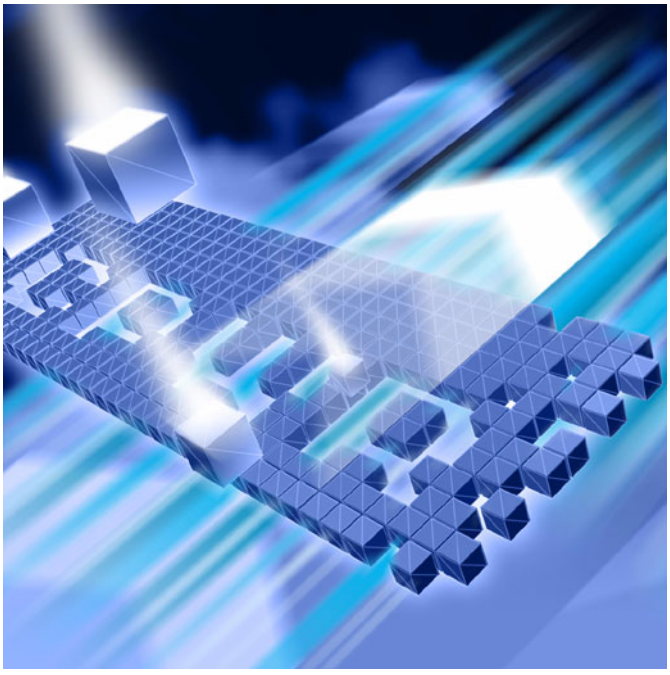


DevPartner®
エラー 検出 ガイド

リリース 9.0.2

Compuware

サポートが必要な場合は、以下のカスタマ サポート ホットラインにお電話をいただくか、弊社 FrontLine サポート Web サイトにアクセスしてください。

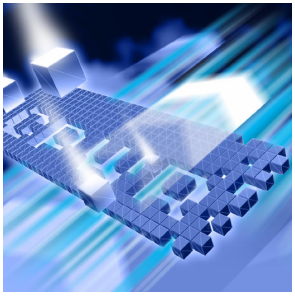
カスタマ サポート ホットライン：
1-800-538-7822

FrontLine サポート Web サイト：
http://frontline.compuware.com

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

発行日 2009.5.18

目次



はじめに

第 1 章
ワークフローと構成ファイルの設定

第 2 章
プログラムのチェックおよび分析

第 3 章
複雑なアプリケーションの分析

第 4 章
ユーザーが作成したアロケータの使用

第 5 章
デッドロック分析

付録 A

エラー検出のトラブルシューティング

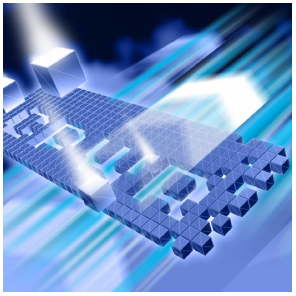
付録 B

重要なエラー検出ファイル

索引

日本コンピュウェア株式会社テクニカル・サポートのご案内

はじめに



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するため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、この製品を新規に使用するDevPartnerエラー検出のユーザー、および新しい機能とインターフェイスの変更の概要について確認するDevPartnerエラー検出の以前のバージョンのユーザーを対象としています。

新しいユーザーは、『DevPartner Studioユーザー ガイド』のエラー検出に関する章を参照して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エラー検出を最も効果的に使用する方法を学習してください。

以前のバージョンのDevPartner Studioのユーザーは、リリース ノートを参照して、DevPartnerエラー検出とBoundsChecker（以前のバージョンに付属のエラー検出ツール）の違いを確認してください。

このマニュアルでは、ユーザーがWindowsインターフェイスおよびソフトウェア開発のコンセプトに精通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

このマニュアルには、以下の章および付録が含まれています。

- 第 1 章「ワークフローと構成ファイルの設定」では、DevPartner エラー検出を設定して、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までのさまざまな問題を解決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および分析」では、DevPartner エラー検出で実行できるエラー検出タスクと、エラー検出以外の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チェックするときに DevPartner エラー検出をより効果的に使用する方法について説明します。
- 第 4 章「ユーザーが作成したアロケータの使用」では、独自のメモリ アロケータを分析できるように `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。この章では、これらのトピックに関する詳細情報を提供する Web アドレスと書籍も紹介します。
- 付録 A「エラー検出のトラブルシューティング」では、問題／解決法の形式でいくつかの最も一般的な問題の回答を示します。
- 付録 B「重要なエラー検出ファイル」では、DevPartner エラー検出に関連付けられた重要なファイルのリストを示し、各ファイルの目的について説明します。

