのフラグの意味については、326 ページの「print コマンド」を参照してください。

ここで、

expression は、有効な式です。

down コマンド

down コマンドは、呼び出しスタックを下方向に移動します (mainから遠ざかる)。

構文

down	呼び出しスタックを 1 レベル下方向に移動します。
down *number*	呼び出しスタックを *number* レベルだけ下方向に移動します。
down -h [*number*]	呼び出しスタックを下方向に移動しますが、隠しフレームをとばすことはしません。

ここで、

number は、呼び出しスタックレベルの数です。

dump コマンド

dump コマンドは、手続きの局所変数すべてを出力します。

構文

dump	現在の手続きの局所変数すべてを出力します。
dump *procedure*	*procedure* の局所変数をすべて出力します。

ここで、

procedure は、手続きの名前です。

edit コマンド

edit コマンドは、ソースファイルに対して$EDITOR を起動します。

dbx が Sun WorkShop Debuggingで動作していない場合、edit コマンドは$EDITOR を使用します。 そうでない場合、editコマンドは該当するファイルを表示することを指示するメッセージをSun WorkShop Debuggingに送信します。 Sun WorkShop Debuggingの起動時に-E オプションを使用した場合、ソース表示を扱う外部エディタが使用されるか、ソース区画が更新されて対象ファイルが表示されます。

構文

`edit`	現在のファイルを編集します。
`edit` *filename*	指定のファイル *filename* を編集します。
`edit` *procedure*	関数または手続き *procedure* が入っているファイルを編集します。

ここで、

filename は、ファイルの名前です。

procedure は、関数または手続きの名前です。

examine コマンド

examine コマンドは、メモリーの内容を表示します。

構文

`examine [ address ] [ / [ count ] [ format ] ]`	*address* を始点とし、*count* 個の項目のメモリー内容を形式 *format* で表示します。
`examine address1 , address2 [ / [ format ] ]`	*address1* から*address2* までのメモリー内容 (address1、address2 を含む) を形式 *format* で表示します。
`examine address = [ format ]`	アドレスを (アドレスの内容ではなく) 指定の形式で表示します。 直前に表示された最後のアドレスを示す+ (省略した場合と同じ) を *address* として使用できます。 xは、`examine` の事前定義別名です。

exception コマンド

exceptionコマンドは、現在の C++ 例外の値を出力します。

構文

`exception [-d | +d]`	現在の C++ 例外がある場合、その値を出力します。

`-d` フラグの意味については、326 ページの「print コマンド」 を参照してください。

exists コマンド

`exists` コマンドは、シンボル名の有無をチェックします。

構文

exists *name*	現在のプログラム内で *name* が見つかった場合は 0、*name* が見つからなかった場合は 1 を返します。

ここで、

name は、シンボルの名前です。

file コマンド

file コマンドは、現在のファイルの表示や変更を行います。

構文

file	現在のファイルの名前を出力します。
file *filename*	現在のファイルを変更します。

ここで、

filename は、ファイルの名前です。

files コマンド

files コマンドは、特定の正規表現に一致するファイル名を一覧表示します。

構文

files	現在のプログラムに対してデバッグ情報を提供したファイルすべての名前を一覧表示します (-g によってコンパイルされたもの)。
files *regular_expression*	指定の正規表現に一致し -g によってコンパイルされたファイルすべての名前を一覧表示します。

ここで、

regular_expression は、正規表現です。

fix コマンド

fix コマンドは、修正されたソースファイルを再コンパイルし、修正された関数をアプリケーションに動的にリンクします。

構文

`fix`	現在のファイルを修正します。
`fix` *filename filename* `...`	*filename* を修正します。
`fix -f`	ソースに手が加えられていない場合にも、ファイルの修正を強制します。
`fix -a`	手が加えられたファイルすべてを修正します。
`fix -g`	`-O` フラグを取り除き、`-g` フラグを追加します。
`fix -c`	コンパイル行を出力します (dbxによる使用を目的として内部的に追加されたオプションの一部が含まれることがあります)。
`fix -n`	compile/link コマンドを実行しません (`-v`を付けて使用)。
`fix -v`	冗長モード (dbx 環境変数 `fix_verbose` の設定より優先されます)。
`fix +v`	簡易モード (dbx 環境変数 `fix_verbose` の設定より優先されます)。

fixed コマンド

`fixed` コマンドは、固定ファイルすべての名前を一覧表示します。

構文

```
fixed
```

frame コマンド

`frame` コマンドは、現在のスタックフレーム番号の表示や変更を行います。

構文

`frame`	現在のフレームのフレーム番号を表示します。
`frame [-h]` *number*	現在のフレームとしてフレーム *number* を設定します。
`frame [-h] +[`*number*`]`	*number* 個のフレームだけスタックを上方向に移動します。デフォルトは 1 です。
`frame [-h] -[`*number*`]`	*number* 個のフレームだけスタックを下方向に移動します。デフォルトは 1 です。
`-h`	フレームが隠されている場合でもフレームに進みます。

ここで、

number は、呼び出しスタック内のフレームの番号です。

func コマンド

`func` コマンドは、現在の関数の表示や変更を行います。

構文

func	現在の関数の名前を出力します。
func *procedure*	現在の関数を関数または手続き *procedure* に変更します。

ここで、

procedure は、関数または手続きの名前です。

funcs コマンド

`funcs` コマンドは、特定の正規表現に一致する関数名をすべて一覧表示します。

構文

`funcs`	現在のプログラム内の関数すべてを一覧表示します。
`funcs [-f` *filename*`] [-g] [`*regular_expression*`]`	`-f` *filename* を指定すると、ファイル内の関数すべてが表示されます。 `-g` を指定すると、デバッグ情報を持つ関数すべてが表示されます。 *regular_expression* を指定すると、この正規表現に一致する関数すべてが表示されます。

ここで、

filename は、一覧表示対象の関数が入っているファイルの名前です。

regular_expression は、一覧表示対象の関数が一致する正規表現です。

gdb コマンド

gdb コマンドは、gdb コマンドセットをサポートします。

構文

`gdb on | off`	`gdb on` を使用すると、dbx が gdb コマンドを理解し受け付ける gdb コマンドモードに入ります。 gdb コマンドモードを終了して dbx コマンドモードに戻るには、`gdb off` を使用します。gdb コマンドモードではdbx コマンドは受け付けられず、dbx コマンドモードでは gdbコマンドが受け付けられません。 ブレークポイントなどのデバッグ設定は、コマンドモードの種類にかかわらず保持されます。

このリリースでは、次の gdb コマンドをサポートしていません。

コマンド
```
define
handle
hbreak
interrupt
maintenance
printf
rbreak
return
```

```
signal
tcatch
until
```

handler コマンド

handler コマンドは、イベントハンドラを変更します (使用可能や使用不可にするなど)。 ハンドラは、デバッグセッションで管理する必要があるイベントそれぞれについて作成されます。 trace、stop、when の各コマンドは、ハンドラを作成します。これらのコマンドはそれぞれ、ハンドラ ID と呼ばれる番号を返します (*handler_id*)。handler、status、delete の各コマンドは、一般的な方法でハンドラの操作やハンドラ情報の提供を行います。

構文

`handler -enable` *handler_id* `...`	特定のハンドラを使用可能にし、全ハンドラを示す all を *handler_id* として指定します。
`handler -disable` *handler_id* `...`	特定のハンドラを使用不可にし、全ハンドラを示す all を *handler_id* として指定します。 *handler_id* の代わりに `$firedhandlers` を使用すると、最後の停止を引き起こしたハンドラが使用不可となります。
`handler -count` *handler_id*	特定のハンドラのトリップカウンタの値を出力します。
`handler -count` *handler_id newlimit*	特定のイベントに対し、新たなカウント制限値を設定します。
`handler -reset` *handler_id*	特定のハンドラのトリップカウンタをリセットします。

