



dbx コマンドによるデバッグ

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

Part No. 806-4834-01
2000 年 6 月 Revision A

本製品およびそれに関連する文書は、その使用、複製、頒布および逆コンパイルを制限するライセンスのもとにおいて頒布されます。サン・マイクロシステムズ株式会社の書面による事前の許可なく、本製品および関連する文書のいかなる部分も、いかなる方法によっても複製することが禁じられます。フォント技術を含む第三者のソフトウェアは、著作権法により保護されており、提供者からライセンスを受けているものです。

本製品の一部は、カリフォルニア大学からライセンスされている Berkeley BSD システムに基づいていることがあります。UNIX は、X/Open Company Limited が独占的にライセンスしている米国ならびに他の国における登録商標です。Netscape™、Netscape Navigator™、および Netscape Communications Corporation のロゴは、次の著作権で保護されています。

© 1995 Netscape Communications Corporation.

Sun、Sun Microsystems、Sun のロゴマーク、docs.sun.com、AnswerBook2、SunOS、JavaScript、SunExpress、Sun WorkShop、Sun WorkShop Professional、Sun Performance Library、Sun Performance WorkShop、Sun Visual WorkShop、および Forte は、米国およびその他の国における米国 Sun Microsystems, Inc. (以下、米国 Sun Microsystems 社とします) の商標もしくは登録商標です。

すべての SPARC 商標は、米国 SPARC International, Inc. のライセンスを受けて使用している同社の米国およびその他の国における商標または登録商標です。SPARC 商標が付いた製品は、米国 Sun Microsystems 社が開発したアーキテクチャに基づくものです。

本書で参照されている製品やサービスに関しては、該当する会社または組織に直接お問い合わせください。

OPEN LOOK および Sun Graphical User Interface は、米国 Sun Microsystems 社が自社のユーザおよびライセンス実施権者向けに開発しました。米国 Sun Microsystems 社は、コンピュータ産業用のビジュアルまたはグラフィカル・ユーザインタフェースの概念の研究開発における米国 Xerox 社の先駆者としての成果を認めるものです。米国 Sun Microsystems 社は米国 Xerox 社から Xerox Graphical User Interface の非独占的ライセンスを取得しており、このライセンスは米国 Sun Microsystems 社のライセンス実施権者にも適用されます。

Sun f90 / f95 は、米国 Silicon Graphics, Inc. の Cray CF90™ に基づいています。

Federal Acquisitions: Commercial Software—Government Users Subject to Standard License Terms and Conditions.

本書は、「現状のまま」をベースとして提供され、商品性、特定目的への適合性または第三者の権利の非侵害の黙示の保証を含み、明示的であるか黙示的であるかを問わず、あらゆる説明および保証は、法的に無効である限り、拒否されるものとします。

本製品が、外国為替および外国貿易管理法 (外為法) に定められる戦略物資等 (貨物または役務) に該当する場合、本製品を輸出または日本国外へ持ち出す際には、サン・マイクロシステムズ株式会社の事前の書面による承諾を得ることのほか、外為法および関連法規に基づく輸出手続き、また場合によっては、米国商務省または米国所轄官庁の許可を得ることが必要です。

原典 : *Debugging a Program With dbx*
Part No: 806-3557-10
Revision A

© 2000 by Sun Microsystems, Inc.



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

製品名の変更について iii

はじめに xvii

1. dbx の起動 1

デバッグセッションを開始する 1

既存のコアファイルのデバッグ 2

プロセス ID の使用 2

dbx 起動時シーケンス 3

起動属性の設定 4

pathmap コマンドによるマッピング 4

dbxenv コマンドによる環境変数の設定 5

alias コマンドを使用してユーザー自身の dbx コマンドを作成 5

デバッグのため、プログラムをコンパイル 6

最適化コードのデバッグ 6

-g オプションを使用しないでコンパイルされたコード 7

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 8

完全にストリップされたプログラム 8

デバッグセッションを終了する 8

- プロセス実行の停止 8
- dbx からのプロセスの切り離し 9
- セッションを終了せずにプログラムを終了する 9
- デバッグ実行の保存と復元 9
 - save コマンドの使用 10
 - 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 11
 - 保存された実行の復元 12
 - replay を使用した保存と復元 13
- 2. dbx のカスタマイズ 15
 - .dbxrc ファイルの使用 15
 - .dbxrc ファイルの作成 16
 - 初期化ファイル 16
 - dbx 環境変数と Korn シェル 17
 - Sun WorkShop での dbx カスタマイズ 17
 - デバッグオプションの設定 17
 - 統一されたオプションセットの維持 17
 - 2 組のオプション維持 18
 - カスタムボタンの保存 18
 - dbxenv コマンドでの dbx 環境変数の設定 19
- 3. コードの表示 27
 - コード位置へのマッピング 27
 - 停止位置とは別の部分のコードを表示する 28
 - ファイルの内容を表示する 28
 - 関数を表示する 29
 - ソースリストの出力 30
 - 呼び出しスタックの操作によってコードを表示する 31
 - スコープ決定演算子を使用してシンボルを修飾する 31

逆引用符演算子	31
コロンを重ねたスコープ決定演算子 (C++)	32
ブロックローカル演算子	32
リンカー名	33
スコープ決定パス	33
シンボルを検索する	34
シンボルの出現を出力する	34
実際に使用されるシンボルを調べる	35
変数、メンバー、型、クラスを調べる	36
変数、メンバー、関数の定義を調べる	36
型およびクラスの定義を調べる	37
自動読み取り機能の使用	40
.。ファイルが存在しない場合のデバッグ	41
モジュールについてのデバッグ情報	42
モジュールのリスト	43
4. プログラムの実行制御	45
dbx でプログラムを実行する	45
動作中のプロセスに dbx を接続する	46
プロセスから dbx を切り離す	47
プログラムのステップ実行	48
シングルステップ	49
プログラムを継続する	49
関数を呼び出す	51
Control+C によってプロセスを停止する	52
5. ブレークポイントとトレースの設定	53
ブレークポイントを設定する	53
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	54

関数に stop ブレークポイントを設定する	54
特定の行に when タイプのブレークポイントを設定する	56
動的にリンクされたライブラリにブレークポイントを設定する	56
C++ プログラムに複数のブレークポイントを設定する	57
コードをトレースする	59
トレースを設定する	59
トレース速度を制御する	60
イベントハンドラのリストの表示とクリア	60
ブレークポイントとトレースポイントの表示	60
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	61
ウォッチポイント	61
高速 modify イベント	63
ブレークポイントフィルタを設定する	64
イベント効率	65
6. イベント管理	67
イベントハンドラ	67
イベントハンドラの作成	68
イベントハンドラを操作するコマンド	69
イベントカウンタ	70
イベント指定の設定	70
ブレークポイントイベント仕様	70
ウォッチポイントイベント仕様	71
システムイベント仕様	73
実行進行状況イベント仕様	76
その他のイベント仕様	78
イベント指定のための修飾子	80
解析とあいまいさに関する注意	82

事前定義済み変数	83
when コマンドに対して有効な変数	84
イベント別の有効変数	85
イベントハンドラの設定例	86
配列メンバーへの書き込みに対するウォッチポイントを設定する	86
単純なトレースを実行する	86
関数の中だけイベントを有効にする (<i>in func</i>)	86
実行された行の数を調べる	87
実行された命令の数をソース行で調べる	87
イベント発生後にブレークポイントを有効にする	88
replay 時にアプリケーションファイルをリセットする	88
プログラムの状態を調べる	89
浮動小数点例外を捕捉する	89
7. 呼び出しスタックの使用	91
スタック上での現在位置の検索	91
スタックを移動してホームに戻る	92
スタックを上下に移動する	92
スタックの上方向への移動	92
スタックの下方向への移動	93
特定フレームへの移動	93
呼び出しスタックのポップ	94
スタックフレームを隠す	94
8. データの評価と表示	97
変数と式の評価	97
実際に使用される変数を確認する	97
現在の関数のスコープ外にある変数	98
変数または式の値を出力する	98

C++ での表示	98
ポインタを間接参照する	100
式を監視する	100
表示を取り消す (非表示)	101
式に値を代入する	101
配列を評価する	102
配列の断面化	102
配列の断面	106
刻み幅	106
9. 実行時検査	109
概要	109
RTC を使用する場合	110
RTC の必要条件	110
制限事項	111
RTC の使用	111
メモリー使用状況とメモリーリーク検査を有効化	111
メモリアクセス検査を有効化	111
すべての RTC を有効化	112
RTC を無効化	112
プログラムを実行	112
メモリアクセスエラーの検出 (SPARC のみ)	116
メモリアクセスエラーの報告	117
メモリアクセスエラー	118
メモリーリークの検査	118
メモリーリーク検査の使用	119
リークの可能性	120
リークの検査	121

メモリリークの報告を理解する	122
メモリリークの修正	125
メモリ使用状況検査の使用	125
エラーの抑止	127
抑止のタイプ	128
エラー抑止の例	129
デフォルトの抑止	130
抑止によるエラーの制御	130
子プロセスにおける RTC の実行	131
接続されたプロセスへの RTC の使用	135
RTC での修正継続機能の使用	137
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	139
バッチモードでの RTC の使用	139
bcheck 構文	140
bcheck 構文	140
dbx からバッチモードを直接有効化	141
トラブルシューティングのヒント	141
RTC の 8M バイト制限	142
RTC エラー	144
アクセスエラー	144
メモリリークエラー	148
10. データの視覚化	151
適切な配列式の指定	151
配列グラフの作成	153
配列グラフ作成の準備	153
さまざまな配列グラフの作成	153

配列表示の自動更新	154
表示の変更	155
視覚化されたデータの分析	159
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	159
シナリオ 2: データのグラフを自動的に更新する	160
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	161
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	162
Fortran のサンプルプログラム	164
C のサンプルプログラム	165
11. 修正継続機能 (fix と continue)	167
修正継続機能の使用	167
fix と continue の働き	168
fix と continue によるソースの変更	168
プログラムの修正	169
修正後の続行	170
修正後の変数の変更	171
ヘッダファイルの変更	173
C++ テンプレート定義の修正	174
12. マルチスレッドアプリケーションのデバッグ	175
マルチスレッドデバッグについて	175
スレッド情報	176
別のスレッドのコンテキストの表示	178
スレッドリストの表示	178
実行の再開	179
LWP 情報について	179
13. 子プロセスのデバッグ	181

単純な接続の方法	181
exec 機能後のプロセス追跡	182
fork 機能後のプロセス追跡	182
イベントとの対話	183
14. シグナルの処理	185
シグナルイベントについて	185
システムシグナルを捕獲する	187
デフォルトの catch リストと ignore リストを変更する	187
FPE シグナルをトラップする	187
プログラム内でシグナルを送信する	189
シグナルの自動処理	190
15. C++ のデバッグ	191
C++ での dbx の使用	191
dbx での例外処理	192
例外処理コマンド	193
例外処理の例	195
C++ テンプレートでのデバッグ	197
テンプレートの例	198
C++ テンプレートのコマンド	199
16. dbx を使用した Fortran のデバッグ	205
Fortran のデバッグ	206
カレントプロシージャとカレントファイル	206
大文字	206
最適化プログラム	207
dbx のサンプルセッション	208
セグメント不正のデバッグ	210

- dbx により問題を見つける方法 211
- 例外の検出 212
- 呼び出しのトレース 212
- 配列の操作 214
 - Fortran 95 割り当て可能配列 215
- 組み込み関数 218
- 複合式 219
- 論理演算子 220
- Fortran 95 構造型の表示 221
- Fortran 95 構造型へのポインタ 223
- 17. 機械命令レベルでのデバッグ 227
 - メモリーの内容を調べる 227
 - examine または x コマンドの使用 228
 - dis コマンドの使用 231
 - listi コマンドの使用 232
 - 機械命令レベルでのステップ実行とトレース 233
 - 機械命令レベルでステップ実行する 233
 - 機械命令レベルでトレースする 234
 - 機械命令レベルでブレークポイントを設定する 235
 - あるアドレスにブレークポイントを設定する 236
 - adb コマンドの使用 236
 - regs コマンドの使用 236
 - プラットフォーム固有のレジスタ 237
 - Intel レジスタ情報 239
- 18. dbx の Korn シェル機能 243
 - 実装されていない ksh-88 の機能 243
 - ksh-88 から拡張された機能 244

名前が変更されたコマンド	244
19. 共有ライブラリのデバッグ	245
動的リンカー	245
リンクマップ	246
起動手順と .init セクション	246
プロシージャ・リンケージ・テーブル	246
読み込み済みの共有オブジェクトのデバッグサポート	247
修正と継続	247
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	248
A. プログラム状態の変更	249
dbx のもとでプログラムを実行することの影響	249
プログラムの状態を変更するコマンドの使用	250
assign コマンド	250
pop コマンド	251
call コマンド	251
print コマンド	251
when コマンド	252
fix コマンド	252
cont at コマンド	252
索引	253

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つプログラマーで、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方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。

マルチプラットフォーム対応

この Sun WorkShop リリースは、Solaris 2.6、7、および 8 のオペレーティング環境 (SPARC™ プラットフォームおよび Intel プラットフォーム) をサポートしています。

注 - Intel アーキテクチャとは、Pentium、Pentium Pro、Pentium II プロセッサおよび、これらと互換性のある AMD および Cyrix 製のマイクロプロセッサチップを含む、Intel 8086 マイクロプロセッサチップ群を意味しています。このマニュアルでは、これらすべてのプラットフォームアーキテクチャを総称して Intel アーキテクチャと呼んでいます。

Sun WorkShop 開発ツールへのアクセス方法

Sun WorkShop 製品コンポーネントとマニュアルページは標準ディレクトリ `/usr/bin` および `/usr/share/man` にはインストールされません。そのため `PATH` および `MANPATH` 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ります。

`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以下を実行します。

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、この節の指示に従って、`PATH` 環境変数を設定してください。

`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以下を実行します。

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。

注 – この節に記載されている情報は Sun WorkShop 6 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 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、システム管理者に連絡してください。

