

DevPartner® エラー 検出ガイド

リリース 8.2.1



技術に関するお問い合わせは、電子メールでお寄せください。

call.center.japan@compuware.com

このドキュメント、およびドキュメントに記載されている製品には、以下が適用されます。

アクセスは、許可されたユーザーに制限されています。この製品の使用には、ユーザーと Compuware Corporation の間で交わされたライセンス契約の条項が適用されます。

© 2007 Compuware Corporation. All rights reserved.

この未公表著作物は、アメリカ合衆国著作権法により保護されています。

アメリカ合衆国政府の権利

アメリカ合衆国政府による使用、複製、または開示に関しては、Compuware Corporation のライセンス契約に定められた制約、および DFARS 227.7202-1(a) および 227.7202-3(a) (1995)、DFARS 252.227-7013(c)(1)(ii)(OCT 1988)、FAR 12.212(a) (1995)、FAR 52.227-19、または FAR 52.227-14 (ALT III) に規定された制約が、適宜、適用されます。

Compuware Corporation.

この製品には、Compuware Corporation の秘密情報および企業秘密が含まれています。Compuware Corporation の書面による事前の許可なく、使用、開示、複製することはできません。

DevPartner Studio[®]、BoundsChecker、FinalCheck、および ActiveCheck は、Compuware Corporation の商標または登録商標です。

Acrobat[®] Reader copyright © 1987-2007 Adobe Systems Incorporated.

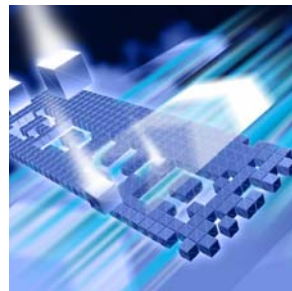
All rights reserved. Adobe、Acrobat、および Acrobat Reader は、Adobe Systems Incorporated の商標です。

その他の会社名、製品名は、関連する各社の商標または登録商標です。

米国特許番号：5,987,249、6,332,213、6,186,677、6,314,558、6,016,466

発行日：2007年10月1日

目次



はじめに

対象読者	vii
このマニュアルの内容	viii
表記方法	ix
マニュアルの記載内容に関する注意事項	ix
サポートのご案内	ix
テクニカル サポート	ix

第 1 章

ワークフローと構成ファイルの設定

DevPartner エラー検出のワークフロー	1
DevPartner エラー検出ワークフローの特長	2
エラー検出構成ファイルの保存	2
コマンドラインからのエラー検出の使用	3
nmvbuild を使用したネイティブ C/C++ のインストール	4
DevPartner エラー検出設定のカスタマイズ	6
全般	7
データ収集	7
API コール レポートイング	7
コール バリデーション	7
COM コール レポートイング	8
COM オブジェクトの追跡	9
デッドロック分析	9
メモリの追跡	9
.NET コール レポートイング	10
.NET 分析	11
リソースの追跡	11
モジュールとファイル	12
フォントと色	12
構成ファイル管理	12

第 2 章

プログラムのチェックと分析

エラー検出タスク	15
リークの検出	15
ポインタ エラーとメモリ エラーの検出	16
メモリ破壊の検出	16
.NET アプリケーションでの従来のコードへの移行の分析	16
Win32 API コールの妥当性検証	18
アプリケーション デッドロックの検出	18
DevPartner エラー検出の拡張機能	18
複雑なアプリケーションの理解	18
リバース エンジニアリング	21
ストレス テスト	24

第 3 章

複雑なアプリケーションの分析

複雑なアプリケーションについて	27
プロセスを待機	29
プログラムの特定部分の分析	30
[モジュールとファイル] の設定の使用	32
監視対象の決定	34
アプリケーションの起動方法	35
サービスの分析	35
要件とガイドライン	36
サービスの分析	36
タイミングの問題と dwWait	36
代替メソッド: ワーカー スレッドからのコントロール ロジックの分離	36
DevPartner エラー検出をログオンまたはログオフするカスタム コード	37
サービス関連の一般的な問題	37
テスト コンテナを使用した ActiveX コントロールの分析	38
テスト コンテナの一般的な問題	39
COM を使用するアプリケーションの分析	40
COM の一般的な問題	41
IIS 5.0 での ISAPI フィルタの分析	42
ISAPI フィルタの一般的な問題	43
IIS 6.0 での ISAPI フィルタの分析	44
IIS 5.0 Isolation Mode	44
IIS 6.0 デフォルト設定	45
IIS 6.0 ISAPI フィルタの一般的な問題	46
よく寄せられる質問 (FAQ)	47

第 4 章

ユーザーが作成したアロケータの使用

概要	49
必要な情報の収集	50
ユーザーが作成したアロケータの名前の検出	50
ユーザーが作成したアロケータによるメモリに関する特別な前提条件	52
UserAllocators.dat のエントリの作成	53
モジュール	53
アロケータ レコード	55
デアロケータ レコード	59
クエリサイズ レコード	62
リアロケータ レコード	64
無視レコード	67
UserAllocator フック要求のコーディング	69
UserAllocator のコード要件	69
アロケータ関数のフック	69
デアロケータ関数のフック	70
リアロケータ関数のフック	71
UserAllocator フックのデバッグ	72
NoDisplay	72
Debug	72
UserAllocators.dat でのエラーの診断方法	73
トークン解析エラー	73
意味的エラー	74
UserAllocators.dat を変更したあと、アプリケーションが不安定になる場合	74

第 5 章

デッドロック分析

背景：シングル スレッド アプリケーションとマルチスレッド アプリケーション	75
スレッド	75
クリティカル セクション	76
デッドロック - 基本定義	77
デッドロックを回避するためのテクニック	78
潜在的なデッドロック	79
食事をする哲学者	79
同期オブジェクトの監視	80
その他の同期オブジェクト	81
追加情報	83
MSDN の参照情報	83
その他の参照情報	83

付録 A

エラー検出のトラブルシューティング

トラブルシューティング	85
-------------------	----

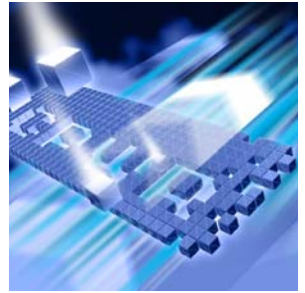
付録 B

重要なエラー検出ファイル

ファイルと用途	95
---------------	----

索引

はじめに



- ◆ 対象読者
- ◆ このマニュアルの内容
- ◆ 表記方法
- ◆ マニュアルの記載内容に関する注意事項
- ◆ サポートのご案内

このマニュアルでは、Compuware® DevPartner エラー検出ソフトウェアの使用方法を理解していただくため、そのコンセプトと手順について説明します。

対象読者

このマニュアルは、DevPartner エラー検出の新規ユーザーのほか、DevPartner エラー検出の前バージョンをご利用で、新しい機能やインターフェイスなどを知りたい方を対象に作成されています。

新規ユーザーの方は、まず、『DevPartner Studio ユーザー ガイド』のエラー検出の章を読んで DevPartner エラー検出のコンセプトの概要を理解してから、このマニュアルで DevPartner エラー検出の最も効果的な使い方を習得してください。

前バージョンからの DevPartner studio ユーザーの場合は、リリース ノートを読んで、前バージョンで使用されていたエラー検出ツールである BoundsChecker と DevPartner エラー検出の相違点を把握してください。

このマニュアルでは、Windows のインターフェイスとソフトウェア開発のコンセプトを理解していることを前提としています。

このマニュアルの内容

このマニュアルは以下の章と付録で構成されています。

- ◆ 第1章「ワークフローと構成ファイルの設定」では、単純なAPIコールバリデーションから複雑なCOMアプリケーションで発生する問題など、さまざまな問題を解決するためにDevPartnerエラー検出を設定する方法について説明します。
- ◆ 第2章「プログラムのチェックと分析」では、DevPartnerエラー検出で実行可能なエラー検出タスクやその他のタスクについて説明します。
- ◆ 第3章「複雑なアプリケーションの分析」では、複雑なアプリケーションを検証する場合に、DevPartnerエラー検出をより効率的に使用するためのヒントを提供します。
- ◆ 第4章「ユーザーが作成したアロケータの使用」では、メモリアロケータを解析できるようにUserAllocators.datファイルをカスタマイズする方法について説明します。
- ◆ 第5章「デッドロック分析」では、デッドロック、潜在的なデッドロック、および同期オブジェクトについて説明します。また、これらのトピックの参照Webアドレスと書籍の一覧も掲載されています。
- ◆ 付録A「エラー検出のトラブルシューティング」では、「問題／対処法」の形式で最も一般的な問題（一部）の解決方法を紹介します。
- ◆ 付録B「重要なエラー検出ファイル」では、DevPartnerエラー検出に関連する重要なファイルのリストを示し、各ファイルの用途について説明します。

このマニュアルの最後には、索引が付いています。

メモ： このマニュアルに記載されている情報は、DevPartner StudioがサポートするすべてのVisual Studioバージョンに適用されます。Visual Studioの特定のバージョンでしか利用できない機能には、注意書きが付いています。

表記方法

このマニュアルの表記方法は以下のとおりです。

- ◆ スクリーン コマンドとメニューの名前は[]で示します。たとえば、以下のよう
に示します。
[ツール]メニューから[ItemBrowser]を選択します。
- ◆ コンピュータのコマンドとファイル名は等幅フォントで示します。たとえば、
以下のよう示します。
『DevPartner エラー検出ガイド』(bc_vc.pdf) で説明します。
- ◆ コンピュータのコマンドとファイル名内の変数 (ユーザーがインストール時に
適切な値を指定するもの) は、イタリックの等幅フォントで示します。たとえば、
以下のよう示します。
[移動先]フィールドに「http://servername/cgi-win/itemview.dll」と入力
します。

マニュアルの記載内容に関する注意事項

このマニュアルは、英語版のマニュアルを基に翻訳され、作成されています。そのた
め、日本では販売されていない製品やサポートされていない機能についての記述が含
まれることがあります。

サポートのご案内

ここでは、テクニカル サポートおよびそれ以外の問題に関するサポートへのお問い合
わせ方法について説明します。

テクニカル サポート

日本コンピュータでは、製品のインストール方法からトラブルシューティングま
で、お客様自身で解決できないような問題をサポートする「テクニカル サポート サ
ービス」を提供しています。なお、このサービスを受けるためには事前にユーザー登録
が必要です。

ユーザー登録の方法

ユーザー登録は、ライセンスファイルの申請と同時に行うことができます。以下の
「ユーザー登録およびライセンスファイル発行の申請」を選択し、必要項目を入力し
ていただくことで登録できます。

オンライン登録の URL :

http://www.compuware.co.jp/products/devpartner_fm/dps_support.html

ユーザー登録に関するお問い合わせ

日本コンピュータ株式会社 DevPartner 担当セールス

Tel: 0120-079009 (03-5473-4527)

* 土日・祝日を除く

テクニカル サポート サービスへのお問い合わせ方法

電子メールでのお問い合わせ :

call.center.japan@compuware.com

Web からのお問い合わせ :

<http://www.compuware.co.jp/products/form/problem.asp>

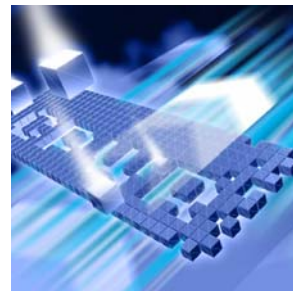
注意 : お電話によるお問い合わせは受け付けておりません。ご了承ください。

お問い合わせの際には、以下の情報をお知らせください。

- ◆ 製品名とそのバージョン
- ◆ 製品のシリアル番号
- ◆ システム構成 (CPU、RAM、OS、コンパイラ、IDE など)
- ◆ 問題の詳細 — 再現性がある場合はその状況

第 1 章

ワークフローと構成ファイルの設定



◆ DevPartner エラー検出のワークフロー

◆ DevPartner エラー検出設定のカスタマイズ

DevPartner エラー検出では、さまざまな種類の問題を識別できます。DevPartner エラー検出のデフォルト設定を使用すると、パフォーマンスにあまり影響を与えない一般的なエラーが検出されます。

この設定を変更して DevPartner エラー検出を微調整すると、特定の種類の問題を探索できます。エラー検出設定を理解すると、DevPartner エラー検出の機能をフルに活用できます。

この章では、単純な API コールバリデーションから複雑な COM アプリケーションで発生した問題まで、さまざまな問題を解消するための DevPartner エラー検出の設定方法について説明します。

メモ： エラー検出により、ターゲットアプリケーションごとにデータファイルが作成されます。エラー検出を開始する前に、ターゲットの実行可能ファイルを含むディレクトリへの書き込みアクセス権があることを確認する必要があります。

DevPartner エラー検出のワークフロー

DevPartner エラー検出のプログラムワークフローは、以前の DevPartner エラー検出バージョンのワークフローをさらに拡張したものとなっています。このメカニズムによって、収集およびレポートされるデータ量を制御できます。

DevPartner エラー検出ワークフローには以下の4つの手順があります。

- 1 対象データを収集するように DevPartner エラー検出を設定します。
 - a 収集するデータのタイプを選択します。
 - b 監視するアプリケーションの部分を定義します。
 - c 適用する[抑制]と[フィルタ]を選択します。
- 2 アプリケーションを実行します。
 - a プログラムを実行しながら、**[検出されたプログラム エラー]**ダイアログボックスに表示されたエラーをレビューします。
 - b 無効なエラーを抑制します。
 - c 必要に応じて、ログを表示し、フィルタを作成します。
 - d メモリとリソースの使用状況を参照します。
- 3 プログラムが終了したら、データを表示します。
 - a ログに表示しないイベントをフィルタします。
 - b アプリケーションを将来実行するときに適用する新しい抑制を作成します。
- 4 必要に応じて、再利用する設定、抑制、およびフィルタを保存します。

DevPartner エラー検出ワークフローの特長

DevPartner エラー検出のワークフローによって、以下のことが可能になります。

- ◆ 収集するデータの量とタイプを選択する。
- ◆ アプリケーションの監視対象部分を選択する。
- ◆ 判明している問題としてレポートされるエラー、条件コードによって処理されるエラー、またはサードパーティコードで生成されたエラーを抑制する。
- ◆ ログ内の不要な情報を非表示にするフィルタを作成する。
- ◆ 設定、抑制、およびフィルタを再利用できるように、異なる構成ファイルを保存する。

DevPartner エラー検出には、ワークフロー プロセスの各ステップにデフォルト設定があります。つまり、デフォルト設定で DevPartner エラー検出を使用することも、設定を変更して DevPartner エラー検出がアプリケーションを分析する方法をカスタマイズすることもできます。

エラー検出構成ファイルの保存

設定 (Visual C++ およびスタンドアロンバージョン) またはオプション (Visual Studio) の固有の組み合わせによるエラー検出構成ファイルを保存しておき、あとで使用することが可能です。

たとえば、メモリ リークとリソース リーク用の構成や、詳細な **lint** タイプ分析のための COM リークとサードパーティ用の構成を作成できます。あとで設定を変更して、

大規模なアプリケーションの特定の部分だけを対象にする構成を定義することもできます。

コマンドラインからのエラー検出の使用

コマンドプロンプトから **BC.exe** (実行可能ファイル) を実行してプログラムをチェックするには、以下のコマンド構文を使用します。角かっこ `[]` は、コマンドが任意選択であることを示します。

```
BC.exe [/?]
```

```
BC.exe sessionlog.DPbcl
```

```
BC [/B sessionlog.DPbcl] [/C configfile.DPbcc] [/M] [/NOLOGO]  
[X[S|D] xmlfile.xml] [/OUT errorfile.txt] [/S] [/W workingdir]  
target.exe [target args]
```

表 1-1. コマンドラインオプション

オプション	説明
<code>/?</code>	使用法情報を表示します。
<code>sessionlog.DPbcl</code>	既存のセッションログを開きます。
<code>/B sessionlog.DPbcl</code>	バッチモードで実行し、セッションログをログファイル <code>sessionlog.DPbcl</code> に保存します。
<code>/C configfile.DPbcc</code>	<code>configfile.DPbcc</code> オプションを使用します。
<code>/M</code>	<code>BC.exe</code> を起動し、実行中は最小化します。
<code>/NOLOGO</code>	<code>BC.exe</code> のロード中にスプラッシュ画面を表示しないようにします。
<code>/X xmlfile.xml</code>	XML 出力を生成し、指定したファイルに保存します。 <ul style="list-style-type: none">実行可能ファイルを指定すると、エラー検出により、その実行可能ファイルでセッションが実行され、その結果から XML 出力が生成されます。セッションログファイル (<code>sessionlog.DPbcl</code>) だけを指定すると、エラー検出により、指定したセッションログが XML に変換され、その出力が保存されます。 メモ：実行可能ファイルを指定する場合には、 <code>/B</code> スイッチを使用して、それに対応するセッションログファイルも指定する必要があります。
<code>/XS xmlfile.xml</code>	エラー検出で <code>/X</code> フラグと <code>S</code> 修飾子を一緒に使用すると、サマリデータのみが XML ファイルに保存されます。つまり、エラー検出セッション (セッションデータ) の実行に関する情報が常にエクスポートされます。
<code>/XD xmlfile.xml</code>	エラー検出で <code>/X</code> フラグと <code>D</code> 修飾子を一緒に使用すると、詳細データのみが XML ファイルに保存されます。つまり、エラー検出セッション (セッションデータ) の実行に関する情報が常にエクスポートされます。

表 1-1. コマンドラインオプション (続き)

オプション	説明
<code>/OUT errorfile.txt</code>	エラーメッセージがある場合、そのメッセージを <i>errorfile.txt</i> という名前のテキストファイルに出力します。このテキストファイルには、エラー検出によって発見されたエラーとリークのリストではなく、エラー検出を実行しようとしたときに生成されたエラーメッセージだけが入ります。
<code>/S</code>	サイレントモードで実行します。エラーが発生しても [検出されたプログラム エラー] ダイアログ ボックスは開きません。
<code>/W workingdirectory</code>	ターゲットの作業ディレクトリを設定します。
<code>target.exe [target args]</code>	起動する実行可能ファイルとその引数です。

メモ： 使用する実行可能プログラムが現在のパスにない場合は、ディレクトリのフルパスを指定する必要があります (実行可能ファイルを探すときにシステムが検索するディレクトリを一覧にする環境変数)。

1つのプログラムに複数のコマンド オプションを指定できます。以下に例を示します。

```
BC.exe /B test.dpbcl /S /M c:¥testdir¥test.exe
```

nmvcbuild を使用したネイティブ C/C++ のインストゥルメント

コマンドラインからのプロジェクトの作成を計画し、そのプロジェクトをエラー検出用にインストゥルメントする場合には、Microsoft vcbuild.exe コンパイラではなく、nmvcbuild.exe を使用する必要があります。vcbuild コンパイラでは、デフォルトのコンパイラとリンカを置換する方法が提供されないため、DevPartner ネイティブ C/C++ インストゥルメンテーションを実行する方法がありません。nmvcbuild.exe は DevPartner 用に特別に設計されたコマンドライン ユーティリティであり、これを使用すると、vcbuild に対してネイティブ C/C++ インストゥルメンテーションを実行することができます。このユーティリティは、開始されている `cl.exe` と `link.exe` を監視する vcbuild のラッパーとして機能し、それらの実行可能ファイルをそれぞれ **nmcl.exe** と `nmclink.exe` に置き換えます。

メモ： nmvcbuild ユーティリティは Visual Studio .NET 2003 と Visual Studio 2005 に有効です。それ以前のバージョンの Visual Studio を使用している場合には、オンラインヘルプのトピック「コマンドラインから FinalCheck を実行する」で説明されているように、メイクファイルを使用する必要があります。

nmvcbuild ユーティリティでは、vcbuild および nmcl と同じコマンドラインパラメータを使用できます。コマンドラインに `vcbuild ?` と `nmcl ?` を入力すると、vcbuild と nmcl のコマンドラインパラメータを表示できます。また、環境変数 nmcl に必須パラメータを埋め込むこともできます。埋め込み後は、nmvcbuild の呼び出

し時に vcbuild パラメータを渡すだけです。たとえば、以下のエントリによって、環境変数に nmcl パラメータが設定されます。

```
set nmcl=/NMignore:StdAfx.cpp
```

詳細については、オンライン ヘルプの「NMCL オプション」を参照してください。

メモ： vcbuild ユーティリティは Visual Studio 2005 に含まれていますが、Visual Studio .NET 2003 には含まれていません。Microsoft のサイトから Visual Studio .NET 2003 に対応するバージョンの vcbuild ユーティリティをダウンロードする必要があります。

前提条件

nmvcbuild.exe を実行するには、以下の前提条件を満たしておく必要があります。

- ◆ DevPartner Studio がシステムにインストールされていること。
- ◆ システム環境が Visual Studio のツールを実行できるように設定されていること。
- ◆ パス設定に vcbuild.exe と nmvcbuild.exe があること。デフォルトでは、nmvcbuild.exe は以下の場所にインストールされます。

```
¥program files¥common files¥compuware¥nmshared
```

例

エラー検出インストゥルメンテーションを含むサンプル プロジェクトのデバッグ構成を構築するには、以下のように入力します。

```
nmvcbuild /nmbcon sample.vcproj debug
```

DevPartner エラー検出設定のカスタマイズ

DevPartner エラー検出の設定では、以下のカスタマイズが可能です。

- ◆ 収集する情報のタイプを制限する（例：メモリ リーク、リソース リーク）。
- ◆ 分析の主なカテゴリごとに収集する情報のタイプを詳細に区別する（例：グラフのコールによって生成されるリソース リークだけを対象にする）。
- ◆ イベントまたはエラーと共に記録するコール スタック、パラメータ データ、戻り値などの追加情報を決める。
- ◆ DevPartner エラー検出ユーザー インターフェイスの外観と使いやすさを制御する。これには、フォントの変更、色の変更、強調、または**[検出されたプログラムエラー]**ダイアログ ボックスの表示/非表示などがあります。
- ◆ 以前に作成した DevPartner エラー検出の設定を保存して復元する。

DevPartner エラー検出設定をカスタマイズすることで、データの収集量やアプリケーションのどの部分を監視するかを制御できます。

DevPartner エラー検出の設定は以下のグループに分けられます。

- ◆ 全般
- ◆ データ収集
- ◆ API コール レポート
- ◆ コール バリデーション
- ◆ COM コール レポート
- ◆ COM オブジェクトの追跡
- ◆ デッドロック分析
- ◆ メモリの追跡
- ◆ .NET 分析
- ◆ .NET コール レポート
- ◆ リソースの追跡
- ◆ モジュールとファイル
- ◆ フォントと色
- ◆ 構成ファイル管理

全般

[全般]のチェックボックスを使用して、以下を制御できます。

- ◆ イベント ログ

メモ： イベント ログをオフにすると、「サイレント モード」でエラー検出が実行されます。イベント ログを再び有効にするまで、エラー検出では何もレポートされません。

- ◆ [検出されたプログラム エラー] ダイアログ ボックスをエラーが発生するたびに表示するかどうか
- ◆ エラー検出または起動中のその他のセッションを終了するときに、プログラムの結果を保存するように指示するプロンプトを表示するかどうか
- ◆ ターゲット アプリケーション終了時に[メモリ リソース ビューア]ダイアログ ボックスを表示するかどうか
- ◆ ソース ファイルとシンボル ファイルを検索するディレクトリ
- ◆ 作業ディレクトリ (DevPartner エラー検出をスタンドアロン モードで使用するときだけに使用可能)
- ◆ コマンドラインの引数を指定する (DevPartner エラー検出をスタンドアロン モードで使用するときだけに使用可能)

データ収集

[データ収集]の設定で、以下を制御できます。

- ◆ さまざまなコール スタックの深さ
- ◆ 構造体、クラス、ポインタなどの非スカラー パラメータに保存するデータの量と戻り値

メモリが制限されたコンピュータや複雑で大規模なアプリケーションを分析する場合は、[メモリ割り当ての最大コール スタック数]のサイズを制限して必要なメモリ容量を下げるすることができます。

API コール レポートリング

[API コール レポートリングを有効にする]をオンにした場合にログに記録される Windows API コールのタイプを制御するには、[API コール レポートリング]の設定を使用します。Windows メッセージのログも制御できます。

ログ ファイルのサイズを小さくするには、特定の Windows モジュールの API コールを有効にします(例: グラフのコールをログに記録する場合は、GDI32 を選択します)。

コールバリデーション

[コールバリデーション]の設定を使用して、DevPartner エラー検出で Windows API パラメータおよび戻り値を検証するかどうかを制御します。デフォルトでは、パラメータは検証されません。

メモリ使用状況を追跡する場合は、**[メモリ ブロック チェックを有効にする]**をオンにします。このオプションをオンにすると、メモリ追跡システムから収集された情報を使用して、より詳細なパラメータ分析が実行されます。この機能を有効にすると、より多くのエラーが検出されますが、パフォーマンスは低下します。

DevPartner エラー検出には、Windows API で実行されるバリデーションのタイプを制限する設定があります。これらの設定によって、疑似エラーを生成するエラー カテゴリの選択を解除できます。たとえば、フラグ チェック、範囲チェック、列挙引数チェックなどがあります。これらのオプションは、ハンドルやポインタを詳細に分析する場合に選択できますが、他のタイプのバリデーションには必要ありません。

DevPartner エラー検出では、チェック対象の Windows API を選択できます。デフォルトでは、すべての Windows API がチェックされます。一部の API コールだけをチェックの対象にするには、それらのモジュールだけを選択します。これによって、検出されるエラーの数が減少し、パフォーマンスが向上します。