ここで、

handler_id は、ハンドラの識別子です。

hide コマンド

hide コマンドは、特定の正規表現に一致するスタックフレームを隠します。

構文

`hide`	現在有効であるスタックフレームフィルタを一覧表示します。
`hide` *regular_expression*	*regular_expression* に一致するスタックフレームを隠します。 正規表現は関数名またはロードオブジェクトの名前を表し、sh または ksh の正規表現スタイルをとります。

ここで、

regular_expression は、正規表現です。

ignore コマンド

`ignore` コマンドは、指定のシグナルを捕獲しないことを dbx プロセスに指示します。 シグナルを無視すると、プロセスがそのシグナルを受信してもdbxが停止しなくなります。

構文

`ignore`	無視するシグナルのリストを出力します。
`ignore` *number*`...`	*number* の番号のシグナルを無視します。
`ignore` *signal*`...`	*signal* という名前のシグナルを無視します。 SIGKILL を捕獲したり無視したりすることはできません。
`ignore $(catch)`	すべてのシグナルを無視します。

ここで、

number は、シグナルの番号です。

signal は、シグナルの名前です。

import コマンド

`import` コマンドは、dbx コマンドライブラリからコマンドをインポートします。

構文

`import` *pathname*	`dbx` コマンドライブラリ *pathname* からコマンドをインポートします。

ここで、

pathname は、`dbx` コマンドライブラリのパス名です。

intercept コマンド

`intercept` コマンドは、指定タイプ (C++のみ) の (C++例外) を送出します。 一致するものがない送出例外は、「処理されない」送出と呼ばれます。 送出元関数の例外仕様に一致しない送出例外は、「予期されない」送出と呼ばれます。

処理されない送出と予期されない送出は、デフォルト時に阻止されます。

構文

`intercept` *typename*	型 *typename* の送出を阻止します。
`intercept -a`	すべての送出を阻止します。
`intercept -x` *typename*	*typename* の阻止を行いません。
`intercept -a -x` *typename*	*typename* 以外の型すべてを阻止します。
`intercept`	阻止対象の型を一覧表示します。

ここで、

typename には、-unhandled または -unexpected を指定できます。

kill コマンド

`kill` コマンドはプロセスにシグナルを送り、ターゲットプロセスを終了します。

構文

`kill -l`	既知の全シグナルの番号、名前、説明を一覧表示します。
`kill`	制御対象プロセスを終了します。
`kill` *job*`...`	一覧表示されているジョブに SIGTERM シグナルを送ります。
`kill -`*signal* *job*`...`	一覧表示されているジョブに指定のシグナルを送ります。

ここで、

job としてプロセスIDを指定するか、または次のいずれかの方法で指定します。

%+	現在のジョブを終了します。
%-	直前のジョブを終了します。
%*number*	*number* の番号を持つジョブを終了します。
%*string*	*string* で始まるジョブを終了します。
%?*string*	*string* を含んでいるジョブを終了します。

signal は、シグナルの名前です。

`language` コマンド

`language` コマンドは、現在のソース言語の表示や変更を行います。

構文

`language`	式の解析と評価に使用する現在の言語の名前を出力します。
`language` *language*	現在の言語として *language* を設定します。

ここで、

language は、c、ansic、c++, pascal、fortran、または fortran90 です。

注 – cは、ansicの別名です。

---

line コマンド

line コマンドは、現在の行番号の表示や変更を行います。

構文

`line`	現在の行番号を表示します。
`line` *number*	現在の行番号として number を設定します。
`line "`*filename*`"`	現在の行番号として行1を *filename* に設定します。
`line "`*filename*`":`*number*	現在の行番号として行 *number* を *filename* に設定します。

ここで、

filename は、変更対象の行番号があるファイルの名前です。 ファイル名を囲んでいる "“ は省略可能です。

number は、ファイル内の行の番号です。

例

```
line 100
```

```
line "/root/test/test.cc":100
```

list コマンド

list コマンドは、ソースファイルの行を表示します。 デフォルト表示行数Nは、dbx環境変数 output_list_size によって制御されます。

構文

`list`	N行を一覧表示します。
`list` *number*	行番号 *number* を表示します。
`list +`	次の N行を一覧表示します。
`list +`*n*	次の *n* 行を一覧表示します。
`list -`	直前の N行を一覧表示します。
`list -`*n*	直前の *n* 行を一覧表示します。
`list` *n1*`,`*n2*	*n1* から *n2* までの行を一覧表示します。
`list` *n1*`,+`	*n1* から *n1* + N までを一覧表示します。
`list` *n1*`,+`*n2*	*n1* から *n1* + *n2* までを一覧表示します。
`list` *n1*`,-`	*n1*-N から *n1* までを一覧表示します。
`list` *n1*`,-`*n2*	*n1*-*n2* から *n1* までを一覧表示します。
`list` *function*	*function* のソースの先頭を表示します。`list` *function* は、現在のスコープを変更します。 詳細については、46 ページの「スコープ」 を参照してください。
`list` *filename*	ファイル *filename* の先頭を表示します。
`list` *filename*`:`*n*	ファイル *filename* を行 *n* から表示します。 ファイルの末尾行を示す '$' を行番号の代わりに使用できます。 コンマは省略可能です。

ここで、

filename は、ソースコードファイルの名前です。

function は、表示対象の関数の名前です。

number は、ソースファイル内の行の番号です。

*n*は、表示対象の行数です。

n1 は、最初に表示する行の番号です。.

n2 は、最後に表示する行の番号です。

オプション

-i または -instr	ソース行とアセンブリコードを混合します。
-w または -w*n*	行または関数のまわりの N (または *n*) 行を一覧表示します。 このオプションを '+' or '-' 構文と併用したり2つの行番号が指定されているときに使用したりすることはできません。

例

```
list                        // 現在の行を先頭とするN行を一覧表示する
list +5                     // 現在の行を先頭とする5行を一覧表示する
list -                      // 直前のN行を一覧表示する
list -20                    // 直前の20行を一覧表示する
list 1000                   // 行1000を表示する
list 1000,$                 // 行1000から末尾行までを一覧表示する
list 2737 +24               // 行2737と次の24行を一覧表示する
list 1000 -20               // 行980から1000までを一覧表示する
list "test.cc":33           // ファイル"test.cc"のソース行33を表示する
list -w                     // 現在行のまわりのN行を一覧表示する
list -w8 `test.cc`func1     // 関数func1のまわりの8行を一覧表示する
list -i 500 +10           //
                              行500から510までを一覧表示する
```

listi コマンド

listi コマンドは、ソース命令と逆アセンブリされた命令を表示します。

詳細については、313 ページの「list コマンド」 を参照してください。

loadobject コマンド

loadobject コマンドは、現在のロードオブジェクトの名前を出力します。

構文

loadobject	現在のロードオブジェクトの名前を出力します。

loadobjectsコマンド

loadobjects は、現在 dbx によって読み込まれているロードオブジェクトを一覧表示します。 ライブラリの自動読み込みに関する動作の一部も制御します。

ロードオブジェクトの一覧表示

`loadobjects [ -a ] [ -v ]`

オプションを付けないで `loadobjects` コマンドを使用すると、現在dbxによって読み込まれているロードオブジェクトすべてが一覧表示されます。 `-v` は、各ロードオブジェクト′のアドレスの下限と上限を冗長モードで出力します。 `-a` は、全ロードオブジェクトに関する情報を出力します。 接頭辞の凡例は、次のとおりです。

m	mapped
u	unmapped
c	cached
x	excluded
*	unexcludable

自動読み込みの制御

`loadobjects [-x | -i | -r | -p | -u ] [ lo-list | -f lo-list-file ]`

-fオプションは、`-x`、`-i`、`-r`、`-p`、または `-u` の後に指定する必要があります。

`loadobjects -x LOs`

除外リストにロードオブジェクトを入れます。

`loadobjects -x`

除外リストにロードオブジェクトを表示します。

`loadobjects -i LOs`

除外リストからロードオブジェクトを外します。つまり、ロードオブジェクトを含めます。

`loadobjects -i`

すべてのロードオブジェクトを除外リストから外します。

`loadobjects -r LOs`

指定ロードオブジェクトのシンボルテーブルの読み込みを強制します。

`loadobjects -r`

全除外ロードオブジェクトのシンボルテーブルの読み込みを強制します。

`loadobjects -p LOs`

指定ロードオブジェクトをあらかじめ読み込みます。

`loadobjects -p`

あらかじめ読み込まれているロードオブジェクトを一覧表示します。

`loadobjects -u` *LOs*
あらかじめ読み込まれているリストからロードオブジェクトを外します。つまり、アンロードします。

`loadobjects -u`
あらかじめ読み込まれるリストからすべてのロードオブジェクトを外します。

ここで、

LOs は1個または複数の共有オブジェクトライブラリ名または構文 `-f` *filename* を指します。*filename* には、1行に1個のロードオブジェクトが入ったロードオブジェクトリストが入っています。 たとえば、

```
loadobjects -x libX.so libXm.so
loadobjects -p -f los_that_my_app_dlopens
```