また、このマニュアルの最後には索引があります。

メモ： **このマニュアルには、DevPartner Studio のすべての Visual Studio バージョンに関する情報が含まれています。テキスト内の「メモ」では、特定のリリースの Visual Studio でのみ使用可能な機能が特定されています。**

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 コマンドやメニュー名などは、[ ] で囲んで示します。以下に例を示します。

  [ツール] メニューから **[オプション]** を選択します。

- コンピュータのコマンドやファイル名などは、等幅フォントで示します。以下に例を示します。

  『DevPartner エラー検出ユーザー 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。以下に例を示します。

  [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。

テクニカル サポート

以下に、技術的な問題とそれ以外の問題についてテクニカル サポートに連絡する方法を示します。

技術的な問題以外

カスタマ サービスでは、アップグレードやその他の受注処理のご要望に関するどのようなご質問にもお答えします。カスタマ サービスは、月曜日から金曜日、米国東部標準時 午前8時から午後6時までご利用になれます。

北米 : 800 538 7822

その他の国 : +1 313 227 5444

その他の国のお客様は、Compuware Worldwide Offices（http://www.compuware.com/corporate/offices）で現地オフィスの電話番号を調べてください。

技術的な問題

インストールからトラブルシューティングに至るまで、テクニカルな問題についてはテクニカル 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をいただく前に、製品マニュアルと ReadMe ファイルで、関連する内容を確認してください。

テクニカル サポートへは以下の方法でお問い合わせいただけます。

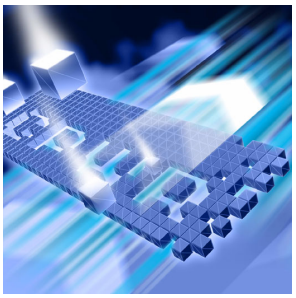
電子メール	詳細な情報をお知らせください。情報はすべて **DevPartnerSupport@compuware.com** に送信してください。
インターネットからのお問い合わせ	**http://frontline.compuware.com** のサポート技術情報にアクセスし、問題を送信してください。
電話	電話サポートは、有料（*）のテクニカル サポート サービスとして月曜日から金曜日、米国東部標準時 午前 8 時から午後 6 時までご利用になれます。製品のバージョン情報とライセンス情報をご用意ください。 米国とカナダでは、以下の番号におかけください。800 538 7822 その他の国では、以下の番号におかけください。+1 313 227 5444 * インストールとセットアップの問題については、無料でテクニカル サポートをご利用になれます。

テクニカル サポートにお問い合わせをいただく前に、以下の情報をご用意ください。

- 製品名（またはエディション）、バージョン、サービス パック
- 製品のライセンス情報
- システム構成：オペレーティング システム、ネットワーク構成、RAM の容量、環境変数、パス
- コンパイラとリンカの名称とバージョン、コンパイルとリンクに使用したオプション
- 問題の詳細情報：設定、エラー メッセージ、スタック ダンプ、診断ウィンドウの内容
- 問題再現手順の詳細（問題の再現が可能な場合）

第 1 章

ワークフローと構成ファイルの設定



- DevPartner エラー検出ワークフロー
- DevPartner エラー検出設定をカスタマイズする

DevPartner エラー検出では、さまざまなタイプの問題を特定できます。DevPartner エラー検出のデフォルト設定では、パフォーマンスへの影響を最小限に抑えて発生頻度の高いエラーを検出するよう設定されています。

特定のタイプの問題を検索するために、この設定を変更して DevPartner エラー検出を微調整できます。エラー検出設定を理解すると、DevPartner エラー検出を最大限に活用できます。

この章では、DevPartner エラー検出を設定して、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までのさまざまな問題を解決する方法について説明します。

メモ： **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、ターゲット実行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 エラー検出ワークフロー

DevPartner エラー検出は、以前のバージョンの DevPartner エラー検出のワークフローよりも広範なプログラム ワークフローに従います。このメカニズムを使用すると、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 つのステップがあります。

1. 必要な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. 監視対象の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 プログラムの実行時に、**[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るエラーを確認します。

   b 有効でないエラーを抑制します。

   c ログを表示し、必要に応じて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確認します。

3 データを表示します（プログラムの終了後）。

   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今後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、今後使用するために設定、抑制、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 エラー検出ワークフローでは、以下のことができます。