COM コール レポートリング

[COM コール レポートリング]の設定を使って、**[選択したモジュールに実装されたオブジェクト上での COM メソッド コールのレポートを有効にする]**が選択されている場合にログに記録する COM インターフェイスを制御します。

[選択したモジュールに実装されたオブジェクト上での COM メソッド コールのレポートを有効にする]が選択されている場合、デフォルトで、すべての既知の COM インターフェイスについてレポートされます。パフォーマンスを向上させるには、チェックが必要な COM インターフェイスだけを選択します。**[COM コール レポート]**に表示されるツリー ビューを使用します。チェックする COM インターフェイスの数を減らすと、ログ ファイルのサイズも小さくなり、パフォーマンスが向上します。

[リストされていないモジュールに実装された COM メソッド コールをレポートする]を選択することもできます。

COMオブジェクトの追跡

DevPartner エラー検出によって、アプリケーション内でCOM使用状況が監視され、インターフェイスがリークしているコードについてレポートされます。インターフェイスのリークが検出されると、アプリケーション内の**AddRef** と **Release** のペアを示すCOM使用回数グラフが表示されます。このグラフを使用すると、アプリケーションの知識を基に**AddRef** または **Release** コールが不足しているかどうかをすばやく確認できます。

デフォルトでは、COMオブジェクト追跡は無効になっています。この機能をアクティブにするには、**[COMオブジェクトの追跡を有効にする]**を選択します。COMオブジェクト追跡がアクティブな場合は、**[すべてのCOMクラス]**を選択するか、または表示されたリストからクラスを個別に選択できます。

デッドロック分析

[デッドロック分析]を使用すると、マルチスレッドアプリケーションのデッドロックを監視できます。これには以下のような分析が含まれます。

- ◆ アプリケーション内で発生するデッドロックを監視し、レポートする
- ◆ アプリケーション内での同期オブジェクトの使用パターンを監視し、潜在的なデッドロックを検出する
- ◆ アプリケーションの同期オブジェクト エラーを監視する

メモリの追跡

[メモリ追跡]の設定を使用して、このアプリケーションで実行されるメモリ リーク検出のタイプを制御します。**[メモリ追跡]**はデフォルトで有効になっています。メモリ リーク検出を実行しない場合は、**[メモリの追跡を有効にする]**をオフにします。

[メモリ追跡]は、ほとんどのアプリケーションで有効な結果が生成されるように設定されています。**[FinalCheckを有効にする]**、**[保護バイト]**、**[確保時にフィルする]**、**[解放時に無効データでフィルする]**の各設定には注意が必要です。

FinalCheckを有効にする

アプリケーションが**FinalCheck**でインストゥルメントされていないならば、**[FinalCheckを有効にする]**をオンにしても効果はありません。エラー検出用にインストゥルメントするオプションを選択した場合、デフォルトでは、**FinalCheck**はオンです。**FinalCheck**を実行せずにインストゥルメントを有効にするには、**[エラー検出設定]**の**[メモリの追跡]**ペインで**FinalCheck**を無効にします。

[FinalCheckを有効にする]は常にオンにしておき、すでにインストゥルメントされているアプリケーションに、**ActiveCheck**分析を大まかに実行するときだけにだけオフにすることを勧めます。

保護バイト

保護バイトは、ActiveCheck 分析でメモリ オーバーランを検出する際に使用します。ヒープ破壊が発生し、問題が検出されない場合は、**【カウント】**設定の値を大きくする必要があります。これらの設定を使用して検出が困難なヒープ エラーを探し出す方法のヒントについては、オンライン ヘルプを参照してください。

「確保時にフィルする」と「解放時に無効データでフィルする」

[確保時にフィルする]は、メモリ確保時にメモリを既知の状態に設定します。[解放時に無効データでフィルする]は、メモリの解放時にメモリを既知の状態に設定します。

使用するバイト パターンを選択する際は、パターンがプログラムの実行中に誤って使用された場合にアプリケーションでエラーが生成されるようなパターンを選択してください。これらの設定の詳細については、オンライン ヘルプを参照してください。

UserAllocators.dat

独自のメモリ割り当てロジックを作成するか、またはグローバルな `operator new` を上書きする場合は、第4章「ユーザーが作成したアロケータの使用」または以下のファイル内のマニュアル（コメント形式）を参照してください。

```
C:¥Program Files¥Compuware¥DevPartner Studio¥BoundsChecker
¥Data¥UserAllocators.dat
```

.NET コール レポートニング

[.NET コール レポートニング]設定を使用して、[.NET メソッドコール レポートニングを有効にする]をオンにしている場合にログに記録される .NET アセンブリを制御します。

.NET と COM コール レポートを組み合わせることによって、COM 相互運用性の両面を監視できます。

.NET ユーザー アセンブリと .NET システム アセンブリは、同じツリー ビュー コントロールの異なる分岐に表示されます。

.NET コール レポートニングにより大量のデータが生成され、システム速度が低下することがあります。フレームワークをデバッグして調べる必要がある場合に限り、.NET コール レポートニングを有効にします。有効にした場合でも、チェックが必要なアセンブリのみを選択してください。[すべてのタイプ]ツリー ビューで選択するアセンブリ数を制限すると、ログ ファイルのサイズが小さくなり、パフォーマンスが向上します。

パフォーマンスに
関するヒント：

.NET 分析

DevPartner エラー検出では、ネイティブ アプリケーションとマネージ アプリケーションを一緒に使用できます。このような環境の場合は、[.NET ランタイム分析を有効にする]を選択できます。DevPartner エラー検出では、以下のタイプの .NET 分析がサポートされています。

- ◆ ネイティブ コードからマネージ コードに渡される、ハンドルされていない例外の監視
- ◆ .NET ファイナライザの分析
- ◆ マネージ コードとネイティブ コード間の相互運用性
- ◆ ガベージ コレクション イベントの監視

.NET 相互運用性

.NET 相互運用性機能によって、アプリケーションがマネージ コードからネイティブ コードに移行した回数が監視されます。この情報を使用して、マネージ コードで書き直すことによって改善される使用パターンおよびターゲット ネイティブ コードを分析します。最良の結果を得るには、**相互運用性レポートのしきい値**パラメータと共にこの機能を使用して、許容範囲内で独自の下限値を指定します。

リソースの追跡

[**リソースの追跡**]の設定を使用して、アプリケーションで実行されるリソース リーク検出のタイプを制御します。[**リソースの追跡**]はデフォルトで有効になっています。リソース リーク検出を実行しない場合は、[リソースの追跡を有効にする]チェックボックスをオフにします。

リソース追跡を有効にした場合は、すべてのリソース リークを検索したり、Windows API の特定のライブラリに関連付けられている特定のリソースに検索を限定したりできます。

リソースはライブラリごとにグループ化されており、ライブラリはリソースの解放に使用される API コールごとにグループ化されています。たとえば、最近、レジストリを操作するためのコードを大量に追加した場合は、**ADVAPI32**を除くすべてのライブラリの選択を解除し、**RegCloseKey**だけを選択できます。

モジュールとファイル

[モジュールとファイル]の設定を使用すると、以下のことが可能です。

- ◆ 監視または無視される、アプリケーション内の実行可能ファイルおよびライブラリを識別する。
- ◆ シンボルが有効な場合に、ソース ファイル レベルまで監視または無視する実行可能ファイルとライブラリのリストを更新する。
- ◆ DevPartnerエラー検出アナライザによって無視される、**[システム ディレクトリ]**のリストを識別する。

[モジュールとファイル]の設定を使用して、DevPartner エラー検出で監視するアプリケーションの部分を制御します。たとえば、大規模なアプリケーションやISAPI フィルタなどのアプリケーションを作成する場合は、**[モジュールとファイル]**の設定を使用します。

メモ： **[モジュールとファイル]**の設定ですべてのモジュールを無効にしても、一部のエラー タイプがレポートされます。エラー検出では常に、あらゆるモジュール内のメモリ オーバーランと、MFCxxxx.dll ライブラリから発生するその他のタイプのイベントがレポートされます。

詳細については、「**[モジュールとファイル]の設定の使用**」(32 ページ)を参照してください。

フォントと色

[フォントと色]設定を使用すると、DevPartner エラー検出ユーザー インターフェイスの各アイテムのフォント、色、および強調表示を変更できます。

構成ファイル管理

[構成ファイル管理]を使用して、各プロジェクトに複数の構成ファイルを作成できます。[図 1-1](#) (13 ページ) は、構成ファイル管理で使用できるオプションを示しています。ソフトウェア開発工程でこれらの構成ファイルを使用すると、さまざまなタイプの分析を実行できます。たとえば以下のように、構成ファイルを作成できます。

- ◆ **[コールバリデーション]**と**[モジュールとファイル]**を使用して、独自のコンポーネントだけを選択できます。これらの設定は、アプリケーションに新しいコードを追加するときに使用します。
- ◆ 新しいコンポーネントが完成するか、または既存のコンポーネントに複雑な変更を加える場合は、**[メモリ追跡]**と**[リソースの追跡]**の設定を使用します。

- ◆ 週末にかけてバッチモードで使用する構成ファイルを作成して、重要なイベントの結果を分析できます。FinalCheckでビルドをインストールメントすると、レポートの分析時に最も詳細な情報を取得できます。
- ◆ 選択したさまざまなモジュールのセットで構成ファイルを作成できますが、すべての分析機能が無効になります。この構成ファイルをロードすると、対話セッション中に必要なオプションを選択できます。これは、複雑なモジュールとファイル設定を管理する必要がある場合に非常に役立ちます。

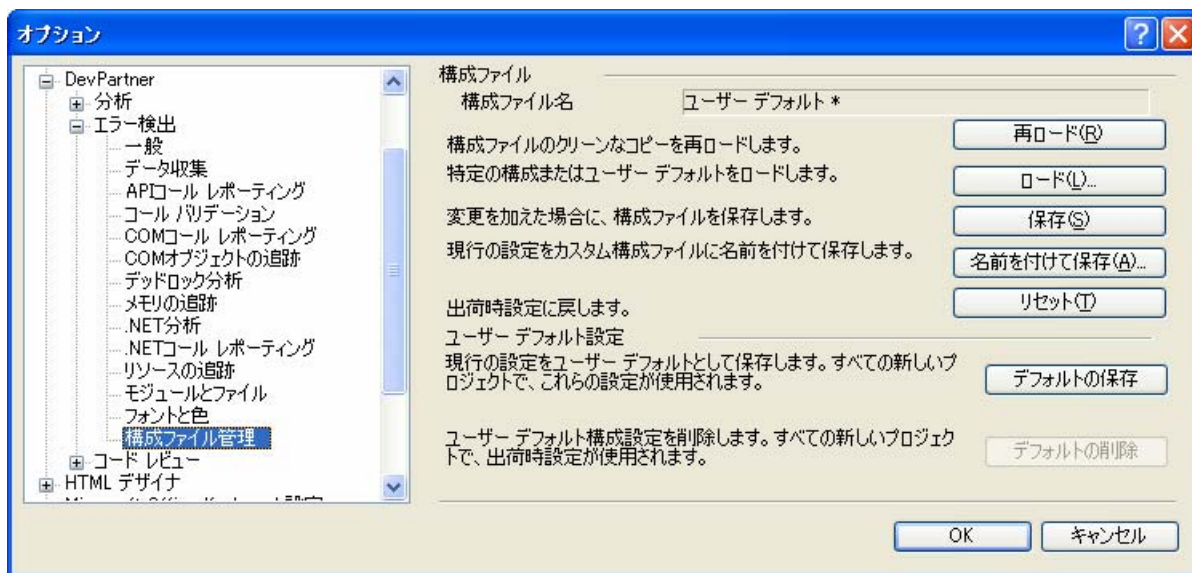


図 1-1. 構成ファイル管理の設定

構成ファイルの機能

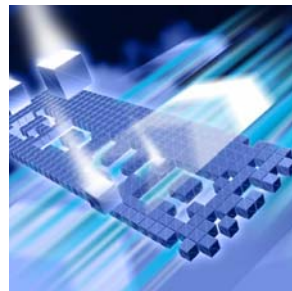
構成ファイル管理ページでは、以下の機能を使用できます。

- ◆ **【構成ファイル名】**: 構成ファイルのフルパスと名前
- ◆ **【再ロード】**: 現在の構成ファイルを再ロードし、変更内容を破棄します。これにより、現在の構成ファイルの前回保存されたバージョンに戻ります。
- ◆ **【ロード】**: **【ロード元】**ダイアログボックスが開きます。**【内部ユーザー デフォルト】**を選択して、ユーザー デフォルト設定をロードします。**【構成ファイル】**を選択した場合、**【構成ファイルのロード】**ダイアログが開きます。このダイアログを使用して、ロードする別の構成ファイルを選択します。
- ◆ **【保存】**: 現在ロードされている構成ファイルにアクティブなすべての変更内容を保存します。
- ◆ **【名前を付けて保存】**: **【構成ファイルの保存】**ダイアログボックスが開きます。このダイアログボックスを使用して、現在の構成設定を別のファイル名で保存します。

- ◆ **【リセット】**: すべてのプログラム プロパティ設定を工場出荷時のデフォルト設定にリセットします。
- ◆ **【デフォルトの保存】**: 現在の設定をユーザー デフォルトとして保存します。すべての新規プロジェクトでデフォルト設定が使用されるようになります。
- ◆ **【デフォルトの削除】**: ユーザー デフォルト構成設定を削除し、工場出荷時の設定に戻します。すべての新規プロジェクトで工場出荷時の設定が使用されるようになります。

第2章

プログラムのチェックと分析



◆ エラー検出タスク

◆ DevPartner エラー検出の拡張機能

この章では、DevPartner エラー検出で実行できるエラー検出タスクについて説明します。また、DevPartner エラー検出で実行できるその他のタスクについても説明します。

エラー検出タスク

DevPartner エラー検出には通常、以下のタスクがあります。

- ◆ メモリ リーク、リソース リーク、およびインターフェイス リークの検出
- ◆ ポインタ エラーとメモリ エラーの検索
- ◆ メモリ破壊の検索
- ◆ .NET アプリケーションでの従来のコードの使用状況の分析
- ◆ Win32 API コールの検証
- ◆ アプリケーション デッドロックの検出

リークの検出

DevPartner エラー検出には、メモリ リーク、リソース リーク、およびインターフェイス リークを検出するための優れた機能があります。デフォルトでは、メモリ リークとリソース リークが検出されますが、インターフェイス リークは検出されません。インターフェイス リークを検出するには、**[COM オブジェクトの追跡]** の設定の **[COM オブジェクトの追跡を有効にする]** をオンにします。

DevPartner エラー検出でメモリ リークを検出するには、ActiveCheck を使用する方法と FinalCheck を使用する方法があります。ActiveCheck を使用すると、Windows アプリケーションのメモリ リークが検出されます。リークはアプリケーションの終了時にレポートされます。FinalCheck を使用すると、アプリケーションでメモリ リークが発生した場合、全体のメモリ リークが実行時にレポートされます。たとえば、ローカル変数が範囲外になったときや、メモリのブロックへの最後のポインタが再び割り当てられたときです。また、ダングリング ポインタ使用や、その他の検出困難なエラーが発見されたときも同様です。

ポインタ エラーとメモリ エラーの検出

DevPartner エラー検出では、ActiveCheck と FinalCheck の両方のテクノロジーを使用してポインタ エラーとメモリ エラーを検索できます。ActiveCheck モードでは、Windows コールに渡されるポインタのエラーが監視されます。DevPartner エラー検出で行われるチェックの量を設定するには、**[コール バリデーション]**と**[メモリ 追跡]**の設定を変更します。

FinalCheck を使用してプログラムを再コンパイルする場合は、プログラム内のすべてのポインタ参照が正しく使用されているかどうかチェックされます。FinalCheck ではプログラムを詳細に分析し、未初期化変数、ダングリング ポインタ、非関連ポインタ比較、配列のインデックス エラーなどの検出困難な問題を検出します。

メモリ破壊の検出

DevPartner エラー検出によって、以下のような問題によって発生するメモリ破壊問題を検出できます。

- ◆ 割り当てられたバッファのオーバーラン
- ◆ メモリの解放後も継続して行われるメモリへのアクセス
- ◆ 複数回のリソース解放（例：二重削除）

DevPartner エラー検出では、ActiveCheck モードでこれらの多くのエラーを検出できますが、FinalCheck を使用するとより詳細に分析できます。

メモリ オーバーラン エラーが発生し、ActiveCheck だけを使用するように制限されている場合は、**[メモリの追跡]**の設定の**[実行時のヒープ ブロック チェック]**について、オンライン ヘルプを参照してください。

.NET アプリケーションでの従来のコードへの移行の分析

DevPartner エラー検出には、ネイティブ アプリケーションからマネージアプリケーション プログラミングへの移行を支援する、以下のような分析があります。

- ◆ Windows アプリケーションのネイティブ部分の包括的な分析
- ◆ 混合コードを使用するアプリケーションのネイティブ セクションとマネージ セクション間の移行層の分析
- ◆ マネージ アプリケーションでのファイナライザの分析

これらの分析によって以下の項目を監視できます。

- ◆ ネイティブ アプリケーションからスローされマネージ コードに渡される、
ハンドルされていない例外
- ◆ パフォーマンス上の問題につながる可能性のあるガベージ コレクタ
- ◆ マネージ コードとネイティブ コード間のCOM 相互運用性
- ◆ マネージ コードから Windows のネイティブ ライブラリに対して行われる
P/Invoke
- ◆ マネージとネイティブの境界を越えたコールの回数

この情報は、アプリケーションの移行プロセスの計画と監視に使用できます。

ネイティブ コードから混合コードまたはマネージ コードへの移行

移行プロセスの手順は以下のとおりです。

- 1 ネイティブ アプリケーションの COM 使用状況を分析して、どのオブジェクトが
使用されているかを調べます。
- 2 P/Invoke と COM を使用してマネージ コード内のアプリケーションのセクションを
修正し、アプリケーションのネイティブ 部分を呼び出します。
- 3 **[.NET 分析]** で **[.NET 分析を有効にする]** と **[P/Invoke 相互運用性の監視]** を選択
し、新たに作成したコードと既存のネイティブ コード間の移行を分析します。
- 4 必要に応じて変更します。
- 5 **[.NET 分析]** で **[COM 相互運用性の監視]** と **[P/Invoke 相互運用性の監視]** を選択
し、マネージ コードとネイティブ コード間で行われるコール回数を監視します。
このパフォーマンス データは、以下の変更が必要かどうかを判断するときに役立
ちます。
 - a どの追加 COM オブジェクトをマネージ コードに移植するかを決めます。
 - b 新しいメソッドを追加してマネージ コードとネイティブ コードの間のコー
ル回数を減らす必要があるかどうかを判断します。たとえば、1 回に 1 項目
ではなく、1 回に 10 ~ 20 項目のデータ レコードを要求するメソッドを追加
する場合があります。
 - c Windows API などのネイティブ API へのコールが効率的に行われているか
どうかを調べます。

また、ネイティブとマネージ間でスローされる、ハンドルされていない例外もチェック
できます。このチェックを実行するには、**[.NET 分析]** で **[例外の監視]** を選択します。
ネイティブ コードで作成されたアプリケーションでは、コールやメソッドの失敗を呼
び出し元に通知するときに例外が使用されます。アプリケーションのセクションをマ
ネージ コードで作成し直すときは、それらのセクションをマネージ コードへ移行す
る前に、例外の使用を監視して例外をキャッチします。

Win32 API コールの妥当性検証

DevPartner エラー検出では多数の Windows コールを識別します。この機能によって、DevPartner エラー検出でポインタ、フラグ、列挙、ハンドル、およびリターンコードの妥当性を検証できます。アプリケーションで Windows コールが正しく使用されているかどうかを確認するには、**[コールバリデーションを有効にする]** をオンにします。

構成可能な **コールバリデーション** 機能は以下のとおりです。

- ◆ 監視する Windows コールのタイプを選択する
- ◆ フラグ、範囲、列挙チェックなどの、さまざまなタイプの妥当性検証を選択して無効にする

これらの機能を使用すると、ハンドルやポインタなどの重要なパラメータの妥当性検証をし、当面の作業に関係のないエラーのレポート数を少なくするように DevPartner エラー検出を設定できます。

アプリケーション デッドロックの検出

DevPartner エラー検出は、アプリケーション内のデッドロックを引き起こすコードを特定できます。**[デッドロック分析を有効にする]** を選択して、デッドロックを見つけます。デッドロック分析を微調整するコントロールもあります。

DevPartner エラー検出の拡張機能

DevPartner エラー検出にはエラー検出の他に、以下の機能もあります。

- ◆ 複雑なアプリケーションを理解するための支援
- ◆ リバース エンジニアリング ツール
- ◆ アプリケーションのストレス テスト ツール

複雑なアプリケーションの理解

DevPartner エラー検出には、大規模で複雑なプログラムの理解力を高めるのに役立つツールがあります。以下に状況の例を挙げます。

- ◆ 既存のチームに加わった新しい開発者は、さまざまな DLL がどのように相互作用しているかを理解する必要があります。
- ◆ クラッシュやメモリ リークなどの問題を解決するためにプロジェクトにコンサルタントを参加させ、エンジニアリングのハードなスケジュールを検討してどこにリソースを集中させればいいのかを理解する必要があります。

- ◆ 開発者は、サードパーティ製ライブラリの使用を開始したあと、そのライブラリがなぜ Windows リソースをリークしているかを理解しなければならないことがあります。多くの場合、問題はライブラリではなくライブラリの使用方法にあります。

DevPartner エラー検出の以下の機能を使用すると、これらのシナリオに対処できます。

COM オブジェクトの追跡

多くのアプリケーションでは、社内の開発者、サードパーティ ベンダ、または Microsoft によって提供された COM オブジェクトを使用します。これらの COM オブジェクトを正しく使用しないと、インターフェイス リークが発生します。インターフェイス リークが発生すると、結果としてメモリ リークとリソース リークが発生します。つまり、ヒープから割り当てられたオブジェクトが適切に解放されず、したがって、それらのオブジェクトによって割り当てられたメモリも適切に解放されなくなります。

[COM オブジェクトの追跡]を使用すると、リークした COM オブジェクトを表示できます。アプリケーション内のどの場所に、**AddRef** に対応する **Release** コールがないのかを調べるのに役立ちます。

デッドロック アナライザ

デュアル プロセッサが一般的に使用されるようになる前に記述された従来のアプリケーションの多くは、新型の高性能なコンピュータ システムで実行すると予期できない動作を引き起こすことがあります。たとえば、同期オブジェクトの使用方法を誤るとデッドロックが発生します。

DevPartner エラー検出によるデッドロック分析では、デッドロックを引き起こす可能性があるコードを特定できます。この分析では、「潜在的」なデッドロックを特定することもできます。潜在的なデッドロックとは、アプリケーションの実行中に、ある好ましくない状態に陥った場合に発生するデッドロックです。DevPartner エラー検出では、実稼働環境で実際にデッドロックが発生する前に潜在的なデッドロックを特定できます。

モジュールとファイル

複雑なアプリケーションは複数の組織間で開発されるため、外部のベンダから購入したライブラリが含まれている場合があります。デフォルトでは、システム以外の DLL のエラーはレポートされません。エラー レポートとコール レポートをアプリケーションの特定のセクションに限定するには、**[モジュールとファイル]**の設定を使用します。これにより、複雑な問題の解決に使用できる、重要なエラー レポートが作成されます。

メモ： **[モジュールとファイル]**の設定ですべてのモジュールを無効にしても、一部のエラー タイプがレポートされます。エラー検出では常に、あらゆるモジュール内のメモリ オーバーランと、MFCxxxx.dll ライブラリから発生するその他のタイプのイベントがレポートされます。

[モジュール]タブ

DevPartnerエラー検出の【モジュール】タブ（図 2-1（20 ページ）を参照）と関連する詳細ペインには、プログラムのビューがあります。このビューには、プログラム実行時にどの DLL がロードされているかが表示されます。このレポートを十分に検討すれば、以下の質問に答えることができ、トレードオフが必要な場合に十分な情報に基づいて判断できます。

- ◆ このモジュールをインストールするか、また、どのようにインストールするか。
- ◆ 特定の DLL が本当に必要か。
- ◆ DLL のメソッドを 1 つだけ呼び出すことが、そのプロセスにロードされるその他の「n」個の DLL を犠牲にするだけの価値があるか。
- ◆ なぜ対象外のロードアドレスに DLL がロードされているか。
- ◆ なぜ同じ DLL の複数のバージョンがメモリにロードされているか。

モジュール名	優先ロード アドレス	実際のロード アドレス	ファイルのバージョン	フルパス
BugBench.exe	00400000	00400000	7.1.0.0	c:\Documents and Settings\...
kernel32.dll	7C800000	7C800000	5.1.2600.2180	C:\WINDOWS\system32\kernel32.dll
ntdll.dll	7C940000	7C940000	5.1.2600.2180	C:\WINDOWS\system32\ntdll.dll
VERSION.dll	77BB0000	77BB0000	5.1.2600.2180	C:\WINDOWS\system32\VERSION.dll
msvcrt.dll	77BC0000	77BC0000	7.0.2600.2180	C:\WINDOWS\system32\msvcrt.dll
MSVCR80D.dll	10200000	10200000	8.00.50727.762	C:\WINDOWS\system32\MSVCR80D.dll
USER32.dll	77CF0000	77CF0000	5.1.2600.2180	C:\WINDOWS\system32\USER32.dll
IMM32.DLL	762E0000	762E0000	5.1.2600.2180	C:\WINDOWS\system32\IMM32.DLL
USP10.dll	73F80000	73F80000	1.0420.2600.2180	C:\WINDOWS\system32\USP10.dll
LPK.DLL	60740000	60740000	5.1.2600.2180	C:\WINDOWS\system32\LPK.DLL
GDI32.dll	77ED0000	77ED0000	5.1.2600.2180	C:\WINDOWS\system32\GDI32.dll
RPCRT4.dll	77E30000	77E30000	5.1.2600.2180	C:\WINDOWS\system32\RPCRT4.dll
ADVAPI32.dll	77D80000	77D80000	5.1.2600.2180	C:\WINDOWS\system32\ADVAPI32.dll
SHLWAPI.dll	77F20000	77F20000	6.00.2900.2180	C:\WINDOWS\system32\SHLWAPI.dll