`PATH` 変数および `MANPATH` 変数は、C シェルを使用している場合はホームディレクトリの下に `.cshrc` ファイルに設定する必要があります。Bourne シェルか Korn シェルを使用している場合は、ホームディレクトリの下に `.profile` ファイルに設定する必要があります。

- Sun WorkShop コマンドを使用するには、`PATH` 変数に以下を追加してください。

```
/opt/SUNWspro/bin
```

- `man` コマンドで、Sun WorkShop マニュアルページにアクセスするには、`MANPATH` 変数に以下を追加してください。

```
/opt/SUNWspro/man
```

`PATH` 変数についての詳細は、`cs(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 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。

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「dbx の起動」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 2 章「dbx のカスタマイズ」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 3 章「コードの表示」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 4 章「プログラムの実行制御」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 5 章「ブレークポイントとトレースの設定」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 6 章「イベント管理」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。

第 7 章「呼び出しスタックの使用」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「データの評価と表示」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「実行時検査」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「データの視覚化」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「修正継続機能 (fix と continue)」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「マルチスレッドアプリケーションのデバッグ」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「子プロセスのデバッグ」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「シグナルの処理」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「C++ のデバッグ」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「`dbx` を使用した Fortran のデバッグ」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「機械命令レベルでのデバッグ」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「dbx の Korn シェル機能」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「共有ライブラリのデバッグ」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「プログラム状態の変更」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。

書体と記号について

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ます。

表 P-1 このマニュアルで使用している書体と記号

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 <i>filename</i> 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』

表 P-1 このマニュアルで使用している書体と記号 (続き)

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。	<pre>machinename% grep `^#define \ XV_VERSION_STRING`</pre>
▶	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

シェルプロンプトの例を以下に示します。

表 P-2 シェルプロンプト

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

関連マニュアル

以下の方法で、関連マニュアルにアクセスすることができます。

- インターネットの docs.sun.com の Web サイトからアクセスできます。特定の本のタイトルで検索するか、主題、マニュアルコレクションまたは製品別にブラウズすることができます。

<http://docs.sun.com>

- ローカルシステムまたはローカルネットワークにインストールされた Sun WorkShop 製品からアクセスできます。Sun WorkShop 6 HTML 文書 (マニュアル、オンラインヘルプ、マニュアルページ、各コンポーネントの README ファイル、リリースノート) が、インストールした Sun WorkShop 6 製品から参照可能です。HTML 文書にアクセスするには、次のいずれかを実行します。
 - Sun WorkShop または Sun WorkShop™ TeamWare ウィンドウで、「ヘルプ」
 - ▶ 「オンラインマニュアルについて」を選択します。
 - Netscape™ Communicator 4.0 またはその互換バージョンのブラウザで、以下のファイルを開きます。

</opt/SUNWspr/docs/ja/index.html>

参照できる Sun WorkShop 6 HTML 文書の一覧がブラウザに表示されます。一覧にあるマニュアルを開くには、マニュアルのタイトルをクリックしてください。

表 P-3 は、Sun WorkShop 6 関連マニュアルをマニュアルコレクション別に一覧にしたものです。

表 P-3 マニュアルコレクション別 Sun WorkShop 6 関連マニュアル

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。

表 P-3 マニュアルコレクション別 Sun WorkShop 6 関連マニュアル (続き)

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	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イ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 <code>er_print</code> 、ループツール、ループレポートユーティリティおよび UNIX プロファイルツール <code>prof</code> 、 <code>gprof</code> 、 <code>tcov</code> についての情報も含んでいます。
	dbx コマンドによるデバッグ	dbx コマンドを使ってプログラムをデバッグする方法について説明しています。参考情報として、同じデバッグ処理を Sun WorkShop デバッグウィンドウを使って実行する方法も記載しています。
	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、 <code>lint</code> 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++ ライブラリ・リファレンス	C++ ライブラリについて説明しています。C++ 標準ライブラリ、Tools.h++ クラスライブラリ、Sun WorkShop Memory Monitor、 <code>Iostream</code> および複素数の情報も含まれます。
	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。

表 P-3 マニュアルコレクション別 Sun WorkShop 6 関連マニュアル (続き)

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
	C++ プログラミングガイド	新しい機能を使ってより効率的なプログラムを記述する方法について説明しています。テンプレート、例外処理、実行時の型識別、キャスト演算、パフォーマンス、およびマルチスレッド対応のプログラムに関する情報も記載されています。
	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。
	Sun WorkShop Memory Monitor ユーザーズガイド	C および C++ のメモリー管理で生じた問題を Sun WorkShop Memory Monitor で解決する方法について説明しています。このマニュアルはインストールした製品 (/opt/SUNWspro/docs/ja/index.html) からのみ参照可能で、 docs.sun.com Web サイトで参照することはできません。
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。
	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。
Forte TeamWare 6 / Sun WorkShop TeamWare 6	Sun WorkShop TeamWare ユーザーズガイド	Sun WorkShop TeamWare コード管理ツールの使用方法について説明しています。

表 P-3 マニュアルコレクション別 Sun WorkShop 6 関連マニュアル (続き)

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 Visual 6	Sun WorkShop Visual ユーザーズガイド	C++ と Java™ の GUI (グラフィカルユーザーインターフェース) を Sun WorkShop Visual を使用して作成する方法について説明しています。このマニュアルには、旧リリース (Sun WorkShop Visual 5.0) から変更のない機能が記載されています。
	Sun WorkShop Visual の新機能	Sun WorkShop Visual 6.0 で追加または変更された機能について説明しています。
Forte / Sun Performance Library 6	Sun Performance Library Reference (英語のみ)	コンピュータによる線形代数および高速フーリエ変換を実行するサブルーチンと関数の最適化ライブラリについて説明しています。
	Sun Performance Library User's Guide (英語のみ)	線形代数で発生した問題の解決に使用されるサブルーチンと関数のコレクションである Sun Performance Library のサン固有の機能の使用方法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。
標準ライブラリ 2	Standard C++ Library Class Reference (英語のみ)	標準 C++ の詳細について説明しています。
	標準 C++ ライブラリ・ユーザーズガイド	標準 C++ ライブラリの使用方法について説明しています。
Tools.h++ 7	Tools.h++ 7.0 ユーザーズガイド	Tools.h++ クラスライブラリの詳細について説明しています。
	Tools.h++ 7.0 クラスライブラリ・リファレンスマニュアル	C++ クラスを使用して、プログラム効率を向上させる方法について説明しています。

表 P-4 は、docs.sun.com の Web サイトからアクセスできる Solaris 関連マニュアルの一覧です。

表 P-4 Solaris 関連マニュアル

マニュアルコレクション	マニュアルタイトル	内容の説明
Solaris ソフトウェア開発	リンカーとライブラリ	Solaris リンクエディタと実行時リンカーの操作およびそれらが操作するオブジェクトについて説明しています。
	プログラミングユーティリティ	Solaris オペレーティング環境で使用可能な特殊組み込みプログラミングツールに関する開発者向けの情報を記載しています。

第1章

dbx の起動

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。このツールを使用することにより、制御された方法でプログラムを実行し、停止したプログラムの状態を調べることができます。dbx は、パフォーマンスデータのコレクションを含む、プログラムの動的な実行を完全に制御することができます。

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。

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
```

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。

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「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 (`pid`) を指定します。

```
$ dbx your_program_name process_ID
```

プログラムの名前を知らなくても、その ID 番号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` のような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。

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、15 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。

プログラムがローディングを終了すると、`dbx` はレディ状態となり、プログラム (C、C++、Fortran 95 については、`main()`、FORTRAN 77 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し、C プログラムに対し、`stop in main` や `run` などの `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。

`pathmap` コマンドによるマッピング

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なオートマウント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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」の「`pathmap` コマンド」を参照してください。

dbxenv コマンドによる環境変数の設定

`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。

```
$ 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、15 ページの第 2 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、19 ページの「`dbxenv` コマンドでの `dbx` 環境変数の設定」および Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」の「`dbxenv` コマンド」を参照してください。

alias コマンドを使用してユーザー自身の 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「`dalias` コマンド」を参照してください。

大域変数を出力し、正規として割り当てることができます。ただし、最終的なメモリーへのレジスタ保存が発生していない場合、それらの変数は正確でない値をとる場合があります。

関数の最初で停止すると、パラメータのみ出力されます。パラメータレジスタは、コンパイラで再使用できるためです。スタック変数の値は、最適化プログラムでは印刷したり、操作することができません。これらの変数のスタックおよびレジスタ位置は、記録されないためです。

『プログラムのパフォーマンス解析』マニュアルには、コンパイラの最適化についての情報が詳しく記載されており、最適化プログラムで Sun WorkShop ツールを使用する場合に役立つでしょう。

デバッグのため、プログラムをコンパイル

プログラムを `-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

ただし、最適化コードでは、`dbx` は次のことを行うことができません。

- 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)
- 局所変数を評価、表示、変更する

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。

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、最適化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。

一般に、パラメータ、局所、大域についてのシンボリック情報は、最適化プログラムに対して提供されます。構造体、共用体、C++ クラスに関する型情報、および局所、大域、パラメータの型と名前が提供されます。プログラムでのこれらの項目の位置に関する完全な情報は、最適化プログラムについては提供されません。

`-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。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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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。11 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_of_commands` 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2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
 - `dbxenv` コマンドでの `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` カスタマイズ変数を変更するコマンドです。
- `input_case_sensitive` は、大文字/小文字の区別を制御する変数です。
- `false` は制御値です。

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。

dbxenv コマンドでの 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 2-1 `dbx` 環境変数

dbx 環境変数	dbx 環境変数の機能
<code>allow_critical_exclusion</code> <code>on off</code>	通常、 <code>loadobject -x</code> は、 <code>dbx</code> 機能に欠かれ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 <code>on</code> に設定すると、その制限が解除されます。この変数が <code>on</code> 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 <code>off</code> です。
<code>aout_cache_size</code> <i>number</i>	<code>a.out</code> ロードオブジェクトのキャッシュサイズ。単一の <code>dbx</code> から順に <code>num</code> 個のプログラムをデバッグする場合は <code>num</code> に設定してください。 <code>num</code> 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。 <code>locache_enable</code> の項目を参照してください。デフォルト値は 1 です。
<code>array_bounds_check</code> <code>on off</code>	パラメータを <code>on</code> 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 <code>on</code> です。
<code>cfront_demangling</code> <code>on off</code>	プログラムの読み込み中、 <code>Cfront</code> (SC 2.0 と SC 2.0.1) 名の復号化を制御します。 <code>Cfront</code> によってコンパイルされたプログラムまたは <code>Cfront</code> コンパイルライブラリによってリンクされたプログラムをデバッグする場合は、パラメータを <code>on</code> に設定するか、または <code>-F</code> オプションをつけて <code>dbx</code> を起動する必要があります。 <code>off</code> に設定すると、 <code>dbx</code> によるプログラムの読み込み速度はおよそ 15% 速くなります。デフォルト値は <code>off</code> です。
<code>delay_xs</code> <code>on off</code>	<code>-xs</code> 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、 <code>dbx</code> 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。

表 2-1 dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
<code>disassembler_version</code> <code>autodetect v8 v9 v9vis</code>	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 <code>autodetect</code> で、 <code>a.out</code> 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、 <code>autodetect</code> だけが有効です。
<code>fix_verbose</code> <code>on off</code>	<code>fix</code> 中のコンパイル行出力を制御します。デフォルト値は <code>off</code> です。
<code>follow_fork_inherit</code> <code>on off</code>	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 <code>off</code> です。
<code>follow_fork_mode</code> <code>parent child both ask</code>	現在のプロセスが <code>fork</code> 、 <code>vfork</code> 、 <code>fork1</code> 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。 <code>parent</code> に設定すると、親を追跡し、 <code>child</code> に設定すると、子を追跡します。 <code>both</code> 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。 <code>ask</code> 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 <code>parent</code> です。
<code>follow_fork_mode_inner</code> <code>unset parent child both</code>	フォークが検出された後、 <code>follow_fork_mode</code> が <code>ask</code> 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、 <code>cont -follow</code> を使用する必要はありません。
<code>input_case_sensitive</code> <code>autodetect true false</code>	<code>autodetect</code> 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 <code>false</code> 、そうでない場合は <code>true</code> です。 <code>true</code> 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 <code>autodetect</code> です。

表 2-1 dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
<code>language_mode</code> <code>autodetect main c </code> <code>ansic c++ objc fortran </code> <code>fortran90 native_java</code>	<p>式の構文解析と評価に使用される言語を指定します。<code>autodetect</code> 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)。<code>main</code> 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。<code>c</code>、<code>c++</code>、<code>ansic</code>、<code>objc</code>、<code>fortran</code>、<code>fortran90</code> および <code>native_java</code> では、それぞれ指定の言語に設定されます。</p>
<code>locache_enable</code> <code>on off</code>	<p>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 <code>on</code> です。</p>
<code>mt_scalable</code> <code>on off</code>	<p>有効の場合、<code>dbx</code> 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 <code>off</code> です。</p>
<code>output_auto_flush</code> <code>on off</code>	<p>呼び出しが行われるたびに、<code>fflush()</code> を自動的に呼び出します。デフォルト値は <code>on</code> です。</p>
<code>output_base</code> <code>8 10 16</code> <code>automatic</code>	<p>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 <code>automatic</code> です (ポインタは 16 進文字、その他すべては 10 進)。</p>
<code>output_dynamic_type</code> <code>on off</code>	<p><code>on</code> の場合、出力、表示、および検査のデフォルト出力を <code>-d</code> にします。デフォルト値は <code>off</code> です。</p>
<code>output_inherited_members</code> <code>on off</code>	<p><code>on</code> の場合、出力、表示、および検査のデフォルト出力を <code>-r</code> にします。デフォルト値は <code>off</code> です。</p>
<code>output_list_size</code> <i>number</i>	<p><code>list</code> 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。</p>
<code>output_log_file_name</code> <i>filename</i>	<p>コマンドログファイルの名前。デフォルト値は <code>/tmp/dbx.log uniqueID</code> です。</p>
<code>output_max_string_length</code> <i>number</i>	<p><code>char *</code> 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。</p>