ロードオブジェクト名には、パスやバージョン番号の接尾辞を使用しません。たとえば、`/usr/lib/libintl.so.1`ではなく `libintl.so` とします。 この点は、ファイル内のロードオブジェクトにも適用されます。

lwp コマンド

`lwp` コマンドは、現在のLWP (軽量プロセス) の表示や変更を行います。

構文

`lwp`
現在のLWP を表示します。

`lwp` *lwp_id*
LWP *lwp_id* に切り替えます。

ここで、

lwp_id 軽量プロセスの識別子です。

lwps コマンド

`lwps` コマンドは、プロセス内のLWP (軽量プロセス) すべてを一覧表示します。

構文

`lwps`	現在のプロセス内の LWP すべてを一覧表示します。

mmapfile コマンド

`mmapfile` コマンドは、コアダンプに存在しないメモリーマップファイルの内容を表示します。

Solarisコアファイルには、読み取り専用のメモリーセグメントは含まれていません。実行可能な読み取り専用セグメント (つまりテキスト) は自動的に処理され、`dbx` は、実行可能ファイルと関連する共有オブジェクトを調べることによってこれらのセグメントに対するメモリーアクセスを解釈処理します。

構文

`mmapfile` *mmapped_file* *address offset length*	コアダンプに存在しないメモリーマップファイルの内容を表示します。

ここで、

mmapped_file は、コアダンプ中にメモリーマップされたファイルのファイル名です。

address は、プロセスのアドレス空間の開始アドレスです。

length は、表示対象アドレス空間のバイト単位による長さです。

offset は、*mmapped_file* の開始アドレスまでのバイト単位によるオフセットです。

module コマンド

`module` コマンドは、1個または複数のモジュールのデバッグ情報を読み込みます。

構文

`module [-v]` 現在のモジュールの名前を出力します。

`module [-f] [-v] [-q]` *name* というモジュールのデバッグ情報を読み込みます。 *name*

`module [-f] [-v] [-q]` 全モジュールのデバッグ情報を読み込みます。 `-a`

ここで、

name は、読み込み対象のデバッグ情報が関係するモジュールの名前です。

`-a` は、すべてのモジュールを指定します。

`-f` は、実行可能ファイルより新しいファイルの場合でもデバッグ情報の読み込みを強制します (注意して使用のこと！)。

`-v` は、言語、ファイル名などを出力する冗長モードを指定します。

`-q` は、静止モードを指定します。

例

読み取り専用データセグメントは、アプリケーションメモリーがデータベースをマップしたときに通常発生します。 たとえば、

```
caddr_t vaddr = NULL;
off_t offset = 0;
size_t = 10 * 1024;
int fd;
fd = open("../DATABASE", ...)
vaddr = mmap(vaddr, size, PROT_READ, MAP_SHARED, fd, offset);
index = (DBIndex *) vaddr;
```

デバッガによってメモリーとしてデータベースにアクセスできるようにするには、以下を入力します。

```
mmapfile ../DATABASE $[vaddr] $[offset] $[size]
```

ここで、以下を入力すれば、データベースの内容を構造的に表示させることができます。

```
print *index
```

modules コマンド

modules コマンドは、モジュール名を一覧表示します。

構文

`modules [-v]`	すべてのモジュールを一覧表示します。
`modules [-v] -debug`	デバッグ情報が入っているモジュールすべてを一覧表示します。
`modules [-v] -read`	すでに読み込まれたデバッグ情報が入っているモジュールの名前を表示します。

ここで、

-v は、言語、ファイル名などを出力する冗長モードを指定します。

next コマンド

next コマンドは、1ソース行をステップ実行します (呼び出しをステップオーバー)。dbx環境変数 step_events は、ステップ実行中にブレークポイントが使用可能であるかどうかを制御します。

構文

`next`	1行をステップ実行します (呼び出しをステップオーバー)。関数呼び出しがステップオーバーされるマルチスレッドプログラムの場合、デッドロック状態を避けるため、その関数呼び出し中は全 LWP (軽量プロセス) が暗黙に再開されます。 非活動状態のスレッドをステップ実行することはできません。
`next` *n*	*n* 行をステップ実行します (呼び出しをステップオーバー)。
`next ... -sig` *signal*	ステップ実行中に指定のシグナルを引き渡します。
`next ...` *thread_id*	指定のスレッドをステップ実行します。
`next ...` *lwp_id*	指定の LWP をステップ実行します。 関数をステップオーバーしたときに全 LWPを暗黙に再開しません。

ここで、

*n*は、ステップ実行対象の行数です。

signal は、シグナルの名前です。

thread_id は、スレッド識別子です。

lwp_id は、LWP識別子です。

明示的な *thread_id* または *lwp_id* が指定されている場合、`next` コマンドによる汎用のデッドロック回避策は無効となります。

マシンレベルの呼び出しステップオーバーについては、322 ページの「nexti コマンド」 も参照してください。

注 – 軽量プロセス (LWP) については、Solaris Multithreaded Programming Guideを参照してください。

`nexti` コマンド

`nexti` コマンドは、1 マシン命令をステップ実行します (呼び出しをステップオーバー)。

構文

`nexti`	マシン命令1個をステップ実行します (呼び出しをステップオーバー)。
`nexti` *n*	*n* 行をステップ実行します (呼び出しをステップオーバー)。
`nexti -sig` *signal*	ステップ実行中に指定のシグナルを引き渡します。
`nexti ...` *lwp_id*	指定の LWP をステップ実行します。
`nexti ...` *thread_id*	指定のスレッドが活動状態である LWP をステップ実行します。 関数をステップオーバーしたときに全 LWP を暗黙に再開しません。

ここで、

*n*は、ステップ実行対象の命令数です。

signal は、シグナルの名前です。

thread_id は、スレッド IDです。

lwp_id は、LWP IDです。

pathmap コマンド

pathmap コマンドは、ソースファイルを検索する場合などに1つのパス名を別のパス名にマッピングします。マッピングは、ソースパス、オブジェクトファイルパス、および現在の作業用ディレクトリ (-c を指定した場合) に適用されます。

pathmap コマンドは、さまざまなホスト上に存在するさまざまなパスを持つ、オートマウントされた明示的な NFS マウント済みファイルシステムを取り扱うときに便利です。 オートマウントされたファイルシステムにおける CWD も不正確であるため、オートマウンタが原因である問題を解決する際には、-c を指定します。 pathmap コマンドは、ソースツリーやビルドツリーを移動した場合にも便利です。

デフォルトの場合、pathmap /tmp_mnt /が存在します。

pathmap コマンドは、dbx 環境変数 core_lo_pathmap が on に設定されているときにロードオブジェクトを検索します。 上記の場合以外では、pathmap コマンドはロードオブジェクト (共有ライブラリ) の検索に対して効果がありません。 17 ページの「一致しないコアファイルのデバッグ」 を参照してください。