右側の詳細ペイン (BugBench.exe) の内容:

- 一般情報
 - 場所: c:\Documents and Settings\...
 - サイズ: 256 KB (262,144 バイト)
 - リンク日: 2007年11月19日 8:22:22
 - 作成日: 2007年11月19日 8:21:33
 - 変更日: 2007年11月19日 8:22:23
 - アクセス日: 2007年11月22日 6:35:47
 - インストールメンテション: エラー検出
 - PDBファイルの場所: c:\Documents and Settings\...
 - 静的フック: PDBを使用
- バージョン情報
 - ファイルのバージョン: 7.1.0.0
 - 説明: BugBench: Compuware Bour...
 - 著作権: Copyright c 1995, 2005 Co...
- その他のバージョン情報:
 - 会社名: Compuware Corporati...
 - 内部名: BugBench
 - 言語: Japanese
 - オリジナルのファイル名: BugBenc...
 - 製品名: BugBench Sample App
 - 製品バージョン: 7.1.0.0
- ロード アドレス情報
 - 実際のロード アドレス: 0x00400000
 - 優先ロード アドレス: 0x00400000

図 2-1. [モジュール]タブと詳細ペイン

検証結果ペインでの表示と並べ替え

DevPartner エラー検出では、アプリケーションで収集したデータをさまざまな方法で表示できます。検証結果ペインには、最初に、高レベルのレポートである【サマリ】タブが表示されます。【サマリ】タブをレビューし、エントリをダブルクリックすると詳細な情報が表示されます。

複数の層の情報を参照するこの機能には、以下のようなさまざまなデータ ビューがあります。

- ◆ 技術リーダーは、ある期間のメモリ リークが多いか少ないかなどの傾向を検索するデータをレビューできます。
- ◆ 開発者は、メモリ オーバーラン エラーやダングリグ ポインタなどを修正する場合があります。

この多階層ビューを使用すると、最も関連性の高いデータを特定し、検証結果ペインにある【メモリ リーク】タブ、【その他のリーク】タブ、【エラー】タブ、【.NET パフォーマンス】タブ、または【モジュール】タブでさらに詳細なビューを参照できます。これらのタブに表示されているデータは、カラム見出しをクリックして、サイズ、発生回数、場所などを基準に並べ替えることができます。

リバース エンジニアリング

DevPartner エラー検出を使用して Windows アプリケーションを分析できます。以下のような設定で構成を作成した場合に DevPartner エラー検出を使用すると、Windows アプリケーションで実行されているオペレーションを監視してレポートできます。

データ収集

より詳細な API パラメータ情報を生成するには、[コール パラメータのデータ表示の深さ] パラメータの値を大きくします。データ表示の深さの値を大きくすると、処理が遅くなり、ログ ファイルのサイズが大きくなります。

API コール レポートティング

API コールと戻り値をログに記録するには、[API コール レポートティングを有効にする] をオンにします。DevPartner エラー検出によってパラメータと、パラメータとして渡されたクラスで収集される詳細な情報の量は、【データ収集】の設定の【コール パラメータのデータ表示の深さ】の値によって決まります。

アプリケーションに送信されるすべてのウィンドウ メッセージを記録するには、[ウィンドウ メッセージを収集する] をオンにします。このオプションをオンにすると、マウス クリックや修復イベントなどのさまざまなウィンドウ イベントに、アプリケーションがどのように応答するのかを表示できます。

メモ: これらのオプションのいずれかをオンにすると、ログ ファイルのサイズが大きくなり、DevPartner エラー検出のパフォーマンスが低下します。

API コール レポートのオーバーヘッドを最小にするには、現在のタスクに最も関連しているシステム DLL だけを選択してください。

COM コール レポート

[選択したモジュールに実装された COM メソッド コールのレポートを有効にする]を選択して、COM メソッド コールを収集できるようにします。

COM コール レポートの情報を管理しやすくするには、最も関連性の高いインターフェイスだけを選択し、**[すべてのコンポーネント]** チェック ボックスをオフにします。

.NET コール レポート

[.NET メソッドコール レポートを有効にする]を選択して、.NET メソッドコールを収集できるようにします。.NET コール レポートの情報を管理しやすくするには、.NET ユーザー アセンブリ (デフォルト設定) のみを選択します。

.NET 分析

ネイティブ コードとマネージ コードが混在したアプリケーションを作成するときは、**[.NET 分析]**機能を使用して、以下の操作を行います。

- ◆ ネイティブ コードからマネージ コードにスローされる、ハンドルされていない例外を監視する。
- ◆ マネージ コードからネイティブ コードに行われるコール (P/Invoke または COM メソッド コール) を監視する。
- ◆ **[例外の監視]**を選択して例外を監視する。

マネージ コードからネイティブ コードへのコールを監視するには、**[COM 相互運用性の監視]**または**[P/Invoke 相互運用性の監視]**を選択してから、**[相互運用性レポートしきい値]**で適切な値を選択します。マネージ コードからネイティブ コードへのコールを監視するときは、レポートしきい値の値を十分に高く設定し、**[モジュールとファイル]**の設定を使用して不要な情報を減らしてください。

リバース エンジニアリング中に無効にする機能グループ

DevPartner エラー検出には、Windows アプリケーションでのさまざまなリークとエラーを監視するツールがあります。ただし、リバース エンジニアリングのセッション中は、DevPartner エラー検出のエラーおよびリーク検出ロジックを無効にすることをお勧めします。DevPartner エラー検出スタンドアロンと Visual C++ 6 IDE の場合は**[プログラム設定]**ダイアログ ボックスで、Visual Studio IDE の場合は**[オプション]**ダイアログ ボックスで、以下の操作を行ってこれらの機能を無効にします。

ヒント: これらの機能は、リバース エンジニアリングセッションの終了後に選択してください。

- 1 **【コールバリデーション】**の**【コールバリデーションを有効にする】**をオフにします。
- 2 **【COMオブジェクトの追跡】**の**【COMオブジェクトの追跡を有効にする】**をオフにします。
- 3 **【メモリの追跡】**の**【メモリの追跡を有効にする】**をオフにします。
- 4 **【リソースの追跡】**の**【リソースの追跡を有効にする】**をオフにします。
- 5 **【デッドロック分析】**の**【デッドロック分析を有効にする】**をオフにします。

これらの機能は、検証しているコードでバグを識別するためのものです。これらの機能を無効にすると、コンポーネントやAPI内のコードがどのように動作しているかを理解するのに役立つ情報だけを取得できます。

モジュールとファイル

デフォルトでは、**【システムディレクトリ】**除外リストにリストされている部分を除いて、アプリケーションのすべての部分についてレポートされます。

リバースエンジニアリングを行うときは、通常除外されるいくつかのDLLを監視することがあります。あるDLLを監視すると、そのDLLを追跡してその動作を確認できます。

たとえば、特定の共通コントロールでWIN32 API コールがどのように使用されているかを理解するには、**COMCTL32.DLL**を明示的に含めたあと、**【API コール レポートニング】**をオンにします。

システムDLLを明示的に監視するには、**【モジュールの追加】**をクリックし、必要なDLLを追加します。

構成ファイル管理

【構成ファイル管理】を使用すると、開発サイクルの特殊なタスク用に設計された設定を作成して保存できます。

以下に例を示します。

- ◆ メモリ リーク、リソース リーク、およびCOM リークの検出
- ◆ メモリと妥当性検証のみ
- ◆ リバース エンジニアリング
- ◆ カスタムの**【モジュールとファイル】**の設定で限定されたDLLのセットを除く、上記のいずれか

業務に不可欠なアプリケーション部分（パスワードチェックなど）を DevPartner エラー検出で監視しないようにするには、実行時に DevPartner エラー検出呼び出し可能インターフェイスを呼び出して、選択した DevPartner エラー検出ログ記録を無効にすることができます。詳細については、以下のファイルのイベント レポートに記載されているコメントを参照してください。

```
C:\Program Files\Compuware\DevPartner Studio\BoundsChecker
\ErptApi\NmApiLib.h
```

ストレス テスト

DevPartner エラー検出の実行に伴う副作用として、過負荷の状態でのみ発生する多くの予期しない状態が、アプリケーション側で強制的に処理させられることがあります。

ゼロ以外の未初期化データの処理

多くのアプリケーションは、動的なメモリ割り当てルーチンから返されるローカル変数とメモリは、何らかの値に初期化されているという誤った前提で作成されています。未初期化データ アクセスを検索するためにメモリを割り当てると、DevPartner エラー検出によってさまざまなタイプのメモリ上に既知の埋め込みパターンが書き込まれます。たとえば、ローカル変数や、**new**、**malloc**、**HeapAlloc**、または **LocalAlloc** によって割り当てられるメモリがあります。

未初期化メモリがゼロになるという前提でアプリケーションを作成すると、DevPartner エラー検出で実行したときにクラッシュしたり、予測しない動作が発生することがあります。このような場合は、**FinalCheck** でアプリケーションをインストールメントし、再度 DevPartner エラー検出でアプリケーションをチェックしてエラーを検索してください。

メモ： これらのルールに従わない独自のメモリ割り当てルーチンを記述した場合、**UserAllocators.dat** ファイルにそのルーチンのエントリを追加してください。詳細については、[第4章「ユーザーが作成したアロケータの使用」](#)を参照してください。

解放時の無効データによるプールのフィル

動的に割り当てられたメモリが解放されたあと、DevPartner エラー検出によって、そのメモリに既知のパターンが書き込まれます。これによって、解放された構造を参照しようとしたアプリケーションでエラーが発生します。多くの場合、ダングリングポインタ エラーの診断と修正は非常に困難です。**FinalCheck** でアプリケーションをインストールメントし、再度 DevPartner エラー検出でアプリケーションをチェックしてエラーを検索してください。

メモ： これらのルールに従わない独自のメモリ割り当てルーチンを記述した場合、**UserAllocators.dat** ファイルにそのルーチンのエントリを追加してください。詳細については、[第4章「ユーザーが作成したアロケータの使用」](#)を参照してください。

CPU 負荷が大きい環境での作業

多くの開発者は、非常に高速で負荷の小さいシステムでアプリケーションを作成します。このため、アプリケーションを運用環境に移行したときに、プログラムに不規則な障害が発生します。タイミングとパフォーマンス関連の問題の追跡は困難で、時間がかかる場合があります。

DevPartner エラー検出ではあらゆる面からプログラム フローを監視し、CPU とメモリの作業負荷が大きい状態にアプリケーションを置きます。同時に、Windows 関数へのコールでエラーの兆候を監視します。エラーは、**【検出されたプログラム エラー】** ダイアログ ボックスにレポートされます。

マルチスレッド コードでの問題の検出

多くのアプリケーションは、マルチプロセッサ アプリケーション サーバーを使用するように作成されています。マルチスレッド アプリケーションは注意深く設計しないと、プログラムがストレス状態に置かれたときに、デッドロックやリソース不足の問題が発生する可能性があります。

DevPartner エラー検出でマルチスレッド アプリケーションを実行すると、さまざまなスレッドのパフォーマンスが低下し、プログラムでタイミング関連の問題が発生する場合があります。そのような多くの問題は通常、運用環境でプログラムがストレス状態に置かれたときに発生します。DevPartner エラー検出を使用すると、開発プロセスで問題を検出し、製品化される前に修正できます。

デッドロック分析を有効にして DevPartner エラー検出でアプリケーションを実行すると、デッドロック、潜在的なデッドロック、その他の同期バグを検出できます。

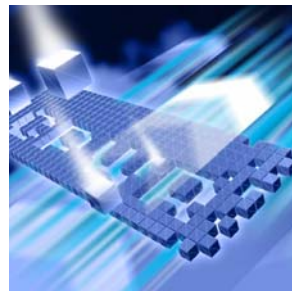
メモリおよびポインタの再利用エラーの検出

アプリケーションがより複雑になってきたため、アプリケーションで使用されるメモリの量とポインタの数が大幅に増大しました。この問題に対処するために、ソフトウェア開発者は DevPartner エラー検出などのツールを使用してメモリ リークとリソース リークを検索します。ただし、リークの検出とプラグはタスクの一部にすぎません。メモリが解放されたら、ブロックへのすべての未解決のポインタを「ダングリング」として宣言する必要があります。ダングリング ポインタを参照しようとする、エラーが生成されます。DevPartner エラー検出の FinalCheck 機能は、ダングリング ポインタを検出してレポートするように設計されています。

未検出のダングリング ポインタを使用すると、プログラムから、システムの他の部分によって解放されたブロック、または解放されて再利用されたブロックをプログラムから参照できます。単純なデバッグ環境でプログラムを実行しても、エラーの兆候が現れない場合があります。ところが、このプログラムを運用環境に移行すると、不規則にクラッシュしたり、データが破壊されたり、予期しない結果が生成されたりするおそれがあります。

第3章

複雑なアプリケーションの分析



- ◆ 複雑なアプリケーションについて
- ◆ プロセスを待機
- ◆ プログラムの特定部分の分析
- ◆ 監視対象の決定
- ◆ サービスの分析
- ◆ テスト コンテナを使用した ActiveX コントロールの分析
- ◆ COM を使用するアプリケーションの分析
- ◆ IIS 5.0 での ISAPI フィルタの分析
- ◆ IIS 6.0 での ISAPI フィルタの分析
- ◆ よく寄せられる質問 (FAQ)

この章では、複雑なアプリケーションを検証する場合に、DevPartner エラー検出をより効率的に使用するために役立つ情報を提供します。

複雑なアプリケーションについて

デフォルトのDevPartnerエラー検出の設定を使用してWindowsアプリケーションをデバッグすると、一般的なプログラミング問題の解決に役立つデータを収集できます。

複雑なアプリケーションをデバッグするときは、エラー検出の設定をカスタマイズすると便利です。

複雑なアプリケーションは以下の2つのグループに分けられます。

- ◆ 多くの複雑なコンポーネントが含まれている大規模なアプリケーション
- ◆ Windows NT サービス、ActiveX コンポーネント、MTS または COM コンポーネント、ISAPI フィルタなどの従来とは異なるアプリケーション

大規模なアプリケーション

大規模な Windows アプリケーションは、大規模であるために監視が難しいという理由だけで特殊です。DevPartner エラー検出を使用すると、アプリケーション全体を一度に分析するのではなく、論理的かつ管理可能なセクションごとに分析できます。たとえば、大規模なアプリケーション用に1つのDLLを記述する場合は、以下の操作を行います。

- ◆ アプリケーションのセクションを分析から除外します。
- ◆ アプリケーションの特定のセクションを監視します。
- ◆ アプリケーション内の特定のトランザクションだけを監視します。

従来と異なるアプリケーション

従来と異なるアプリケーションは、スタートアップまたは構成が複雑なので、別のエラー検出対策が必要な場合があります。このようなタイプのアプリケーションの監視に必要な特別なデバッグまたは分析の操作を実行するように DevPartner エラー検出を設定できます。

DevPartner エラー検出の機能と複雑なアプリケーション

エラー検出の以下の機能は、複雑なアプリケーションの分析に役立ちます。

- ◆ プロセスを待機する機能
- ◆ アプリケーションで監視するモジュールとファイルを制限する機能
- ◆ 実行時にエラー検出のログを有効または無効にする機能

プロセスを待機

エラー検出を指定してプログラムを実行する代わりに、エラー検出自体をアプリケーション用に初期化して、その処理が完了するまで待機する方法を使用できます。初期化が終了したら、手動でアプリケーションを起動します。または、サービス コントロール マネージャなどの手段を使用することもできます。このオプションを使用すると、IISなどのサービスをデバッグできます。

メモ： [プロセスを待機]を使用するときには、起動するアプリケーションのフルパス名とエラー検出によって検索されるアプリケーションのフルパス名が完全に一致している必要があります。

メモ： このオプションは、BoundsCheckerおよびDevPartnerエラー検出の以前のリリースにあったImage File Execution Optionsの代わりに導入された機能です。

このオプションは、DevPartnerエラー検出のアプリケーション（BC.EXE）を使用している場合にだけ有効です。Visual Studioに統合されているエラー検出を使用している場合は、利用できません。

「初期化して待機」する方法でエラー検出を使用してアプリケーションやサービスをデバッグするには、以下の手順を実行します。

- 1 エラー検出アプリケーション（BC.EXE）でテストするイメージを開きます。
- 2 エラー検出を設定し、関心のあるエラーを監視します。
- 3 [プログラム]メニューから[プロセスを待機]を選択します。
エラー検出自体が初期化され、セッションをキャンセルするかどうかを確認するダイアログボックスが表示されます。

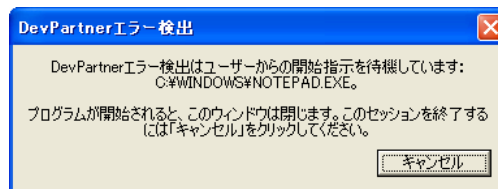


図 3-1. プロセスを待機するダイアログボックス

- 4 通常どおり、アプリケーションを開始します。
通常はサービス コントロール マネージャを使用してアプリケーションを開始している場合は、そのように操作します。アプリケーションを起動すると、エラー検出のダイアログボックスは閉じます。
- 5 アプリケーションを実行し、終了します。

プログラムの特定部分の分析

大規模なアプリケーションまたは複雑なアプリケーションの問題となる特定の領域を DevPartner エラー検出に指示して、アプリケーションのその他の部分を無視することができます。DevPartner エラー検出には、プログラムの特定部分を分析するときに役立つ4つのメカニズムがあります。

- ◆ **[モジュールとファイル]**を使用してプログラムのセクションを分析から除外します。
- ◆ **[抑制]**と**[フィルタ]**を使用して、不要な情報がログに記録されたり表示されたりしないようにします。
- ◆ **[プログラム]>[イベントをログに記録]**メニュー項目または**[イベントをログに記録]**ツールバー ボタンを使用して、エラー検出のログ記録を切り替えます。
- ◆ アプリケーションに**StartEvtReporting**と**StopEvtReporting**を呼び出す条件コードを追加します。

メモ： **StartEvtReporting**と**StopEvtReporting**は、アプリケーション内から呼び出して DevPartner エラー検出ログへのデータの書き込みを制御できる DevPartner エラー検出関数です。DevPartner のエラー検出がアクティブでないと、これらの呼び出しはすぐに戻ります。

モジュールとファイル

大規模なアプリケーションで作業する場合は、**[モジュールとファイル]**の設定を使用してアプリケーションのセクションを分析しないようにできます。これにより、分析時間が短縮され、不要なエラー メッセージが削減されます。除外できるセクションは以下のとおりです。

- ◆ サードパーティ製 DLL などの不要な DLL
- ◆ DLL または EXE からの個々のソース ファイル
- ◆ DLL ディレクトリ ツリー全体
- ◆ ソース コードがない場合のエラーは除外する

メモ： **[モジュールとファイル]**の設定ですべてのモジュールを無効にすると、一部のエラー タイプがレポートされてしまいます。エラー検出では、常にあらゆるモジュール内、MFCxxxx.dll ライブラリから発生するその他のイベント内でのメモリ オーバーランをレポートします。

[モジュールとファイル]の設定の使用 (32 ページ) を参照してください。

抑制とフィルタ

DevPartner エラー検出がレポートするエラーとイベントを非表示にするには、以下の2つの方法があります。

- ◆ **抑制**は、特定のタイプのエラーまたはイベントがエラー検出ログに記録されないようにします。抑制されたエラーを表示するには、抑制の指定を解除し、DevPartner エラー検出でアプリケーションを再実行します。
- ◆ **フィルタ**は、すでにログに記録されているエラーまたはイベントを表示しないようにします。フィルタされたエラーを非表示にしたり表示したりできます。

特定のイベント ログ

大規模なアプリケーションの中の小さなセクションを監視するには、**[イベントをログに記録]**メニューまたはツールバー ボタンを使用して、エラー検出ログのオン/オフを切り替えます。この方法は、以下の設定を選択するとき特に役に立ちます。

- ◆ API コールまたは COM コールのログ
- ◆ コール バリデーション

特定のイベント ログを、メモリ追跡、リソース追跡、COM インターフェイス追跡などのリーク検出機能に使用する場合、ほとんどのリークがプログラムの最後で検出されることに注意してください。プログラム終了時にログがオフの場合、検出しようとしているリークの多くはレポートされません。

リーク検出時には、**[モジュールとファイル]**または**[抑制]**を使用して不要な情報を除外します。

条件コード

DevPartner エラー検出のデータ収集エンジンへのコールを作成するようにプログラムを変更して、エラー検出ログを有効または無効にすることができます。以下のサンプル コードは、不要な領域のエラー検出ログを無効にする方法を示しています。

```
// Requires library [installation directory]
¥ERptApi¥NMApiLib.lib
// Include file is located in [installation
directory]¥ErptApi
#include "nmapilib.h"
... [Code that can be monitored]
StopEvtReporting()
... [Code that should not be monitored]
StartEvtReporting()
... [Code that can be monitored]
```

StartEvtReporting または **StopEvtReporting** API コールを使用して、DevPartner エラー検出で業務に不可欠なアプリケーションのセクションを記録しないようにすることもできます。たとえば、パスワードの検証、暗号化ルーチンなどです。DevPartnerのエラー検出がアクティブでないと、API コールはすぐに戻ります。



[モジュールとファイル]の設定の使用

アプリケーションのどの部分を除外するかを指定するには、以下の操作を行います。

- 1 DevPartner エラー検出で実行可能ファイルを開きます。
- 2 すべてのデータ収集を無効にします。
 - ◇ Visual C++ 6.0 の場合
[DevPartner]>[エラー検出設定]を選択します。
 - ◇ DevPartner エラー検出スタンドアロンの場合
[プログラム]>[設定]>[エラー検出]を選択します。
 - ◇ Visual Studio の場合
[DevPartner]>[オプション]を選択します。
[オプション]または[設定]ダイアログボックスで、[API コール レポートिंग]、[コールバリデーション]、[COM コール レポートिंग]、[COM オブジェクトの追跡]、[デッドロック分析]、[メモリの追跡]、および[リソースの追跡]の設定をオフにします。
- 3 DevPartner エラー検出でプログラムを実行します。
エラー検出によってアプリケーションで使用されるすべての DLL が記録されます。その DLL がすべてロードされるような方法でプログラムを実行し、アプリケーションを終了します。
- 4 DevPartner エラー検出の [設定] または [オプション] ダイアログボックスを開き、データ収集の設定を選択します。
- 5 [設定] または [オプション] ダイアログボックスの [モジュールとファイル] を選択します。DevPartner エラー検出によって、システム ディレクトリに保存されているファイル以外の、アプリケーションが使用するすべての実行可能ファイルと DLL が自動的にリストされます。
- 6 モジュールとファイルのリストを調べます。当面の作業に関係のない DLL をオフにします。少なくなった DLL のリストから、各 DLL を展開し、監視するソースファイルを選択します。

- 7 特定のディレクトリ内のDLLをすべて除外するには、**[システム ディレクトリ]**をクリックし、除外するディレクトリのリストにそのディレクトリを追加します。特定のファイルをシステム ディレクトリに含める場合は、**[モジュールの追加]**をクリックして、監視するDLLのリストにそのファイルを追加します。フォルダアイコンをクリックすると、単一フォルダアイコンと複数フォルダアイコンが切り替わります。表3-1に、アイコンの意味を示します。

表 3-1. このダイアログ ボックスでのフォルダ アイコンの意味

アイコン	説明
	選択したディレクトリはテストから除外されます（特定のdllが[モジュール]ダイアログにもリストされている場合は除きます）。
	選択したディレクトリとすべてのサブディレクトリはテストから除外されます。

- 8 ソース コードのないプログラム部分のリークやエラーを除外するには、**[ソースコードがある場合のみリークとエラーを表示する]**を選択します。
- 9 アプリケーションの論理サブセットの作成後、**[構成ファイル管理]**を使用して設定を保存します。

[モジュールとファイル]の設定方法について、表3-2にリストを示します。

表 3-2. [モジュールとファイル]の設定方法

デバッグの対象	除外対象
ActiveX コントロール	ActiveX テスト コンテナ実行可能ファイルなどの ActiveX コントロールが含まれている DLL 以外のすべてのモジュール
Windows NT サービス	デバッグしているサービスのセクションに直接関連付けされていないモジュール
ISAPI フィルタ	該当する ISAPI フィルタ以外の IIS または W3WP 内のすべての実行可能ファイルと DLL
複雑なアプリケーション	解決しようとしている問題とは無関係のアプリケーションのセクション
アウトオブプロセス COM オブジェクト	DLLHOST.exe や MTX.exe など、DLL に直接関連付けされていないモジュール