- 収集するデータのタイプと量の選択
- 監視対象のアプリケーションの部分の選択
- 既知の問題をレポートするエラー、条件付きコードによって処理されるエラー、サードパーティ コードで生成されるエラーの抑制
- ログ内の無関係な情報を非表示にするフィルタの作成
- 設定、抑制、フィルタを再利用するためのさまざまな構成の保存

DevPartner エラー検出では、ワークフロー プロセスのステップごとにデフォルトが用意されています。つまり、DevPartner エラー検出をデフォルト設定で使用することも、設定を変更して DevPartner エラー検出でのアプリケーションの分析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（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 リーク用の設定、COM リーク用の別の設定、詳細な **`lint`** タイプの分析を行うためのさらに別の設定を作成できます。より精密な設定を行い、大規模なアプリケーションの特定のセクションのみを調べる設定を定義できます。

コマンド ラインからエラー検出を使用する

BC.exe（実行可能ファイル）でプログラムをチェックするには、コマンド プロンプトから以下のコマンド構文を使用します。[ ] で囲んだコマンドはオプションです。

```
BC.exe [/?]

BC.exe sessionlog.DPbcl

BC.exe [/B sessionlog.DPbcl] [/C configfule.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
/?	使用状況を表示します。
sessionlog.DPbcl	既存のセッション ログを開きます。
/B sessionlog.DPbcl	バッチ モードで実行して、セッション ログをログ ファイルの **sessionlog.DPbcl** に保存します。
/C configfile.DPbcc	**configfile.DPbcc** オプションを使用します。
/M	**BC.exe** を起動し、実行中は最小化します。
/NOLOGO	**BC.exe** 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。
/X xmlfile.xml	XML 出力を生成して指定されたファイルに保存します。 • 実行ファイルを指定した場合は、エラー検出によって、その実行ファイル上の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（sessionlog.DPbcl）だけを指定した場合は、エラー検出によって、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**：実行ファイルを指定した場合は、/B スイッチを使用して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。
/XS xmlfile.xml	/X フラグと S 修飾子を一緒に使用すると、サマリ データだけが XML ファイルに保存されます。エラー検出セッションの実行に関する情報（セッション データ）はすべてエクスポートされます。
/XD xmlfile.xml	/X フラグと D 修飾子を一緒に使用すると、詳細データだけが XML ファイルに保存されます。エラー検出セッションの実行に関する情報（セッション データ）はすべて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
/OUT errorfile.txt	エラー メッセージを **errorfile.txt** という名前のテキスト ファイルに出力します。このファイルには、エラー検出によって検出されたエラーやリークのリストではなく、エラー検出を実行しようとしているときに生成されたエラー メッセージのみが含まれています。
/S	サイレント モードで実行します。エラー発生時に [検出されたプログラム エラー] ダイアログ ボックスを表示しないようにします。
/W workingdirectory	ターゲットの作業ディレクトリを設定します。
***target*.exe [target args]**	起動する実行可能ファイルとその引数

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。次に例を示します。

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

nmvcbuild を使用してネイティブ C/C++ コードをインストゥルメントする

コマンド ラインからプロジェクトを構築して、エラー検出用としてインストゥルメントする場合は、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 には、デフォルトのコンパイラとリンカを置き換える方法はないため、DevPartner のネイティブ C/C++ インストゥルメンテーションを実行できません。nmvcbuild.exe は、vcbuild のネイティブ C/C++ インストゥルメンテーションを実行できる、DevPartner 専用に設計されたコマンド ライン ユーティリティです。cl.exe と link.exe の起動を監視し、これらを nmcl.exe と nmlink.exe に置き換える vcbuild 用のラッパーとして機能します。

nmvcbuild ユーティリティは、vcbuild および nmcl と同じコマンド ライン パラメータを受け入れます。vcbuild と nmcl のパラメータは、コマンド ラインで vcbuild ? および nmcl ? と入力すると表示できます。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。この場合、nmvcbuild の呼び出し時に vcbuild パラメータのみを渡します。たとえば、次のエントリは、環境変数で nmcl パラメータを設定します。

```
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。

前提条件

nmvcbuild.exe を実行するには、以下の環境を整える必要があります。

- システムに DevPartner Studio をインストールする。
- Visual Studio ツールを実行するようにシステム環境を設定する。
- パス設定に vcbuild.exe と nmvcbuild.exe を含める。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