表 2-1 dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
<code>output_pretty_print on off</code>	出力、表示、および検査の出力デフォルトを <code>-p</code> に設定します。デフォルト値は <code>off</code> です。
<code>output_short_file_name on off</code>	ファイル名を表示する時に短形式で表示しません。デフォルト値は <code>on</code> です。
<code>overload_function on off</code>	C++ の場合、 <code>on</code> 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 <code>on</code> です。
<code>overload_operator on off</code>	C++ の場合、 <code>on</code> 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 <code>on</code> です。
<code>pop_auto_destruct on off</code>	<code>on</code> 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。
<code>proc_exclusive_attach on off</code>	<code>on</code> に設定すると、別のツールがすでに接続されている場合、 <code>dbx</code> 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 <code>on</code> です。
<code>rtc_auto_continue on off</code>	<code>rtc_error_log_file_name</code> にエラーを記録して続行します。デフォルト値は <code>off</code> です。
<code>rtc_auto_suppress on off</code>	<code>on</code> 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 <code>off</code> です。
<code>rtc_biu_at_exit on off verbose</code>	<code>check -memuse</code> が明示的に、または <code>check -all</code> によって <code>on</code> になっている場合に使用されます。この値が <code>on</code> 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 <code>verbose</code> 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 <code>off</code> の場合、出力は生成されません。デフォルト値は <code>on</code> です。

表 2-1 dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
<code>rtc_error_limit number</code>	報告される RTC エラーの数。デフォルト値は 1000 です。
<code>rtc_error_log_file_name filename</code>	<code>rtc_auto_continue</code> 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 <code>/tmp/dbx.errlog.uniqueID</code> です。
<code>rtc_error_stack on off</code>	<code>on</code> 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 <code>off</code> です。
<code>rtc_inherit on off</code>	<code>on</code> 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、 <code>LD_PRELOAD</code> が継承されます。デフォルト値は <code>off</code> です。
<code>rtc_mel_at_exit on off verbose</code>	リーク検査がオンの場合に使用されます。この値が <code>on</code> 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 <code>verbose</code> 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。 <code>off</code> の場合は出力は生成されません。デフォルト値は <code>on</code> です。
<code>rtc_use_traps on off</code>	ランタイムチェック時の 8 MG コード制限への対策を有効にします。 <code>on</code> に設定すると、 <code>dbx</code> は UltraSparc トラップ命令 (提供される場合) を使用します。デフォルト値は <code>off</code> です。
<code>run_autostart on off</code>	<code>dbx</code> で実行中でないプログラムで <code>on</code> の場合、 <code>step</code> 、 <code>next</code> 、 <code>stepi</code> 、および <code>nexti</code> を実行した場合、暗黙指定で <code>run</code> を実行し、言語依存のメインルーチンで停止します。 <code>on</code> の場合、 <code>cont</code> は必要に応じて <code>run</code> を暗黙指定します。デフォルト値は <code>off</code> です。

表 2-1 dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
<code>run_io stdio pty</code>	ユーザープログラムの入出力が、 <code>dbx</code> の <code>stdio</code> か、または特定の <code>pty</code> にリダイレクトされるかどうかを指定します。 <code>pty</code> は、 <code>run_pty</code> によって指定します。デフォルト値は <code>stdio</code> です。
<code>run_pty ptyname</code>	<code>run_io</code> が <code>pty</code> に設定されているときに使用する <code>pty</code> の名前を設定します。 <code>pty</code> は GUI のラッパで使用されます。
<code>run_quick on off</code>	<code>on</code> 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、 <code>prog -readsysms</code> 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 <code>dbx</code> 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 <code>off</code> です。
<code>run_savetty on off</code>	<code>dbx</code> と デバッグ対象の間で、 <code>tty</code> 設定、プロセスグループ、およびキーボード設定 (<code>-kbd</code> 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。 <code>dbx</code> が <code>SIGTTIN</code> または <code>SIGTTOU</code> を取得しシェルに戻る場合は、 <code>on</code> に設定します。速度を多少上げるには <code>off</code> に設定します。 <code>dbx</code> 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 <code>on</code> です。
<code>run_setpgrp on off</code>	<code>on</code> の場合、プログラムが実行時に、フォークの直後に <code>setpgrp(2)</code> が呼び出されます。デフォルト値は <code>off</code> です。
<code>scope_global_enums on off</code>	<code>on</code> 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 (<code>~/ .dbxrc</code>)。デフォルト値は <code>off</code> です。
<code>scope_look_aside on off</code>	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 <code>on</code> です。

表 2-1 dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
<code>session_log_file_name filename</code>	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。
<code>satck_find_source on off</code>	on に設定した場合、デバッグ中のプログラムが <code>-g</code> 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。
<code>stack_max_size number</code>	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。
<code>stack_verbose on off</code>	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。
<code>step_events on off</code>	step と next の間ブレークポイントを許可します。デフォルト値は off です。
<code>step_granularity statement line</code>	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 <pre>a(); b();</pre> 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。
<code>suppress_startup_message number</code>	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。
<code>symbol_info_compression on off</code>	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。
<code>trace_speed number</code>	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3章

コードの表示

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、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ト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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「pathmap コマンド」を参照してください。

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。

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、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
```


ファイルコマンドを単独で使用すると、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
```

関数を表示する

`func` コマンドを使用すると、関数を表示することができます。コマンド `func` に続けて、関数名を入力します。次の例を示します。

```
(dbx) func adjust_speed
```

`func` コマンドを単独で使用すると、現在表示中の関数が表示されます。

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、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。

指定の関数があいまいな場合は、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、次のように入力することができます。

```
(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中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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」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。

```
(dbx) list [-i | -instr] [+] [-] number [ function | filename ]
```

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「`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。

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、92 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 (`whatis` 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++ のメンバー関数またはトップレベル関数を修飾するときには、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

同じ名前が多くのある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。

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
```

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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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ブにない場合、スコーブ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せん。

決定パスに含まれる任意の場所で、同じスコーブ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 'fid' 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中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。

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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」の「`whatis` コマンド」および「C++ で使用される `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。「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();  
Notice that table::pos is inherited from block:  
(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。この例の場合、`whatis` からの出力は、コンパイラがレジスタに `this` 変数を自動的に割り当てたことを示しています。

```
(dbx) stop in brick::draw  
(dbx) cont  
// expose the blocks window (if exposed, hide then expose) to  
force program to hit the breakpoint.  
(dbx) where 1  
brick::draw(this = 0x48870, pw = 374752)、line 124 in  
"block_draw.cc"  
(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せん。

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、そのクラスが親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、 subclasses が継承するメンバーを表示するために `whatis -r` がその subclasses 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();
// exampleprotected から削除されたいいくつかのメンバー:
    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。「`.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、

- 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4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。このマニュアルで後述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++, Fortran 90 の場合は `main()`、FORTRAN 77 の場合は `MAIN`) におとずれます。`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。ここで、別の部分のコードを表示したり、イベント管理コマンドを使用したりすることができます。

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。準備ができたなら、`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。`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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」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 (`pid`) 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` は終了前に自動的に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 program_name process_ID
```

`program_name` を - (ダッシュ) で置換することができます。`dbx` は、`process_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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」の「`debug` コマンド」および「`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。

「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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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 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。

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がある `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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」の「`next` コマンド」および「`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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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。

- 54 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」
- 57 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」
- 57 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」
- 58 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」
- Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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` コマンドを使用する (Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」の「`print` コマンド」または「`call` コマンド」参照) 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は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5章

ブレークポイントとトレースの設定

ブレークポイントは、アクションが発生する場所であり、そのポイントでプログラムは実行を停止します。トレースを設定することにより、変数の値の変更など、プログラムでのイベントに関する情報を表示することができます。トレースの動作は、ブレークポイントの動作とは異なりますが、トレースとブレークポイントは同様のイベントハンドラを共有します。

この章では、ブレークポイントとトレースの設定方法とウォッチポイントの使用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。

- ブレークポイントを設定する
- コードをトレースする
- イベントハンドラのリストの表示とクリア
- ブレークポイントフィルタを設定する
- イベント効率

ブレークポイントを設定する

`dbx` では、ソースレベルのブレークポイントを設定するため、3種類のブレークポイントアクション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 - プログラムは停止し、`dbx` が1つまたは複数の `dbx` コマンドを実行した後、プログラムは再開します。プログラムは、コマンドの中に `stop` が含まれていない限り処理を続けます。

- `trace` ブレークポイント - プログラムは停止し、イベント固有のトレース情報行を出力した後、プログラムは再開します。

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

`dbx 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
```

`stop` または `when` コマンドに指定された行が、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。

「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せん。

- 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

次の例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せん。

特定の行に `when` タイプのブレークポイントを設定する

`when` タイプのブレークポイントは、ほかの `dbx` コマンドを副作用コマンドとして利用できます。たとえば、`when` で `list` コマンドを副作用コマンドとして用いれば、独自のトレースを実現することができます。

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

`when` は `cont` コマンドを暗黙指定して動作します。上の例の場合、現在行のコードを出力した後、プログラムの実行が続けられます。

動的にリンクされたライブラリにブレークポイントを設定する

実行時リンカーとの間でプログラムインタフェースを使用するコード、すなわち `dlopen()`、`dlclose()`、およびそれらに関連する関数を呼び出すコードをデバッグするときは、`dbx` の全機能を利用できます。実行時リンカーは、プログラムの実行中に共有ライブラリをリンクしたり、そのリンクを解除したりします。`dlopen()` と `dlclose()` のサポートにより、動的にリンクされている共有ライブラリの関数もプログラムの起動時にリンクされたライブラリの関数と同様に、ステップ実行したり、ブレークポイントを設定したりできます。

例外が 3 つあります。

- `dlopen()` によって読み込まれるライブラリには、そのライブラリが `dlopen()` によって読み込まれる前にブレークポイントを設定することはできません。
- `dlopen()` によって読み込まれた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を設定することはできません。

- ライブラリが `dlopen()` によって読み込まれるとき、`_init()` という名前の初期化ルーチンが呼び出されます。このルーチンは、ライブラリ内のほかのルーチンを呼び出すことがあります。この初期化が終わるまで、`dbx` は読み込まれたライブラリにブレークポイントを置くことができません。`dlopen()` によって読み込まれたライブラリの `_init()` 内で `dbx` を停止させることはできません。

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。

`when` ブレークポイントを設定するには、`when` コマンドをキーワード `inmember` とともに使用します。

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
```

「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。

`when` タイプのブレークポイントを設定する場合は、`when` コマンドをキーワード `inclass` とともに使用します。

クラス `wedge` 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 wedge
```

「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。

ブレークポイントは、該当するクラスで定義されているクラスメンバー関数にだけ挿入されます。基底クラスから継承された関数には挿入されません。

`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 (65 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。

`when` タイプのブレークポイントを設定する場合は、`when` コマンドをキーワード `infunction` とともに使用します。

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) when infunction sort {cmd;}
```

「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て、特定オブジェクトに適用される演算を確認します。`In Object` ブレークポイントは、オブジェクトから呼び出されると、そのオブジェクトクラスのすべての非静的メンバー関数によるプログラムの実行を中断します。

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
```

「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。

コードをトレースする

トレースは、実行されるコード行、呼び出される関数、変更される変数についての情報を表示します。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「コードのトレース」を参照してください。

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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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 (70 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

多くのプログラムは高速に実行されるため、表示がよく見えません。環境変数 `dbxenv trace_speed` を使用して、トレーススコープ内の各行の実行間隔を制御することができます。デフォルトの実行間隔は 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。

イベントハンドラのリストの表示とクリア

`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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」の「`delete` コマンド」を参照してください。

ウォッチポイント

ウォッチポイントは、変数または式の値が変化したときにイベントを発生させる `dbx` の一般的な機能です。`stop` コマンドを使用してウォッチポイントを設定することができます。

アドレスの内容に書き込みが行われたときに実行を停止する

アドレスの内容に書き込みが行われたときにプログラムの実行を停止するには、次のように入力します。