- メモ：** コード以外のすべてを除外する場合は、アプリケーションのセクションによって間接的に発生するメモリ リークまたはリソース リークをチェックできません。

ヒント： 複数の構成ファイルを作成する場合は、いずれかの構成ファイルに Base Configuration という名前を付けます。この Base Configuration 設定を基にして、他の構成ファイルを作成できます。

監視対象の決定

複雑なアプリケーションを処理する場合は、アプリケーションのどのセクションを監視対象にするかが重要です。どこを監視対象セクションにするかで、リークとエラーを正常に追跡できるかどうかが決まります。

監視対象セクションを決定するには、アプリケーションに関する以下の項目を検討します。

- ◆ アプリケーションはどのような方法で起動しますか。
 - ◇ 直接起動しますか。
 - ◇ 他のプログラムを実行して起動しますか。
 - ◇ コントロール パネルから起動しますか。
 - ◇ 間接的に起動しますか。
- ◆ アプリケーションには何個のモジュールとファイルがありますか。
 - ◇ システム モジュール以外の、アプリケーションのすべてのモジュールを所有していますか。
 - ◇ すべてのモジュールのソースを持っていますか。
- ◆ 監視対象はアプリケーション全体ですか、または特定の部分ですか。
 - ◇ 管理外のモジュールのエラーを考慮しますか。
 - ◇ アプリケーションはトランザクションですか。その場合、アプリケーション全体を監視しますか、または特定のトランザクションを監視しますか。
 - ◇ アプリケーションで、管理外のコードから渡されたリソースが使用されますか。

これらの質問について検討してから、**DevPartner** エラー検出でのアプリケーションの監視を設定します。

監視対象を決定するときは、プログラムのその他の部分がアプリケーションにリソースを提供する可能性があることも考慮します。監視対象を狭めると、選択した分析サブセットとその他のアプリケーションの間で渡されるリソースを見逃す可能性があることに注意してください。

たとえば、**ActiveX** コントロールを作成し、それをテスト コンテナで実行する場合は、**DLL** で何が起きるかを把握しておく必要があります。ただし、オブジェクトを誤って呼び出すと、リソース リークとインターフェイス リークが発生することがあります。コントロールだけを監視する場合は、エラーを検出できますが、コントロールの誤使用によって発生するエラーは検出できません。

アプリケーションの起動方法

コンソールまたはWindowsアプリケーションで作業する場合は、**【ファイル】>【開く】**を選択してアプリケーションを監視するようにエラー検出を設定できます。DevPartnerエラー検出でアプリケーションが開き、直接リンクされているDLLが分析されます。

従来とは異なるアプリケーションを使用している場合は、以下のいずれかに該当します。

- ◆ 制御プログラムから直接起動される。
- ◆ システムの動作に基づいて、間接的に起動される。

最初のタイプのアプリケーションは、一部のテストアプリケーションによって起動されるActiveXコントロールやDLLなどです。たとえば、ActiveXコントロールを作成した場合は、Visual Studioのテストコンテナアプリケーション (**TSTCON32.EXE**)を使用して分析できます。

アプリケーションがシステムの動作によって間接的に起動される場合は、エラー検出の**【プロセスを待機】**オプションを使用して、アプリケーションの開始を待機します (**「プロセスを待機」** (29ページ) を参照してください)。この例は以下のとおりです。

- ◆ Windows NT サービス
- ◆ アウトオブプロセスCOMサーバー

サービスやCOMサーバーなどの多くの特殊なアプリケーションはスピードが重視されます。スピード重視のアプリケーションで最良の結果を得るには、DevPartnerエラー検出を使用するときにタイムアウトロジックを無効にします。

サービスの分析

DevPartnerエラー検出ではWindows NTサービスを監視できます。サービスの監視時は、以下の項目を検討します。

- ◆ そのサービスは起動時に開始しますか、要求時に開始しますか。
- ◆ そのサービスには特定のセキュリティコンテキストが必要ですか。
- ◆ そのサービスは、インタラクティブに実行されますか。
- ◆ そのサービスはサービスとしてでなくても実行できますか。
- ◆ そのサービスにはタイミングの問題がありますか。

DevPartnerエラー検出では、システムを起動し実行したあとに開始するサービスを分析できます。最良の結果を得るには、デバッグプロセス時に、サービスを手動で開始または停止できるようにする必要があります。

要件とガイドライン

サービスを監視するには、DevPartner エラー検出の実行時に使用するアカウントは、管理者権限である必要があります。また、アプリケーションのタイミング要件が厳しい場合は、他の問題が生じる可能性があることに注意してください。

サービスの分析

DevPartner エラー検出でサービスを分析するには、以下の操作を行います。

- 1 サービスを停止します。
- 2 サービスのデバッグ構成を、シンボルを使用し最適化しないで（オプションで **FinalCheck** を使用）ビルドします。
- 3 エラー検出を使用してサービスのイメージを開き、セッションに合わせて設定を更新します。
- 4 [プログラム]メニューから [プロセスを待機] を選択します。
エラー検出自体が初期化され、セッションをキャンセルするかどうかを確認するダイアログ ボックスが表示されます。
- 5 通常どおり、サービスを開始します。
通常はサービス コントロール マネージャを使用してサービスを開始している場合は、そのように操作します。アプリケーションを起動すると、エラー検出のダイアログ ボックスは閉じます。

タイミングの問題と dwWait

サービスが起動しない場合、または起動してもすぐに終了してしまう場合は、**SetServiceStatus** に渡す **ServiceStatus** ブロックの **dwWait** パラメータを変更する必要があります。サービスに指定されている値が小さすぎると、Windows NT のサービス コントロール マネージャによってサービスが中止されます。DevPartner エラー検出を使用するときは、**dwWait** を 400 万程度の大きな値に設定します。

メモ： **dwWait** の値は、DevPartner エラー検出の使用後に通常の値に戻してください。

代替メソッド：ワーカー スレッドからのコントロール ロジックの分離

サービスがモジュール方式で作成されている場合は、ワーカー スレッドからサービスのコントロール ロジックを分離できます。これには、単純なコンソール アプリケーションでワーカー スレッド ロジックをラッピングする方法があります。これにより、DevPartner エラー検出を使用して、Windows のコンソール プログラムであるかのようにサービスのワーカー スレッドをチェックできます。

DevPartner エラー検出をログオンまたはログオフするカスタム コード

インタラクティブでないサービスを処理するときは、サービスの実行中に DevPartner エラー検出をログオンまたはログオフするカスタム コードを作成できます。 `dwControl` パラメータから `ControlService` に渡す制御コードに応答するカスタム コードを作成します。

サービスのコントロール ロジックで、開始および停止イベントをレポートする API を呼び出すことができます。「条件コード」(31 ページ) を参照してください。

サービス関連の一般的な問題

サービスが起動するとすぐにハングする：

管理者権限でサービスを実行します。管理者権限を取得できない場合は、前述の代替メソッドを使用します。

サービスが起動するとすぐに終了する：

一般的に考えられる原因は、Windows NT のサービス コントロール マネージャによるサービスの中止です。サービスの初期化ロジックの `dwWait` の値を大きくしてサービスを再実行します。

また、有効な作業ディレクトリが DevPartner エラー検出にあることも確認します。[プログラム]メニューの[設定]にある[全般]の設定を使用して、作業ディレクトリを指定します。

問題が解決しない場合は、前述の代替メソッドを使用してサービスを変更することを検討してください。

サービスが実行してしばらく経つと、突然終了する：

サービス状態を要求するコントロール メッセージへのサービスの応答が遅すぎる可能性があります。サービス状態の要求に応答する場合は、`dwWait` のタイムアウト値を大きくします。

また、DevPartner エラー検出によってアプリケーションのメモリが解放時に無効データによってフィルされ、クラッシュすることもあります。エラー検出の設定の **[メモリ追跡]** 機能を無効にします。これでクラッシュが解消したら、`FinalCheck` でサービスをインストールし、アプリケーションを再実行して、初期化されていないメモリ リファレンス、バッファ オーバーラン、およびダングリング ポインタを探します。

問題が解決しない場合は、前述の代替メソッドを使用してサービスを変更することを検討してください。

サービスは正常に実行されるが、シャットダウン時に突然終了する：

サービス コントロール マネージャからシャットダウン要求を受け取ったときのサービスの応答時間が制限されています。アプリケーションの終了時は、DevPartner エラー検出によって、メモリ リーク、リソース リーク、およびインターフェイス リークの検出や、割り当て済みのメモリ ブロックを再チェックしメモリ オーバーランが発生していないかどうかの確認が行われます。シャットダウン要求に応答するために指定されている `dwWait` 値が小さすぎると、サービス コントロール マネージャによってサービスが中止されます。この場合は、`dwWait` 値を大きくします。

問題が解決しない場合は、前述の代替メソッドを使用してサービスを変更することを検討してください。

テスト コンテナを使用した ActiveX コントロールの分析

Visual Studio のテスト コンテナ ユーティリティと共に DevPartner エラー検出を使用して、テスト コンテナと併用できる ActiveX コントロールとその他の COM オブジェクトを監視します。

DevPartner エラー検出をテスト コンテナと併用するには、以下の操作を行います。

- 1 DevPartner エラー検出を実行します。
- 2 **[ファイル]>[開く]** を選択し、テスト コンテナを選択します。
Visual Studio を標準のディレクトリにインストールした場合、テスト コンテナは以下のいずれかの場所にあります。
C:\Program Files\Microsoft Visual Studio\Common\Tools\1\TestCon32.exe
C:\Program Files\Microsoft Visual Studio .NET\Common7\Tools\TestCon32.exe
C:\Program Files\Microsoft Visual Studio .NET 2003\Common7\Tools\TestCon32.exe
C:\Program Files\Microsoft Visual Studio 8\Common7\Tools\TestCon32.exe
- 3 **[モジュールとファイル]** の設定を表示します。
- 4 **TestCon32.exe** の選択を解除します。
- 5 **[モジュールの追加]** をクリックします。
- 6 ActiveX または COM コントロールが含まれている DLL をモジュールとファイルのリストに追加します。
- 7 コントロールに必要な他の DLL を追加します。
- 8 アプリケーションを実行します。

テスト コンテナのアプリケーションを起動するには、以下の操作を行います。

- 1 ツールバーの**[新規コントロール]**をクリックします。
- 2 表示されているリストからコントロールを追加します（たとえば、Calendar Control 8.0）。
- 3 ツールバーの**[メソッドの起動]**と**[プロパティ]**ボタンを使用してコントロールを操作します。
- 4 コントロールの操作が終了したら、テスト コンテナを終了します。

実行時、エラーが検出されると DevPartner エラー検出からレポートされます。テスト コンテナを終了するときは、実行時にレポートされなかったメモリ リーク、リソース リーク、およびインターフェイス リークがレポートされます。

テスト コンテナの一般的な問題

なぜ DevPartner エラー検出から TestCon32.exe のエラーがレポートされるのか：

デフォルトでは、**[モジュールとファイル]**または**[システム ディレクトリ]**で DLL と EXE が明示的に除外されていない場合は、プロセスに関連付けられている実行可能ファイルとすべての DLL のエラーがレポートされます。DevPartner エラー検出で **TestCon32.exe** のエラーがレポートされないようにするには、チェックするモジュールのリストから該当する実行可能ファイルを除外します。

なぜ DevPartner エラー検出 COM コール レポートではオブジェクトへのコールがログに記録されないのか：

DevPartner エラー検出は、認識するように指示されている COM インターフェイスのみのメソッドを記録します。DevPartner エラー検出に ActiveX コントロールについて通知するには、**[COM コール レポート]**の設定の**[選択したモジュールに実装された COM メソッド コールのレポートを有効にする]**を選択して、メソッドのログを記録します。

なぜ DevPartner エラー検出からオブジェクトの COM インターフェイス リークがレポートされないのか：

COM インターフェイス リーク情報を収集するには、**[COM オブジェクトの追跡]**の設定の**[COM オブジェクトの追跡を有効にする]**をオンにして、監視する COM クラスを選択します。

独自のオブジェクトを追跡するには、**[COM オブジェクトの追跡]**の設定の COM クラスのリストから、該当するクラスだけを選択します。選択するクラスがわからない場合は、**[すべての COM クラス]**を選択します。

COMを使用するアプリケーションの分析

DevPartner エラー検出ではCOM コンポーネントを分析できます。DevPartner エラー検出でCOMコンポーネントを分析するには、**[コンポーネント サービス]**のCOMの設定を編集し、DevPartner エラー検出をCOM コンポーネントのデバッグとして設定します。

DevPartner エラー検出をCOM コンポーネントのデバッグとして設定するには、以下の操作を行います。

- 1 **[スタート]>[設定]>[コントロール パネル]>[管理ツール]>[コンポーネント サービス]**を選択します。
- 2 **[コンポーネント サービス]** ウィンドウのツリー コントロールを使用して、**[COM+ アプリケーション]**を開きます。
- 3 ツリー コントロールからコンポーネントを選択します。
- 4 コンポーネントを右クリックして、**[プロパティ]**を選択します。
- 5 コンポーネントのプロパティ シートで、**[詳細設定]**タブをクリックします。
- 6 **[詳細設定]** タブで、**[デバッグ内で実行する]**を選択します。
- 7 **[デバッグ パス]**を変更して**bc.exe**を指定します。フル パスを入力します。DevPartner エラー検出をインストールしたときのデフォルト パスを選択した場合、このパスは以下のようになります。

```
C:¥Program Files¥Compuware¥DevPartner Studio  
¥BoundsChecker¥bc.exe
```

注意： デバッグ パスの最後から `dllhost.exe /ProcessID:{...}` を削除しないでください。

- 8 **[OK]**をクリックして、変更内容を保存します。

DevPartner エラー検出をコンポーネントのデバッグとして設定したら、以下の操作を行います。

- 1 以下のいずれかのメソッドを使用して、コンポーネントを開始します。
 - ◇ コンポーネントを使用するアプリケーションを実行します。
 - ◇ **[コンポーネント サービス]**を使用してアプリケーションを開始します。
 - ツリー コントロールからコンポーネントを選択します。
 - コンポーネントを右クリックして、**[開始]**を選択します。
- 2 DevPartner エラー検出の起動時に、使用する設定を選択し、エラー検出を開始します。

メモ： COMコンポーネントだけのエラーとイベントを監視するには、DevPartner エラー検出の**[モジュールとファイル]**設定のモジュールのリストから `dllhost.exe` と他のDLLを削除します。

ヒント: `dllhost.exe` を誤って削除しないようにするため、**[参照]**を使用しないで、パスを切り取って貼り付けるか、または入力します。

- 3 コンポーネントの実行が終了したら、コンポーネントをシャット ダウンします。以下の操作を行います。
 - a **【コンポーネント サービス】**ウィンドウで、ツリー コントロールからコンポーネントを選択します。
 - b コンポーネントを右クリックして、**【シャットダウン】**を選択します。
- 4 DevPartner エラー検出によって、通常のプロセス終了時のエラーおよびリーク検出が実行されます。
- 5 デバッグが終了したら、**【デバッガ内で実行する】**チェック ボックスをオフにします。
 - a **【コンポーネント サービス】**ウィンドウのツリー ビューで、コンポーネントを選択します。
 - b コンポーネントを右クリックして、**【プロパティ】**を選択します。
 - c プロパティ シートの**【詳細設定】**タブをクリックして、**【デバッガ内で実行する】**をオフにします。
 - d **【OK】**をクリックします。

COMの一般的な問題

なぜ DevPartner エラー検出から `dllhost.exe` のエラーがレポートされるのか：

デフォルトでは、**【モジュールとファイル】**または**【システム ディレクトリ】**でDLLとEXEが明示的に除外されていない場合、DevPartner エラー検出によって、プロセスに関連付けられている実行可能ファイルとすべてのDLLのエラーがレポートされます。DevPartner エラー検出で`dllhost.exe`のエラーがレポートされないようにするには、チェックするモジュールのリストから該当する実行可能ファイルを除外します。

なぜ DevPartner エラー検出 COM コール レポートではコンポーネントへのコールがログに記録されないのか：

DevPartner エラー検出では認識できるインターフェイスのCOMメソッド コールだけがログに記録されます。メソッドのログをアクティブにするには、**【COM コール レポート】**の設定の**【選択したモジュールに実装されたCOMメソッド コールのレポートを有効にする】**を選択します。

なぜ DevPartner エラー検出ではコンポーネントの COM インターフェイス リークがレポートされないのか：

DevPartner エラー検出では、**【COM オブジェクトの追跡】**設定の**【COM オブジェクトの追跡を有効にする】**がオンになっている場合だけ、COM インターフェイス リーク情報がレポートされます。監視するCOMクラスを指定することも必要です。

独自のインターフェイスだけを追跡するには、**【COM オブジェクトの追跡】**設定のCOMクラスのリストから、該当するクラスだけを選択します。選択するクラスがわからない場合は、**【すべてのCOMクラス】**を選択します。

コンポーネントの実行を停止すると、DevPartner エラー検出がハングしたかのように長時間応答しない：

DevPartner エラー検出は `dllhost.exe` がタイムアウトし、プロセスを終了するまで待機します。`dllhost.exe` が終了すると、メモリ リーク、リソース リーク、およびインターフェイス リークの最終的な検出が行われます。

タイムアウトする前に `dllhost.exe` を終了するには、[コンポーネント サービス] ウィンドウからコンポーネントを探し、該当するコンポーネントを右クリックし、[シャットダウン] を選択します。

[プロセスを待機] を使用して `dllhost.exe` をデバッグする方法はあるのか：

[プロセスを待機] を使用して `dllhost.exe` をデバッグすることは絶対に避けてください。Windows 2000 や Windows XP システムで作成されるコンポーネント数を考慮すると、コンポーネントサービスのデバッグ オプションを使用して、COM でサポートされているメカニズムを使用する方が安全です。

サポートされているデバッグメカニズムを使用できない場合は、他の COM コンポーネントが要求されたときに、異常なシステム障害が発生することがあります。`dllhost.exe` のすべてのインスタンスが DevPartner エラー検出に関連付けられているため、コンポーネントは正常に起動できません。

IIS 5.0 での ISAPI フィルタの分析

DevPartner エラー検出を使用して IIS プロセス内の ISAPI フィルタを分析できます。DevPartner エラー検出で ISAPI フィルタを分析するには、以下の操作を行います。

- 1 デバッグ構成を持つ ISAPI フィルタを、シンボルを使用し最適化しないで (オプションで `FinalCheck` を使用) ビルドします。
- 2 Internet Information Server (IIS) サービスを停止します。
- 3 `inetinfo.exe` 用にエラー検出を構成します。
 - a エラー検出アプリケーション (BC.EXE) で `inetinfo.exe` を開きます。`inetinfo.exe` は、以下の場所にあります。

```
%WINDIR%\System32\Inetsrv\inetinfo.exe
```
 - b [オプション] または [設定] で [モジュールとファイル] を開き、すべての EXE と DLL をオフにします。
 - c [モジュールの追加] をクリックし、モジュールのリストに ISAPI フィルタを追加します。
 - d ISAPI フィルタに合わせて残りの設定を更新します。

- 4 テスト対象の ISAPI 拡張機能が入っている仮想ディレクトリの保護レベルを高く設定 ([XXXApplication Protection] で [XXXHigh (Isolated)] を選択) します。
 - a [インターネット インフォメーション サービス マネージャ]を開きます。
 - b その仮想ディレクトリを参照します。
 - c 右クリックして、[プロパティ]を選択します。
 - d このダイアログ ボックスの [仮想ディレクトリ] タブで [XXXApplication Protection] を [XXXHigh (Isolated)] に設定します。
- 5 [プログラム] メニューから [プロセスを待機] を選択します。
エラー検出自体が IIS 用に初期化されます。IIS は開始を待機します。
- 6 [サービス] コントロール パネルから IIS Admin Service を起動します。
- 7 IIS サーバーへの一連の HTTP リクエストを生成し、ISAPI フィルタを実行します。
- 8 ISAPI フィルタの実行が終了したら、[サービス] コントロール パネルを使用して IIS サービスを停止します。
- 9 エラー検出によって、プロセス終了時のエラーおよびリーク検出が実行されます。

ISAPI フィルタの一般的な問題

ISAPI フィルタのデバッグに関連付けられている一般的な問題の多くについては、「サービス関連の一般的な問題」で説明しました。

IIS と ISAPI フィルタのデバッグに特有の問題を以下に示します。

なぜ IIS は起動してすぐハングするのか：

DevPartner エラー検出でサービスをデバッグするには管理者権限が必要です。管理者権限のないアカウントを使用すると、IIS はハングするか、エラーの発生と同時に終了します。

なぜ DevPartner エラー検出のログには不要な情報が多く含まれているのか：

[モジュールとファイル] 設定で、`inetinfo.exe` や、ISAPI フィルタ以外のすべての DLL を除外できます。

`inetinfo.exe` をはじめて実行したときは、DevPartner エラー検出によって、プロセスに動的にロードされた DLL がモジュールとファイルのリストに自動的に追加されます。[モジュールとファイル] 設定ダイアログを使用して、不要な DLL をオフにします。リストからは削除しないでください。そうすれば、次に実行するときに、リストに追加して有効にすることができます。

サービスと ISAPI フィルタのデバッグについて、その他の参考資料があるのか：

- ◆ IIS と ISAPI フィルタのデバッグ方法については、多くの優れた記事が MSDN に掲載されています。
- ◆ コンピューウェアの Web サイトには、多くのナレッジ ベース情報があります。

DevPartner エラー検出を使用してインタラクティブに IIS をデバッグするためのヒントは何か：

- ◆ 管理者権限を持つアカウントでログインする必要があります。
- ◆ これらの点に注意しても問題が解消されない場合は、以下の Microsoft MSDN Library 「ISAPI アプリケーションのデバッグ」をお読みください。
<http://msdn.microsoft.com/library/ja/default.asp?url=/library/ja/vsdebug/html/vxoriDebuggingISAPIApplication.asp>

IIS 6.0 での ISAPI フィルタの分析

以下のいずれかの方法で IIS 6.0 を設定した場合、DevPartner エラー検出を使って ISAPI フィルタを分析できます。

- ◆ IIS 5.0 Isolation Mode
- ◆ IIS 6.0 デフォルト設定

DevPartner エラー検出で ISAPI フィルタを分析するには、まず、デバッグ構成を持つ ISAPI フィルタを、最適化しないでビルドします（オプションで FinalCheck を使用）。その後、使用する IIS 構成の指示に従ってください。

IIS 5.0 Isolation Mode

IIS 5.0 Isolation Mode で IIS 6.0 を実行すると場合、**inetinfo.exe** 実行可能ファイルに対して DevPartner エラー検出を実行します。

ISAPI フィルタを分析するには、以下の操作を行います。

- 1 **inetinfo.exe** 用にエラー検出を構成します。
 - a エラー検出アプリケーション（BC.EXE）で **inetinfo.exe** を開きます。**inetinfo.exe** は、以下の場所にあります。
`%WINDIR%\System32\Inetsrv\inetinfo.exe`
 - b [オプション] または [設定] で [モジュールとファイル] を開き、すべての EXE と DLL をオフにします。
 - c [モジュールの追加] をクリックし、モジュールのリストに ISAPI フィルタを追加します。
 - d ISAPI フィルタに合わせて残りの設定を更新します。
- 2 テスト対象の ISAPI 拡張機能が入っている仮想ディレクトリの保護レベルを高く設定（[XXXApplication Protection] で [XXXHigh (Isolated)] を選択）します。
 - a IIS Admin ユーティリティを開きます。
 - b その仮想ディレクトリを参照します。
 - c 右クリックして、[プロパティ] を選択します。
 - d このダイアログ ボックスの [仮想ディレクトリ] タブで [XXXApplication Protection] を [XXXHigh (Isolated)] に設定します。

- 3 [コントロールパネル]で[サービス]ダイアログ ボックスを開きます。
 - 4 [プログラム]メニューから[プロセスを待機]を選択します。
エラー検出自体がIIS用に初期化されます。IISは開始を待機します。
 - 5 IIS Admin と World Wide Web Publishing Service を開始します。
DevPartner エラー検出が自動的に起動され、inetinfo.exe プロセスと ISAPI フィルタを監視します。
 - 6 Web サーバーへの一連の HTTP リクエストを生成し、ISAPI フィルタを実行します。
 - 7 ISAPI フィルタの使用を終了したあと、IIS マネージャで IIS を停止します。
[シャットダウン中]ダイアログ ボックスが開いたら、[今すぐ終了]をクリックします。これによって、inetinfo.exe プロセスが終了します。
- メモ：** DevPartner エラー検出で[停止]をクリックすると、DevPartner エラー検出と inetinfo.exe プロセスが両方とも停止されるので、収集したデータが失われます。
- 8 エラー検出によって、プロセス終了時のエラーおよびリーク検出が実行されます。

IIS 6.0 デフォルト設定

デフォルト構成モードで IIS 6.0 を実行するときは、W3WP.exe 実行可能ファイルに対して DevPartner エラー検出を実行します。

ISAPI フィルタを分析するには、以下の操作を行います。

- 1 W3WP.exe 用にエラー検出を設定します。
 - a エラー検出アプリケーション (BC.EXE) で W3WP.exe を開きます。
W3WP.exe は、以下の場所にあります。
`%WINDIR%\System32\Inetsrv\W3WP.exe`
 - b [オプション]または[設定]で[モジュールとファイル]を開き、すべての EXE と DLL をオフにします。
 - c [モジュールの追加]をクリックし、モジュールのリストに ISAPI フィルタを追加します。
 - d ISAPI フィルタに合わせて残りの設定を更新します。
- 2 テスト対象の ISAPI 拡張機能を含む仮想ディレクトリに対し、MSSharePointAppPool を使用するように設定します。
 - a [インターネットインフォメーションサービス (IIS) マネージャ]を開きます。
 - b その仮想ディレクトリを参照します。
 - c 右クリックして、[プロパティ]を選択します。
 - d [仮想ディレクトリ]タブの[アプリケーションプール]を MSSharePointAppPool に設定します。