  メモ： 64 ビット バージョンの Windows では、このファイルは以下の場所にインストールされます。

  ¥Program Files (x86)¥Common Files¥Compuware¥NMShared

例

エラー検出のインストゥルメンテーションで sample プロジェクトのデバッグ構成をビルドするには

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

DevPartner エラー検出設定をカスタマイズする

DevPartner エラー検出設定では、以下のタイプのカスタマイズを行うことができます。

- 収集する情報のタイプの制限（メモリ リークやリソース リークなど）。
- 分析の主な各カテゴリで収集する情報のタイプの絞り込み（グラフィック コールによって生成されるリソース リークだけを検索するなど）。
- イベントや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の量の決定。
- DevPartner エラー検出ユーザー インターフェイスのルック アンド フィールの制御。これには、フォント、色、強調表示の変更や、**[検出されたプログラム エラー]** ダイアログ ボックスを表示するかどうかが含まれます。
- 以前に作成した DevPartner エラー検出設定の保存と復元。

DevPartner エラー検出設定をカスタマイズして、収集するデータの量と監視するアプリケーションの部分を制御します。

DevPartnerエラー検出設定は、以下のグループに分かれています。

- 全般
- データ収集
- API コール レポーティング
- コール バリデーション
- COM コール レポーティング
- 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 レポーティング
- 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]設定の下のチェック ボックスを使用して、以下を制御します。

- イベント ログ

  メモ： イベント ログ「サイレンス」エラー検出を無効にします。イベント ログを再度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。

- 各エラーに**[検出されたプログラム エラー]**ダイアログ ボックスを表示するかどうか。
- エラー検出の終了時または別のセッションの開始時にプログラム検証結果の保存を確認するプロンプトを表示するかどうか。
- ターゲット アプリケーションが存在する、または存在しない場合に**[メモリおよびリソース 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 モードで使用する場合にのみ使用可能）。
- コマンド ライン引数の指定（DevPartnerエラー検出をスタンドアロン モードで使用する場合にのみ使用可能）。

データ収集

[データ収集]設定を使用して、以下の機能を制御します。

- 各種のコール スタックの数
- スカラー以外のパラメータ（構造、クラス、ポインタなど）の保存するデータ量と戻り値

メモリが限られたコンピュータを使用している場合、または大規模で複雑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]**を制限してメモリの所要量を削減できます。

API コール レポーティング

[API コール レポーティング]設定を使用して、**[API コール レポーティングを有効にする]**が選択されている場合に記録する Windows API コールのタイプを制御します。Windows メッセージのログを制御することもできます。

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 コールを選択して有効にします（たとえば、グラフィック コールを記録するには GDI32 を選択）。

コール バリデーション

[コール バリデーション]設定を使用して、DevPartner エラー検出で Windows API パラメータと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、DevPartner エラー検出はパラメータを検証しません。

メモリの使用も追跡している場合は、**[メモリ ブロック チェックを有効にする]**を選択できます。このオプションを選択すると、DevPartner エラー検出ではメモリ追跡システムから収集し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、パフォーマンスに影響を与えます。

DevPartner エラー検出には、Windows API で実行される検証のタイプを制限できる設定が含まれています。これらの設定を使用すると、不正なエラーを生成する可能性のあるエラーのカテゴリを選択解除できます。この例には、フラグ チェック、範囲チェック、列挙チェックがあります。ハンドルとポインタを詳細に分析するが、その他のタイプの検証は必要ない場合は、これらのオプションを確認してください。

DevPartner エラー検出では、チェックする 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。限定的な API コールのセットが対象の場合は、それらのモジュールだけを選択します。これにより、検出されるエラーの数は減少しますが、パフォーマンスは向上します。

COM コール レポーティング

[COM コール レポーティング]設定を使用して、**[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]**が選択されている場合に記録する COM インターフェイスを制御します。

デフォルトでは、**[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]**が選択されている場合、DevPartner エラー検出ではすべての既知の COM インターフェイスがレポートされます。パフォーマンスを向上させるには、チェックする必要がある COM インターフェイスだけを選択します。**[COM コール レポーティング]**の下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 ファイルのサイズが減少し、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 エラー検出では、アプリケーション内の COM の使用を監視し、インターフェイスをリークしているすべてのコードをレポートできます。インターフェイス リークが検出された場合、DevPartner エラー検出によってアプリケーション内のすべての **`AddRef`** と **`Release`** を示した COM 使用回数グラフが表示されます。このグラフを使用して、欠落している **`AddRef`** コールや **`Release`** コールをアプリケーションの情報に基づいてすばやく特定できます。