```
(dbx) stop modify & variable
```

`stop modify` を使用するときの注意事項

- 変数 *variable* にそれまで設定されていた値と同じ値が書き込まれた場合でもイベントは発生します。
- `dbx` が命令をエミュレートすることによって、メモリーに新しい内容が設定された場合でも、変数への書き込みを行なった命令が実行される前にイベントが発生しません。
- 関数に局所的な自動変数など、スタック変数のアドレスを使用することはできません。

「ブレークポイント」ウィンドウで、「指定範囲変更時」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 オンラインヘルプの「デバッグウィンドウの使い方」の「指定範囲変更時に停止」を参照してください。

指定した変数の値が変化したときに実行を停止する

指定した変数の値が変化したときにプログラムの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` を使用するときの注意事項

- `dbx` は、指定された変数の値に変更があった行の後ろの行でプログラムを停止します。
- 変数が関数の局所変数の場合、その関数に最初に制御が移り、変数のための記憶領域が割り当てられた時点で変更があったと見なします。パラメータについても、同じことが言えます。

「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 オンラインヘルプの「デバッグウィンドウの使い方」の「指定変数の値変更時に停止」を参照してください。

`dbx` は、自動的なステップ実行を行い、各ステップで値を検査することによって、`stop change` を実現しています。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、ステップ実行では、ライブラリ呼び出しはスキップされます。したがっ

て、制御の流れが次のようになっているとき、トレース時にライブラリ呼び出しと入れ子になった `user_routine2` の呼び出しがスキップされるため、`dbx` は入れ子になった `user_routine2` をトレースしません。

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

`variable` の値の変更は、`user_routine2` の途中ではなく、ライブラリ呼び出しから復帰した後に発生します。

`dbx` は、ブロック局所変数、すなわち `{ }` 内で入れ子に定義された変数の変化に対するブレークポイントは設定できません。このような変数にブレークポイント (またはトレース) を設定しようとする、その操作ができないことを示すエラーメッセージが出されます。

指定した条件が発生した場合に実行を停止する

条件文が真になったときにプログラムの実行を停止するには、次のように入力します。

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

「ブレークポイント」ウィンドウで、カスタムの「指定条件成立時」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 オンラインヘルプの「デバッグウィンドウの使い方」の「指定条件成立時カスタムブレークポイントの設定」を参照してください。

高速 `modify` イベント

ウォッチポイントは、`modify` イベントを使用すると簡単に設定できます。Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」の「定義済みイベント」を参照してください。

`modify` イベントは、プログラムを自動的にステップ実行する代わりに、はるかに高速のページ保護構造を使用します。処理速度は、デバッグ中のプログラムのシステム呼び出し率だけでなく、監視中の変数の存在するページが何回修正されるかによって異なります。

ブレークポイントフィルタを設定する

`dbx` では、イベント管理コマンドのほとんどで (省略可能な) イベントフィルタの修飾文も利用することができます。最も単純なフィルタは次のいずれかの時点で、`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。イベント修飾子についての詳細は、80 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。

このフィルタ条件が真 (非 0) と評価されると、イベントコマンドが適用されます。条件が偽 (0) と評価されると、`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続けます。

条件つき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ためには、`-if condition` 指示子文を `stop` コマンドまたは `trace` コマンドの終わりに追加します。

`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。

注 - `in` や `at` など位置にもと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構文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。

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 modify &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` (ウォッチポイントと呼ばれるイベントのクラス) 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 `expression` や変数 `variable` 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6章

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。

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
- 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(70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(80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。

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。(70 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。

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「`when` コマンド」、「`stop` コマンド」、「`trace` コマンド」を参照してください。

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。

`stop` と `when` コマンドをハンドラモデルに合致させるよう試みましたが、以前の `dbx` リリースとの下位互換性を保つため、完全にはできませんでした。

一例として、初期の `dbx` バージョンで使用されていたハンドラと、それに相当する新しいハンドラを次の表に示します。

旧ハンドラ	新ハンドラ
<code>when cond body</code>	<code>when step -if cond body</code>
<code>when cond in func body</code>	<code>when next -if cond -in func body</code>

この例は、`cond` が純粋なイベントではないことを示しています。条件に対する内部ハンドラはありません。

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。各コマンドについての詳細は、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」の「コマンド」リストを参照してください。

- `status` - ハンドラを表示（「`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（「`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（「`clear` コマンド」参照）。
- `handler - enable` - ハンドラを有効にします（「`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（「`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 (81 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。

ブレイクポイントイベント仕様

次に、ブレイ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。

`in function`

関数が入力され、最初の行が実行される直前。`-instr` 修飾子を使用される場合 (81 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。

ウォッチポイントイベント仕様

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 addr-exp, [byte-size-exp]`

`addr-exp` 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` 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。シンボリック式を使用すると、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 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。

`cond cond-expr`

`cond-expr` によって示される条件が真と評価されます。`cond-expr` 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。

`modify addr-exp [, byte-size]`

指定されたアドレス範囲で変更があったとき。これは古いウォッチポイント機能です。

`addr-exp` 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、監視対象の領域のサイズが自動的に推定されます。このサイズは `' , '` 構文を使用して変更できます。また、記号を使用しない、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 modify 0x5678, sizeof(Complex)
```

注 - マルチスレッドアプリケーションは、デッドロックする傾向があります。そのため、`mt` ウォッチポイントは公式には許可されていません。`mt` ウォッチポイントは、`dbx` 環境変数 `mt_watchpoints` を設定することにより、オンにすることができます。

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。

`fault` *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
<code>FLTILL</code>	不正命令
<code>FLTPRIV</code>	特権つき命令
<code>FLTBPT*</code>	ブレークポイント命令
<code>FLTTRACE*</code>	トレーストラップ (ステップ実行)
<code>FLTWATCH</code>	ウォッチポイントトラップ
<code>FLTACCESS</code>	メモリアクセス (境界合わせなど)
<code>FLTBOUNDS*</code>	メモリー境界 (無効なアドレス)
<code>FLTIOVF</code>	整数オーバーフロー
<code>FLTIZDIV</code>	整数ゼロ除算
<code>FLTPE</code>	浮動小数点例外
<code>FLTSTACK</code>	修復不可能なスタックフォルト
<code>FLTPAGE</code>	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、ステップ実行、ウォッチポイントを実現するため、`dbx` で使用されます。これらを実行すると、`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/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。特定の関数と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	<code>\$o0</code>
Intel	<code>\$eax</code>

これは、次のように書くことができます。

```
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; }
```

`step` を有効にするということは、次に `cont` が使用されるときに自動的にステップ実行するように `dbx` に命令することと同じです。`step` コマンドは、次のように指定できます。

```
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。

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 (70 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`

指定されたイベントがスレッド識別子 `thread_ID` に一致するアクティブなスレッドで発生した場合にコマンドを実行します。

`-lwp lwp_ID`

指定されたイベントが LWP 識別子 `lwp_ID` に一致するアクティブな LWP で発生した場合にコマンドを実行します。

`-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` という名前の変数で使用されていても、`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
<code>\$ins</code>	現在の命令の逆アセンブル
<code>\$lineno</code>	現在の行番号 (10 進数)
<code>\$line</code>	現在の行の内容
<code>\$func</code>	現在の関数の名前
<code>\$vfunc</code>	現在「表示している」関数
<code>\$class</code>	<code>\$func</code> が所属するクラスの名前
<code>\$vclass</code>	<code>\$vfunc</code> が所属するクラスの名前
<code>\$file</code>	現在のファイルの名前
<code>\$vfile</code>	現在表示しているファイルの名前
<code>\$loadobj</code>	現在のロードオブジェクトの名前
<code>\$vloadobj</code>	現在表示している現在のロードオブジェクトの名前
<code>\$scope</code>	逆引用符表記での現在の PC のスコープ
<code>\$vscope</code>	現在表示している逆引用符表記での PC のスコープ
<code>\$funcaddr</code>	<code>\$func</code> のアドレス (16 進数)
<code>\$caller</code>	<code>\$func</code> を呼び出している関数の名前
<code>\$dlist</code>	<code>dlopen</code> イベントまたは <code>dlclose</code> イベントの後、 <code>dlopen</code> または <code>dlclose</code> 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。 <code>\$dlist</code> 中の先頭の単語は実際には "+" または "-" です。これは、 <code>dlopen</code> と <code>dlclose</code> のどちらが発生したかを示します。
<code>\$newhandlerid</code>	最後に作成されたハンドラの ID
<code>\$firedhandlers</code>	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、 <code>status</code> コマンドの出力時に「*」が付きます。
<code>\$proc</code>	現在デバッグ中のプロセスの ID
<code>\$lwp</code>	現在の LWP の ID

変数	定義
<code>\$thread</code>	現在のスレッドの ID
<code>\$prog</code>	デバッグ中のプログラムの絶対パス名
<code>\$oprogram</code>	<code>\$prog</code> の古い値または元の値。これは、 <code>exec()</code> に続いて、デバッグしていたものに戻る場合に便利です。
<code>\$exitcode</code>	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 6-1 `sig` イベントに固有の変数

変数	説明
<code>\$sig</code>	イベントを発生させたシグナル番号
<code>\$sigstr</code>	<code>\$sig</code> の名前
<code>\$sigcode</code>	適用可能な場合、 <code>\$sig</code> のサブコード
<code>\$sigcodestr</code>	<code>\$sigcode</code> の名前
<code>\$sigsender</code>	必要であれば、シグナルの送信者のプロセス ID

表 6-2 `exit` イベントに固有の変数

変数	説明
<code>\$exitcode</code>	<code>_exit(2)</code> または <code>exit(3)</code> に渡された引数の値、または <code>main</code> の戻り値

表 6-3 `dlopen` および `dlclose` イベントに固有の変数

変数	説明
<code>\$dlobj</code>	<code>dlopen</code> または <code>dlclose</code> されたロードオブジェクトのパス名

表 6-4 `sysin` および `sysout` イベントに固有の変数

変数	説明
<code>\$syscode</code>	システムコールの番号
<code>\$sysname</code>	システムコールの名前

表 6-5 `proc_gone` イベントに固有の変数

変数	説明
<code>\$reason</code>	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。

配列メンバーへの書き込みに対するウォッチポイントを設定する

配列 `array[99]` にウォッチポイントを設定します。

```
(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 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
...
(dbx) restore # run が自動的に行われ、sync イベントが
              # 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
...
```


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和、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。

呼び出しスタックがメモリー上位からメモリー下位に成長することから、「上に移動する」(`up`) はメモリーを上位に移動して呼び出し側に向かうこと、「下に移動する」(`down`) はメモリー内を下位に移動することを意味します。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] [-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。(2 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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」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
```

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「`up` コマンド」を参照してください。

スタックの下方方向への移動

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
```

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「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 (94 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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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。

正規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、`sh` または `ksh` の正規表現スタイルをとります。

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 ]
```


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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」の「`which` コマンド」および「`where` 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。31 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。28 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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」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ます、`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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「whatis コマンド」および「C++ で使用される 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せん。

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っ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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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。

配列の例：

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

配列を評価するには、次のようにします。

```
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]
```

<i>arr-exp</i>	配列またはポインタ型に評価されるべき式
<i>first-exp</i>	印刷される最初の要素。デフォルトは 0
<i>last-exp</i>	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
<i>stride-exp</i>	刻み幅の長さ (スキップされる要素の数は <i>stride-expression-1</i>)。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)
```

<i>array-expression</i>	配列ポインタ型に評価される式
<i>first-expression</i>	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
<i>last-expression</i>	範囲内の最後の要素。ただし刻み幅が 1 より大きい場合、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
<i>stride-expression</i>	刻み幅 (スキップされる要素の数は <i>stride-expression</i> -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
Reading symbolic information for a.out
(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
Running: a.out
stopped in MAIN at line 7 in file "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(3:3, 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。

	100	101	102	103	104	105	106
200	□	□	□	□	□	□	□
201	□	⊗	⊗	⊗	⊗	⊗	□
202	□	⊗	⊗	⊗	⊗	⊗	□
203	□	⊗	⊗	⊗	⊗	⊗	□
204	□	□	□	□	□	□	□
205	□	□	□	□	□	□	□

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* 個の要素のうち先頭の1つが評価され、残りの *stride-expression* - 1 個の要素は評価されません。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。

	100	101	102	103	104	105	106
200	□	□	□	□	□	□	□
201	□	⊗	□	⊗	□	⊗	□
202	□	⊗	□	⊗	□	⊗	□
203	□	⊗	□	⊗	□	⊗	□
204	□	□	□	□	□	□	□
205	□	□	□	□	□	□	□

`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

<code>print arr</code>	デフォルトの境界で配列全体を出力します。
<code>print arr(:)</code>	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。
<code>print arr(:,:,stride-expression)</code>	配列全体を <i>stride-expression</i> 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) を使用すると、開発段階でアプリケーションの実行時エラーを自動的に検出できます。RTC では、メモリアクセスエラーおよびメモリーリークエラーなどの実行時エラーを検出して、メモリーの使用状況を監視できます。

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
- システムライブラリなど、ソースを持たないコードで動作する
- 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せん。

制限事項

RTC は、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。詳細については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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」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

librt.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 0xeffff068 で しようとした

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 サイズ	ブロック 数	リーク ブロック アドレス	割り当て呼出しスタック
-----------	-----------	---------------------	-------------

```
=====
      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 サイズ	割合 %	ブロック 数	平均 サイズ	割り当て呼出しスタック
-----------	---------	-----------	-----------	-------------

```
=====
      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 割り当て	エラー時の呼び出しスタック (パッチモード)。 <i>addr</i> 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;
```

メモリアクセスエラー

RTC は、以下のメモリ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、アレイ境界チェックは行いません。したがって、アレイ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。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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」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 サイズ	ブロック 数	リーク ブロック アドレス	割り当て呼出しスタック	
=====	=====	=====	=====	
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 サイズ	ブロック 数	リーク ブロック アドレス	割り当て呼出しスタック	
=====	=====	=====	=====	
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 (me1):		
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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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増えるのを避けるため、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せん。

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となります。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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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

メモリー使用状況検査は、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、Sun WorkShop オンラインヘルプのdbx コマンドの使い方」の「shownyuse 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 サイズ	割合 %	ブロック 数	平均 サイズ	割り当て呼び出しスタック
=====	=====	=====	=====	=====
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続きます。

エラーは `suppress` コマンド (Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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 (Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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_limit` 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回報告されません。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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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 `ru` や共有ライブラリの `ru`, `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「非初期化メモリからの読み取り (`ru`)」を抑制しません。

```
unsuppress -d ru
```

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「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 `ru`、`rua`、`wua` に関連したもので、この順序で検出されます。`ru` 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 `rua` 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 `rua` 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 `rua` 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 }
%

```