構文

`pathmap [ -c ] [-`*index*`] `*from*	*from* から *to* への新たなマッピングを作成します。
from to	
`pathmap [ -c ] [-`*index*`]`	すべてのパスを *to* にマッピングします。
`to`	
`pathmap`	既存のパスマッピングすべてを一覧表示します (インデックス別に)。
`pathmap -s`	上記と同じですが、出力をdbx によって読み込むことができます。
`pathmap -d` *from1*	指定のマッピングをパス単位で削除します。
from2...	
`pathmap -d` *index1 index2*	指定のマッピングをインデックス単位で削除します。
`...`	

ここで、

from と *to* は、ファイルパス接頭辞です。*from* は実行可能ファイルやオブジェクトファイルにコンパイルされたファイルパス、*to* はデバッグ時におけるファイルパスを示します。

from1 は、最初に削除するマッピングのファイルパスです。

from2 は、最後に削除するマッピングのファイルパスです。

index は、マッピングをリストに挿入する際に使用するインデックスを指定します。 インデックスを指定しなかった場合、リスト末尾にマッピングが追加されます。

index1 は、最初に削除するマッピングのインデックスです。

index2 は、最後に削除するマッピングのインデックスです。

`-c` を指定すると、現在の作業用ディレクトリにもマッピングが適用されます。

`-s` を指定すると、dbx が読み込める出力形式で既存のマッピングがリストされます。

`-d` を指定すると、指定のマッピングが削除されます。

例

```
(dbx) pathmap /export/home/work1 /net/mmm/export/home/work2

  # /export/home/work1/abc/test.c を
/net/mmm/export/home/work2/abc/test.c へマップする
```

```
(dbx) pathmap /export/home/newproject
  # /export/home/work1/abc/test.c を
/export/home/newproject/test.c へマップする
(dbx) pathmap
(1) -c /tmp_mnt /
(2) /export/home/work1 /net/mmm/export/home/work2
(3) /export/home/newproject
```

pop コマンド

pop コマンドは、1 個または複数のフレームを呼び出しスタックから削除します。

-g を使ってコンパイルされた関数の場合、フレームにポップできるだけです。 プログラムカウンタは、呼び出し場所におけるソース行の先頭にリセットされます。 デバッガによる関数呼び出しを越えてポップすることはできません。pop -c を使用してください。

通常、pop コマンドはポップ対象フレームに関するC++デストラクタをすべて呼び出します。dbx 環境変数 pop_auto_destruct を off に設定すれば、この動作を変更できます (22 ページの「dbx 環境変数の設定」参照)。

構文

pop	現在のトップフレームをスタックからポップします。
pop *number*	*number* 個のフレームをスタックからポップします。
pop -f *number*	指定のフレーム *number* までフレームをスタックからポップします。
pop -c	デバッガが行った最後の呼び出しをポップします。

ここで、

number は、スタックからポップするフレームの数です。

print コマンド

print コマンドは、式の値を出力します。

構文

`print` *expression*`, ...`	式 *expression, ...*の値を出力します。
`print -r` *expression*	継承メンバーを含み、式*expression* の値を出力します (C++ のみ)。
`print +r` *expression*	`dbx` 環境変数 `output_inherited_members` が on であるときは、継承メンバーを出力しません (C++ のみ)。
`print -d [-r]` *expression*	式 *expression* の静的型ではなく動的型を表示します (C++のみ)。
`print +d [-r]` *expression*	`dbx` 環境変数 `output_dynamic_type` が on であるときは、式*expression* の動的型を使用しません (C++のみ)。
`print -p` *expression*	prettyprint 関数を呼び出します。
`print +p` *expression*	`dbx` 環境変数 `output_pretty_print` が on であるときは、prittyprint 関数を呼び出しません。
`print -l` *expression*	('Literal') 左側を出力しません。 式が文字列である場合 (`char *`)、アドレスの出力は行わず、文字列内の文字だけを引用符なしで出力します。
`print -f`*format expression*	整数、文字列、浮動小数点の式の形式として *format* を使用します (オンラインヘルプの Output Formats 参照)。
`print -F`*format expression*	指定の形式を使用しますが、左側 (変数名や式) は出力しません (オンラインヘルプの Output Formats 参照)。
`print -o` *expression*	*expression* の値を出力します。これは、序数としての列挙式でなければなりません。ここでは、形式文字列を使用することもできます (`-f`*format*)。 非列挙式の場合、このオプションは無視されます。
`print --` *expression*	'--' は、フラグ引数の終わりを示します。 これは、*expression* がプラスやマイナスで始まる可能性がある場合に便利です (スコープ解釈処理ルールについては、46 ページの「スコープ」 を参照してください。 `print` コマンドの出力先を切り替えてトレーリングコメントを追加するには、オンラインヘルプの Redirection を参照してください)。

ここで、

expression は、出力対象の値を持つ式です。

format は、式の出力時に使用する形式です。

prog コマンド

prog コマンドは、デバッグ中のプログラムとその属性を管理します。

構文

`prog -readsyms`	据え置きされていたシンボリック情報を、dbx 環境変数 run_quick を on に設定することによって読み込みます。
`prog -executable`	-の使用がプログラムに設定されている場合、実行可能ファイルのフルパス- を出力出力します。
`prog -argv`	argv[0] を含む argv 全体を出力します。
`prog -args`	argv[0] を含まない argv を出力します。
`prog -stdin`	< *filename* を出力します。stdin が使用されている場合は、空にします。
`prog -stdout`	> *filename* または >> *filename* を出力します。stdout が使用されている場合は、空にします。 -args、-stdin、-stdout の出力は、組み合わせて run コマンドで使用できるようになっています (332 ページの「run コマンド」参照)。

quit コマンド

quit コマンドは、dbxを終了します。

dbx がプロセスに接続されている場合、このプロセスを切り離してから終了が行われます。 保留状態のシグナルは取り消されます。 微調整を行うには、detach コマンドを使用します (300 ページの「detach コマンド」参照)。

構文

`quit`	リターンコード0を出力してdbx を終了します。*exit*と同じです。
`quit` *n*	リターンコード*n*を出力して終了します。 `exit` *n*と同じです。

ここで、

n は、リターンコードです。

regs コマンド

`regs` コマンドは、レジスタの現在値を出力します。

構文

```
regs [-f] [-F]
```

ここで、

`-f` には、浮動小数点レジスタ (単精度) が含まれます (SPARCプラットフォームのみ)。

`-F` には、浮動小数点レジスタ (倍精度) が含まれます (SPARCプラットフォームのみ)。

例 (SPARCプラットフォーム)

```
dbx[13] regs -F
現スレッド: t@1
現フレーム:  [1]
g0-g3             0x00000000 0x0011d000 0x00000000 0x00000000
g4-g7             0x00000000 0x00000000 0x00000000 0x00020c38
o0-o3             0x00000003 0x00000014 0xef7562b4 0xefffff420
o4-o7             0xef752f80 0x00000003 0xefffff3d8 0x000109b8
l0-l3             0x00000014 0x0000000a 0x0000000a 0x00010a88
l4-l7             0xefffff438 0x00000001 0x00000007 0xef74df54
```

```
i0-i3             0x00000001 0xeffff4a4 0xeffff4ac 0x00020c00
i4-i7             0x00000001 0x00000000 0xeffff440 0x000108c4
y                 0x00000000
psr               0x40400086
pc                0x000109c0:main+0x4mov     0x5, %l0
npc               0x000109c4:main+0x8st      %l0, [%fp - 0x8]
f0f1              +0.00000000000000e+00
f2f3              +0.00000000000000e+00
f4f5              +0.00000000000000e+00
f6f7              +0.00000000000000e+00
```

replay コマンド

replay コマンドは、最後の run、rerun、または debug コマンド以降のデバッグコマンドを再現します。

構文

replay [-*number*] 最後のrun コマンド、rerun コマンド、または debug コマンド以降のコマンドすべてを再現するか、またはそれらのコマンドから *number* 個のコマンドを差し引いたコマンドを再現します。