- 3 [プログラム]メニューから[プロセスを待機]を選択します。
エラー検出自体が IIS 用に初期化されます。IIS は開始を待機します。
 - 4 IIS Admin と World Wide Web Publishing Service を開始します。
エラー検出が自動的に起動され、W3WP.exe プロセスと ISAPI フィルタを監視します。
 - 5 IIS サーバーへの一連の HTTP リクエストを生成し、ISAPI フィルタを実行します。
 - 6 ISAPI フィルタの使用を終了したあと、IIS マネージャで IIS を停止します。
[シャットダウン中]ダイアログ ボックスが開いたら、[今すぐ終了]をクリックします。これによって、W3WP.exe プロセスが終了します。
- メモ：** DevPartner エラー検出で[停止]をクリックすると、エラー検出と W3WP.exe プロセスが両方とも停止されるので、収集したデータが失われます。
- 7 エラー検出によって、プロセス終了時のエラーおよびリーク検出が実行されます。

IIS 6.0 ISAPI フィルタの一般的な問題

IIS 5.0 の ISAPI フィルタの一般的な問題がすべて含まれます。場合によっては、W3WP.exe を inetinfo.exe に代用する必要があります。

「ISAPI フィルタの一般的な問題」(43 ページ) を参照してください。

IIS 6.0 に関する以下の問題点が新たに確認されました。

- ◆ Microsoft は IIS 6.0 のデフォルト設定を再設計し、セキュリティを強化しました。この変更の一部として、ISAPI 拡張機能がデフォルトで無効に設定されるようになりました。ISAPI 拡張機能をデバッグするには、IIS 管理ツールの [Web サービス拡張] タブを開いて、不明な ISAPI 拡張機能を有効にするように IIS を変更してください。
 - ◆ IIS マネージャ ツールを使用して、IIS を起動、停止、再起動できます。これらの操作を行うには、ツリーで <コンピュータ名> ノードを右クリックし、[すべてのタスク] > [IIS の再起動] を右クリックします。これによって、IIS の起動や停止が可能なコントロールを含むダイアログ ボックスが開きます。
 - ◆ 最良の結果を得るために、IIS の監視を開始する前に API コール ログなどのログ機能をオフにしてください。ログ機能がオンになっていると、DevPartner エラー検出によって非常にサイズの大きいログ ファイル (.DPBcl) が作成されるので、IIS サーバーのパフォーマンスに影響します。
- メモ：** 一般的な設定を行うダイアログで [イベントをログに記録] を無効にしないでください。[イベントをログに記録] が無効になっていると、エラー検出では何もレポートされません。この機能は、メニュー バー ボタンを使用してイベント ログを有効にするまですべてのレポートを抑制したいときにだけ、使用してください。

よく寄せられる質問 (FAQ)

DevPartner エラー検出 ActiveCheck と FinalCheck の違い (テクノロジーの違い) は何ですか:

DevPartner エラー検出には以下の2つの動作モードがあります。

- ◆ **ActiveCheck** –このモードでは、DevPartner エラー検出は任意の32ビット Windows プログラム上で動作し、オペレーティング システムおよびCランタイム ライブラリへのすべてのコールをインターセプトし、有効でない (または割り当て解除された) 機能に渡されたメモリ リーク、リソース リーク、およびポインタの使用状況を監視します。
- ◆ **FinalCheck** –このモードでは、DevPartner エラー検出 FinalCheck インストールメンテーション ロジックを使ってCやC++プログラムを再コンパイルする必要があります。FinalCheck を使用してビルドするには、以下の手順を実行します。
 - ◇ Visual C++ 6.0の場合、[DevPartner] > [*build preference*] > [エラー検出] を選択します。

メモ: *build preference* は、[ビルド]、[すべてリビルド]、[バッチ ビルド]のうち、利用できるオプションを示します。
 - ◇ Visual Studio 2003 と Visual Studio 2005 の場合は、[DevPartner] > [ネイティブC/C++インストールメンテーションマネージャ]を選択します。FinalCheck インストールメンテーションを使用すると、DevPartner エラー検出で、ターゲット モジュール内で行われるすべてのポインタの *fetch*、*store*、*indirect* を監視できます。さらに、変数の有効範囲内外の移動も監視できます。

注意: 評価順序を明確にしないでコードをインストールメントすると、エラーデータ、ハング、さらにはクラッシュなどの予期しない結果が生じる可能性があります。

「CとC++では基本的に、変数への書き込みもしている1つの式の中で変数を2回読んだ結果は未定義です」 – Bjarne Stroustrup

C/C++ 標準では、オブジェクトに値を保存するなどの「副次的な影響」がある場合、評価順序は明確に定義されません。たとえば、`i = ++i + 2;` のステートメントは評価順序が明確に定義されていません。

値が変数「i」に保存され、言語によって値の発生順序が定義されていない場合、このステートメントには2つの意味が存在します。このようにコードをインストールメントすると、評価順序が変わり、異なる結果が生じる場合があります。

FinalCheck モードで実行するとき、拡張 FinalCheck 分析と同時に、すべての ActiveCheck 分析も実行されます。

FinalCheck は、ダングリング ポインタ、二重解放、ポインタ オーバーラン、未初期化メモリ エラー、未割り当てメモリの読み取り／書き込みの検出専用です。

コールバリデーションを有効にするのはどのような場合ですか：

コールバリデーションを有効にすると、プログラム内でより多くのメモリおよびポインタ エラーを検出できるようになります。検出するイベント量が多くなるので、この機能はデフォルトでオフになっています。

[コールバリデーション]で[メモリブロックチェックを有効にする]機能を選択すると、DevPartner エラー検出はどのように動作しますか：

[メモリブロックチェックを有効にする] (デフォルトでオフ) を選択すると、DevPartner エラー検出でより詳細な ActiveCheck 分析が行われます。ただし、DevPartner エラー検出の実行速度が20%まで低下する可能性があります。

メモリ追跡実行時、DevPartner エラー検出の[保護バイト]の設定はどのように使用しますか：

[メモリの追跡]設定で保護バイトの設定を変更するには、まず[保護バイトを有効にする]がオンになっていることを確認してください。

[カウント]を4より大きくして、8または16にします。

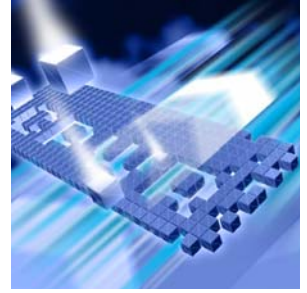
保護バイトを増やすと、ヒープブロック間の間隔が増え、DevPartner エラー検出でオーバーランを監視するブロック間の間隔が増えます。

[実行時のヒープブロックチェック]の設定を[適応分析の使用]、または[すべてのメモリ API コール時]に変更します。

このオプションは、DevPartner エラー検出に、メモリ関数を呼び出すたびに各ヒープブロックを検証するように指示します。これによって、プログラムの実行速度は大幅に低下しますが、ヒープ破壊をプログラム内の限定された領域に切り分けるので、追跡が簡単になります。

第4章

ユーザーが作成したアロケータの使用



- ◆ 概要
- ◆ 必要な情報の収集
- ◆ UserAllocators.datのエントリの作成
- ◆ UserAllocator フック要求のコーディング
- ◆ UserAllocator フックのデバッグ
- ◆ UserAllocators.datでのエラーの診断方法

この章では、ユーザーが作成したメモリ アロケータを実装する際に役立つ情報を提供します。

概要

DevPartner エラー検出では、ユーザーが作成したメモリ アロケータのメモリ分析を実行できます。これには、メモリ アロケータの記述を UserAllocators.dat というテキスト ファイルに追加します。このファイルは DevPartner エラー検出がインストールされているディレクトリの Data サブディレクトリにあります。ユーザーが作成したアロケータをこのファイルに追加すると、DevPartner エラー検出はこれらのアロケータを、オペレーティング システムで提供されているメモリ割り当てルーチンと同様に処理します。UserAllocators.dat に記述されている、ユーザーが作成したアロケータによって発生したリークが検出されると、ユーザーが作成したアロケータの下位レベルアロケータではなく、ユーザーが作成したアロケータが [検出されたプログラム エラー] ダイアログ ボックスに表示されます。

必要な情報の収集

ユーザーが作成したアロケータを `UserAllocators.dat` に追加する前に、以下の情報を収集する必要があります。

- 1 ユーザーが作成したアロケータの割り当て、解放、再割り当て、およびサイズ変更関数の正確な名前を指定します。
- 2 ユーザーが作成したメモリ アロケータが静的にアプリケーションにリンクされているか、別のモジュール (DLL) に提供されているかを調べます。
- 3 ユーザーが作成したアロケータを含んでいるモジュール (DLL) の名前を指定します。
- 4 ルーチンのパラメータを調べて、メモリ ブロックに関連付けられているブロックサイズとポインタがどのようにして渡されるか、または呼び出し側に返されるかを判断します。
- 5 アロケータによる特別な前提条件を調べます。これには、割り当て時にメモリをゼロにするなどの条件があります。ユーザーが作成したアロケータの場合は、解放されたブロックにデータを保存することなどです。

ユーザーが作成したアロケータの名前の検出

`UserAllocators.dat` にレコードを追加するには、割り当て、解放、再割り当て、およびサイズ関数の正確な名前を指定する必要があります。

以下の手順で、ルーチン名を検出します。

- 1 以下の関数の名前を確認します。
 - ◇ 割り当て関数 (`malloc`、`calloc`、`new` など)
 - ◇ 解放関数 (`free` または `delete`)
 - ◇ 再割り当て関数 (`realloc` または `recalloc`)
 - ◇ メモリ ブロック サイズ関数 (`_msize` など)
- 2 以下のどちらかを確認します。
 - ◇ 定義する関数を含むモジュールにシンボル (PDB ファイル) がある場合。PDB ファイルを使用できる場合、内部シンボルを使用できます。関数の短縮名または拡張名を検索するには、モジュールのコンパイル時にリンカ/デバッガのオプションを使用してマップ ファイルを作成します。その後、マップ ファイルの **Publics by Value** セクションを調べ、その関数の名前を判断します。このメソッドでは、常に `userAllocator` 関数定義において `Static` キーワードが必要です。

◇ シンボルがない、またはシンボルが無効である場合。

PDB ファイルを使用できない場合、Visual Studio コマンドプロンプトで `dumpbin /exports yourlibrary.dll` と入力します。出力に表示された関数名を使用します。

割り当て関数名は、使用する呼び出しルールと言語によって、短縮されているものと短縮されていないものがあります。C++を使用する場合、通常、名前は短縮されています。以下の小さいC++プログラムを見ていきましょう。

```
#include <malloc.h>
#include <memory.h>

class SampleClass
{
public:
    SampleClass() {}
    void *operator new( size_t stAllocateBlock );
    void operator delete( void * pBlock);
};
void *SampleClass::operator new( size_t stAllocateBlock )
{
    void *pvTemp = malloc( stAllocateBlock );
    if( pvTemp != 0 )
        memset( pvTemp, 0, stAllocateBlock );
    return pvTemp;
}
void SampleClass::operator delete(void * pBlock)
{
    free(pBlock);
    return;
}
int main(int argc, char * argv[])
{
    SampleClass *pClass = new SampleClass;
    return 0;
}
```

アプリケーションを作成する前に、Visual Studio のプロジェクトの設定で [マップ ファイルの生成] を選択します。アプリケーションの作成後、マップ ファイルを開いて、`operator new` と `operator delete` メソッドを検索します。これらは以下のようになります。

```
グローバル演算子 new:      ??2SampleClass@@SAPAXI@Z
グローバル演算子 delete:  ??3SampleClass@@SAXPAX@Z
```

これらの演算子は、以下のようにUserAllocators.datに記述することができます。

```
Allocator
  Module=myModule
  Function=??2SampleClass@@SAPAXI@Z
  MemoryType=MEM_CUSTOM1
  NumParams=1
  Size=1
  NoFill Static Debug
Deallocator
  Module=myModule
  Function=??3SampleClass@@SAXPAX@Z
  MemoryType=MEM_CUSTOM1
  NumParams=1
  Address=1
  Static Debug
```

ユーザーが作成したアロケータによるメモリに関する特別な前提条件

DevPartner エラー検出では、通常、ポインタをユーザー プログラムに返す前に割り当てられたメモリ ブロックに充てん文字をフィルし、解放されたあとにそのメモリ ブロックに無効データをフィルします。

メモリ アロケータがブロックを特殊データで初期化する場合、ブロックの上書きと特殊データの消失が発生しないようにNOFILLフラグを使用する必要があります。

ブロックが解放されたあと、そのブロックから読み込むことがメモリ アロケータの前提条件となっている場合、NOPOISONフラグを使用する必要があります。無効データをフィルしないことが必要なのは、以下のような場合です。

- ◆ メモリ アロケータが、割り当てステータスを追跡するために解放されたブロックにデータを保存し、解放された他のブロックにリンクする場合。
- ◆ ブロックが再割り当てされるまで、解放されたブロックを参照できることがアプリケーションの前提条件となっている場合。解放されたブロックを継続して参照しようとすることは危険ですが、多数のアプリケーションでまだ使用されています。

UserAllocators.datのエントリの作成

「必要な情報の収集」(50ページ)で説明されている情報を収集したら、UserAllocators.datにアロケータを記述することができます。

関数を記述するには、1つの関数につき1つのレコードをこのファイルに作成します。各レコードの構文は以下のとおりです。

```
Record_Type Parameter_1 Parameter_2 Etc...
```

*Record_Type*では、作成する関数の種類を指定します。続くパラメータでは、アロケータ関数に関する追加情報を指定します。

レコードを作成するとき、各フィールドを1つまたは複数の空白文字またはタブで区切ります。レコードには、複数の行を指定できます。

表 4-1に現在定義されているレコードタイプを示します。

表 4-1. レコードタイプ

レコードタイプ	説明
Allocator	メモリを割り当てる関数。
Deallocator	メモリを解放する関数。
QuerySize	以前にアロケータ関数によって割り当てられていたメモリブロックのサイズを照会できる関数。
Reallocator	以前にアロケータ関数によって割り当てられていたメモリブロックのサイズを調整できる関数。 リアロケータ関数はいつも同じメモリブロックを返すとは限りません。
Ignore	DevPartnerエラー検出に無視させる(メモリを追跡させない)アロケータまたはデアロケータ関数。

モジュール

各UserAllocatorのレコードタイプには、記述される関数を含むモジュールを指定する必要があります。モジュールには3種類あります。

表 4-2. モジュールタイプ

モジュールタイプ	説明
名前付きモジュール	ユーザー割り当て関数またはメソッドを含む、明示的な名前が付けられているモジュール(DLL)(foo.dllなど) メモ :モジュール名にワイルドカードを使用することはできません。

表 4-2. モジュール タイプ (続き)

モジュール タイプ	説明
<p>静的にリンクされた ユーザー アロケータ</p>	<p>ユーザー割り当て関数またはメソッドを含む、明示的な名前が付けられているモジュール (DLL または実行可能ファイル)。ただし、この場合、関数またはメソッドは本来ライブラリ (.lib file) の一部です。いったんモジュールにリンクすると、カスタマ コードで関数またはメソッドを参照できますが、外部からはアクセスできません。モジュールのデバッグ シンボルを指定し、オプションの STATIC キーワードを使用して、そのデバッグ シンボル内で関数またはメソッドを検索するように、DevPartner エラー検出に警告します。</p> <p>メモ：レコードのオプションパラメータに STATIC キーワードを含めないと、DevPartner エラー検出はユーザーが作成した割り当て関数やメソッドを正しく監視できません。</p>
<p>*CRT</p>	<p>関数がアプリケーション内のどこにある場合でも、その関数を参照できる特殊なモジュールです。</p> <p>メモ：*CRT の * はワイルドカード文字ではありません。</p> <p>*CRT は、以下の 3 つの場合に適用されます。</p> <ul style="list-style-type: none"> • Microsoft C ランタイム ライブラリ • 静的にリンクされている C ランタイム ライブラリ • パッチ対象と同じ短縮名を持つユーザー関数 (たとえば、<code>global operator new</code>)。

アロケータ レコード

メモリを割り当てる関数を記述するアロケータ レコードを作成します。以下の形式で指定します。

```
Allocator Module=module_name Function=func_name  
MemoryType=mem_type NumParams=param_num Size=size_value  
[Count=count_num] [BufferLoc=buffer_loc] [Optional  
Parameters]
```

メモ： デフォルトでは、割り当てられたブロックのアドレスが指定した関数の戻り値となります。この動作を上書きするには、割り当てられたブロックのアドレスがパラメータに返されることを示す BufferLoc パラメータを指定します (UserAllocators.dat ファイルの MAPIAllocateBuffer を参照)。

DevPartner エラー検出では、コール後に、割り当てられたブロックのアドレスがあると予期される場所が NULL の場合 (戻り値、または BufferLoc で指定されたパラメータの値)、割り当てが失敗したとみなされます。

表 4-3. アロケータ レコード パラメータ

パラメータ	説明
Allocator	レコードの最初のパラメータは Allocator にして、割り当てルーチンの記述であることを示す必要があります。
module_name	ユーザーが作成したアロケータを含むモジュール (実行可能ファイルや DLL) の名前。 メモ： モジュール名にワイルドカードを使用することはできません。
func_name	ユーザーが作成したアロケータにある、メモリ ブロックを割り当てる関数名。C++ を使用している場合、これはその関数の「短縮」名となります。 このパラメータでは、大文字と小文字が区別されます。

表 4-3. アロケータ レコード パラメータ (続き)

パラメータ	説明
mem_type	<p>このパラメータでは、割り当てるメモリの種類を記述します。DevPartner エラー検出では、現在、以下の種類のメモリが定義されています。</p> <ul style="list-style-type: none"> MEM_MALLOC malloc、calloc、strdup などのルーチンから返されるメモリブロック。この種類のメモリは、C ランタイム ライブラリ解放ルーチンのようなルーチンを使用して解放されます。 MEM_NEW 演算子 new によって返され、演算子 delete によって解放されるメモリ ブロック。 MEM_CUSTOM1 ~ MEM_CUSTOM9 特定のデアロケータと組み合わせる必要があるメモリブロック。この種類のメモリでは、開発者は、上記の標準メモリアロケータと相互に影響を及ぼさない独自のカスタムメモリアロケータを宣言できます。 <p>DevPartner エラー検出は、ある種類のメモリで割り当てられたメモリ ブロックが同じ種類の関数で解放されることを確認します。メモリの種類が一致しない場合、実行時にメモリ不一致エラーが表示されます。</p>
param_num	<p>関数に渡されるパラメータの数。この値は 1 ~ 32 でなければなりません。</p> <p>ユーザーが作成したアロケータ関数の記述に使用するパラメータの数を指定します。正しい数値を指定しないと、予期しない結果になる場合があります。</p>
size_value	<p>割り当てるブロックのサイズを含むパラメータの番号。最初のパラメータ番号は 1 です。</p>
count_num	<p>このオプション パラメータは、size パラメータおよび count パラメータを受け入れる calloc に類似した関数の記述に使用します。このパラメータを指定する場合は、指定したサイズのメモリ ブロックをいくつ割り当てるかを示します。このパラメータを指定しない場合、DevPartner エラー検出では、カウントは常に 1 であるとみなされます。</p>
buffer_loc	<p>割り当てられたブロックのアドレスを持つアドレスを含むパラメータの番号。最初のパラメータ番号は 1 です。</p>

表 4-3. アロケータ レコード パラメータ (続き)

パラメータ	説明
[Optional Parameters]	<p>レコードの終わりに以下のオプションパラメータを含めることができます。</p> <ul style="list-style-type: none"> • DEBUG このパラメータを指定すると、フックに関する追加情報が表示されます。出力ウィンドウ (または dbgview) には、フックしようとしたときに発生したエラーに関する情報が表示されます。[通知情報] タブには、関数が正常にフックされたかどうかや、フックされた関数のコール回数など、フック別の統計が表示されます。 • NODISPLAY このパラメータを指定すると、要求された各フックに関する詳細情報が [通知情報] タブの上部に表示されません。 • NOFILL このパラメータを指定すると、返されるバッファが充てん文字でフィルされません。 メモ : calloc と同様に、ユーザーが作成したアロケータがブロックをデータで初期化する場合は、NOFILL を指定して、そのデータが破壊されないようにします。 • NOGUARD このパラメータを指定すると、割り当て関数によって作成されたブロックの最後に保護バイトが追加されません。 • STATIC ユーザーが作成したアロケータが静的にパッチされます。ユーザーが作成したアロケータがアプリケーションにリンクされており、エクスポートされたインターフェイスを含む別個の DLL に提供されていない場合は、STATIC オプションを指定します。

アロケータ レコードの例

以下に、架空のアロケータ レコード関数の例を示します。

例 1

この例では、関数 `mallocXX` が `MyAlloc.dll` というライブラリにあります。この関数はタイプ `malloc` の演算子で、1つのパラメータを持ち、最初のパラメータでサイズを渡します。`DevPartner` エラー検出は、アプリケーションに返す前にメモリ ブロックをフィルできません。この関数によって割り当てられたブロックは、任意のタイプ `MEM_MALLOC` の関数で解放できます。

```
Allocator Module=MyAlloc.dll Funtion=mallocXX
MemoryType=MEM_MALLOC NumParams=1 Size=1 NOFILL
```

例 2

この例は、`Microsoft iostream` コード内のカスタム グローバル演算子 `new` の追跡に使用されているファイルから抽出したものです。この関数は `C` ランタイム ライブラリにあります。このレコードは、モジュール名として `*CRT` を指定しているため、この関数が `Microsoft C` または `C++` ランタイム ライブラリの1つにあるものとみなされます。

この関数は、4つのパラメータを持ち、最初のパラメータにサイズが保存されています。`DevPartner` エラー検出では、メモリを要求しているアプリケーションに返す前にそのメモリ ブロックをフィルできます。

```
Allocator Module=*CRT Function=??2@YAPAXIHDPBDH@Z
MemoryType=MEM_NEW NumParams=4 Size=1
```

例 3

この例は、1つのパラメータを持ち、最初のパラメータでサイズが渡される、`CustomAllocXX` という関数です。

`DevPartner` エラー検出は、アプリケーションに返す前にバッファをフィルできません。このレコードでは、`MemoryType` として `MEM_CUSTOM1` が指定されています。この関数で割り当てられたメモリが、メモリ タイプ `MEM_CUSTOM1` のルーチンによって解放されることが確認されます。他の解放ルーチンを使用すると、メモリ解放後にメモリ解放関数の不一致メッセージが表示されます。

```
Allocator Module=foo.dll Function=CustomAllocXX
MemoryType=MEM_CUSTOM1 NumParams=1 Size=1 NOFILL
```

例 4

この例では、`MyAlloc` という関数が `.LIB` ファイルとして作成され、`DataStore.dll` というデータ収集コンポーネントに静的にリンクされています。`MyAlloc` では4つのパラメータを指定できます。最初のパラメータはデータ レコードのサイズで、2番目のパラメータは1つのブロックに割り当てられるレコードの数です。3番目のパラメータには、割り当てられたブロックのアドレスがある場所のアドレスが含まれます。アプリケーションから取得したメモリは事前に初期化されているため、`DevPartner` エラー検出はブロックをフィルできません。

```
Allocator Module=DataStore.dll Function=MyAlloc BufferLoc=3
MemoryType=MEM_MALLOC NumParams=4 Size=1 Count=2
NOFILL STATIC
```

メモ： 関数名が `DLL` からエクスポートされない場合は、`STATIC` キーワードを指定する必要があります。

デアロケータ レコード

メモリを解放する関数を記述するデアロケータ レコードを作成します。以下の形式で指定します。

```
Deallocator Module=module_name Function=func_name  
MemoryType=mem_type NumParams=param_num  
Address=address_value [Optional Parameters]
```

表 4-4. デアロケータ レコード パラメータ

パラメータ	説明
Deallocator	レコードの最初のパラメータは Deallocator にして、割り当てルーチンの記述であることを示す必要があります。
module_name	ユーザーが作成したアロケータを含むモジュール（実行可能ファイルやDLL）の名前。 メモ ：モジュール名にワイルドカードを使用することはできません。
func_name	ユーザーが作成したアロケータにある、メモリ ブロックを解放する関数名。C++ を使用している場合、これはその関数の「短縮」名となります。 このパラメータでは、大文字と小文字が区別されます。
mem_type	このパラメータでは、解放するメモリの種類を記述します。DevPartner エラー検出では、現在、以下の種類のメモリが定義されています。 <ul style="list-style-type: none">MEM_MALLOC malloc、calloc、strdup などのルーチンから返されるメモリ ブロックを記述します。この種類のメモリは、C ランタイム ライブラリ解放ルーチンのようなルーチンを使用して解放されます。MEM_NEW 演算子 new によって返され、演算子 delete によって解放されるメモリ ブロックを記述します。MEM_CUSTOM1 ~ MEM_CUSTOM9 特定のデアロケータと組み合わせる必要があるメモリ ブロックを記述します。この種類のメモリでは、開発者は、上記の標準メモリ アロケータと相互に影響を及ぼさない独自のカスタム メモリ アロケータを宣言できます。 ある種類のメモリで割り当てられたメモリ ブロックが同じ種類の関数で解放されることが確認されます。メモリの種類が一致しない場合は、実行時にメモリ不一致エラーが表示されます。
param_num	関数に渡されるパラメータの数。この値は 1 ~ 32 でなければなりません。 ユーザーが作成したアロケータ関数の記述に使用するパラメータの数を指定します。正しい数値を指定しないと、予期しない結果になる場合があります。