```

% 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
librt.c.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 0xeffff110 で しようとした
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;

```

RTC は親プロセス、
[program1](#) の最初のエラー
を報告します。

`follow_fork_mode` が `child` に設定されているため、フォークが起ると、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、librttc.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

RTC は接続されたプロセスに機能します。ただし、接続されたプロセスに RTC を使用するには、`librtc.so` を事前に読み込んでおく必要があります。たとえば、`librtc.so` は、Sun WorkShop の `lib` ディレクトリにインストールされます。したがって、製品を `/opt` にインストールすると、`librtc.so` は `/opt/SUNWspro/lib` にインストールされます。

`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...
% start-your-application
% 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います
librttc.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アccess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、メモリアccess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イクファイルで `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

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。そ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. トラップを使用します。

UltraSparc ハードウェアでは、ブランチではなくトラップハンドラを呼び出す機能が `dbx` に追加されました。トラップハンドラへの制御の移行には非常に時間がかかりますが (最大で 10 分)、8M バイトの制限がなくなります。トラップハンドラを有効にするには、`dbx rtc_use_traps` 環境変数を使用します (19 ページの「`dbxenv` コマンドでの `dbx` 環境変数の設定」参照)。ハードウェアが UltraSparc であれば、この変数は 32 ビットプログラムと 64 ビットプログラムで使用できます。ハードウェアを検査するには、システムコマンド `uname -m` を使用し、結果が `sun4u` であるかを確認します。

3. 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り、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。

4. 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ることができます。

5. 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有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るようとした。

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 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るうとした。

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、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、すべてのスカラー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。

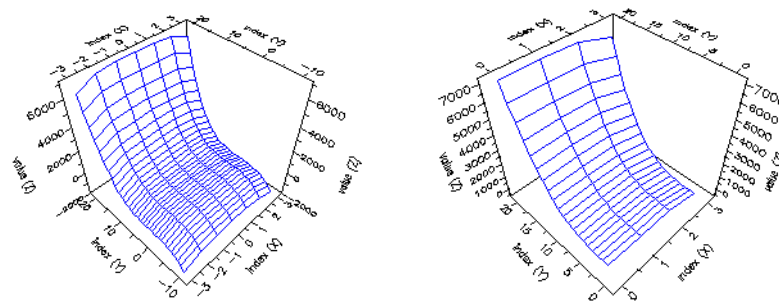


図 10-1 bf 配列名だけのグラフ(左)、bf(0:3,1:20)のグラフ(右)

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。

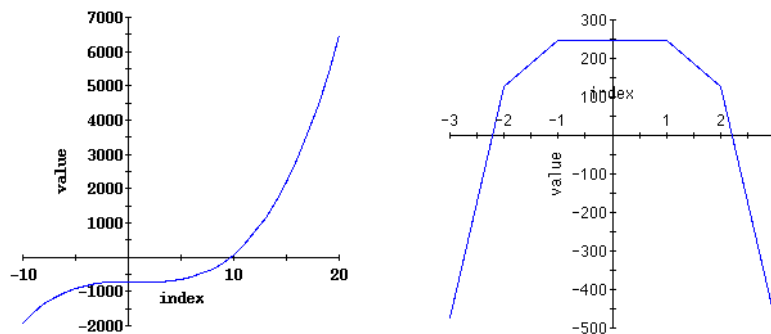


図 10-2 配列 bf : bf(3,:) (左)、bf(:,7) (右)

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と比較できます。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 (Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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」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」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。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、標準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、「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。

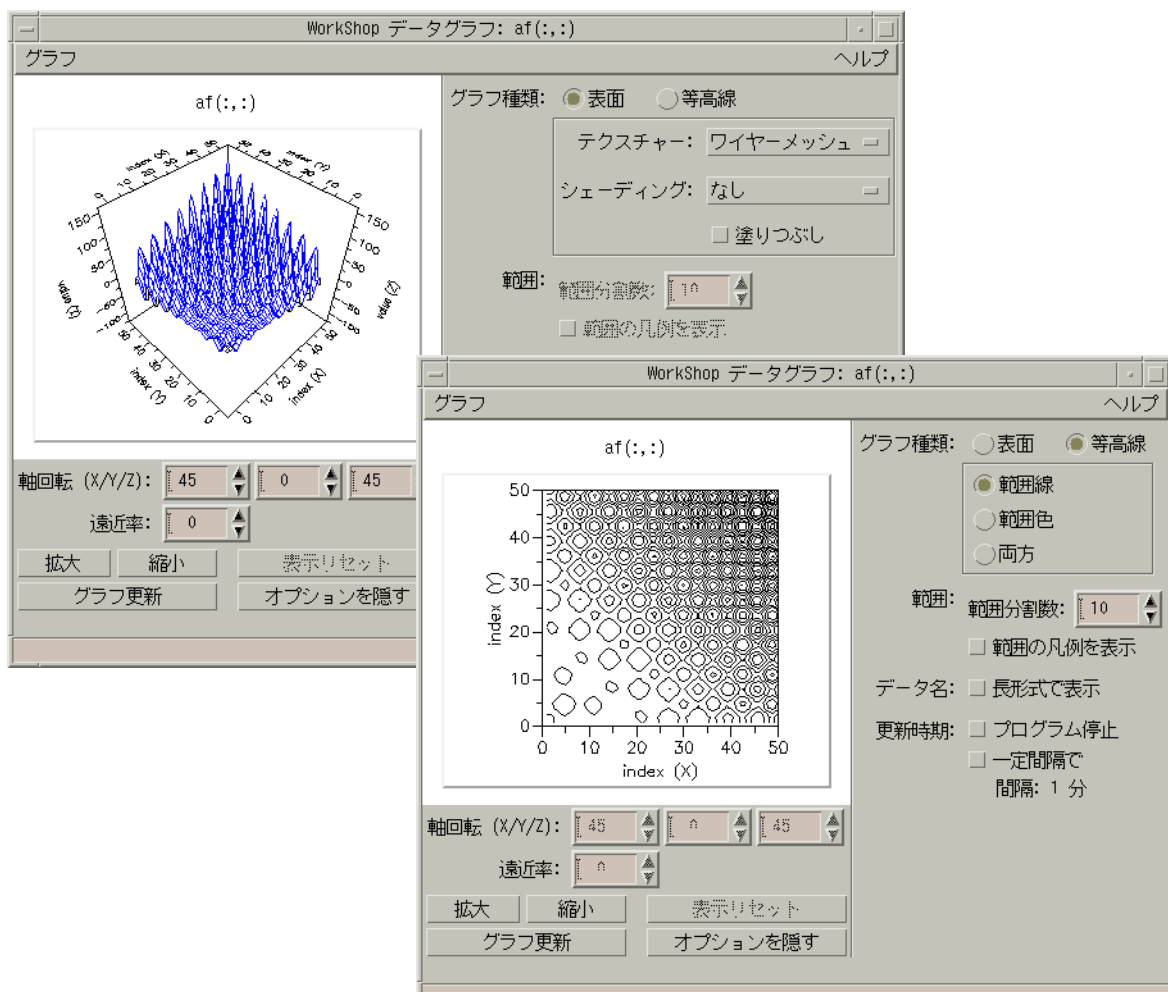


図 10-3 同じ配列の2種類の表示画面

「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。

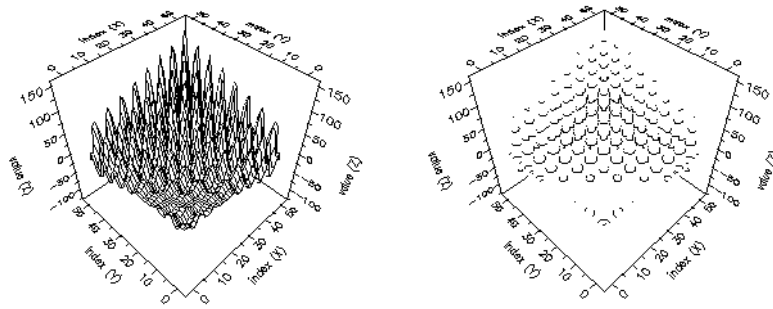


図 10-4 ワイヤーメッシュ(左)と範囲線(右)の「表面」グラフ

「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。

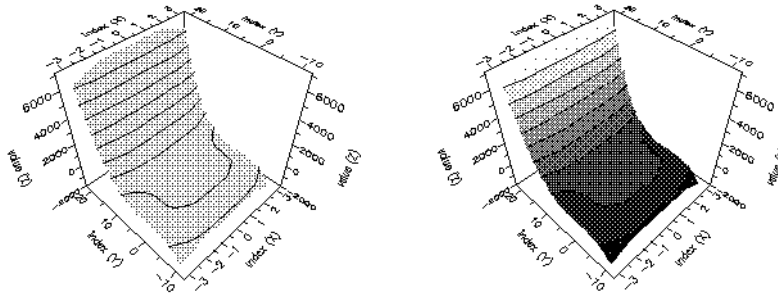


図 10-5 「光源」(左)と「範囲色」(右)の「表面」グラフ

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。

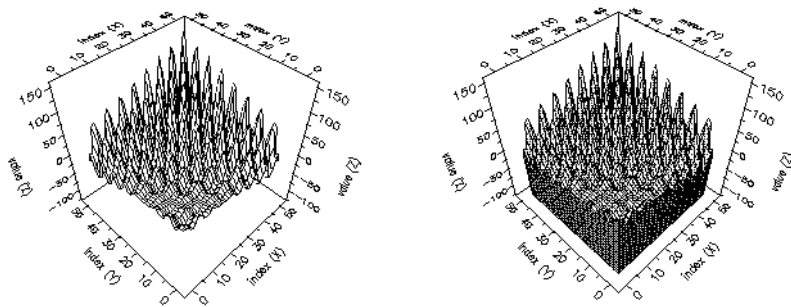


図 10-6 陰影付き (左) と表面を塗りつぶした (右) 「表面」グラフ

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。

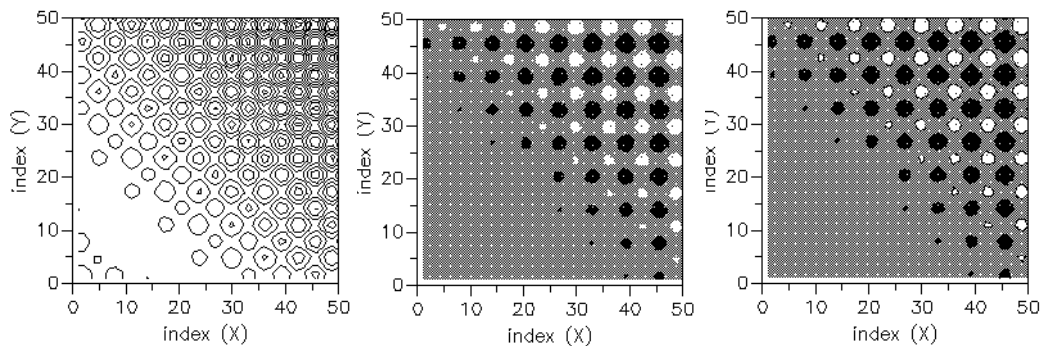


図 10-7 ライン (左)、カラー (中央)、両方 (右) による「等高線」グラフ

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。

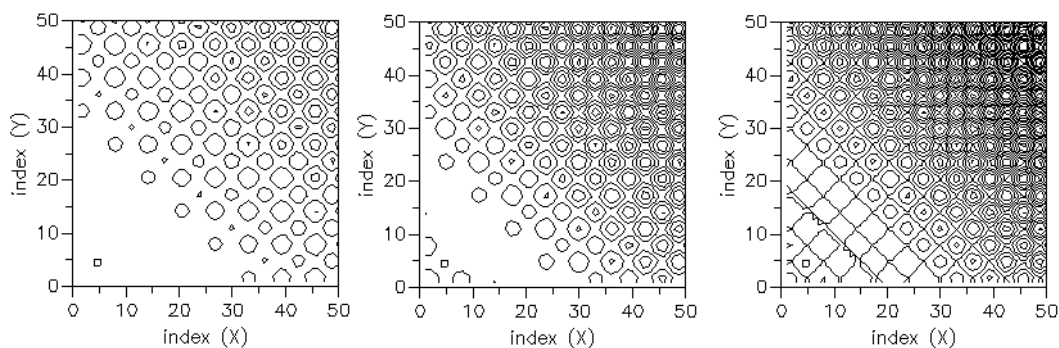


図 10-8 「範囲: 分割数」が (左から右に) 5、10、15 の「等高線」グラフ
 分割数を大きくすると、カラーマップに悪影響が出ることがあります。

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/SUNWsprow/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「データグラフィック」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 番目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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」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。Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」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 (Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」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 (Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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ず。

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。(Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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にアクセスするには、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_swch()
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	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りソースとして LWP を待機しています。
ゾンビ		結合されていないスレッド (<code>thr_exit()</code>) がある場合、 <code>thr_join()</code> で再結合するまでゾンビ状態になります。 <code>THE_DETACHED</code> は、スレッド作成時に指定するフラグです (<code>thr_create()</code>)。非結合のスレッドは、再実行されるまでゾンビ状態です。
<code>syncobj</code> 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。 <code>libthread</code> と <code>libthread_db</code> によるサポートレベルにより、 <code>syncobj</code> 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、 <code>dbx</code> は LWP をアクセスできません。
未知		<code>dbx</code> では状態を判定できません。
<code>lwpstate</code>		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 <code>num</code>		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 <code>num</code> 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 <code>num</code>		LWP に所定の障害番号が発生しました。
シグナル <code>name</code>		LWP に所定のシグナルが発生しました。
プロセス <code>sync</code>		この LWP が所属するプロセスの実行が開始しました。
LWP デス		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] tid
```

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 *tid* に切り替えるには、次のように入力します。

```
thread tid
```

`thread` コマンドの詳細については、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」の「`thread` コマンド」を参照してください。

スレッドリストの表示

次のコマンドは、`threads` リストを表示するためのものです。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
```

`threads` コマンドの詳細については、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」の「`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イ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 ブレイクポイント：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。
- 177 ページの「スレッドの状態と LWP の状態」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 progname pid
```

コマンド行からは次のように入力します。

```
(dbx) debug progname pid
```

どちらの場合も *prognam*e 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 (*pid*) 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。

- | | |
|-----------------------------|---|
| <code>parent</code> (親プロセス) | 従来の動作です。 <code>dbx</code> は <code>fork</code> を無視し、親プロセスを追跡します。 |
| <code>child</code> (子プロセス) | <code>dbx</code> 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。 |
| <code>both</code> (両方) | このモードは、Sun WorkShop から <code>dbx</code> 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かせてきます。`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、第 6 章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。