ここで、

number は、再現しないコマンドの数です。

rerun コマンド

rerun コマンドは、引数を付けないでプログラムを実行します。

構文

rerun	引数を付けないでプログラムの実行を開始します。
rerun *arguments*	`save` コマンドで新しい引数を付けてプログラムの実行を開始します (333 ページの「save コマンド」参照)。

`restore` コマンド

`restore` コマンドは、以前に保存されていた状態に dbx を復元します。

構文

restore [*filename*]	保存されていたときの状態に`dbx` を復元します。

ここで、

filename は、最後の `run` コマンド、`rerun` コマンド、または `debug` コマンドが保存されてから実行された `dbx` コマンドの実行対象ファイルの名前です。

`rprint` コマンド

`rprint` コマンドは、シェル引用規則を使用して式を出力します。

構文

`rprint [-r|+r|-d|+d|` `-p|+p|-l|-f`*format*`|` `-F`*format*`|--] `*expression*	式の値を出力します。 特別な引用規則は適用されないので、`rprint a > b` の場合、`a` の値 (存在する場合) がファイル`b` に入れられます (フラグの意味については326 ページの「print コマンド」参照)。

ここで、

expression は、出力対象の値を持つ式です。

format は、式の出力時に使用する形式です。

run コマンド

run コマンドは、引数を付けてプログラムを実行します。

Control-C を使用すると、プログラムの実行が停止します。

構文

run	現在の引数を付けてプログラムの実行を開始します。
run *arguments*	新規の引数を付けてプログラムの実行を開始します。
run ... > | >> *input_file*	出力先の切り替えを設定します。
run ... < *output_file*	入力元の切り替えを設定します。

ここで、

arguments は、ターゲットプロセスの実行時に使用する引数です。

input_file は、入力元ファイルの名前です。

output_file は、出力先ファイルの名前です。

注 – 現在、run コマンドや runargs コマンドによって stderr の出力先を切り替えることはできません。

---

runargs コマンド

runargs コマンドは、ターゲットプロセスの引数を変更します。

ターゲットプロセスの現在の引数を調べるには、引数を付けないで debug コマンドを使用します (298 ページの「debug コマンド」参照)。

構文

`runargs` *arguments*	runコマンドで使用する現在の引数を設定します (332 ページの「run コマンド」参照)。
`runargs ... >|>>` *file*	run コマンドで使用する出力先を設定します。
`runargs ... <` *file*	run コマンドで使用する入力元を設定します。
`runargs`	現在の引数をクリアします。

ここで、

arguments は、ターゲットプロセスの実行時に使用する引数です。

file は、ターゲットプロセスからの出力またはターゲットプロセスへの入力の切り替え先です。

save コマンド

save コマンドは、コマンドをファイルに保存します。

構文

`save [-`*number*`] [`*filename*`]`	最後の run コマンド、rerun コマンド、または debug コマンド以降のコマンドすべて、またはそれらのコマンドから*number*個のコマンドを差し引いたコマンドを、デフォルトファイルまたは*filename*に保存します。

ここで、

number は、保存しないコマンドの数です。

filename は、最後の runコマンド、rerun コマンド、または debug コマンドの後に実行されるdbx コマンドを保存するファイルの名前です。

scopes コマンド

scopes コマンドは、活動状態にあるスコープのリストを出力します。

構文

```
scopes
```

search コマンド

search コマンドは、現在のソースファイルにおいて順方向検索を行います。

構文

`search` *string*	現在のファイルの中で、*string* を順方向で検索します。
`search`	最後の検索文字列を使用して検索を繰り返します。

ここで、

string は、検索対象の文字列です。

showblock コマンド

showblock コマンドは、特定のヒープブロックが割り当てられた場所を示す実行時検査結果を表示します。

メモリー使用状況検査やメモリーリーク検査がオンになっているときに showblock コマンドを使用すると、指定アドレスのヒープブロックに関する詳細が表示されます。 ブロックの割り当て場所とそのサイズが示されます。 284 ページの「check コマンド」を参照してください。

構文

```
showblock -a address
```

ここで、

address は、ヒープブロックのアドレスです。

showleaks コマンド

showleaks コマンドは、最後の showleaks コマンド実行後のメモリーリークについて報告します。 デフォルトの簡易形式では、1行に1つのリークレコードを示すレポートが出力されます。 実際に発生したリークの後に、発生する可能性のあるリークが報告されます。 リークの合計サイズに基づいてレポートがソートされます。

構文

```
showleaks [-a] [-m m] [-n number] [-v]
```

ここで、

-a は、これまでに発生したリークすべてを表示します (最後の showleaks コマンドを実行した後のリークだけではなく)。

-m *m* は、複数のリークをまとめます。2 個以上のリークに対する割り当て時の呼び出しスタックが *m* 個のフレームに一致するとき、これらのリークは1つのリークレポートにまとめて報告されます。 -m オプションを指定すると、check コマンドで指定した*m* の大域値が無効となります (284 ページの「check コマンド」参照)。 *m* のデフォルト値は 2 または check コマンドで最後に指定した大域値です (指定されている場合)。

-n *number* は、最大 *number* 個のレコードをレポートに表示します。 デフォルトの場合、すべてのレコードが表示されます。

-v 冗長出力を生成します。 デフォルトの場合、簡易出力が表示されます。

showmemuse コマンド

showmemuse コマンドは、最後の showmemuse コマンド実行後に使用したメモリーを表示します。 1行に1つの「使用中ブロック」を示すレコードが出力されます。 このコマンドは、ブロックの合計サイズに基づいてレポートをソートします。 最後の showleaks (335 ページの「showleaks コマンド」参照) コマンド実行後にリークしたブロックもレポートに含まれます。

構文

showmemuse [-a] [-m <m>] [-n *number*] [-v]

ここで、

-a は、使用中ブロックすべてを表示します(最後の showmemuse コマンド実行後のブロックだけではなく)。

-m *m* は、使用中ブロックレポートをまとめます。 *m* のデフォルト値は2または check コマンドで最後に指定した大域値です (284 ページの「check コマンド」参照)。 2個以上のブロックに対する割り当て時の呼び出しスタックが*m* 個のフレームに一致するとき、これらのブロックは1つのレポートにまとめて報告されます。 -m オプションを使用すると、*m*の大域値が無効となります。

-n *number* は、最大*number*個のレコードをレポートに表示します。 デフォルト値は20です。-v は、冗長出力を生成します。 デフォルトの場合、簡易出力が表示されます。

source コマンド

sourceコマンドは、指定ファイルからコマンドを実行します。

構文

source *filename* — ファイル *filename* からコマンドを実行します。 $PATH は検索されません。

status コマンド

status コマンドは、イベントハンドラ (ブレークポイントなど) を一覧表示します。

構文

`status`	活動中の trace、when、および stop ブレークポイントを出力します。
`status` *handler_id*	ハンドラ *handler_id* のステータスを出力します。
`status -h`	隠れているものを含み、活動中の trace、when、および stop ブレークポイントを出力します。
`status -s`	上記と同じですが、出力をdbx によって読み込むことができます。

ここで、

handler_id は、イベントハンドラの識別子です。

例

```
(dbx) status -s > bpts
...
(dbx) source bpts
```

step コマンド

step コマンドは、1 ソース行または1文をステップ実行します (呼び出しにステップイン)。 dbx 環境変数 step_events は、ステップ実行中にブレークポイントが使用可能であるかどうかを制御します。