デフォルトでは、DevPartner エラー検出では COM オブジェクトの追跡は有効ではありません。この機能を有効にするには、**[COM オブジェクトの追跡を有効にする]**を選択します。COM オブジェクトの追跡が有効な場合、**[すべての COM クラス]**を選択することも、表示されたリストからクラスを個別に選択することもできます。

デッドロック分析

マルチスレッドのアプリケーションでデッドロックを監視するには、**[デッドロック分析]**を使用します。これには次のような分析が含まれます。

- アプリケーションでデッドロックの発生を監視してレポートします。
- アプリケーション内で同期オブジェクトの使用パターンを監視して潜在的なデッドロックを検出します。
- アプリケーションを監視して同期オブジェクトのエラーを検出します。

メモリの追跡

[メモリの追跡]設定を使用して、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。**[メモリの追跡]**は、デフォルトで有効です。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。

[メモリの追跡]設定はあらかじめ設定されており、大部分のアプリケーションについて適切な結果が生成されます。**[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をフィルする]**の各設定については特に注意が必要です。

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場合、**[FinalCheck を有効にする]**をオンにしても影響はありません。**[Instrumenting for Error Detection（エラー検出用にインストゥルメントする）]**を選択した場合は、FinalCheck がデフォルトでオンになります。FinalCheck を実行せずにインストゥルメンテーションを有効にするには、**[エラー検出設定]**の**[メモリの追跡]**ペインで FinalCheck を無効にする必要があります。

[FinalCheck を有効にする]をオンにしたままにし、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で詳細でない ActiveCheck 分析を実行する場合にのみオフにして使用することをお勧めします。

保護バイト

保護バイトは、ActiveCheck 分析でのメモリ オーバーランの検出に使用されます。ヒープ破壊が発生し、DevPartner エラー検出で問題が検出されない場合は、**[回数]**設定をより大きい値にすることを検討してください。これらの設定を使用して検索の困難なヒープ エラーを追跡する方法のヒントについては、オンライン マニュアル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をフィルする]

[確保時にフィルする]では、メモリが割り当て時の既知の状態に設定されます。**[解放時に無効データをフィルする]**では、メモリが解放時の既知の状態に設定されます。

使用されているバイト パターンは、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に、慎重に選択されています。これらの設定の詳細については、オンライン マニュアル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場合、またはグローバル **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」を参照し、以下のファイルのドキュメント（コメントの形式）を確認してください。

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

メモ： 64 ビット バージョンの Windows では、このファイルは以下の場所にインストールされます。**¥Program Files (x86)¥Compuware¥DevPartner Studio¥BoundsChecker¥Data¥UserAllocators.dat**

.NET コール レポーティング

[.NET コール レポーティング]設定を使用して、**[.NET メソッド コール レポーティングを有効にする]**が選択されている場合に記録する .NET アセンブリを制御します。

.NET コール レポーティングと COM コール レポーティングを組み合わせると、両方の側の COM 相互運用を確認できます。

.NET ユーザー アセンブリと .NET システム アセンブリは、ツリー ビュー コントロールの別の分岐に表示されます。

パフォーマンスのヒント：

.NET コール レポーティングは、大量のデータを生成して、システムを減速させる可能性があります。フレームワークをデバッグして理解する必要がある場合にだけ .NET コール レポーティングを有効にして、チェックが必要なアセンブリだけを選択します。**[すべてのタイプ]**ツリー ビューで選択するアセンブリの数を減らせば、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。

.NET 分析

DevPartner エラー検出は、ネイティブ アプリケーションとマネージ アプリケーションの混在をサポートしています。混在環境で作業している場合は、**[.NET ランタイム分析を有効にする]**を選択できます。DevPartner エラー検出は以下のタイプの .NET 分析をサポートしています。

- ネイティブ コードからマネージ コードに渡された未処理の例外の監視
- .NET ファイナライザの分析
- マネージ コードからネイティブ コードへの相互運用性
- ガベージ コレクション イベントの監視

.NET 相互運用性

DevPartner エラー検出の .NET 相互運用性機能では、マネージ コードからネイティブ コードへのアプリケーション移行の回数が監視されます。この情報を使用して、使用パターン、およびマネージ コードで書き直すことが効果的なターゲットのネイティブ コードを分析します。最良の結果を得るには、この機能を**[相互運用性レポートのしきい値]**パラメータと共に使用して、条件を満たした使用の独自の下限を指定します。