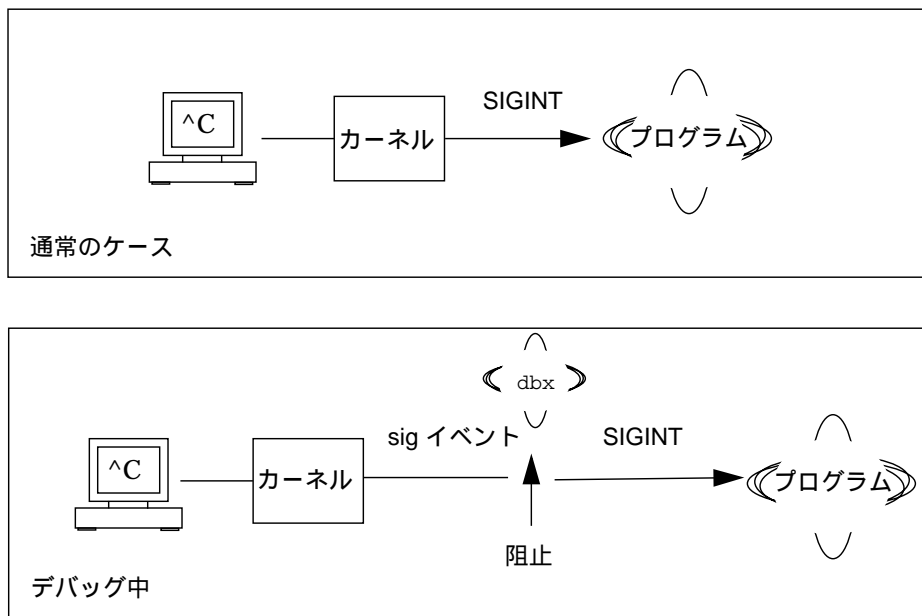


図 14-1 SIGINT シグナルの阻止と取り消し

- 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 (Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」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 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。(Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」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、6 ページの「最適化コードのデバッグ」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 (37 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 (51 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 (57 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 (57 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 (58 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 (54 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 (59 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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」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、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」の「`exception` 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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「exception 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。

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 番目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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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<int>  
class Array<int> {  
public:  
    int Array<int>::getlength();  
    int &Array<int>::operator[] (int i);  
    Array<int>::Array<int>(int l);  
    Array<int>::~~Array<int>();  
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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「whatis コマンド」と「C++ で使用する whatis コマンド」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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「stop コマンド」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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「stop コマンド」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 1)
(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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「stop コマンド」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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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せん。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、`-On` を付けずにメインプログラムをコンパイルする。
- 適切な `-On` 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、`-On` 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
```

デバッグの関数

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
wewidth.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、104 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
```

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

```
(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
```

組み込み関数

`dbx` は、Fortran の組み込み関数を認識します。

`dbx` での組み込み関数を示します。

```
(dbx) list 1,$
      1      INTEGER i
      2      i = -2
      3      END
(dbx) stop in MAIN
(2) stop in MAIN
(dbx) run
実行中: a.out
(プロセス id 23536)
mb.so.1 の読み込み中
wewidth.so.1 の読み込み中
MAIN で停止しました 行番号 2   ファイル "ShowIntrinsic.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 "ShowIntrinsic.f"
      3      END
(dbx) print i
i = -2
(dbx) print abs(i)
abs(i) = 2
(dbx) quit
demo%
```

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
wccwidth.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% f77 -g -silent 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) "ShowLogical.f" で停止しました:
(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

構造型、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

構造型、Fortran 95 構造型およびポインタを `dbx` で表示できます。

`DebStruct.f95`

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)

```
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
```

`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せん。

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。(221 ページの「`dis` コマンドの使用」と221 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。

<code>0xff99</code>	絶対アドレス
<code>main</code>	関数のアドレス
<code>main+20</code>	関数アドレス + オフセット
<code>&errno</code>	変数のアドレス
<code>str</code>	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。

<code>i</code>	アセンブラ命令として表示
<code>d</code>	10 進表記の16 ビット (2 バイト) で表示
<code>D</code>	10 進表記の32 ビット (4 バイト) で表示
<code>o</code>	8 進表記の16 ビット (2 バイト) で表示