表 4-4. デアロケータ レコード パラメータ (続き)

パラメータ	説明
address_value	解放するブロックへのポインタを含むパラメータの番号。最初のパラメータ番号は1です。
[Optional Parameters]	<p>以下のオプション パラメータをレコードの終わりに含めることができます。</p> <ul style="list-style-type: none"> • DEBUG このパラメータを指定すると、フックに関する追加情報が表示されます。出力ウィンドウ (または dbgview) には、フックしようとしたときに発生したエラーに関する情報が表示されます。[通知情報] タブには、関数が正常にフックされたかどうかや、フックされた関数のコール回数など、フック別の統計が表示されます。 • NODISPLAY このパラメータを指定すると、要求された各フックに関する詳細情報が [通知情報] タブの上部に表示されません。 • NOPOISON NOPOISON を指定すると、DevPartner エラー検出は解放されたメモリ ブロックに無効データをフィルしません。 ユーザーが作成したアロケータがブロックの解放後そのブロックにデータを保存したり、アプリケーションが解放されたブロックのデータを使い続ける場合、NOPOISON を指定してデータが破壊されないようにします。 • STATIC ユーザーが作成したアロケータが静的にバッチされます。ユーザーが作成したアロケータがアプリケーションにリンクされており、エクスポートされたインターフェイスを含む別個の DLL に提供されていない場合は、STATIC オプションを指定します。

DevPartner エラー検出は、メモリ解放関数からの戻り値をチェックしません。

デアロケータ レコードの例

以下に、架空のデアロケータ レコードの例を示します。

例 1

この例では、関数 `freeXX` が `MyAlloc.dll` というライブラリにあります。この関数は、1つのパラメータを持ち、解放するブロックのポインタが最初のパラメータで渡されます。**DevPartner** エラー検出は、アプリケーションに返す前に無効データをフィルできません。

```
Deallocator Module=MyAlloc.dll Function=freeXX
MemoryType=MEM_MALLOC NumParams=1 Address=1 NOPOISON
```

例 2

この例は、`foo.dll` にある `MyFree` という関数です。この関数は、1つのパラメータを持ち、解放するブロックのポインタが最初のパラメータで渡されます。**DevPartner** エラー検出は、アプリケーションに返す前にメモリに無効データをフィルします。ブロックが解放されると、ブロックがタイプ `MEM_CUSTOM1` のアロケータで割り当てられたことが確認されます。ブロックがこのグループのものでない場合、エラーが発生します。

```
Deallocator Module=foo.dll Funtion=MyFree
MemoryType=MEM_CUSTOM1 NumParams=1 Address=1
```

例 3

この例では、`MyFree` という関数が `.LIB` ファイルとして作成され、`DataStore.dll` というデータ収集コンポーネントに静的にリンクされています。`MyFree` では3つのパラメータを指定できます。最初と最後のパラメータは **DevPartner** エラー検出には関係ありません。2番目のパラメータには解放するメモリ ブロックのアドレスを指定します。また、プライベート メモリ割り当てルーチンでは、メモリ ブロックが解放されたあと、その中の秘密情報は保持されます。

```
Deallocator Module=DataStore.dll Function=MyFree
MemoryType=MEM_MALLOC NumParams=3 Address=2 NOPOISON
STATIC
```

メモ： 関数名が DLL からエクスポートされない場合は `STATIC` を指定します。

クエリサイズ レコード

クエリサイズ レコードを作成し、割り当てられたメモリ ブロックのサイズを返す関数を記述します。以下の形式で指定します。

```
QuerySize Module=module_name Function=func_name
          MemoryType=mem_type NumParams=param_num
          Address=address_value [Optional Parameters]
```

メモ： ユーザーが作成したアロケータのクエリサイズ レコードを省略すると、その関数に誤ったブロック サイズが返されます。

表 4-5. クエリサイズ レコード

パラメータ	説明
QuerySize	レコードの最初のパラメータはQuerySizeにして、サイズルーチンの記述であることを示す必要があります。
module_name	ユーザーが作成したアロケータを含むモジュール（実行可能ファイルやDLL）の名前。 メモ： モジュール名にワイルドカードを使用することはできません。
func_name	ユーザーが作成したアロケータにある、割り当てられたメモリブロックのサイズを返す関数名。C++を使用している場合、これはその関数の「短縮」名となります。 このパラメータでは、大文字と小文字が区別されます。
mem_type	このパラメータでは、照会するメモリの種類を記述します。DevPartnerエラー検出では、現在、以下の種類のメモリが定義されています。 <ul style="list-style-type: none">MEM_MALLOC malloc、calloc、strdupなどのルーチンから返されるメモリブロックを記述します。この種類のメモリは、Cランタイムライブラリ解放ルーチンのようなルーチンを使用して解放されます。MEM_NEW 演算子newによって返され、演算子deleteによって解放されるメモリ ブロックを記述します。MEM_CUSTOM1～MEM_CUSTOM9 特定のデアロケータと組み合わせる必要があるメモリブロックを記述します。この種類のメモリでは、開発者は、上記の標準メモリ アロケータと相互に影響を及ぼさない独自のカスタム メモリ アロケータを宣言できます。 ある種類のメモリで割り当てられたメモリ ブロックが同じ種類の関数で解放されることが確認されます。メモリの種類が一致しない場合は、実行時にメモリ不一致エラーが表示されます。

表 4-5. クエリサイズ レコード (続き)

パラメータ	説明
param_num	関数に渡されるパラメータの数。この値は1～32でなければなりません。 ユーザーが作成したアロケータ関数の記述に使用するパラメータの数を指定します。正しい数値を指定しないと、予期しない結果になる場合があります。
address_value	照会するブロックへのポインタを含むパラメータの番号。最初のパラメータ番号は1です。
[Optional Parameters]	以下のオプションパラメータをレコードの終わりに含めることができます。 <ul style="list-style-type: none"> • DEBUG このパラメータを指定すると、フックに関する追加情報が表示されます。出力ウィンドウ (または dbgview) には、フックしようとしたときに発生したエラーに関する情報が表示されます。[通知情報] タブには、関数が正常にフックされたかどうかや、フックされた関数のコール回数など、フック別の統計が表示されます。 • NODISPLAY このパラメータを指定すると、要求された各フックに関する詳細情報が[通知情報]タブの上部に表示されません。 • STATIC ユーザーが作成したアロケータが静的にパッチされます。ユーザーが作成したアロケータがアプリケーションにリンクされており、エクスポートされたインターフェイスを含む別個のDLLに提供されていない場合は、STATICオプションを指定します。

この関数からの戻り値は、ブロック サイズ `size_t` であるとみなされます。

クエリサイズ レコードの例

以下に、架空のクエリサイズ レコードの例を示します。

例 1

この例では、関数 `MySize` が `foo.dll` というライブラリにあります。この関数は1つのパラメータを持ち、最初のパラメータで照会されるメモリ ブロックのポインタが指定されます。

```
QuerySize Module=foo.dll Function=MySize
      MemoryType=Mem_Custom1 NumParams=1
      Address=1
```

例 2

この例では、関数 `MySize` は `DataStore.dll` というデータ収集コンポーネントに静的にリンクされています。関数 `MySize` は2つのパラメータを持ち、照会するアドレスが最初のパラメータで渡されます。

```
QuerySize Module=DataStore.dll Function=MySize
MemoryType=MEM_NEW NumParams=2
Address=1 STATIC
```

メモ： 関数名がDLLからエクスポートされない場合はSTATICを指定します。

リアロケータ レコード

メモリを再割り当てする関数を記述するリアロケータ レコードを作成します。以下の形式で指定します。

```
Reallocator Module=module_name Function=func_name
MemoryType=mem_type NumParams=param_num
Address=address_value Size=size_value
[Count=count_num] [BufferLoc=buffer_loc] [Optional
Parameters]
```

メモ： デフォルトでは、割り当てられたブロックのアドレスが指定した関数の戻り値となります。この動作を上書きするには、割り当てられたブロックのアドレスがパラメータに返されることを示すBufferLocパラメータを指定します (UserAllocators.dat ファイルのMAPIAllocateBufferを参照)。

表 4-6. リアロケータ レコード パラメータ

パラメータ	説明
Reallocator	レコードの最初のパラメータはReallocatorにして、再割り当てルーチンの記述であることを示す必要があります。
module_name	ユーザーが作成したアロケータを含むモジュール（実行可能ファイルやDLL）の名前。 メモ： モジュール名にワイルドカードを使用することはできません。
func_name	ユーザーが作成したアロケータにある、メモリ ブロックを再割り当てする関数名。C++を使用している場合、これはその関数の「短縮」名となります。 このパラメータでは、大文字と小文字が区別されます。

表 4-6. リアロケータ レコードパラメータ (続き)

パラメータ	説明
mem_type	<p>このパラメータでは、再割り当てするメモリの種類を記述します。DevPartner エラー検出では、現在、以下の種類のメモリが定義されています。</p> <ul style="list-style-type: none"> MEM_MALLOC malloc、calloc、strdup などのルーチンから返されるメモリブロックを記述します。この種類のメモリは、C ランタイムライブラリ解放ルーチンのようなルーチンを使用して解放されます。 MEM_NEW 演算子 new によって返され、演算子 delete によって解放されるメモリ ブロックを記述します。 MEM_CUSTOM1 ~ MEM_CUSTOM9 特定のデアロケータと組み合わせる必要があるメモリ ブロックを記述します。この種類のメモリでは、開発者は、上記の標準メモリ アロケータと相互に影響を及ぼさない独自のカスタム メモリ アロケータを宣言できます。 <p>ある種類のメモリで割り当てられたメモリ ブロックが同じ種類の関数で解放されることが確認されます。メモリの種類が一致しない場合は、実行時にメモリ不一致エラーが表示されます。</p>
param_num	<p>関数に渡されるパラメータの数。この値は 1 ~ 32 でなければなりません。</p> <p>ユーザーが作成したアロケータ関数の記述に使用するパラメータの数を指定します。正しい数値を指定しないと、予期しない結果になる場合があります。</p>
address_value	<p>再割り当てするブロックへのポインタを含むパラメータの番号。最初のパラメータ番号は 1 です。</p>
size_value	<p>再割り当てするブロックのサイズを含むパラメータの番号。最初のパラメータ番号は 1 です。</p>
count_num	<p>このオプションパラメータは、size パラメータおよび count パラメータを受け入れる calloc に類似した関数の記述に使用します。このパラメータを指定する場合は、指定したサイズのメモリ ブロックをいくつ割り当てるかを示します。このパラメータを指定しない場合、DevPartner エラー検出では、カウントは常に 1 であるとみなされます。</p>
buffer_loc	<p>再度割り当てられたブロックのアドレスを持つアドレスを含むパラメータの番号。最初のパラメータ番号は 1 です。</p>

表 4-6. リアロケータ レコード パラメータ (続き)

パラメータ	説明
[Optional Parameters]	<p>以下のオプション パラメータをレコードの終わりに含めることができます。</p> <ul style="list-style-type: none"> DEBUG このパラメータを指定すると、フックに関する追加情報が表示されます。出力ウィンドウ (または dbgview) には、フックしようとしたときに発生したエラーに関する情報が表示されます。[通知情報] タブには、関数が正常にフックされたかどうかや、フックされた関数のコール回数など、フック別の統計が表示されます。 NODISPLAY このパラメータを指定すると、要求された各フックに関する詳細情報が [通知情報] タブの上部に表示されません。 NOFILL このパラメータを指定すると、DevPartner エラー検出は以前に割り当てられたメモリ ブロックの最後に追加されたバイトに、充てん文字をフィルしません。 メモ : calloc と同様に、ユーザーが作成したアロケータがブロックをデータで初期化する場合は、NOFILL を指定して、そのデータが破壊されないようにします。 NOGUARD このパラメータを指定すると、割り当て関数によって作成されたブロックの最後に保護バイトが追加されません。 STATIC ユーザーが作成したアロケータが静的にパッチされます。ユーザーが作成したアロケータがアプリケーションにリンクされており、エクスポートされたインターフェイスを含む別個の DLL に提供されていない場合は、STATIC オプションを指定します。

DevPartner エラー検出は、再割り当て関数からの戻り値をチェックし、NULL 値が返された場合はエラーとみなします。NULL でないアドレスは、新しく割り当てられたメモリ ブロックのアドレスとみなされます。

リアロケータ レコードの例

以下に、架空のリアロケータ レコードの例を示します。

例 1

foo.dll というモジュールで宣言された reallocXX という関数です。この関数では 2 つのパラメータを指定できます。最初のパラメータは既存のメモリ ブロックのアドレスで、2 番目のパラメータは要求されたブロックのサイズです。オプションパラメータが指定されていないので、DevPartner エラー検出は、アプリケーションに制御を

戻す前に、新しいメモリ ブロックすべてに充てんパターンをフィルします (新しいメモリ ブロックの方が大きいとします)。

```
Reallocator Module=foo.dll Function=reallocXX
MemoryType=MEM_MALLOC NumParams=2 Address=1 Size=2
```

例 2

foo.dll というモジュールで宣言された reallocClear という関数です。この関数では 3 つのパラメータを指定できます。最初のパラメータは既存のメモリ ブロックのアドレスで、3 番目のパラメータは要求されたブロックのサイズです。この再割り当てルーチンは新しいメモリ ブロックに割り当てられた追加のメモリに対し独自の充てんを行うので、DevPartner エラー検出は新しいブロックに追加メモリをフィルできません。

メモ： パラメータ 2 の内容は DevPartner エラー検出に関係ないので、無視されます。

```
Reallocator Module=foo.dll Function=reallocClear
MemoryType=MEM_MALLOC NumParams=3 Address=1 Size=3
NOFILL
```

例 3

MyRealloc という関数は .LIB ファイル内で作成され、DataStore.dll というデータ収集コンポーネントに静的にリンクされています。MyRealloc では、4 つのパラメータを使用できます。最初と 4 番目のパラメータは DevPartner エラー検出には関係ありません。2 番目のパラメータには既存のブロックのアドレスが含まれ、3 番目のパラメータにはブロックの新しいサイズが含まれます。データ収集ルーチンは、再割り当て時、新しいデータをこのブロックに事前にロードします。

```
Reallocator Module=DataStore.dll Function=MyRealloc
MemoryType=MEM_ALLOC NumParams=4 Address=2 Size=3
NOFILL STATIC
```

メモ： 関数名が DLL からエクスポートされない場合は STATIC を指定します。

無視レコード

無視レコードを作成し、DevPartner エラー検出メモリ追跡システムが無視すべき関数を記述します。以下の形式で指定します。

```
Ignore Module=module_name Function=func_name [Optional Parameters]
```

無視レコードを使用して、DevPartner エラー検出にユーザーが作成したアロケータを無視するか、ユーザーが作成したアロケータで使用される下位レベル アクセスルーチンを無視するように指示します。無視レコードは、DevPartner エラー検出メモリ追跡システムに、通常は監視される API を追跡しないように指示します。

注意： 無視レコードを作成すると、DevPartner エラー検出メモリ追跡システムはこれらの API から割り当てられた、または解放されたメモリを追跡しません。このため、DevPartner エラー検出はこのメモリを認識しなくなります。このため、コールバリデーションおよび FinalCheck 分析モジュールは正しくないか完全でないエラーメッセージを表示します。この機能の使用についての質問がある場合は、テクニカル サポートまでお問い合わせください。

表 4-7. 無視レコードのパラメータ

パラメータ	説明
Ignore	レコードの最初のパラメータはIgnoreにして、無視するAPIを記述することを示す必要があります。
module_name	無視する関数を含むモジュール（実行可能ファイルやDLL）の名前。 メモ ：モジュール名にワイルドカードを使用することはできません。
func_name	メモリ追跡システムで無視する関数名。C++を使用している場合、これはその関数の「短縮」名となります。 このパラメータでは、大文字と小文字が区別されます。
[Optional Parameters]	<ul style="list-style-type: none"> • DEBUG このパラメータを指定すると、フックに関する追加情報が表示されます。出力ウィンドウ（またはdbgview）には、フックしようとしたときに発生したエラーに関する情報が表示されます。[通知情報]タブには、関数が正常にフックされたかどうかや、フックされた関数のコール回数など、フック別の統計が表示されます。 • NODISPLAY このパラメータを指定すると、要求された各フックに関する詳細情報が[通知情報]タブの上部に表示されません。 • STATIC ユーザーが作成したアロケータが静的にパッチされます。ユーザーが作成したアロケータがアプリケーションにリンクされており、エクスポートされたインターフェイスを含む別個のDLLに提供されていない場合は、STATICオプションを指定します。

無視レコードのサンプル

以下の例では、架空の無視レコードを示します。

例 1

この例では、DevPartner エラー検出にGlobalAllocによって割り当てられたメモリを監視させるが、GlobalFreeを使用してオペレーティングシステムに返されたメモリを解放する要求はチェックさせない無視レコードを作成します。

メモ： これによって、DevPartner エラー検出は多数のメモリ リークを誤検出してレポートします。

```
Ignore Module=Kernel32.dll Function=GlobalFree
```

例 2

この例では、DevPartner エラー検出に、GlobalAllocファミリのコールによって操作されるメモリを無視するように指示します。

メモ： これによって、DevPartner エラー検出は多数のコールバリデーションおよびFinalCheckエラーを誤検出してレポートするようになります。

```
Ignore Module=Kernel32.dll Function=GlobalAlloc
Ignore Module=Kernel32.dll Function=GlobalReAlloc
Ignore Module=Kernel32.dll Function=GlobalFree
```

メモ： これら3行をUserAllocators.datに追加することは推奨できません。

例3

この例では、指定されたモジュール内で静的にリンクされている関数によって操作されているメモリを無視するように、DevPartnerエラー検出に指示します。独自の代替メモリ割り当てライブラリを作成して標準Cランタイムライブラリと同じ名前を使用し、DevPartnerエラー検出にライブラリの使用状況を監視させない場合は、以下のような行を追加する必要があります。

```
Ignore Module=MyDLL.dll Function=Malloc STATIC
Ignore Module=MyDLL.dll Function=free STATIC
Ignore Module=MyDLL.dll Function=realloc STATIC
```

メモ： UserAllocators.datファイルにこのような行を追加する場合は、事前にテクニカルサポートにお問い合わせください。

UserAllocator フック要求のコーディング

UserAllocator フック要求を作成する場合には、以下の重要事項に留意してください。

- ◆ MEM_CUSTOM1～MEM_CUSTOM9を使用して、システムの割り当てから独自のメモリ割り当てを分離します。
- ◆ アロケータタイプごとに固有のカスタムアロケータタイプを指定します。
- ◆ 常にアロケータに対応するアロケータタイプを使用して、メモリを解放します。たとえば、アロケータがMEM_CUSTOM1タイプでフックされた場合、同様にMEM_CUSTOM1タイプでフックされるデアロケータを使用してメモリを解放する必要があります。デアロケータのタイプが異なると、割り当ての競合エラーがレポートされます。

UserAllocator のコード要件

UserAllocatorメモリ割り当てのフックを使用するには、メモリの割り当てと解放を制御する関数をアプリケーションコードに含める必要があります。

アロケータ関数のフック

- ◆ コールするアロケータ関数には、要求メモリのバイト数を指定するパラメータを含める必要があります。
- ◆ アロケータ関数は、割り当てられたメモリの場所を返すか、または場所を示すアドレスの入ったパラメータを含む必要があります。
- ◆ アロケータ関数には、他のパラメータも含めることができます。

例

```
void *GetMemory(int BytesRequested);  
Allocator  
    Module=bcheap.dll  
    Function=GetMemory  
    MemoryType=MEM_CUSTOM1  
    NumParams=1  
    Size=1  
    Static  
    Noguard  
    NoFill  
    Debug
```

または

```
void *pVoid;  
HRESULT GetMemoryAgain(int BytesRequested, &pVoid);  
// in the above example, the memory allocator will place the  
// location of the allocated memory in the pVoid pointer.  
Allocator  
    Module=bcheap.dll  
    Function=GetMemoryAgain  
    MemoryType=MEM_CUSTOM1  
    NumParams=2  
    Size=1  
    BufferLoc=2  
    Static  
    Noguard  
    NoFill  
    Debug
```

デアロケータ関数のフック

- ◆ コールするデアロケータ関数には、解放メモリのアドレスを指定するパラメータを含める必要があります。
- ◆ アロケータ関数には、他のパラメータも含めることができます。

例

```
; static void *mark_free(void *p, int nLen, void *pExclude, int  
nExcludeLen)
```

Deallocator

```
Module=bcheap.dll  
Function=mark_free  
MemoryType=MEM_CUSTOM1  
NumParams=4  
Address=1  
Static  
NoPoison  
Debug
```

リアロケータ関数のフック

- ◆ コールするリアロケータ関数には、要求メモリのバイト数を指定するパラメータを含める必要があります。
- ◆ リアロケータ関数には、再割り当てする「旧」メモリのアドレスを指定するパラメータを含める必要があります。
- ◆ リアロケータ関数は、割り当てられた「新しい」メモリの場所を返すか、または場所を示すアドレスを含む必要があります。
- ◆ リアロケータ関数には、他のパラメータを含めることもできます。

例

```
; void *DoMyRealloc(MyHeap *me, void *p, uint32 uSize)
```

ReAllocator

```
Module=bcheap.dll  
Function=DoMyRealloc  
MemoryType=MEM_CUSTOM1  
NumParams=3  
Address=2  
Size=3  
NoFill  
NoGuard  
Static  
Debug
```

UserAllocator フックのデバッグ

エラー検出では、**UserAllocator** フックに関する情報表示の変更に使用できる2つのキーワードが提供されます。これにより、**UserAllocator** フックのデバッグが容易になります。

NoDisplay

デフォルトでは、ファイル要求から解釈されたフックの詳細が [通知情報] タブの上部に表示されます。

例：

```
Allocator Module=bcheap.dll Function=MarkNodeAllocated  
MemoryType=MEM_CUSTOM1 NumParams=3 Size=1 BufferLoc=3 NoFill  
NoGuard Static Debug
```

フックの詳細情報を [通知情報] タブの上部に表示しない場合、**NoDisplay** キーワードをフック要求に追加します。

Debug

エラー検出では、**Debug** キーワードも提供されます。このキーワードをフック要求に追加すると、詳細が表示されます。

[通知情報] タブ

フック要求に **Debug** キーワードを追加すると、エラー検出実行の完了後に詳細が [通知情報] タブの下部に表示されます。[通知情報] タブの下部に表示される詳細には、関数が正常にフックされたかどうか、フックされた関数のコール回数などの各フックの統計が含まれます。

エラー詳細

[通知情報] タブの下部に関数がフックされなかったことが示された場合、エラー検出では、失敗したフックのデバッグをサポートする詳細が以下のように表示されます。

Visual Studio 内で使用している場合には、出力ウィンドウにエラー詳細が表示されます。

エラー検出のスタンドアロンバージョンを実行している場合には、**DbgView** ツールによってエラー詳細が表示されます。

UserAllocators.datでのエラーの診断方法

UserAllocators.datにレコードを追加すると、1つまたは複数の以下のようなエラーが表示されることがあります。

- ◆ ファイル アクセス エラー

UserAllocators.datは、DevPartnerエラー検出がインストールされているディレクトリのDataサブディレクトリに保存されているテキストファイルです。ファイルが削除されるか読み取り不可能になった場合、DevPartnerエラー検出はエラーをレポートします。

- ◆ ファイル書き込みエラー

DevPartnerエラー検出セッションを開始すると、UserAllocators.nlbという名前のファイルが作成されます。**【オプション】>【データ】>【NLBファイルディレクトリ】**で指定した場所が無効または読み取り専用である場合、エラーが発生します。このエラーを解決するには、**【オプション】**のディレクトリ指定を変更するか、またはディレクトリのプロパティを書き込み可能に変更します。

- ◆ 解析エラー

UserAllocators.datファイルの解析中にエラーが発生した場合、DevPartnerエラー検出は**【エラー】**タブにエラーを記録します。UserAllocators.datエラーが発生したときに、メモリの追跡またはリソースの追跡が有効になっていた場合は無効にされます。

トークン解析エラー

DevPartnerエラー検出は、以下のルールを使用して、ファイルを1行ずつ解析します。

- ◆ 空白行とセミコロンで始まる行は無視されます。
- ◆ 追加されるUserAllocatorの定義はすべて、有効なレコードタイプで開始しなければなりません。
- ◆ 各定義のパラメータはすべて、1つまたは複数空白文字またはタブで分離し、各レコードタイプのルールに従う必要があります。

意味的エラー

各レコードタイプは、各パラメータのルールに従って解析されます。パラメータの中には大文字と小文字が区別されるものがあり、有効範囲が限定されているものもあります（たとえば、1つの関数で使用できるパラメータの最大数は32です）。

同じファイル内に重複エントリが存在し、レコードが競合する場合にもエラーが発生します。UserAllocators.datは高度な機能であるため、綿密なチェックは実行されません。