構文

`step`	1行をシングルステップ実行します (呼び出しにステップイン)。 関数呼び出しがステップオーバーされるマルチスレッドプログラムの場合、デッドロック状態を避けるため、その関数呼び出し中は全 LWP (軽量プロセス) が暗黙に再開されます。 非活動状態のスレッドをステップ実行することはできません。
`step` *n*	*n* 行をシングルステップ実行します (呼び出しへのステップイン)。
`step up`	ステップアップし、現在の関数から出ます。
`step ... -sig` *signal*	ステップ実行中に指定のシグナルを引き渡します。
`step ...` *thread_id*	指定のスレッドをステップ実行します。 `step up` には適用されません。
`step ...` *thread_id*	指定の LWP をステップ実行します。 関数をステップオーバーしたときに全 LWPを暗黙に再開しません。
`step to` [*func*]	現在のソースコード行の *func* へのステップインを試行します。 *func* が指定されていない場合、現在のソースコード行のアセンブリコードに従って呼び出された最後の関数へのステップインが試行されます。

ここで、

n は、ステップ実行対象の行数です。

signal は、シグナルの名前です。

thread_id は、スレッド IDです。

lwp_id は、LWP IDです。

func は、関数名です。

明示的な *thread_id* または *thread_id* が指定されている場合、`step` コマンドによる汎用のデッドロック回避策は無効となります。

`step to` コマンドを実行した際、最後のアセンブル呼び出し命令へのステップインや現在のソースコード行の関数 (指定されている場合) へのステップインが試行されている間、条件付き分岐があると呼び出しが受け付けられないことがあります。 呼び出しが受け付けられない場合や現在のソースコード行に関数呼び出しがない場合、`step to` コマンドが現在のソースコード行をステップオーバーします。 `step to`コマンドを使用する際は、ユーザー定義演算子にとくに注意してください。

マシンレベルの呼び出しステップ実行については、339 ページの「stepi コマンド」 も参照してください。

stepi コマンド

nexti コマンドは、1マシン命令をステップ実行します (呼び出しにステップイン)。

構文

`stepi`	1つのマシン命令をシングルステップ実行します (呼び出しにステップイン)。
`stepi` *n*	*n* 個のマシン命令をシングルステップ実行します (呼び出しへのステップイン)。
`stepi -sig` *signal*	ステップ実行し、指定のシグナルを引き渡します。
`stepi ...` *lwp_id*	指定の LWP をステップ実行します。
`stepi ...` *thread_id*	指定のスレッドが活動状態である LWP をステップ実行します。

ここで、

n は、ステップ実行対象の命令数です。

signal は、シグナルの名前です。

lwp_id は、LWP IDです。

thread_id は、スレッド IDです。

stop コマンド

stop コマンドは、ソースレベルのブレークポイントを設定します。

構文

stop コマンドの一般構文は、次のとおりです。

`stop` *event-specification* [*modifier*]

指定イベントが発生すると、プロセスが停止されます。

次の構文が有効です。

`stop [ -update ]`	実行をただちに停止します。 whenコマンドの本体内でのみ有効です。
`stop -noupdate`	`-update` と同じですが、Sun WorkShop Debugging 表示の更新は行いません。
`stop at` *line*	実行を指定行で停止します。 73 ページの「関数に stop ブレークポイントを設定する」 も参照してください。
`stop in` *function*	*function* が呼び出されたときに実行を停止します。 73 ページの「関数に stop ブレークポイントを設定する」 も参照してください。
`stop inclass` *classname*	C++のみ： class/struct/union/template のいずれかのクラスのメンバー関数すべてにブレークポイントを設定します。 75 ページの「同じクラスのメンバー関数にブレークポイントを設定する」 も参照してください。
`stop inmember` *name*	C++のみ：すべてのメンバー関数 *name* にブレークポイントを設定します。75 ページの「同じクラスのメンバー関数にブレークポイントを設定する」 も参照してください。
`stop infunction` *name*	C++のみ：すべての非メンバー関数 *name* にブレークポイントを設定します。

ここで、

line は、ソースコード行の番号です。

signal は、シグナルの名前です。

classname は、C++のclass、struct、union、または template クラスの名前です。

name は、C++関数の名前です。

マシンレベルのブレークポイントの設定については、341 ページの「stopi コマンド」も参照してください。

全イベントのリストと構文については、260 ページの「イベント指定の設定」を参照してください。

stopi コマンド

stopi コマンドは、マシンレベルのブレークポイントを設定します。

構文

stopi コマンドの一般構文は、次のとおりです。

stopi *event-specification* [*modifier*]

指定イベントが発生すると、プロセスが停止されます。

次の構文が有効です。

stopi at *address*	*address* の場所で実行を停止します。
stopi in *function*	*function* が呼び出されたときに実行を停止します。

ここで、

address は、アドレスとなった式またはアドレスとして使用可能な式です。

function は、関数の名前です。

全イベントのリストと構文については、260 ページの「イベント指定の設定」を参照してください。

suppress コマンド

suppress コマンドは、実行時検査中のメモリーエラーの報告を抑止します。 dbx 環境変数 rtc_auto_suppress がonである場合、指定場所におけるメモリーエラーは1度だけ報告されます。

構文

`suppress`	`suppress` コマンドと `unsuppress` コマンドの履歴 (`-d` オプションと `-reset` オプションを指定するものは含まない)。
`suppress -d`	デバッグ用にコンパイルされなかった関数で抑止されているエラーのリスト (デフォルト抑止)。 このリストは、ロードオブジェクト単位です。 これらのエラーの抑止を解除する方法は、`-d` オプションを付けて unsuppress を使用することだけです。
`suppress -d` *errors*	*errors* をさらに抑止することによって、全ロードオブジェクトに対するデフォルト抑止を変更します。
`suppress -d` *errors* `in` *loadobjects*	*errors* をさらに抑止することによって、*loadobjects* のデフォルト抑止を変更します。
`suppress -last`	エラー位置における現在のエラーを抑止します。
`suppress -reset`	デフォルト抑止としてオリジナルの値を設定します (起動時)。
`suppress -r <id> ...`	識別子によって指定される抑止解除イベントを削除します (識別子は '(un)suppress' を行うと取得できます)。
`suppress -r 0 | all | -all`	unsuppress コマンドによって指定される抑止解除イベントすべてを削除します。
`suppress` *errors*	あらゆる場所における *errors* を抑止します。
`suppress` *errors* `in` [*functions*] [*files*] [*loadobjects*]	*functions* リスト、*files* リスト、*loadobjects* リストにおける *errors* を抑止します。
`suppress` *errors* `at` *line*	*line* における *errors* を抑止します。
`suppress` *errors* `at` "*file*" `:`*line*	*file* の *line* における *errors* を抑止します。
`suppress` *errors* `addr` *address*	*address* における *errors* を抑止します。

ここで、

address は、メモリーアドレスです。

errors は空白文字で区切られた以下の要素で構成されます。

all	すべてのエラー
`aib`	メモリーリークの可能性 - ブロック中のアドレス
`air`	メモリーリークの可能性 - レジスタ中のアドレス
`baf`	不正解放

`duf`	重複解放
`mel`	メモリーリーク
`maf`	境界整列を誤った解放
`mar`	境界整列を誤った読み取り
`maw`	境界整列を誤った書き込み
`oom`	メモリー不足
`rua`	非割り当てメモリーからの読み取り
`rui`	非初期化メモリーからの読み取り
`wro`	読み取り専用メモリーへの書き込み
`wua`	非割り当てメモリーへの書き込み
`biu`	ブロック使用状況 (割り当てられているメモリー) `biu` はエラーではありませんが、*errors* とまったく同じように suppress コマンドで使用できます。

file は、ファイルの名前です。

files は、1個または複数のファイル名です。

functions は、1個または複数の関数名です。

line は、ソースコード行の番号です。

loadobjects は、1個または複数のロードオブジェクト名です。

エラーの抑止解除については、350 ページの「unsuppress コマンド」 を参照してください。

sync コマンド

`sync` コマンドは、指定の同期オブジェクトに関する情報を表示します。

構文

`sync -info` *address*	address における同期オブジェクトに関する情報を表示します。

ここで、

address は、同期オブジェクトのアドレスです。

syncs コマンド

syncs コマンドは、同期オブジェクト (ロック) すべてを一覧表示します。

構文

```
syncs
```

thread コマンド

thread コマンドは、現在のスレッドの表示や変更を行います。

構文

`thread`	現在のスレッドを表示します。
`thread` *tid*	スレッド *tid* に切り替えます。

以下の構文で*tid* がない場合は、現在のスレッドが仮定されます。

`thread -info [` *tid* `]`	指定スレッドに関する既知情報すべてを出力します。
`thread -hide [` *tid* `]`	指定 (または現在の) スレッドを隠します。 このスレッドは、汎用スレッドリストに表示されません。
`thread -unhide [` *tid* `]`	指定 (または現在の) スレッドを隠ぺい解除します。
`thread -unhide all`	すべてのスレッドを隠ぺい解除します。
`thread -suspend` *thread_id*	指定スレッドが実行されないようにします。 中断されているスレッドは、スレッドリストに S の文字とともに表示されます。
`thread -resume` *thread_id*	`-suspend` の効果を解除します。
`thread -blocks [` *thread_id* `]`	他のスレッドをブロックしている指定スレッドが保持しているロックすべてを一覧表示します。
`thread -blockedby [` *thread_id* `]`	指定スレッドをブロックしている同期オブジェクトがある場合、そのオブジェクトを表示します。