<code>O</code>	8 進表記の32 ビット (4 バイト) で表示
<code>x</code>	16 進表記の16 ビット (2 バイト) で表示
<code>X</code>	16 進表記の32 ビット (4 バイト) で表示 (デフォルトの書式)
<code>b</code>	8 進表記のバイトで表示
<code>c</code>	1 バイトの文字で表示
<code>w</code>	ワイド文字で表示
<code>s</code>	NULL バイトで終わる文字列で表示
<code>W</code>	ワイド文字列で表示
<code>f</code>	単精度浮動小数点数として表示
<code>F, g</code>	倍精度浮動小数点数として表示
<code>E</code>	拡張精度浮動小数点数として表示
<code>ld, lD</code>	10 進数として 32 ビット (4 バイト) で表示 (<code>D</code> と同じ)
<code>lo, lO</code>	8 進数として 32 ビット (4 バイト) で表示 (<code>O</code> と同じ)
<code>lx, lX</code>	16 進数として 32 ビット (4 バイト) で表示 (<code>X</code> と同じ)
<code>Ld, LD</code>	10 進数として 64 ビット (8 バイト) で表示
<code>Lo, LO</code>	8 進数として 64 ビット (8 バイト) で表示
<code>Lx, LX</code>	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。30 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]
```

Intel 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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「next コマンド」と「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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「nexti コマンド」と「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、59 ページの「コードをトレースする」と、Sun WorkShop オンラインヘルプの「`dbx` コマンドの使い方」の「`nexti` コマンド」と「`trace` 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。

<code>tracei step</code>	各命令をトレース
<code>tracei next</code>	各命令をトレースするが、呼び出しを飛び越します。
<code>tracei at address</code>	指定のコードアドレスをトレース

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「`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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」の「stopi 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
stopped in hand::ungrasp at 0x12638
(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、Sun WorkShop オンラインヘルプの「dbx コマンドの使い方」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、「Sun WorkShop オンラインヘルプ」の「dbx コマンドの使い方」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  regs -F
current thread: t@1
current frame: [1]
g0-g3  0x00000000 0x0011d000 0x00000000 0x00000000
g4-g7  0x00000000 0x00000000 0x00000000 0x00020c38
o0-o3  0x00000003 0x00000014 0xef7562b4 0xffff420
o4-o7  0xef752f80 0x00000003 0xffff3d8 0x000109b8
l0-l3  0x00000014 0x0000000a 0x0000000a 0x00010a88
l4-l7  0xffff438 0x00000001 0x00000007 0xef74df54
i0-i3  0x00000001 0xffff4a4 0xffff4ac 0x00020c00
i4-i7  0x00000001 0x00000000 0xffff440 0x000108c4
y      0x00000000
psr    0x40400086
pc     0x000109c0:main+0x4      mov      0x5, %l0
npc    0x000109c4:main+0x8      st       %l0, [%fp - 0x8]
f0f1   +0.000000000000000e+00
f2f3   +0.000000000000000e+00
f4f5   +0.000000000000000e+00
f6f7   +0.000000000000000e+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
<code>\$g0-\$g7</code>	「大域」レジスタ
<code>\$o0-\$o7</code>	「出力」レジスタ
<code>\$l0-\$l7</code>	「局所」レジスタ
<code>\$i0-\$i7</code>	「入力」レジスタ

レジスタ	説明
<code>\$fp</code>	フレームポインタ (レジスタ <code>\$i6</code> と等価)
<code>\$sp</code>	スタックポインタ (レジスタ <code>\$o6</code> と等価)
<code>\$y</code>	Y レジスタ
<code>\$psr</code>	プロセッサ状態レジスタ
<code>\$wim</code>	ウィンドウ無効マスクレジスタ
<code>\$tbr</code>	トラップベースレジスタ
<code>\$pc</code>	プログラムカウンタ
<code>\$npc</code>	次のプログラムカウンタ
<code>\$f0-\$f31</code>	FPU “f” レジスタ
<code>\$fsr</code>	FPU 状態レジスタ
<code>\$fq</code>	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 “double” 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ユアルバージョン 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
<code>\$gs</code>	代替データセグメントレジスタ
<code>\$fs</code>	代替データセグメントレジスタ
<code>\$es</code>	代替データセグメントレジスタ
<code>\$ds</code>	データセグメントレジスタ
<code>\$edi</code>	デスティネーションインデックスレジスタ
<code>\$esi</code>	ソースインデックスレジスタ
<code>\$ebp</code>	フレームポインタ
<code>\$esp</code>	スタックポインタ
<code>\$ebx</code>	汎用レジスタ
<code>\$edx</code>	汎用レジスタ
<code>\$ecx</code>	汎用レジスタ
<code>\$eax</code>	汎用レジスタ
<code>\$trapno</code>	例外ベクトル番号
<code>\$err</code>	例外を示すエラーコード
<code>\$eip</code>	命令ポインタ
<code>\$cs</code>	コードセグメントレジスタ
<code>\$eflags</code>	フラグ
<code>\$es</code>	代替データセグメントレジスタ
<code>\$uesp</code>	ユーザースタックポインタ
<code>\$ss</code>	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
<code>\$sp</code>	スタックポインタ (<code>\$uesp</code> と同じ)。
<code>\$pc</code>	プログラムカウンタ (<code>\$eip</code> と同じ)。
<code>\$fp</code>	フレームポインタ (<code>\$ebp</code> 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
<code>\$ax</code>	汎用レジスタ
<code>\$cx</code>	汎用レジスタ
<code>\$dx</code>	汎用レジスタ
<code>\$bx</code>	汎用レジスタ
<code>\$si</code>	ソースインデックスレジスタ
<code>\$di</code>	デスティネーションインデックスレジスタ
<code>\$ip</code>	命令ポインタ (下位 16 ビット)
<code>\$flags</code>	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せん。

レジスタ	説明
<code>\$al</code>	レジスタの下位 (右) 部分 <code>\$ax</code>
<code>\$ah</code>	レジスタの上位 (左) 部分 <code>\$ax</code>
<code>\$cl</code>	レジスタの下位 (右) 部分 <code>\$cx</code>
<code>\$ch</code>	レジスタの上位 (左) 部分 <code>\$cx</code>
<code>\$dl</code>	レジスタの下位 (右) 部分 <code>\$dx</code>
<code>\$dh</code>	レジスタの上位 (左) 部分 <code>\$dx</code>
<code>\$bl</code>	レジスタの下位 (右) 部分 <code>\$bx</code>
<code>\$bh</code>	レジスタの上位 (左) 部分 <code>\$bx</code>

80387 用レジスタは次のとおりです。

レジスタ	説明
<code>\$fctrl</code>	コントロールレジスタ
<code>\$fstat</code>	状態レジスタ
<code>\$ftag</code>	タグレジスタ
<code>\$fip</code>	命令ポインタオフセット
<code>\$fcs</code>	コードセグメントセクタ
<code>\$fopoff</code>	オペランドポインタオフセット
<code>\$fopsel</code>	オペランドポインタセクタ
<code>\$st0 - \$st7</code>	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 -Aname` 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、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。これはデバッグ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ため、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る場合は、`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。

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さなくなります。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、`expressions` 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。

<code>pop</code>	カレントフレームをポップ
<code>pop number</code>	<code>number</code> 個のフレームをポップ
<code>pop -f number</code>	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。

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。

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。

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。

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。

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。

索引

記号

`::` (二重コロン) C++ 演算子 32

A

`adb` コマンド 236

`adb` モード 236

`alias` コマンド 5

`allow_critical_exclusion` 環境変数 20

`aout_cache_size` 環境変数 20

`array_bounds_check` 環境変数 20

`assign` コマンド 101, 171, 173, 250

B

`bcheck` コマンド 140

構文 140

例 140

`button` コマンド 18

C

C++

`dbx` 使用 191

`-g0` オプションでコンパイル 6

`-g` オプションでコンパイル 6

印刷 98

オブジェクトポインタ型 98

関数テンプレートインスタンス化, リスト 36

逆引用符演算子 31

クラス

印刷の宣言 37

継承されたすべてのデータメンバーを表示
99

継承メンバーの表示 39

宣言, 検索 36

直接定義されたすべてのデータメンバーを
表示 99

定義, 検索 37

表示 36

継承メンバー 39

さまざまな名前 33

テンプレート定義

修正 174

表示 36

テンプレート デバッグ 197

二重コロンスコープ決定演算子 32

複数のブレークポイントの設定 57

無名引数 99

メンバー関数のトレース 59

例外処理 192

C++

あいまいな名前または多重定義関数 29

`call` コマンド 51, 203, 251

`catch` コマンド 187, 189

`cfront_demangling` 環境変数 20

`check` コマンド 111, 112

cont コマンド 49, 113, 170, 171, 173, 179, 252
デバッグ情報なしでコンパイルできるファイルの制限 168
C 配列例プログラム 165

D

dbx

- Korn シェル機能 243

dbx, 起動 1

- process_ID でのみ 3
- コアファイル名 2

dbxenv コマンド 5, 19

.dbxrc ファイル 15, 18

- 作成 16
- サンプル 16

.dbxrc ファイルの例 16

dbx が使用するシンボルの決定 35

dbx から切り離す 47

dbx からのプロセスの切り離し 9

dbx からプロセスを切り離す 47

dbx 環境変数

- rtc_auto_continue 141

dbx 環境変数 20

- allow_critical_exclusion 20
- aout_cache_size 20
- array_bounds_check 20
- cfront_demangling 20
- dbxenv コマンドで設定 19
- delay_xs 20, 41
- disassembler_version 21
- fix_verbose 21
- follow_fork_inherit 21, 183
- follow_fork_mode 21, 131, 182
- follow_fork_mode_inner 21
- input_case_sensitive 21, 206, 207
- Korn シェル 17
- language_mode 22
- locache_enable 22
- mt_scalable 22
- output_auto_flush 22
- output_base 22
- output_derived_type 99
- output_dynamic_type 22, 193
- output_inherited_members 22
- output_list_size 22
- output_log_file_name 22
- output_max_string_length 22
- output_pretty_print 23
- output_short_file_name 23
- overload_function 23
- overload_operator 23
- pop_auto_destruct 23
- proc_exclusive_attach 23
- rtc_auto_continue 23, 112
- rtc_auto_suppress 23, 129
- rtc_biu_at_exit 23, 125
- rtc_error_limit 24, 129
- rtc_error_log_file_name 24, 113, 141
- rtc_error_stack 24
- rtc_inherit 24
- rtc_mel_at_exit 24
- rtc_use_traps 24
- run_autostart 24
- run_io 25
- run_pty 25
- run_quick 25
- run_savetty 25
- run_setpgrp 25
- scope_global_enums 25
- scope_look_aside 25
- session_log_file_name 26
- stack_find_source 26
- stack_max_size 26
- stack_verbose 26
- step_events 26, 65
- step_granularity 26, 49
- suppress_startup_message 26
- symbol_info_compression 26
- trace_speed 26, 60

dbx セッションの終了 8

debug コマンド 181

delay_xs 環境変数 20, 41

detach コマンド 9, 47

disassembler_version 環境変数 21

display コマンド 100

dis コマンド 231

dlopen()

ブレイクポイントの設定 56
ブレイクポイントの例外 56
down コマンド 93

E

examine コマンド 228
exception コマンド 193
exec 関数, 追跡 182

F

fflush(stdout), dbx の呼び出し後 51
file コマンド 29
fix_verbose 環境変数 21
fix コマンド 168, 169, 252
 効果 169
 デバッグ情報なしでコンパイルできるファイルの制限 168
follow_fork_inherit 環境変数 21, 183
follow_fork_mode_inner 環境変数 21
follow_fork_mode 環境変数 21, 131, 182
fork 関数, 追跡 182

Fortran

 大文字小文字を区別 206
 組み込み関数 218
 構造 221
 配列断面化の構文 104
 配列例プログラム 164
 派生型 221
 複雑な式 219
 論理演算子 220
 割り当て可能な配列 215
FPE シグナル, トラップ 187
frame コマンド 93
func コマンド 29

G

-g オプションなしでコンパイルされたコード 7

-g コンパイラオプション 6

H

handler コマンド 70
hide コマンド 94

I

ignore コマンド 186, 187
input_case_sensitive 環境変数 21, 206, 207
Intel レジスタ 239
intercept コマンド 193

K

kill コマンド 9, 121
Korn シェル
 拡張機能 244
 実装されていない機能 243
 名前が変更されたコマンド 244

L

language_mode 環境変数 22
LD_PRELOAD 136
librt.so, 読み込み済み 135
librtld_db.so 246
libthread.so 175
libthread_db.so 175
listi コマンド 232
list コマンド 30, 204
loadobject 246
locache_enable 環境変数 22
LWPs (軽量プロセス) 175
 情報を表示 179
lwps コマンド 179

M

- `modify` コマンド 63
- `modules` コマンド 42, 43
- `module` コマンド 42
- `mt_scalable` 環境変数 22

N

- `nexti` コマンド 233

O

- `output_auto_flush` 環境変数 22
- `output_base` 環境変数 22
- `output_derived_type` 環境変数 99
- `output_dynamic_type` 環境変数 22, 193
- `output_inherited_members` 環境変数 22
- `output_list_size` 環境変数 22
- `output_log_file_name` 環境変数 22
- `output_max_string_length` 環境変数 22
- `output_pretty_print` 環境変数 23
- `output_short_file_name` 環境変数 23
- `overload_function` 環境変数 23
- `overload_operator` 環境変数 23

P

- `pathmap` コマンド 4, 28, 170
- `pop_auto_destruct` 環境変数 23
- `popping`
 - 1つの呼び出しスタックフレーム 173
 - 呼び出しスタック 171, 251
- `pop` コマンド 94, 173, 251
- `print` コマンド 98, 100, 103, 104, 203, 251
- `proc_exclusive_attach` 環境変数 23

R

- `regs` コマンド 236

- `replay` コマンド 9, 13
- `restore` コマンド 9, 12
- `rtc_auto_continue` 環境変数 23, 112, 141
- `rtc_auto_suppress` 環境変数 23, 129
- `rtc_biu_at_exit` 環境変数 23
- `rtc_error_limit` 環境変数 24, 129
- `rtc_error_log_file_name` 環境変数 24, 113, 141
- `rtc_error_stack` 環境変数 24
- `rtc_inherit` 環境変数 24
- `rtc_mel_at_exit` 環境変数 24
- `rtc_use_traps` 環境変数 24
- `rtld` 245
- `run_autostart` 環境変数 24
- `run_io` 環境変数 25
- `run_pty` 環境変数 25
- `run_quick` 環境変数 25
- `run_savetty` 環境変数 25
- `run_setpgrp` 環境変数 25
- `run` コマンド 45

S

- `save` コマンド 9
- `scope_global_enums` 環境変数 25
- `scope_look_aside` 環境変数 25
- `session_log_file_name` 環境変数 26
- `showblock` コマンド 111
- `showleaks` コマンド 120, 124, 125, 129
- `showmemuse` コマンド 125
- SPARC レジスタ 237
- `stack_find_source` 環境変数 26
- `stack_max_size` 環境変数 26
- `stack_verbose` 環境変数 26
- `step_events` 環境変数 26, 65
- `step_granularity` 環境変数 26, 49
- `stepi` コマンド 233
- `step` コマンド 192
- `stop at` コマンド 54

`stop change` コマンド 62
`stop inclass` コマンド 57
`stop inmember` コマンド 57
`stopi` コマンド 235
`stop modify` コマンド 61
`stop` コマンド 202
Sun WorkShop デバッグオプションダイアログ
ボックス 17
`suppress_startup_message` 環境変数 26
`suppress` コマンド 113, 128, 130
`symbol_info_compression` 環境変数 26

T

`thread` コマンド 178
`tracei` コマンド 234
`trace_speed` 環境変数 26, 60
`trace` コマンド 59

U

`uncheck` コマンド 112
`undisplay` コマンド 101
`unhide` コマンド 95
`unintercept` コマンド 194
`unsuppress` コマンド 128, 130
`up` コマンド 92

V