UserAllocators.datを変更したあと、アプリケーションが不安定になる場合

UserAllocators.datにレコードを追加すると、DevPartnerエラー検出にユーザーが作成したアロケータとの間のコールを監視するように指示することになります。DevPartnerエラー検出にAPIを正しく記述しなかった場合、アプリケーションはクラッシュするか、予期しない動作をすることがあります。そのような問題の最も一般的な原因の1つとして、関数にパラメータの数を誤って指定した場合があります。

また、メモリ残量の定数値に依存した場合や、NOFILLやNOPOSITIONを記述に追加しなかった場合にも問題が発生することがあります。

エラーが発生し、対処方法がわからない場合は、テクニカルサポートにお問い合わせください。テクニカルサポートにお問い合わせの際には、以下の情報をお知らせください。

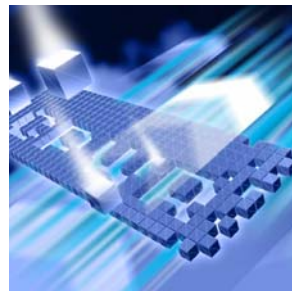
- ◆ DevPartnerのバージョン
- ◆ UserAllocators.datファイルのコピー
- ◆ 発生している問題の概要

場合によっては、ユーザーが作成したアロケータ関数が入っているDLLとDLLにリンクするために使用するマップファイルのコピーが必要になることがあります。

無視レコードの使用はなるべく避けてください。無視レコードを指定すると、アプリケーションを分析するときにDevPartnerエラー検出が予期されない動作を行うことがあります。

第5章

デッドロック分析



- ◆ 背景：シングルスレッドアプリケーションとマルチスレッドアプリケーション
- ◆ デッドロック - 基本定義
- ◆ デッドロックを回避するためのテクニック
- ◆ 潜在的なデッドロック
- ◆ その他の同期オブジェクト
- ◆ 追加情報

デッドロック分析は、カスタムアプリケーションのデッドロック、潜在的なデッドロック、その他の同期エラーを自動的に検索する方法を提供します。

この章では、以下について説明します。

- ◆ デッドロック分析で使用される用語の概要
- ◆ デッドロックおよび潜在的デッドロックの例
- ◆ 同期に関する追加情報の入手先

背景：シングルスレッドアプリケーションとマルチスレッドアプリケーション

古いスタイルのCおよびC++プログラムは、多くの関数を呼び出し、多様な演算を実行して終了する単純なメインルーチンを利用していました。これらのプログラムは、実行にシングルスレッドを使用していました。これは、プログラムが、一度に1つの命令を実行することを意味します。デバッガを使用してプログラムをステップごとに検証すれば、映画のフレームのように、すべての動作を確認できます。

スレッド

新しいアプリケーションでは、マルチスレッドを使用できます。「スレッド」とは、制御の流れです。マルチスレッドアプリケーションは複数の制御の流れを利用します。

Windowsの**CreateThread**関数を呼び出して、追加のスレッドを作成することが可能です。**CreateThread**には、新しく作成されたスレッドで実行させる関数のアドレスなどの一連のパラメータを指定できます。**CreateThread**関数が正しく実行されると、アプリケーションが追加スレッドを実行できるようになります。

スレッドを自動的に作成する方法は複数あります。たとえば、**_beginthread**を呼び出す方法や、サードパーティのライブラリ、COMまたはDCOM、あるいは共通言語ランタイムを使用する方法などがあります。

プログラムで複数のスレッドを利用すると、2つのスレッドが同時に同じリソースにアクセスしようとする状況が発生する可能性があります。リソースには、変数、ファイル、ハンドル、Windowsリソースなどがあります。複数のスレッドが同時に同じリソースにアクセスしようとする、同期の問題が発生します。たとえば、T1とT2という2つのスレッドの両方が、1～100の数字を印刷しようとする、各スレッドからの出力は以下ようになります。

```
1 2 3 4 5 6 7 8 9 10 11 12 ... 95 96 97 98 99 100
```

スレッドが同時に実行されると、下の例のように出力が混同します（スレッドT1からの出力は標準で、スレッドT2からの出力は斜体で示します）。

```
1 2 3 4 1 2 5 6 3 4 5 6 7 8 7... 95 96 97 94 95 96 97 98 9998 99 100 100
```

クリティカル セクション

このような問題を防止するため、スレッド間の相互作用を整理する必要があります。最近のオペレーティングシステムの大半には、共有リソースへのアクセスを調整するために呼び出す、一連の同期関数が装備されています。最も簡単でよく使用される同期オブジェクトは、「クリティカルセクション」と呼ばれるものです。クリティカルセクションは、一度にリソースにアクセスできるスレッドを1つに制限する簡単な関数です。

前述の、共に1～100の数字を印刷するスレッドT1とT2の例を考えてみましょう。クリティカルセクションC1を定義すると、両方のスレッドが実行される時の出力の混同を防止できます。このクリティカルセクションは、出力ストリームへのアクセスを制御します。スレッドT1とT2によって実行される関数は、以下のように変更する必要があります。

- 1 スレッドのいずれかが、クリティカルセクションC1を作成する。
- 2 その後、それぞれのスレッドが、次のステップを実行する。
 - a クリティカルセクションC1を要求する。
 - b 1～100の数字リストを印刷する。
 - c クリティカルセクションC1を解放する。
- 3 両スレッドが解放され、他方のスレッドとの相互作用が発生しない残りのプロセスを実行する。

手順 2-a は、スレッドにクリティカル セクション C1 への排他的なアクセスを与えるようにオペレーティング システムに要求する、**EnterCriticalSection** コールに変換されます。クリティカル セクションが利用できない場合は、オペレーティング システムがスレッドを一時停止して、C1 が利用できるようになるまで待機します。

1つのスレッドがクリティカル セクションにアクセスできるようになると、C1 のクリティカル セクションルールに従う他のスレッドは、出力を印刷しようとしなくなります。スレッドが 1 ~ 100 の数字を印刷したら、手順 2-c では **LeaveCriticalSection** をオペレーティング システムに指示します。これは、クリティカル セクションを他のスレッドで使用できるように解放します。

プログラムの全スレッドが、端末に出力を印刷するためにクリティカル セクションを使用しなければならないというルールはありませんが、このルールに従うと、出力は常に正しく表示されます。

このルールは、変数、構造、ファイル、その他の共有リソースにアクセスする場合にも適用できます。

メモ： 2つの出力ストリームが相反するコードを記述する以外は、ほとんどの場合、コンソール出力にクリティカル セクションを使用する必要はありません。

デッドロック - 基本定義

前出の例を見ると、クリティカル セクションは、共有リソースへのアクセスを与えるための非常に簡単なメカニズムのようですが、問題が発生する可能性もあります。

C1、C2、C3 という複数のクリティカル セクションを作成するプログラムを考えてみましょう。それぞれのクリティカル セクションは、スレッド間で共有される別々のリソースへのアクセスを保護するために使用されます。

スレッドが1つのクリティカル セクション (C1 など) へのアクセスを与えられ、他のクリティカル セクション (C2 など) へのアクセスを得ようとする場合、C2 がすでに他のスレッドに割り当てられている可能性があります。他のスレッドがすばやく C2 を解放する場合は問題はありません。最初のスレッドは、C2 が利用できるようになるまで待機したあとで C2 へのアクセスが与えられれば、操作を続行できます。

一方、C2 を与えられているスレッドが、他の同期オブジェクト (C1 など) を利用できるようになるのを待機する必要がある場合、両方のスレッドが、必要なリソースへのアクセスを得るために待機し続ける状態が発生します。2つ以上のスレッドが利用可能になることのないリソースを待機し続ける状態のことを、「デッドロック」と呼びます。

デッドロックを回避するためのテクニック

デッドロックは、複数のスレッドが共有リソースを使おうとしたときに、これらのリソースへのアクセスを得ることができない場合に発生します。デッドロックを回避するには、数多くの方法があります。

- ◆ 必要なときにだけ、同期オブジェクトへのアクセスを要求します。オブジェクトへのアクセスを得たら、他のスレッドがそのオブジェクトを使用できるように、できるだけすばやくオブジェクトを使用して解放します。
- ◆ 特定の操作を実行するために、複数の同期オブジェクトへのアクセスを一度に得る必要がある場合、まず最初のオブジェクトを要求してから、2番めのオブジェクトへのアクセスを得るようにします。2番めのオブジェクトが利用できない場合は、両方のオブジェクトを解放して、短いランダムな間隔で待機します。待機したあとで、再度これらのリソースへのアクセスを試みます。スレッドが他のリソースを待機しているときにそのリソースへのアクセスがブロックされた場合は、所有しているリソースを解放することが非常に重要となります。オブジェクトの解放に失敗すると、デッドロック状態をさらに悪化させる事態になります。
- ◆ 常に同じ順番でリソースを要求します。たとえば、操作を実行するために C1、C2、C3 へのアクセスを得る必要がある場合、常に同じ順序 (C1、C2、C3) でアクセスし、逆の順序 (C3、C2、C1) で解放します。
- ◆ 操作を実行するために必要なすべての同期オブジェクトを取得したら、他のリソースに対する待機をブロックする可能性のある操作は実行しないようにします。

同期オブジェクトの利用方法には、これら以外にも多くのテクニックがあります。「[追加情報](#)」(83 ページ) には、同期オブジェクトに関する MSDN リソースと参考文献が記載されています。

潜在的なデッドロック

DevPartner エラー検出は、安全な方法でリソースにアクセスしていない状態が検出されると、「潜在的なデッドロック」をレポートします。この例として、クリティカルセクション C1、C2、C3 によって制御される一連のリソースを使用するスレッド T1、T2、T3 を持つアプリケーションを示します。

表 5-1 では、各スレッドが指定された操作を実行するために必要となるクリティカルセクションを示しています。

表 5-1. 潜在的なデッドロックの例：スレッドと必要なクリティカルセクション

スレッド	クリティカルセクション
T1	C1、C2
T2	C2、C3
T3	C3、C1

各スレッドは、独立して実行でき、指定されたタスクを実行するために必要なクリティカルセクションを取得しますが、すべてのスレッドが同時にこれらの操作を実行しようとする、問題が発生します。

食事をする哲学者

「食事をする哲学者」は、コンピュータ科学の授業で潜在的なデッドロックを説明する際によく使用される、有名な例です。DevPartner エラー検出ソフトウェアには、食事をする哲学者のサンプルコードが含まれています。これは、以下の場所にあります。

...DevPartner StudioExamplesDeadlockPhilosophers

食事をする哲学者の問題は、複数の哲学者が円形のテーブルにつき、そのテーブルの中央に食べ物を盛った大きな皿が置かれている状態から始まります。各哲学者の間には、はしが1本ずつ置かれています。

テーブルについている哲学者は、3つのことができます。

- 1 休む：休んでいる哲学者はただ座っているだけで何もしません。休んでいる時間はランダムです。
- 2 話す：会話をする哲学者は、話を聞きたい他の哲学者に対して話をします。話をしている時間はランダムです。

3 食べる：空腹な哲学者は食事をしようとします。このため、はしを取ろうとします。最も簡単な例として、哲学者は、まず最初に左のはしを取ろうとします。左のはしを取れたら、右のはしも取ろうとします。右と左のはしを持った哲学者は、ランダムな時間だけ食べ物を食べてから、はしを置き、休むか話を始めます。

最初のはしを取れなかった哲学者は、数秒待機してから、もう一度はしを取ろうとします。最初のはしを取れたら、右のはしを取ろうとします。右のはしがあれば、哲学者は数秒待ってからもう一度はしを取ろうとします。

問題は、すべての哲学者が同時に左のはしを取ろうとしたときに発生します。この状態が発生すると、左のはしを置くことができなくなるため、だれも食べ物を食べるができなくなります（デッドロック）。

食事をする哲学者アルゴリズムをどのように構成するかにより、ただちにデッドロック状態になるか、またはプログラムを数分実行できるが、その後デッドロック状態になることが考えられます。哲学者とはしを食卓に追加すると、実際のデッドロック数は減少しますが、だれも食べられなくなる可能性はまだ存在します。これを「潜在的なデッドロック」と呼びます。

潜在的なデッドロックは、非常に負荷の高い実稼働システムで発生する傾向があるため、多くの場合、最も追跡が難しいデッドロックです。開発システムでこれらの問題を再現しようとする、時間がかかり、問題の本当の原因をつかめないことがよくあります。

DevPartner エラー検出は、実際のデッドロックが発生するかなり前に、潜在的デッドロックを検出した場合にレポートします。また、実際のデッドロックがどのように発生するかについての詳細情報も提供します。これにより、問題を回避するためのコードの書き換えが簡単にできるようになります。

同期オブジェクトの監視

デッドロック分析では、アプリケーションの全同期オブジェクトも監視し、以下のようなエラーや問題のある可能性がある使用状況を検出します。

- ◆ ユーザーが指定した時間を超えた待機
- ◆ すでに所有されているクリティカル セクションを再入力するスレッド
- ◆ すでにスレッドによって所有されているミューテックスの待機
- ◆ 同期オブジェクトを解放せずにスレッドを終了する

さらに、名前を付けることができる同期オブジェクトが、命名規則に従って命名されていることを確認するように DevPartner エラー検出を設定することもできます。たとえば、プロセス外からのアクセスを防ぐため、同期オブジェクトに名前を付けないようにしたとします。名前の付いた同期オブジェクトは、潜在的なエラーとしてフラグが付けられます。次に、このリストを使って、システムの他のプロセスによる望ましくないアクセスを防ぐために、必要なセキュリティ記述子を含んでいるかどうかを確認できます。

同期エラーの完全なリストは、オンライン ヘルプの「検出されるエラーの説明」セクションにある「デッドロック エラー」に記載されています。

その他の同期オブジェクト

Windows オペレーティング システムは、[76 ページ](#)で説明するクリティカルセクション以外にも、多くの同期オブジェクトを提供しています。以下に、同期オブジェクトのリストとMSDNから抜粋した定義を示します。抜粋したテキストは、「」で囲っています。各用語について、定義の全文の URL と関連コード例の URL も記載しています。

クリティカル セクション

「クリティカル セクション オブジェクトは、ミューテックス オブジェクトが提供するのと同様の同期を提供します。ただし、クリティカル セクションは、単一プロセスのスレッドでのみ使用できます。」

定義の全文：

```
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/Critical\_Section\_Objects.asp
```

コード例：

```
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using\_critical\_section\_objects.asp
```

イベント

「イベント オブジェクトは、SetEvent 関数を使用してシグナル状態を設定できる同期オブジェクトです。イベント オブジェクトは、特定のイベントが発生したことを示し、スレッドにシグナルを送る場合に使用します。」

定義の全文：

```
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/event\_objects.asp
```

コード例：

```
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using\_event\_objects.asp
```

ミューテックス

「ミューテックス オブジェクトは、スレッドに所有されていない場合に、シグナル状態を設定する同期オブジェクトです。所有されている場合は、非シグナル状態になります。1つのスレッドで同時に所有できるミューテックス オブジェクトは1つです。

ミューテックス オブジェクトは、クリティカル セクションよりも処理が遅くなります。」

定義の全文：

```
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/mutex\_objects.asp
```

以下の URL は、ミューテックス オブジェクトを使う例を示しています。

```
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using\_mutex\_objects.asp
```

セマフォ

「セマフォ オブジェクトは、ゼロと指定した最大値の間でカウントを維持する同期オブジェクトです。カウントは、スレッドがセマフォ オブジェクトの待機を完了するたびに1つ減り、スレッドがセマフォを解放するたびに1つ増えます。」

定義の全文：

```
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/semaphore\_objects.asp
```

コード例：

```
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using\_semaphore\_objects.asp
```

追加情報

MSDNの参照情報

同期オブジェクトについての詳細は、以下のMSDNリンクを参照してください。

同期の概要 :	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/about_synchronization.asp
同期オブジェクト :	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/synchronization_objects.asp
待機関数 :	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/wait_functions.asp
同期オブジェクトの使用 :	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using_synchronization.asp
同期についての参照情報 :	http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/synchronization_reference.asp

その他の参照情報

以下の書籍には、同期オブジェクトの詳細が記載されています。

『Win32 Multithreaded Programming』、Aaron Cohen、Mike Woodring 共著

『Debugging Applications for Microsoft .NET and Microsoft Windows』、John Robbins 著

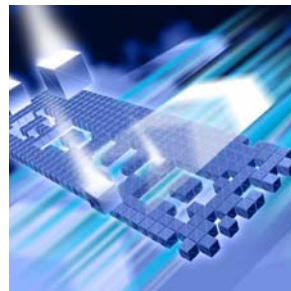
『Debugging Windows Applications, 1st Edition』、John Robbins 著

『Operating Systems, 4th Edition』、William Stallings 著

『Foundations of Multithread, Parallel and Distributed Programming』、Gregory R. Andrews 著

付録 A

エラー検出のトラブルシューティング



トラブルシューティング

以下の問題が発生した場合は、そこに示す対処法を試してみてください。さらに困難な問題が発生する場合は、弊社のテクニカル サポートにお問い合わせください。

問題	対処法
エラー検出がメモリ上でステップングする	<ul style="list-style-type: none">• [オプション]または[設定]ダイアログの[メモリの追跡]で、[解放時に無効データでフィルする]を無効にします。• [オプション]または[設定]ダイアログの[メモリの追跡]で、[確保時にフィルする]を無効にします。
[デバッグ] ボタンを使用して [検出されたプログラム エラー] ダイアログ ボックスを非表示にすると、デバッグでエラー検出が停止しない	<ul style="list-style-type: none">• [デバッグ]ボタンを使用して[検出されたプログラム エラー]を非表示にすると、アプリケーションでは、エラー検出によってエラー後の最初の行の初めにブレークポイントが設定されます。ブレークポイントは、アプリケーションがデバッグで停止したときに自動的に削除されます。ブレークポイント例外をキャッチして実行を継続する例外ハンドラがアプリケーションにある場合、デバッグでブレークポイントをキャッチし、アプリケーションを停止することができなくなります。
「メモリ割り当ての競合：関数の不一致」というエラー メッセージが表示され、エラー検出ではブロックがFree関数によって解放されたことが報告される	<ul style="list-style-type: none">• newとdeleteを使用している場合、エラー検出は、ブロックがfree関数によって解放されたことをレポートしています。また、Cランタイム ライブラリDLL (/MD) のリリース バージョンを使用している可能性があります。この問題を修正するには、Cランタイム ライブラリDLL (/MDd) のデバッグ バージョンに対してビルドします。これは、C/C++ コード生成のプロパティ ページのランタイム ライブラリ エントリで制御できます。
エラー検出で、演算子newとfreeの間での割り当ての競合がレポートされる	<ul style="list-style-type: none">• userallocators.datに、ユーザーが作成したすべてのアロケータを追加していることを確認します。 ...¥Data¥UserAllocators.dat ファイルのコメントを参照してください。

エラー検出でメモリ割り当てとリソース割り当てが無視される

- C++アプリケーションでマネージ デバッガを使用してエラー検出を実行している場合、アプリケーションではなく、mscorwks.dll や mscorsrv.dll のメモリ割り当てとリソース割り当てが使用され、その結果、エラー検出でアプリケーションのメモリ割り当てとリソース割り当てが無視されます。混合コードのアプリケーションやマネージ コードのアプリケーションでエラー検出を使用するには、以下の手順を実行します。

- 1 Visual Studio バージョン (2003 または 2005) を開き、プロセスのデバッガに使用します。
- 2 DevPartner エラー検出のスタンドアロン アプリケーションを起動します。
- 3 エラー検出内でターゲット プロセスを開き、[プログラム]>[開始] を選択してターゲット プロセスを実行します。
- 4 デバッガを以下のようにターゲット プロセスにアタッチします。
 - Visual Studio 2003 – [ツール]>[デバッグ プロセス] を選択します。
 - Visual Studio 2005 – [ツール]>[プロセスにアタッチ] を選択します。

メモ : [マネージ コード デバッグ] を指定して ([混合] や [自動] ではなく)、デバッガがマネージ コードのブレークポイントに正確に到達できるようにします。

これらの手順を完了すると、マネージ コードのブレークポイントを設定し、そのブレークポイントに到達することができます。その後、エラー検出でメモリ割り当てとリソース割り当てが検出されます。

エラー検出でアプリケーションを実行すると、「不正な文字列」というエラーメッセージが表示される

- ASCII文字列をWide Char文字列にキャストし、APIパラメータとして使用すると、実際の問題に関連しないエラーがレポートされる場合があります。キャストとは、「文字列のデータ型を正確に認識する」ことを意味するため、キャストはエラー検出によって検出されません。以下に例を示します。

```
BSTR m_DSNName;  
m_DSNName=SysAllocString(BSTR(""));
```

上記の文字列はコンパイラによって以下のように解釈されます。

```
m_DSNName=SysAllocString((wchar_t *) "");
```

以下のエラーがレポートされます。

```
Invalid String: In call to SysAllocString, address  
0x0FE2751F is not null terminated in block  
0x0FDF0000 (290816).
```

SysAllocStringステートメントでは、ASCIIの空文字列をWide Char文字列にキャストし、SysAllocStringへの入力パラメータとして使用します。SysAllocStringでは、最初の2バイトのNULLを検索し、wchar_t文字列の終わりを見つけます。さらに、この2バイトのNULLの場所によって決定された長さを使用して、新しいBSTRのサイズを決定します。SysAllocStringでは、m_DSNNameに対して新しいBSTRを有効に作成します。これには、あとでメモリに発生する、1バイトのASCII文字列の始まりと最初の2バイトのNULL間に存在するすべての不正な文字列のコピーが保存されます。

上記の例では、エラー検出によって、入力されたwchar_t文字列（実際は1バイトのASCII空文字列）が調べられ、不正な文字列（NULLで終了しない）がレポートされます。この場合に使用されたテストコードでは、1バイトのASCII文字列の始まりと予期されない2バイトのNULL間に物理的に存在したメモリのブロックは有効であると認識されています。メモリが他の用途で割り当てられていないことを「認識」する限りは、入力文字列のみがスキャンされます。そのため、この場合は数バイトしかスキャンされていないので、そのメモリ領域で2バイトのNULLが見つかりませんでした。この問題を回避するには、Wide Char文字列を入力として使用するようそのコードを変更してください。

エラー検出でエラーが何も報告されない

- **[オプション]**または**[設定]**ダイアログの**[全般]**で、**[イベントをログに記録]**を有効にします。
- DCOMまたはCOMベースのアプリケーションやコンポーネントを使って、aspnetアカウントの制限下で実行を試みた場合は、2つの対処法があります。デフォルトでは、DCOMまたはCOMのアプリケーション／コンポーネントは、ASP.NETを有効にしたWebページから起動され、aspnetのコンテキストで実行されます。セキュリティ上の理由から、aspnetアカウントは制限されています（このアカウントは、Usersグループのメンバであり、同等の権限内容が設定されています）。この状況では、COMコンポーネントは、エラー検出を適切に実行するために必要なセキュリティ権限を持っていないこととなります。この問題を回避するには、DCOMまたはCOMのアプリケーション／コンポーネントを構成し、(dcomcnfg.exe経由の)対話ユーザーのコンテキストで実行するようにします。これを行うには、以下の手順を実行します。
 - 1 コマンド プロンプトを開き、dcomcnfg.exeを実行します。
 - 2 **[コンポーネント サービス]>[コンピュータ]>[マイ コンピュータ]>[DComの構成]**の順に選択します。
 - 3 COMコンポーネントを右クリックして、**[プロパティ]**を選択します。
 - 4 **[ID]**タブを選択します。
 - 5 **[対話ユーザー]**を選択していることを確認します。
 - 6 **[OK]**をクリックします。

エラー検出の実行が極めて遅い

- エラー検出のすべてのオプションを有効にしているかを確認します。オプションの中には、プロセッサやメモリを大量に消費するものがあります。本当に必要なオプションだけを使用してください。
- 割り当ての最大コール スタック数を制限します（**[オプション]**または**[設定]**ダイアログの**[データ収集]**を使用）。コール スタック数が大きいと、メモリを大量に消費します。
- モジュールやファイルを除外して、分析を制限します（**[オプション]**または**[設定]**ダイアログの**[モジュールとファイル]**を使用）。
- FinalCheckは、開発の重要なマイルストーンでのみ使用します。
- **[.NET コール レポート]**を無効にするか、または**[すべてのタイプ]** ツリー ビューで選択したアセンブリ数を制限します。
- アプリケーションでリークの検出だけを実行する場合には、リークのみ分析を有効にすることを検討してください。**[メモリの追跡]**オプションの**[リーク分析のみを有効にする]**チェックボックスをオンにすると、メモリ追跡でリークの監視を除くすべての分析が無効になります。メモリ追跡では、オーバーラン、初期化されていないメモリ、ダングリング ポインタは検索されません。メモリ追跡では、システム モジュールによって割り当てられたメモリを評価しないため、**[コール バリデーション]**の**メモリ ブロック チェック**も無効になります。

問題

対処法

ログのサイズが大きすぎる

- **[オプション]**または**[設定]**ダイアログの**[全般]**で、**[イベントをログに記録]**を無効にします。
- データ表示の深さを制限します (**[オプション]**または**[設定]**ダイアログの**[データ収集]**を使用)。
- モジュールやファイルを除外して、分析を制限します (**[オプション]**または**[設定]**ダイアログの**[モジュールとファイル]**を使用)。
- フィルタと抑制を使用して、レポート範囲を制限します。
- **[.NET コール レポーターティング]**を無効にするか、または**[すべてのタイプ]**ツリー ビューで選択したアセンブリ数を制限します。
- **[API コール レポーターティング]**の**[API メソッドのコールとリターンを収集する]**を無効にします。
- **[リソースの追跡]**を無効にするか、または追跡対象のリソースを制限します。