ここで、

thread_id は、スレッドIDです。

threads コマンド

threads コマンドは、すべてのスレッドを一覧表示します。

構文

`threads`	既知のスレッドすべてのリストを出力します。
`threads -all`	通常出力されないスレッド (ゾンビ) を出力します。
`threads -mode all|filter`	全スレッドを出力するか、またはスレッドをフィルタリングするかを制御します。 デフォルトの場合、スレッドがフィルタリングされます。
`threads -mode auto|manual`	Sun WorkShop Debuggingでは、スレッドリスト表示の自動更新を使用可能にします。
`threads -mode`	現在のモードをエコーします。

trace コマンド

traceコマンドは、実行したソース行、関数呼び出し、変数の変更を表示します。

トレース速度は、dbx 環境変数 trace_speed によって設定します。

構文

traceコマンドの一般構文は、次のとおりです。

`trace` *event-specification* [*modifier*]

指定イベントが発生すると、トレースが出力されます。

次の構文が有効です。

`trace -file` *filename*	指定 *filename* に全トレース出力を送ります。トレース出力を標準出力に戻すには、*filename* として - を使用します。トレース出力は、必ず *filenam e*の後ろに付加されます。トレース出力は、dbx がプロンプト表示するたび、またアプリケーションが終了するたびにフラッシュされます。*filename* は、接続後の新規実行時や再開時に必ずオープンしなおされます。
`trace step`	各ソース行をトレースします。
`trace next -in` *function*	指定 *function* の中で各ソース行をトレースします。
`trace at` *line*	指定のソース *line* をトレースします。
`trace in` *function*	指定 *function* の呼び出しとこの関数からの戻り値をトレースします。
`trace inmember` *function*	*function* という名前のメンバー関数の呼び出しをトレースします。
`trace infunction` *function*	*function* という名前の関数が呼び出されるとトレースします。
`trace inclass` *class*	*class* のメンバー関数の呼び出しをトレースします。
`trace change` *variable*	*variable* の変更をトレースします。

ここで、

filename は、トレース出力の送信先ファイルの名前です。

function は、関数の名前です。

line は、ソースコード行の番号です。

class は、クラスの名前です。

variable は、変数の名前です。

全イベントのリストと構文については、260 ページの「イベント指定の設定」を参照してください。

tracei コマンド

tracei コマンドは、マシン命令、関数呼び出し、変数の変更を表示します。

tracei は、trace *event-specification* -instr の省略形です。 ここで、-instr 修飾子を指定すると、ソース行の細分性ではなく命令の細分性でトレースが行われます。

構文

`tracei step`	各ソース行をトレースします。
`tracei next -in` *function*	指定 *function* の中で各ソース行をトレースします。
`tracei at` *address*	*address* にあるソース行をトレースします。
`tracei in` *function*	指定 *function* の呼び出しとこの関数からの戻り値をトレースします。
`tracei inmember` *function*	*function* という名前のメンバー関数の呼び出しをトレースします。
`tracei infunction` *function*	*function* という名前の関数が呼び出されるとトレースします。
`tracei inclass` *class*	*class* のメンバー関数の呼び出しをトレースします。
`tracei change` *variable*	*variable* の変更をトレースします。

ここで、

filename は、トレース出力の送信先ファイルの名前です。

function は、関数の名前です。

line は、ソースコード行の番号です。

class は、クラスの名前です。

variable は、変数の名前です。

詳細については、345 ページの「trace コマンド」 を参照してください。

uncheck コマンド

uncheck コマンドは、メモリーのアクセス、リーク、使用状況の検査を使用不可にします。

構文

`uncheck`	検査の現在のステータスを出力します。
`uncheck -access`	アクセス検査を停止します。
`uncheck -leaks`	リーク検査を停止します。
`uncheck -memuse`	memuse 検査を停止します (リーク検査も停止されます)。
`uncheck -all`	`uncheck -access`、`uncheck -memuse` と同じです。
`uncheck` [*functions*] [*files*] [*loadobjects*]	*functions files loadobjects* に対する `suppress all` と同じです。

ここで、

functions は、1 個または複数の関数名です。

files は、1 個または複数のファイル名です。

loadobjects は、1個または複数のロードオブジェクト名です。

検査をオンにする方法については、284 ページの「check コマンド」を参照してください。

エラーの抑止については、341 ページの「suppress コマンド」 を参照してください。

実行時検査の概要については、112 ページの「すべての RTC を有効化」 を参照してください。

undisplay コマンド

`undisplay` コマンドは、`display` コマンドを取り消します。

構文

`undisplay` *expression*`, ...`	`display` *expression* コマンドを取り消します。
`undisplay` *n*`, ...`	*n* 個の`display` コマンドを取り消します。
`undisplay 0`	すべての`display` コマンドを取り消します。

ここで、

expression は、有効な式です。

unhide コマンド

unhide コマンドは、 hide コマンドを取り消します。

構文

`unhide 0`	すべてのスタックフレームフィルタを削除します。
`unhide` *regular_expression*	スタックフレームフィルタ *regular_expression* を削除します。
`unhide` *number*	スタックフレームフィルタ番号 *number* を削除します。

ここで、

regular_expression は、正規表現です。

number は、スタックフレームフィルタの番号です。

hideコマンド (309 ページの「hide コマンド」参照) は、番号を持つフィルタを一覧表示します。

unintercept コマンド

uninterceptコマンドは、intercept コマンドを取り消します (C++のみ)。

構文

`unintercept` *typename*	*typename* を intercept リストから削除します。
`unintercept -a`	すべての型を intercept リストから削除します。
`unintercept -x` *typename*	*typename* を intercept -x リストから削除します。
`unintercept -x -a`	すべての型を intercept -x リストから削除します。
`unintercept`	阻止対象の型を一覧表示します。

ここで、

typename には、-unhandled または -unexpectedを指定できます。

unsuppress コマンド

unsuppress コマンドは、suppressコマンドを取り消します。

構文

`unsuppress`	suppress コマンドとunsuppress コマンドの履歴 (`-d` オプションと `-reset` オプションを指定するものは含まない)。
`unsuppress -d`	デバッグのためのコンパイルが行われない関数で抑止解除されているエラーのリスト このリストは、ロードオブジェクト単位です。 エラーを抑止する方法は、`-d` オプションを付けて suppress コマンド (341 ページの「suppress コマンド」参照) を使用することだけです。
`unsuppress -d` *errors*	*errors* をさらに抑止解除することによって、全ロードオブジェクトに対するデフォルト抑止を変更します。
`unsuppress -d` *errors* `in` *loadobjects*	*errors* をさらに抑止解除することによって、*loadobjects* のデフォルト抑止を変更します。
`unsuppress -last`	エラー位置における現在のエラーを抑止解除します。
`unsuppress -reset`	デフォルト抑止マスクとしてオリジナルの値を設定します (起動時)。
`unsuppress` *errors*	あらゆる場所における *errors* を抑止解除します。
`unsuppress` *errors* `in` [*functions*] [*files*] [*loadobjects*]	*functions* リスト、*files* リスト、*loadobjects* リストにおける *errors* を抑止します。
`unsuppress` *errors* `at` *line*	*line* における *errors* を抑止解除します。
`unsuppress` *errors* `at` "*file*":*line*	*file* の *line* における *errors* を抑止解除します。
`unsuppress` *errors* `addr` *address*	*address* における *errors* を抑止解除します。

up コマンド

up コマンドは、呼び出しスタックを上方向に移動します (`main` に近づく)。

構文

up	呼び出しスタックを1レベル上方向に移動します。
up *number*	呼び出しスタックを *number* レベルだけ上方向に移動します。
up -h [*number*]	呼び出しスタックを上方向に移動しますが、隠しフレームをとばすことはしません。