`vitem` コマンド 154

W

`whatis` コマンド 36, 37, 99, 200
`when infunction` コマンド 58
`when` コマンド 56, 68, 252
`when` ブレークポイント行, 設定 56
`whereis` コマンド 34, 97, 199

`where` コマンド 92, 213
`which` コマンド 30, 35, 97
`whocatches` コマンド 194
`.workshoprc` ファイル 17

X

`x` コマンド 228

あ

あいまいな関数名リストから選択 29
アクセス検査 116
アセンブリ言語 デバッグ 227
アドレス
内容を調べる 227
表示書式 229
アプリケーションファイルを再設定して再実行
88

い

移動

呼び出しスタックの指定フレーム 93
呼び出しスタックを上へ 92
呼び出しスタックを下へ 93

イベント

あいまいさ 82
ウォッチポイント 61
解析 82
子プロセスの対話 183

イベントカウンタ 70

イベント指定 67, 68, 70, 235

イベントの他の型 78
ウォッチポイントイベント 71
キーワード, 定義 70
システムイベント 73
修飾子 80
進行イベント実行 76
設定 70

- 定義済み変数の使用 83
- ブレークポイントイベント 70
- イベント指定のための定義済み変数 83
- イベント指定 変数 84
- イベント発生後にブレークポイントを有効にする 88
- イベントハンドラ
 - クリア 60
 - 作成 68
 - 設定, 例 86
 - 操作 69
 - リスト 60
- イベントハンドラのクリア 60
- イベントハンドラの操作 69
- 印刷
 - インスタンス化された指定関数のソースリスト 204
 - 型または C++クラスの宣言 37
 - 関数テンプレートインスタンス化の値 197
 - 既知のすべてのスレッドのリスト 178
 - 現在のモジュールの名前 42
 - 式の値 251
 - シンボルの出現リスト 34
 - すべての機械レベルレジスタの値 236
 - すべてのクラスと関数テンプレートインスタンス化のリスト 197, 199
 - ソースリスト 30
 - 通常印刷されない (ゾンビ) 印刷リスト 178
 - データメンバー 36
 - 配列 102
 - フィールド型 37
 - 変数型 37
 - 変数または式の値 98
 - ポインタ 225
 - メンバー関数 36
- インスタンス, 定義を表示 197, 200

- う
- ウォッチポイントイベント指定 71

- え
- エラー抑制 127, 128
 - 型 128
 - デフォルト 130
 - 例 129
- 演算子
 - C++ 二重コロンスコープ決定 32
 - 逆引用符 31
 - ブロックローカル 32

- お
- 大文字小文字を区別, Fortran 206
- オブジェクトポインタ型 98

- か
- 開始
 - メモリアクセス検査 111, 112
 - メモリー使用状況検査 111, 112
 - メモリーリーク検査 112

- カスタマイズ
 - dbx 15, 17
 - Sun WorkShop 17
 - 配列の表示 155
- 「カスタムボタン」ウィンドウ 18
- カスタムボタンを保存 18

- 型
 - 印刷の宣言 37
 - 検索の宣言 36
 - 検索の定義 37
 - 宣言, 検索 36
 - 派生, Fortran 221
 - 表示 36

- 関数
 - C++ コードでのブレークポイントの設定 57
 - あいまいな名前または多重の 29
 - インスタンス化
 - ソースリストを出力 204
 - 評価 203
 - 呼び出し 203
 - 組み込み, Fortran 218

- クラステンプレートのメンバー, 評価 203
- クラステンプレートのメンバー, 呼び出し 203
- 検索の定義 36
- コンパイラで割り当てられた名前を保持 99
- 実行, 変更 170
- 実行中, 変更 170
- スタックにある, 変更 171
- 表示 29
- ブレークポイント設定 54
- 呼び出されていない, 変更 170
- 呼び出し 51
- 関数テンプレートインスタンス化
 - 値を出力 197
 - ソースコードを表示 197
 - リスト出力 197, 199
- 関数引数, 無名
 - 評価 100
 - 表示 100

き

- 機械命令レベル
 - Intel レジスタ 239
 - SPARC レジスタ 237
 - アドレスでブレークポイント設定 235, 236
 - シングルステップ 233
 - すべてのレジスタの値を出力 236
 - デバッグ 227
 - トレース 234
- 機械命令レベルでトレース 234
- 逆引用符演算子 31
- 共有オブジェクト
 - 修正と継続 247
 - デバッグサポート 247
 - ブレークポイントで設定 247
- 共有オブジェクト, 修正 168
- 共有ライブラリ, dbx 用にコンパイル 8

く

- クラス

- 印刷の宣言 37
- 継承されたすべてのデータメンバーを表示 99
- 継承メンバーの表示 39
- 検索の宣言 36
- 検索の定義 37
- 直接定義されたすべてのデータメンバー 99
- 表示 36
- クラステンプレートインスタンス化, リスト出力 197, 199

け

- 継承メンバー
 - 表示 38, 39
- 決定
 - 実行行数 87
 - 実行命令数 87
 - ソース行実行の精度 49
- 現在の手順とファイル 206

検索

- `this` ポインタ 37
- オブジェクトファイル 4
- 型の定義 37
- 関数の定義 36
- クラス定義 37
- ソースファイル 4
- 変数の定義 36
- メンバーの定義 36
- 呼び出しスタックの位置 91

検出

- オブジェクトファイル 4
- ソースファイル 4

こ

- コアファイルデバッグ 2
- 子プロセス
 - イベントと対話 183
 - 実行時検査を使用 131
 - 接続 `dbx` to 181
 - デバッグ 181

コマンド

adb 236
adb(1) 構文に入力 236
alias 5
assign 101, 171, 173, 250
bcheck 140
button 18
call 51, 203, 251
catch 187, 189
check 111, 112
cont 49, 113, 170, 171, 173, 179, 252
 デバッグ情報なしでコンパイルできるファ
 イルの制限 168
dbxenv 5, 19
debug 181
detach 47
dis 231
display 100
down 93
examine 228
exception 193
file 29
fix 168, 169, 252
 効果 169
 デバッグ情報なしでコンパイルできるファ
 イルの制限 168
frame 93
func 29
handler 70
hide 94
ignore 186, 187
intercept 193
kill 9, 121
list 30, 204
listi 232
lwps 179
modify 63
module 42
modules 42, 43
nexti 233
pathmap 4, 28, 170
pop 94, 173, 251
print 98, 100, 103, 104, 203, 251
regs 236
replay 9
restore 9, 12
run 45

save 9
showblock 111
showleaks 120, 124, 125, 129
showmemuse 125
step 192
stepi 233
stop 202
stop at 54
stop change 62
stopi 235
stop inclass 57
stop inmember 57
stop modify 61
suppress 113, 128, 130
thread 178
trace 59
tracei 234
uncheck 112
undisplay 101
unhide 95
unintercept 194
unsuppress 128, 130
up 92
vitem 154
whatis 36, 37, 99, 200
when 56, 68, 252
when infunction 58
where 92, 213
whereis 34, 97, 199
which 30, 35, 97
whocatches 194
x 228
 detach 9
 replay 13
 プログラムの状態を変更 250
 プロセス制御 45
コンパイラで割り当てられた関数名を保持 99
コンパイル
 -g オプションで 6
 -O オプションで 6
 最適化コード 6

さ
再開

- 指定した行でプログラム実行 50
- マルチスレッド化プログラムの実行 179
- 最適化コード
 - コンパイル 6
 - デバッグ 6
- 削除
 - 指定ブレークポイントをハンドラ ID を使用して 61
 - すべての呼び出しスタックフレームフィルタ 95
 - 阻止リストから例外型を 194
 - 呼び出しスタックから停止した関数 94
 - 呼び出しスタックフレーム 95
- 作成
 - .dbxrc ファイル 16
 - イベントハンドラ 68
- し
- 式
 - 値を監視 100
 - 値を出力 98, 251
 - 表示 100
 - 表示の終了 101
 - 複雑な, Fortran 219
 - 変更を監視 100
- 式の値を監視 100
- シグナル
 - FPE, トラップ 187
 - キャンセル 186
 - 自動ハンドリング 190
 - 取得 187
 - デフォルトリスト変更 187
 - 転送 186
 - トラップされているリスト 187
 - プログラム送信 189
 - 無視 187
 - 無視されているリスト 187
- シグナルリスト捕捉 187
- シグナルリスト無視 187
- システムイベント指定 73
- 実行時検査
 - 8M バイト制限 142
 - アクセス検査 116
 - アプリケーションプログラミングインタフェース 139
 - エラー 144
 - エラー抑制 127
 - デフォルト 130
 - 例 129
 - エラー抑制のタイプ 128
 - 子プロセス 131
 - 修正と継続を使用 137
 - 終了 112
 - 障害追跡のヒント 141
 - 条件 110
 - 使用時 110
 - 制限 111
 - 接続プロセス 135
 - 前回のエラー抑制 128
 - バッチモード
 - dbx から直接 141
 - バッチモードを使用 139
 - メモリーアクセス
 - エラー 118, 144
 - エラーレポート 117
 - 検査 116
 - メモリー使用状況検査 125
 - メモリーリーク
 - エラー 119, 148
 - エラーレポート 122
 - 検査 118, 121
 - メモリーリーク修正 125
 - リークの可能性 120
- 指定 151
- 指定オブジェクトで停止ブレークポイント 59
- 指定型の例外の捕捉 193
- 指定関数中で停止ブレークポイント 54, 58
- 指定条件で停止ブレークポイント 63
- 指定範囲変更時に停止ブレークポイント 62
- 指定メンバーで停止ブレークポイント 57
- 自動読み取り機能
 - .o ファイル 40
 - アーカイブファイル 40
 - 実行可能ファイル 41

- デフォルトの動作 40
 - 修正
 - C++テンプレート定義 174
 - 共有オブジェクト 168
 - プログラム 169, 252
 - 修正と継続 167
 - 共有オブジェクトで使用 247
 - 実行時検査で使用 137
 - 制限 168
 - ソースコードの修正 168
 - 動作方法 168
 - 終了
 - 監視中のすべての変数の表示 101
 - 実行時検査 112
 - 特定の変数または式の表示 101
 - プログラム 9
 - プログラムのみ 9
 - 障害追跡のヒント, 実行時検査 141
 - シングルステップ
 - 機械命令レベル 233
 - 進行イベント指定実行 76
 - シンボル
 - dbx が使用するシンボルの決定 35
 - 出現リストの印刷 34
 - 多重発生から選択 30
 - シンボルの検索順序 33
 - シンボルの多重発生から選択 30
 - シンボル名の修飾 31
- す
- スコープ決定演算子 31
 - スコープ決定検索パス 33
 - スタックトレース 213
 - ストリップされたプログラム 8
 - スレッド
 - リスト, 表示 178
 - 既知のすべての印刷リスト 178
 - 現在, 表示 178
 - 情報を表示 176
 - スレッド ID で切り換え 178
- 通常印刷されない (ゾンビ) 印刷リスト 178
表示コンテキストを切り換え 178
- せ
- セグメンテーションフォルト
 - Fortran, 原因 210
 - 行番号検索 211
 - 生成 211
 - セッション, [dbx](#)
 - 起動 1
 - 終了 8
 - 接続
 - [dbx](#) 実行中の子プロセスへ 181
 - [dbx](#) 実行中のプロセスに 46, 47
 - 実行中のプロセスへ [dbx](#) を 2
 - 接続プロセス, 実行時検査を使用 135
 - 設定
 - [dbxenv](#) コマンドでの [dbx](#) 環境変数 19
 - トレース 59
 - 非メンバー関数の複数のブレークポイント 58
 - ブレークポイント
 - 同じクラスのメンバー関数 57
 - オブジェクト内 59
 - 異なるクラスのメンバー関数 57
 - すべてのインスタンス関数テンプレート 202
 - 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 202
 - ブレークポイントフィルタ 64
 - 前回のエラー抑制 128
 - 宣言, 検索 (表示) 36
- そ
- ソースリスト, 印刷 30
- た
- 単一の実行
 - プログラム 49

断面化

- C と C++ 配列 103
- Fortran 配列 104
- 配列 106

ち

- チェックポイント, デバッグ実行を保存 11

つ

追跡

- `exec` 関数 182
- `fork` 関数 182

て

停止

- テンプレートクラスのすべてのメンバー関数 202
- プログラム 実行
 - 指定した変数の値が変更された場合 62
 - 条件文が真と評価された場合 63
- プログラムの実行
 - アドレスの内容を書き込み 61
- プロセスの実行 8
- プロセスを Ctrl+C で 52

- ディレクトリからディレクトリへ新しいマッピングを確立する 4, 28

データ

- 異なる表示の比較 159
- データグラフを自動的に更新 160
- 表示, 分析 159
- プログラムの異なる時点でのグラフの比較 161

- データグラフウィンドウ 151

- データグラフを自動的に更新 160

- データ表示 151

- データメンバー, 印刷 36

- 手順, 呼び出し 251

- 手順リンクテーブル 246

デバッグ

- `-g` オプションなしでコンパイルされたコード 7

- アセンブリ言語 227

- 機械命令レベル 227, 233

- コアファイル 2

- 子プロセス 181

- 最適化コード 6

- 最適化プログラム 207

- マルチスレッド化プログラム 175

デバッグオプション

- 2 セットを保持 18

- 単一化されたセットを保持 17

デバッグ実行

- 保存 9

- 再実行 13

- 復元 12

デバッグ情報

- すべてのモジュールの, 読み込み 42

- モジュールの, 読み込み 42

- デフォルト `dbx` 設定のアジャスト 15

テンプレート

- インスタンス化 197

- リスト印刷 197, 199

- 関数 197

- クラス 197

- メンバー関数内で停止 202

- 宣言の検索 37

- 定義を表示 197, 200

と

- 等高線グラフタイプで配列表示

- カラーで表示 158

- 動的リンカー 245

- どの変数を `dbx` が評価したか確認 97

- トリップカウンタ 70

トレース

- 実装 86

- 設定 59

- 速度の制御 60

- リスト 60

- トレース速度の制御 60

は

配列

- 2次元, グラフ化 152
 - Fortran 213
 - Fortran 95 割り当て可能な 215
 - 値の範囲, グラフ化の例 152
 - 刻み 102, 106
 - 境界, 超過 210
 - グラフ化 153
 - グラフ化の準備 153
 - グラフ化の方法 153
 - スライス
 - グラフ化の例 152
 - 単次元, グラフ化 152
 - 断面化 102, 106
 - C と C++ の構文 103
 - Fortran 構文 104
 - 断面化の構文, 刻み 102
 - 等高線グラフタイプ
 - カラーで表示 158
 - 行表示 158
 - 配列表示の変更 155
 - 評価 102
 - 表示の回転 155
 - 表示の自動更新 154
 - 表示の変更 155
 - 表面グラフタイプ 157
 - 選択のシェーディング 157
 - テキストチャー選択 157
- ### 配列 式 151
- 配列のグラフ化 153
 - dbx コマンド行 154
 - 配列の断面化の刻み 106
 - 配列表示の回転 155
 - 配列表示の等高線グラフタイプ
 - 行表示 158
 - データ範囲値の表示の変更 158
 - 配列表示の表面グラフタイプ 157
 - 選択のシェーディング 157
 - テキストチャー選択 157
- ### ハンドラ 67
- 関数内で有効にする 86

作成 68

ハンドラ ID, 定義 68

ひ

比較

- 同じデータの異なる表示 159
- データグラフ
 - 同じプログラムの別の実行 162
 - プログラムの異なる時点 161

評価

- 関数のインスタンス化またはクラステンプレート
のメンバー関数 203
- 配列 102
- 無名関数引数 100

表示

- 型 36
 - 関数 29
 - 呼び出しスタックの操作による 31
 - 関数テンプレートインスタンス化のソース
コード 197
 - 基底クラスから継承されたすべてのデータメン
バー 99
 - クラス 36
 - クラスで直接定義されたすべてのデータメン
バー 99
 - 継承メンバー 38
 - シンボル, 出現 34
 - スレッドリスト 178
 - 宣言 36
 - 他のスレッドのコンテキスト 178
 - テンプレート定義 36
 - テンプレートとインスタンス定義 197, 200
 - ファイル 28
 - 変数 36
 - 変数型 37
 - 変数と式 100
 - 無名関数引数 100
 - メンバー 36
 - 例外処理の型 193
- ### 表示データの分析 159

ふ

ファイル

アーカイブ, 自動読み取り機能 40

位置 27

検索 4

名前の修飾 31

表示 28

フィールド型

印刷 37

表示 37

浮動小数点例外 (FPE)

位置を決定 188

原因を決定 188

取得 89

浮動小数点例外 (FPE) 原因を決定 188

浮動小数点例外 (FPE) の位置を決定 188

ブレークポイント

`stop`型 53

`trace` 54

`when`型

設定行 56

`when`型 53

イベント効率 65

イベント指定 70

イベント発生後に有効にする 88

ウォッチポイント 61

概要 53

指定オブジェクトで停止 59

指定関数中で停止 54, 58

指定条件で停止 63

指定範囲変更時に停止 62

指定メンバーで停止 57

条件, 設定 61

設定

C++ コードでの複数のブレーク 57

アドレス 236

同じクラスのメンバー関数 57

オブジェクト内 59

関数テンプレートインスタンス化 197, 202

関数テンプレートのすべてのインスタンス
202

関数内 54

機械レベル 235

行 54

クラステンプレートインスタンス化 197,
202

異なるクラスのメンバー関数 57

条件ブレーク 62

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 202

動的にリンクされたライブラリ 56, 248

フィルタ 64

読み込み済み共有オブジェクト 247

ハンドラ ID を使用してハンドラを削除, ハン
ドラ ID を使用 61

複数, 非メンバー 関数で設定 58

リスト 60

例外 56

プログラム

最適化, デバッグ 207

実行 45, 48

`dbx`, インバクト 249

実行時検査 112

実行継続 49

修正後 170

実行停止

アドレスの内容を書き込み 61

条件文が真と評価された場合 63

実行を継続

指定行 252

指定した行で実行を再開 50

修正 169, 252

終了 9

状態, チェック 89

ストリップされた 8

単一の実行 49

停止 実行

指定した変数の値が変更された場合 62

マルチスレッド化

実行の再開 179

デバッグ 175

プログラムの実行 45, 48

実行時検査 112

引数のない `dbx` 内 45

プログラムの実行継続 49

指定行 252

指定した行 49

- 修正後 170
- プロセス
 - Ctrl+C で停止 52
 - dbx の切り離し 9
 - 子
 - 実行時検査を使用 131
 - 接続 dbx 181
 - 実行, dbx を接続 46, 47
 - 実行の停止 8
 - 接続, 実行時検査を使用 135
- プロセス制御コマンド, 定義 45
- ブロック捕捉 192
- ブロックローカル演算子 32

- へ
- ヘッダファイルの修正 173
- 変更
 - 関数実行中 170
 - 実行関数 170
 - 修正後の変数 171
 - スタックにある関数 171
 - デフォルトシグナルリスト 187
 - 等高線グラフタイプで表示されるデータ範囲
 - 値の変更 158
 - 配列表示 155
 - 呼び出されていない関数 170
- 変数
 - 値を出力 98
 - 値を割り当て 101, 250
 - イベント指定 84, 85
 - 検索の宣言 36
 - 検索の定義 36
 - 修正後の変数 171
 - 条件ブレークポイント 62
 - スコープ外 98
 - 宣言, 検索 36
 - 定義された表示関数とファイル 97
 - どの変数を dbx が評価したか決定 97
 - 名前の修飾 31
 - 表示 36
 - 表示の終了 101

- 変更を監視 100
- 変数型, 表示 37
- 変数に値を割り当て 101, 250

- ほ
- ポインタ
 - 印刷 225
 - 間接参照 100
- ポインタを間接参照 100
- 保持
 - 2 セットのデバッグオプション 18
 - デバッグオプションの単一化されたセット 17
- 保存
 - デバッグ実行をチェックポイントとして 11
 - デバッグ実行をファイルへ 9, 11
- 保存されたデバッグ実行を再実行 13
- 保存されたデバッグ実行を復元 12
- 「ボタン編集」ダイアログ 18
- ポップ
 - 呼び出しスタック 94

- ま
- マルチスレッド化プログラム, デバッグ 175

- め
- メモリー
 - アドレスで内容を調べる 227
 - 状態 116
 - 表示書式 229
 - 表示モード 227
- メモリーアクセス
 - エラー 118, 144
 - エラーレポート 117
 - 検査 116
 - 開始 111, 112
- メモリー使用状況検査
 - 開始 111
- 125

- 開始 112
- メモリーの内容を調べる 227
- メモリーリーク
 - エラー 119, 148
 - 検査 118, 121
 - 開始 111, 112
 - 修正 125
 - レポート 122
- メンバー
 - 検索の宣言 36
 - 検索の定義 36
 - 宣言, 検索 36
 - 表示 36
- メンバー関数
 - 印刷 36
 - トレース 59
 - 複数のブレークポイントの設定 57
- メンバーテンプレート 関数 197

も

- モジュール
 - dbx に読み込まれたデバッグ情報を含む, リスト 43
 - 現在, 名前を印刷 42
 - すべてのプログラム, リスト 43
 - デバッグ情報リスト 42

よ

- 呼び出し
 - 関数 51
 - 関数のインスタンス化またはクラステンプレートのメンバー関数 203
 - 手順 251
 - メンバーテンプレート 関数 197
- 呼び出しスタック 91
 - popping 171, 251
 - 1フレーム 173
 - 位置を検索 91
 - 移動
 - 上へ 92

- 下へ 93
- 指定フレームへ 93
- 削除
 - すべてのフレームフィルタ 95
 - フレーム 95
- 操作 31, 92
- 停止された関数削除 94
- フレームを隠す 95
- ポップ 94
- 呼び出しスタックの操作 31, 92
- 呼び出しスタックフレームを隠す 95
- 読み込み
 - すべてのモジュールのデバッグ情報 42
 - モジュールのデバッグ情報 42
- 読み込み済み librttc.so 135

ら

ライブラリ

- 共有, dbx 用にコンパイル 8
- 動的にリンクされた, ブレークポイントの設定 56

り

リスト

- C++ 関数テンプレートインスタンス化 36
- dbx に読み込まれたデバッグ情報を含むモジュール名 43
- イベントハンドラ 60
- すべてのプログラムのモジュール名 43
- デバッグ情報を含むすべてのプログラムのモジュール 43
- トラップされているシグナル 187
- トレース 60
- ブレークポイント 60
- 無視されているシグナルリスト 187
- モジュールのデバッグ情報 42
- リンカー名 33
- リンクマップ 246

れ

例外

Fortran プログラムでの, 検索 212

型, 表示 193

型が取得される場所のレポート 194

指定型, 取得 193

阻止リストから型を削除 194

例外型が取得される場所のレポート 194

例外処理 192

例 195

レジスタ

Intel 239

SPARC 237

値を出力 236

ろ

ロードオブジェクト 245