エラー検出でUserAllocators.log ファイルの作成が失敗したことを示すエラーが表示される

- 書き込み権がないディレクトリでエラー検出を使用してアプリケーションを実行しようとすると、以下のようなエラーが表示される場合があります。

```
UserAllocatorsError: An error was discovered when processing the UserAllocators.dat file. Failed to create UserAllocators.log file Error:0x00000005
```

[データ収集]設定ページで**[NLB ファイル ディレクトリ]**を設定することで、UserAllocators ファイルを書き込むディレクトリを制御することができます。

Windows Vistaでエラー検出を使用してターゲットアプリケーションを開こうとすると、エラーが表示される

- エラー検出により、ターゲット アプリケーションごとにデータ ファイルが作成されます。エラー検出を開始する前に、ターゲットの実行可能ファイルを含むディレクトリへの書き込みアクセス権があることを確認する必要があります。

エラー検出が完了しない、またはコール スタックにエラーがある

- エラー検出がシンボルを特定できなかったか、シンボルが古くなっています。Microsoft Symbol Server を有効にし、現在のファイルに一致するシンボルを取得します。そして、アプリケーションを再実行してください。
- テスト中のアプリケーションで、マネージコードとアンマネージコードが混在して使用されています。2種類のモジュール間で発生したトランザクションが、コース スタックを正常に処理できません。

Symbol Server での処理に時間がかかり過ぎる

- エラー検出は、実行のたびに、Microsoft Symbol Server からシンボルの取得を試みます。現在のファイルに一致するシンボルを取得したら、Microsoft Symbol Server を無効にして、ローカルなシンボルを使用してアプリケーションを実行してください。
- 一部のオペレーティング システムについては、シンボル パッケージを
- ダウンロードし、永久的にインストールすることができます。詳細については、Microsoft 社のシンボル パッケージ ダウンロード サイト (<http://www.microsoft.com/japan/whdc/devtools/debugging/symbolpkg.mspx>) を参照してください。

問題

対処法

COM オブジェクトの追跡（または COM コール レポートリング）が正常に機能していないように見える	<ul style="list-style-type: none">• [オプション]または[設定]ダイアログの[モジュールとファイル]の設定を調べ、特定のモジュールについてイベントのレポートを無効に設定していないことを確認します。• [オプション]または[設定]ダイアログの[抑制とフィルタ]の設定を調べ、特定の COM 追跡イベントについてフィルタや抑制を設定していないことを確認します。• [オプション]または[設定]ダイアログの[COM コール レポートリング]にある[すべてのインターフェイス]チェックボックスを使用して、エラー検出で複数の COM インターフェイスを監視するように設定します。
アプリケーションが開始しない、または開始してもすぐにクラッシュする	<ul style="list-style-type: none">• [オプション]または[設定]ダイアログの[メモリの追跡]で、[解放時に無効データでフィルする]チェックボックスをオフにします。アプリケーションが解放したメモリを参照している可能性があります。• [オプション]または[設定]ダイアログの[コールパリデーション]で、[コール前に出力情報を入力する]チェックボックスを無効にします。• ユーザーが作成したアロケータがあるかどうかを調査します（...¥Data¥UserAllocators.dat ファイルを参照してください）。• システム上、OS シンボルが存在するかを確認します。存在する場合、そのシンボル ファイルに不具合がある可能性があります。• クラッシュのタイミングに関連があるかを確認します。関連がある場合は、エラー検出機能の一部を無効にし、もう一度、アプリケーションの実行を試してください。
サービスが起動するとすぐにハングする	<ul style="list-style-type: none">• 管理者権限でサービスを実行していることを確認します。
サービスが起動するとすぐに終了する	<ul style="list-style-type: none">• 一般的に考えられる原因は、Windows NT のサービス コントロール マネージャによるサービスの中止です。サービスの初期化ロジックの dwWait の値を大きくしてサービスを再実行します。• 有効な作業ディレクトリがエラー検出にあることを確認します。[プログラム]メニューの[設定]にある[全般]の設定を使用して、作業ディレクトリを指定します。
サービスが実行してしばらく経つと、突然終了する	<ul style="list-style-type: none">• サービス状態を要求するコントロール メッセージへのサービスの応答が遅すぎる可能性があります。サービス状態の要求に応答する場合は、dwWait のタイムアウト値を大きくします。• エラー検出によってアプリケーションのメモリが解放時に無効データによってフィルされ、クラッシュすることもあります。[オプション]または[設定]ダイアログの[メモリの追跡]機能を無効にします。これでクラッシュが解消したら、FinalCheck でサービスをインストールし、アプリケーションを再実行して、初期化されていないメモリ リファレンス、バッファ オーバーラン、およびダンangling ポインタを探します。
サービスは正常に実行されるが、シャットダウン時に突然終了する	<ul style="list-style-type: none">• サービス コントロール マネージャからシャットダウン要求を受け取ったときのサービスの応答時間が制限されています。アプリケーションの終了時は、エラー検出によって、メモリ リーク、リソース リーク、およびインターフェイス リークの検出や、割り当て済みのメモリ ブロックの再チェックによるメモリ オーバーランが発生していないかどうかの確認が行われます。シャットダウン要求に応答するために指定されている dwWait 値が小さすぎると、サービス コントロール マネージャによってサービスが中止されます。この場合は、dwWait 値を大きくします。

エラー検出でサービスを分析できない	<ul style="list-style-type: none"> エラー検出では、サービスを分析できない場合があります。一般的な原因としては、プロセスの監視に使用する1つまたは複数のディレクトリが、作成されるプロセスに対して書き込み可能になっていないことが挙げられます。以下のディレクトリは、書き込み可能にする必要があります。 <ul style="list-style-type: none"> TMP 環境変数と TEMP 環境変数は、プロセスに書き込むことができるディレクトリを参照する必要があります。LOCAL_SYSTEM コンテキストを実行するためにサービスを設定している場合には、それらの環境変数をシステム全体に割り当てる必要があります。 サービスには、NLB ファイル ディレクトリへの書き込みアクセス権が必要です。 サービスには、作業ディレクトリへの書き込みアクセス権が必要となる場合があります。
[プロセスを待機]を使用しているときに、エラー検出でアプリケーションを分析できない	<ul style="list-style-type: none"> エラー検出のスタンドアロンバージョンで[プロセスを待機]を使用しているときに、アプリケーションを正常に監視するには、ターゲットの名前とパスが、エラー検出に対して指定したものと完全に一致している必要があります。ターゲットの名前とパスを変更する中間手順を実行した場合、エラー検出でアプリケーションの起動が認識されない可能性があります。たとえば、実行可能ファイル (C:¥¥MyApplications¥foo.exe) をターゲットとして選択し、[プロセスを待機]を選択してから、完全修飾名を短縮名に変更する中間アプリケーションと C:¥¥MyApplication¥foo.exe を起動すると、エラー検出でアプリケーションの起動が認識されません。
エラー検出を指定してアプリケーションを実行しようとすると、Visual Studio がクラッシュする	<ul style="list-style-type: none"> 以下に挙げる方法の少なくとも1つを使用して、クラッシュの原因を絞り込みます。 <ul style="list-style-type: none"> アプリケーション テスト コードのさまざまな場所にブレークポイントを設定する。 エラー検出の設定を再構成し、いくつかのサブシステムやモジュールを無効にする。詳細については、ヘルプの「プログラム設定の変更」に関する説明を参照してください。
タスク マネージャではメモリ使用量の増加を確認したが、エラー検出ではリークについて何も報告されない	<ul style="list-style-type: none"> FinalCheck でアプリケーションを完全にインストールしていることを確認します。 [オプション]または[設定]ダイアログの[全般]で、[イベントをログに記録]の設定を調べます。 アプリケーション テスト モジュールが有効になっていることを確認します([オプション]または[設定]ダイアログの[モジュールとファイル]を使用)。
マネージC++アプリケーションに対してエラー検出を実行したが、期待していたリークやエラーが報告されない	<ul style="list-style-type: none"> エラー検出の既知の制限事項です。メモリ割り当てを調べ、適切なレポートが出力されるように、ネイティブ デバッガ (またはマネージ デバッガとネイティブ デバッガ) を使用していると考えられます。マネージ デバッガを使用している場合、メモリ割り当てとリソース割り当ては (アプリケーションではなく) mscorewks.dll の情報が使用され、結果としてアプリケーションの情報は無視されます。
ActiveCheck でメモリ リークが検出されるが、FinalCheck では検出されない	<ul style="list-style-type: none"> FinalCheck でインストールすると、より確実なメモリ追跡を行うことができます。FinalCheck は範囲を認識し、エラー検出ではメモリ使用状況だけでなく各ポイントの追跡も可能になります。詳細については、『クイック リファレンス』の「ActiveCheck と FinalCheck によるエラー検出」を参照してください。

問題

エラー検出を指定して実行するとアプリケーションが失敗する（通常はメモリ追跡を有効にしている）

対処法

- **[オプション]**または**[設定]**ダイアログの**[メモリの追跡]**で、**[解放時に無効データでフィルする]**を無効にします。
- **[オプション]**または**[設定]**ダイアログの**[メモリの追跡]**で、**[保護バイトを有効にする]**と**[確保時にフィルする]**の充てんパターンを入れ替えてみてください。
- 不具合のあるPDBファイルを使用している可能性があります。ASCIIテキストファイルを作成し、NMSYMDATA.DATという名前をNMSHARED¥4.7ディレクトリに保存します。このファイルには、不具合のあるPDBファイルに関連付けられたモジュール名を含めます。後ろに「,0x0」を付加します。
例：ADVAPI32.DLL,0x0
- Visual Studio MFCアプリケーションウィザードを使用して生成されたアプリケーションをデバッグし、メモリ追跡サブシステムのメモリフィル機能を有効にしている場合、エラー検出でアプリケーションがクラッシュする可能性があります。MFCであいまいなpragmaを設定すると、コンパイラで「最小デバッグ情報」が生成される原因となります。使用しているOS構造に追加フィールドが追加されている場合、エラー検出でその構造のサイズを判断するときに、そのデバッグ情報から不正な構造サイズを取得する可能性があります。これは、タッチすべきでないメモリをフィルする原因となります。プロジェクトのC++プリプロセッサ設定ページで_AFX_FULLTYPEINFOをプリプロセッサ定義に追加してから、ソリューションをリビルドします。
- NMSYMDATA.DATファイルを作成しても問題が解決されない場合は、モジュール全体を除外する必要があるかもしれません。アプリケーションモジュールを除外するには、ASCIIテキストファイルを作成し、EXCLUDEDMODULES.DATという名前をDataディレクトリに保存します。このディレクトリは、エラー検出がインストールされたところにあります。以下に例を示します。

```
<InstallationRoot>¥Data¥EXCLUDEDMODULES.DAT
```

除外するモジュールの名前を追加します。モジュール名は、1行に1つ指定します。

例：MYCUSTOMDRIVER.DLL

Visual Studioに統合したエラー検出を使用してWebアプリケーションをデバッグできない

- アプリケーションやサービス（Webアプリケーションなど）をデバッグするには、**[プロセスを待機]**オプションを使用します（**「サービスの分析」**（35ページ）を参照）。このオプションは、エラー検出のアプリケーション（BC.EXE）から利用できます。Visual Studioに統合したエラー検出を実行している場合は、このオプションは利用できません。
-

デバッグしたいモジュールが System ファイルのディレクトリにあるのだが、これらのファイルは制限されているため、モジュールをデバッグできない	<ul style="list-style-type: none"> • [オプション] または [設定] ダイアログの [モジュールとファイル] を使用してモジュールを有効にすると、現在のプロジェクトに対してデバッグできます。 • すべてのプロジェクトとソリューションに対してモジュールを有効にするには、エラー検出のインストール先である Data ディレクトリにある <code>Unrestricted_modules.txt</code> というファイルを編集します。以下に例を示します。
	<pre><InstallationRoot>%Data%\Unrestricted_modules.dat</pre> <p>含めるモジュールの名前を追加します。モジュール名は、1行に1つ指定します。</p> <p>例：MYCUSTOMDRIVER.DLL</p>
エラー検出で <code>dllhost.exe</code> または <code>TestCon32.exe</code> のエラーがレポートされる	<ul style="list-style-type: none"> • エラー検出で <code>dllhost.exe</code> または <code>TestCon32.exe</code> のエラーがレポートされないようにするには、チェックするモジュールのリストから該当する実行可能ファイルを除外します。
COM コール レポートでオブジェクトやコンポーネントへのコールがログに記録されない	<ul style="list-style-type: none"> • エラー検出は、認識するように指示されている COM インターフェイスのみのメソッドを記録します。エラー検出に ActiveX コントロールについて知らせるには、[オプション] または [設定] ダイアログの COM コール レポートで、[選択したモジュールに実装された COM メソッド コールのレポートを有効にする] を選択します。
エラー検出からオブジェクトまたはコンポーネントの COM インターフェイス リークがレポートされない	<ul style="list-style-type: none"> • COM インターフェイス リーク情報を収集するには、[オプション] または [設定] ダイアログの [COM オブジェクトの追跡] で、[COM オブジェクトの追跡を有効にする] を選択します。そして、監視する COM クラスを選択します。 • 独自のオブジェクトを追跡するには、[COM オブジェクトの追跡] の設定の COM クラスのリストから、該当するクラスだけを選択します。選択するクラスがわからない場合は、[すべての COM クラス] を選択します。
コンポーネントの実行を停止すると、エラー検出がハングしたかのように長時間応答しない	<ul style="list-style-type: none"> • エラー検出は <code>dllhost.exe</code> がタイムアウトし、プロセスを終了するまで待機します。<code>dllhost.exe</code> が終了すると、メモリ リーク、リソース リーク、およびインターフェイス リークの最終的な検出が行われます。
なぜ IIS は起動してすぐハングするのか	<ul style="list-style-type: none"> • エラー検出でサービスをデバッグするには管理者権限が必要です。管理者権限のないアカウントを使用すると、IIS はハングするか、エラーの発生と同時に終了します。
エラー検出によってシンボル情報が返されない	<ul style="list-style-type: none"> • プロジェクト設定でシンボルパスがソリューション レベルではなく、プロジェクト レベルで定義されていることを確認してください。
コマンドラインで (VCBUILD.EXE を使用して) バッチ プロセスを自動的にコンパイルするときに、エラー検出のコードをインストールできない	<ul style="list-style-type: none"> • Visual Studio .NET 2003 または Visual Studio 2005 を使用している場合：コマンドラインからコンパイラを呼び出すときに、コードをインストールするには、DevPartner でインストールされる NMVCBUILD パリアントを使用する必要があります。NMVCBUILD を使用したコンパイルに関する詳細については、「nmvcbuild を使用したネイティブ C/C++ のインストール」を参照してください。 • 以前のバージョンの Visual Studio を使用している場合：コマンドラインからコンパイラを呼び出すときに、コードをインストールするには、メイクファイルを使用する必要があります。メイクファイルの詳細については、オンライン ヘルプの「コマンドラインからの FinalCheck の実行」を参照してください。

Windows Vista でエラー検出のインストゥルメントーションを使用してターゲットアプリケーションを作成しようとする、エラーが表示される

エラー検出でアプリケーションを実行すると、不正なデータや予期されない結果が返される

エラー検出によって、混合モード（マネージとアンマネージ）の環境で不正なリークとオーバーランがレポートされる

• エラー検出では、ターゲットアプリケーションごとにデータ ファイルが作成されます。エラー検出を開始する前に、ターゲットの実行可能ファイルを含むディレクトリへの書き込みアクセス権があることを確認する必要があります。

• 定義されていない評価順序に依存する式がコードに存在しないことを確認してください。評価順序を明確にしないでコードをインストゥルメントすると、エラー データ、ハング、さらにはクラッシュなどの予期しない結果が生じる可能性があります。

C/C++ 標準では、オブジェクトに値を保存するなどの「副次的な影響」がある場合、評価順序は明確に定義されません。たとえば、`i = ++i + 2;` のステートメントは評価順序が明確に定義されていません。

値が変数「i」に保存され、言語によって値の発生順序が定義されていない場合、このステートメントには2つの意味が存在します。このようにコードをインストゥルメントすると、評価順序が変わり、異なる結果が生じる場合があります。

• C++ アプリケーションでマネージ デバッガを使用してエラー検出を実行している場合、アプリケーションではなく、`mscorwks.dll` や `mscorsrv.dll` のメモリ割り当てとリソース割り当てが使用され、その結果、エラー検出でアプリケーションのメモリ割り当てとリソース割り当てが無視されます。混合コードのアプリケーションやマネージコードのアプリケーションでエラー検出を使用するには、以下の手順を実行します。

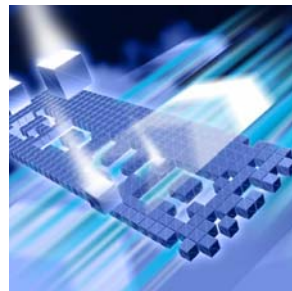
- 1 Visual Studio バージョン（2003 または 2005）を開き、プロセスのデバッグに使用します。
- 2 DevPartner エラー検出のスタンドアロン アプリケーションを起動します。
- 3 エラー検出内でターゲット プロセスを開き、**[プログラム]>[開始]** を選択してターゲット プロセスを実行します。
- 4 デバッガを以下のようにターゲット プロセスにアタッチします。
 - Visual Studio 2003 – **[ツール]>[デバッグ プロセス]** を選択します。
 - Visual Studio 2005 – **[ツール]>[プロセスにアタッチ]** を選択します。

メモ： **[マネージコード デバッグ]** を指定して（**[混合]** や **[自動]** ではなく）、デバッガがマネージコードのブレークポイントに正確に到達できるようにします。

これらの手順を完了すると、マネージコードのブレークポイントを設定し、そのブレークポイントに到達することができます。その後、エラー検出でメモリ割り当てとリソース割り当てが検出されます。

付録 B

重要なエラー検出ファイル



ファイルと用途

以下の表に、セッション時に動作を制御したり定義したりするためにエラー検出で使用されるファイルを示します。また、ファイルの場所、ファイル名、用途、ユーザーによる変更が可能かどうかについても説明します。

ファイル名とパス	用途
<code><Program Root>\¥Data¥CTISafe.dat</code>	ポインタ値を保存せずにポインタを受け入れる関数を指定します。MemTrackとFinalCheckでは、このファイルを使用してsafe関数を追跡します。この情報により、ポインタが不明な関数にアクセスしたときにエラーの発生が回避されます。関数がこのファイルにリストされている場合には、MemTrackとFinalCheckでは、関数にポインタのコピーが保存されていないものとみなします。 必要な場合には、関数を保存することができます。このリストからデフォルトの関数を削除する場合には、事前にコンピュータのテクニカルサポートに問い合わせてください。
<code><Program Root>\¥Data¥BCDefault.DPRul</code>	抑制のデフォルトセットと、エラー検出がロードするフィルタ ファイルをリストします。 エラー検出の抑制に関するダイアログとフィルタの編集に関するダイアログを使用して、このリストに追加することができます。追加した情報は、現在のプロジェクトディレクトリにのみ有効です。追加した情報をシステム全体で有効にするには、手でファイルを編集し、追加する抑制ファイルまたはフィルタ ファイルへのフルパスを指定します。
<code><Program Root>\¥Data¥*.DPFlt</code> <code><Program Root>\¥Data¥*.DPSup</code>	エラー検出で使用するフィルタと抑制を定義します。これらの .DPFlt ファイルと .DPSup ファイルにはそれぞれ、システム モジュール固有のフィルタと抑制が含まれます。 エラー検出の抑制とフィルタの編集ダイアログを使用して、抑制とフィルタの追加、変更、削除を行います。これらのファイルを手入力編集しないでください。

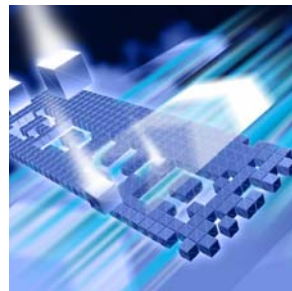
ファイル名とパス	用途
<Program Root>%Data%\Unrestricted_modules.txt	システム ディレクトリに配置されているモジュールの中で、制限のないDLLのモジュールを指定します。システム ディレクトリに既存のモジュールは、エラー調査対象外としてマークする必要があります。デフォルトでは、Unrestricted_modules.txtにMFCモジュールの各種バージョンがリストされます。 このファイルを手入力で編集し、システム ディレクトリに既存の特定モジュール名を追加すると、エラー検出で、それらのモジュール名にエラーがないか調べられます。
<Program Root>%Data%\UserAllocators.dat	カスタム アロケータを指定します。このファイルの詳細とエラー検出での使用方法については、第4章「ユーザーが作成したアロケータの使用」を参照してください。
<Program Root>%ERptApi%\NMApiLib.*	ユーザーによる呼び出しが可能なエラー検出インターフェイスのアクセスを提供します。NMApiLib.hには、ユーザーによる呼び出しが可能なエラー検出インターフェイスが定義され記述されています。このファイルは、NMApiLib.libをプロジェクトにリンクすることによって実装されます。ユーザーによる呼び出しが可能なインターフェイスの詳細については、第3章「複雑なアプリケーションの分析」を参照してください。
<Program Root>%Data%\ExcludedModules.dat	(ユーザー作成ファイル) 除外されたモジュールのリストが含まれます。各モジュールは、このファイルの別々の行にリストされます。 例: MYCUSTOMDRIVER.DLL
<Program Root>%DPSErrorDetection.xsd	セッション ファイル データのXMLへのエクスポート時に使用するスキーマ情報が含まれます。このファイルを編集しないでください。
<NMShared Root>%NMSymData.dat	(ユーザー作成ファイル) 不具合のあるPDBファイルに関連付けられたモジュール名が含まれます。モジュール名の後に「,0x0」が付きます。 例: ADVAPI32.DLL,0x0
<Program Root>%DPSErrorDetection.xsd	XML機能へのデータ エクスポートで使用するスキーマを指定します。

キー

<Program Root> = C:\Program Files\Compuware\DevPartner Studio\BoundsChecker

<NMShared Root> = C:\Program Files\Common Files\Compuware\NMShared\4.7

索引



A

ActiveCheck.....	47
ActiveX.....	35
コントロールのデバッグ.....	33
コンポーネント.....	28

B

BCDefault.DPRul.....	95
----------------------	----

C

CLR 分析.....	17
COM.....	
コンポーネント.....	28
サービス.....	35
使用状況.....	9
CTISafe.dat.....	95

D

dwWait.....	36
-------------	----

E

ExcludedModules.dat.....	96
--------------------------	----

F

FinalCheck.....	47
-----------------	----

I

IIS.....	29
プロセス.....	42
ISAPI フィルタ.....	12, 28, 33, 42, 43

N

NMApiLib.....	96
NMSymData.dat.....	96
NMVCBUILD.....	4

P

P/Invoke コール.....	17, 22
相互運用性の監視.....	17

S

StartEvtReporting.....	32
StopEvtReporting.....	32

U

Unrestricted_modules.txt.....	96
UserAllocators.dat.....	96

V

VCBUILD.....	4
--------------	---

W

Windows NT

サービス	28, 35
サービス コントロール マネージャ	37, 90
サービスのデバッグ	33

あ

アプリケーション

シングルスレッド	75
トランザクション	34
複雑な	28
マルチスレッド	25, 75

い

インターフェイス

コマンドライン	3
リーク	19

か

管理者権限	37, 43, 93
-------------	------------

く

クリティカル セクション、同期オブジェクト	76
-----------------------------	----

こ

構成ファイル管理	12
コール パラメータのデータ表示の深さ	21
コール バリデーション	16, 18
コマンドライン	3

さ

サードパーティ ソフトウェア	2, 19, 30
サービス、デバッグ	29
サービスのコントロール ロジック	36

し

実行可能ファイル

dllhost.exe	40
-------------------	----

重要なファイル

BCDefault.DPRul	95
CTISafe.dat	95
ExcludedModules.dat	96
NMApiLib	96
NMSymData.dat	96
Unrestricted_modules.txt	96
UserAllocators.dat	96

条件コード	30
-------------	----

す

スレッド、定義	75
---------------	----

せ

設定	12, 13
デフォルト	2
変更	2
潜在的なデッドロック	79

た

ダングリリング ポインタ	25
--------------------	----

て

テクニカル サポート サービス	ix
テスト コンテナ	34
デッドロック	77
潜在的	79
デフォルト設定	15

と

トラブルシューティング	85
トランザクション アプリケーション	34

ね

ネイティブ コード	11, 16, 17
ネイティブ コードのインストゥルメント	4

ふ

ファイル拡張子	
.DPFlt	95
.DPRul	95
.DPSup	95
フィルタ	30
複雑なアプリケーション	28
デバッグ	33
分析	18

ほ

ポインタ、ダンダリング	25
保護バイト	10

ま

マネージ コード	11, 16, 17
マルチスレッド アプリケーション	25, 75
マルチプロセッサ アプリケーション サーバー	25

め

メモリ	
オーバーラン	16
トラッカ	12
無効データでのフィル	10
リーク	19
メモリの無効データでのフィル	10, 24

も

[モジュール] タブ	20
モジュールとファイル	12, 30, 32
複雑なアプリケーション	19
リバース エンジニアリング	23

よ

抑制	30
----------	----

り

リソース トラッカ	12
リソース リーク	11, 19

ろ

ログ ファイル	21
---------------	----

わ

ワーカー スレッド	36
-----------------	----