ここで、

number は、呼び出しスタックレベルの数です。

use コマンド

use コマンドは、ディレクトリ検索パスの表示や変更を行います。

このコマンドは時代遅れなので、このコマンドは次の pathmap コマンドにマッピングしてあります。

use は、pathmap -s と同じです。

use *directory* は、 pathmap *directory*と同じです.。

vitem コマンド

vitem コマンドは、Sun WorkshopのData Visualizer のインタフェースです。

構文

`vitem -new` *array-expression*新たな視覚化項目を作成します。

`vitem <id> -replace` *array-expression*　既存項目を置換します。

`vitem <id>|all ... -update ontimer|onstop|ondemand`　指定項目の更新モードを設定します。

`vitem <id>|all ... -update now`　ondemand 項目をすべて再表示します。

`vitem <id>|all ... enable`　項目が指定されている場合、再表示を使用可能にします。

`vitem <id>|all ... -disable`　項目が指定されている場合、再表示を使用不可にします。

`vitem <id>|all -list`　既知項目をすべて一覧表示します。

`vitem <id>|all -list -s`　`source` コマンドで再読み込みできる様式で既知項目すべてを一覧表示します (336 ページの「source コマンド」参照)。

`vitem <id>|all -delete`　指定項目を削除します。

`vitem -timer` *seconds*　インターバルタイマーを設定します。dbx' のシングルインターバルタイマーは、`timer` イベントと共有されます (260 ページの「イベント指定の設定」 および 289 ページの「collector コマンド」参照)。 これらの機能のどれか1つを使用すると、他の機能が排除されます。

ここで、

array-expression は、グラフィック表示が可能な式です。

seconds は、秒数です。

whatis コマンド

`whatis` コマンドは、式の型または型の宣言を出力します。

構文

`whatis [-n] [-r]` *name* 型ではない *name* の宣言を出力します。
`whatis -t [-r]` *type* 型 *type* の宣言を出力します。
`whatis -e [-r] [-d]` 式 *expression* の型を出力します。
expression

ここで、

name は、型ではない名前です。

type は、型の名前です。

expression は、有効な式です。

`-d` は、静的型ではなく動的型を表示します (C++のみ)。

`-e` は、式の型を表示します。

`-n` は、型ではない宣言を表示します。 `-n` はオプションを付けないで `whatis` コマンドを使用したときのデフォルト値であるため、-nを指定する必要はありません。

`-r` は、基底クラスに関する情報を出力します (C++のみ)。

`-t` は、型の宣言を表示します。

C++のクラスや構造体に対して `whatis` コマンドを実行すると、定義済みメンバー関数すべて (未定義メンバー関数は除く)、静的データメンバー、クラスのフレンド、およびそのクラス内で明示的に定義されているデータメンバーのリストが表示されます。

`-r` (recursive) オプションを指定すると、継承クラスからの情報が追加されます。

`-d` フラグを `-e` フラグを併用すると、式の動的型が使用されます。

C++の場合、テンプレート関係の識別子は次のように表示されます。

- テンプレート定義は `whatis -t` によって一覧表示されます。
- 関数テンプレートのインスタンス化は、`whatis` によって一覧表示されます。
- クラステンプレートのインスタンス化は、`whatis -t` によって一覧表示されます。

when コマンド

when コマンドは、指定イベント発生時にコマンドを実行します。

構文

when コマンドの一般構文は、次のとおりです。

when *event-specification* [*modifier*]{ *command* ... ; }

指定イベントが発生すると、コマンドが実行されます。

次の構文が有効です。

when at *line* { *command*; } — *line* に到達したら、*command*(s) を実行します。

when in *procedure* { *command*; } — *procedure* が呼び出されたら、*command*(s) を実行します。

ここで、

line は、ソースコード行の番号です。

command は、コマンドの名前です。

procedure は、手続きの名前です。

全イベントのリストと構文については、260 ページの「イベント指定の設定」を参照してください。

ローレベルイベントの発生時にコマンドを実行する方法については、354 ページの「wheni コマンド」を参照してください。

wheni コマンド

wheni コマンドは、指定のローレベルイベント発生時にコマンドを実行します。

wheni コマンドの一般構文は、次のとおりです。

構文

`wheni` *event-specification* `[` *modifier* `]{` *command* `...` `;` `}`

指定イベントが発生すると、コマンドが実行されます。

次の構文が有効です。

`wheni at` *address* `{` *command* `;` `}` — *address* に到達したら、*command*(s) を実行します。

ここで、

address は、アドレスとなった式またはアドレスとして使用可能な式です。

command は、コマンドの名前です。

全イベントのリストと構文については、260 ページの「イベント指定の設定」を参照してください。

where コマンド

where コマンドは、呼び出しスタックを出力します。

構文

ここで、	手続きトレースバックを出力します。
`where` *number*	トレースバックの上から *number* 個のフレームを出力します。
`where -f` *number*	フレーム *number* からトレースバックを開始します。
`where -h`	隠しフレームを含めます。
`where -l`	関数名を持つライブラリ名を含めます。
`where -q`	クィックトレースバック (関数名のみ)。
`where -v`	冗長トレースバック (関数の引数と行情報を含む)。

ここで、

number は、呼び出しスタックフレームの数です。

これらの構文の後にスレッドや LWP ID を指定すれば、指定エンティティのトレースバックを取り出せます。

whereami コマンド

whereami コマンドは、現在のソース行を表示します。

構文

whereami	現在の位置 (スタックのトップ) に該当するソース行、および現在のフレームに該当するソース行を表示します (前者と異なる場合)。
whereami -instr	上記と同じ。ただし、ソース行ではなく現在の逆アセンブル命令が出力されます。

whereis コマンド

whereis コマンドは、特定の名前の使用状況すべて、またはアドレスの英字名を出力します。

構文

whereis *name*	*name* の宣言をすべて出力します。
whereis -a *address*	*address* 式の場所を出力します。

ここで、

name は、変数、関数、クラステンプレート、関数テンプレートといった、スコープ内の読み込み可能オブジェクトの名前です。

address は、アドレスとなった式またはアドレスとして使用可能な式です。

which コマンド

which コマンドは、指定の名前の完全修飾形を出力します。

構文

`which` *name*　　*name* の完全修飾形を出力します。

ここで、

name は、変数、関数、クラステンプレート、関数テンプレートといった、スコープ内の物の名前です。

whocatches コマンド

`whocatches` コマンドは、C++例外が捕獲される場所を示します。

構文

`whocatches` *type*　　型 *type* の例外が現在の実行点で送出された場合にどこで捕獲されることになるかを示します (捕獲されるとしたら)。次に実行される文が `throw` *x* であり (*x* の型は*type*)、これを捕獲する `catch` 節の行番号、関数名、フレーム番号を表示するとします。

このとき、送出を行う関数の中に捕獲点がある場合には、"*type* `is unhandled`" が返されます。

ここで、

type は、例外の型です。

索引

記号

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

く

け

こ

さ

し

す

せ

そ

た

ち

つ

て

と

は

ひ

ふ

へ

ら

り

れ

ろ