リソースの追跡

[リソースの追跡]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、デフォルトで選択されています。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。

リソースの追跡を選択した場合、すべてのリソース リークを検索することも、Windows API の特定のライブラリに関連付けられた特定のリソースに限定して検索することもできます。

リソースはライブラリ別にグループ化され、各ライブラリ内でリソースの解放に使用された API コール別にグループ化されています。たとえば、レジストリを操作する大量のコードを作成した場合、**ADVAPI32** 以外のすべてのライブラリを選択解除し、**RegCloseKey** だけを選択することもできます。

モジュールとファイル

[モジュールとファイル]の設定を使用して、以下の操作を実行します。

- アプリケーション内の監視または無視する必要がある実行ファイルとライブラリを特定します。
- シンボルを使用できる場合、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ソース ファイル レベルまで絞り込みます。
- DevPartner エラー検出アナライザで無視する必要がある**システム ディレクトリ**のリストを特定します。

[モジュールとファイル]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ときに**[モジュールとファイル]**設定の使用を検討します。

> **メモ：[モジュールとファイル]設定内のすべてのモジュールを無効にした場合は、エラーの種類によって報告されるものとされないものがあります。エラー検出では、任意のモジュール内のメモリ オーバーランと、`MFC`*xxxx*`.dll` ライブラリが原因で発生したその他のイベントは必ずレポートされます。**

詳細については、「[モジュールとファイル] 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]設定を使用して、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、強調を変更します。

構成ファイル管理

[構成ファイル管理]を使用して、プロジェクトごとに複数の設定ファイルを作成します。図 1-1（12ページ）に、使用可能な[構成ファイル管理]オプションを示します。ソフトウェア開発サイクル全体をとおしてこれらの設定ファイルを使用して、さまざまなタイプの分析を実行できます。作成できる以下の設定ファイルの例を検討してください。

- **[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、ユーザーのコンポーネントのみを選択します。アプリケーションに新しいコードを追加するときに、これらの設定を日常的に使用します。
- 新しいコンポーネントを完成させるとき、または既存のコンポーネントに比較的大きな変更を加えるとき、**[メモリの追跡]**と**[リソースの追跡]**の設定を使用します。
- 主なマイルストーンの結果を分析するために週末にバッチ　モードで使用する設定ファイルを作成します。ビルドをFinalCheckでインストゥルメントして、レポートの分析時に最も詳細な情報を取得することもできます。
- さまざまなモジュールのセットを選択し、すべての分析機能を無効にした設定ファイルを作成します。この設定ファイルをロードし、対話的な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、複雑なモジュールやファイルの設定を管理する必要がある場合に特に役立ちます。

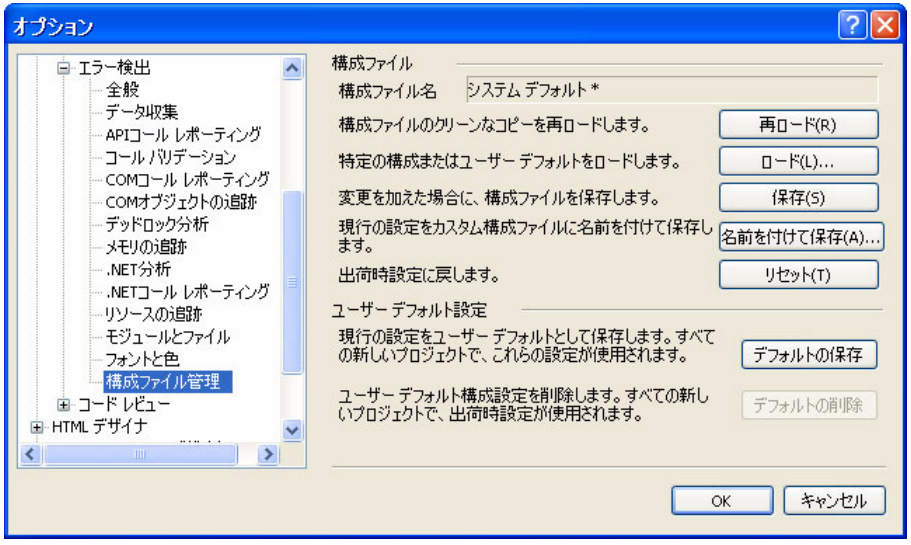


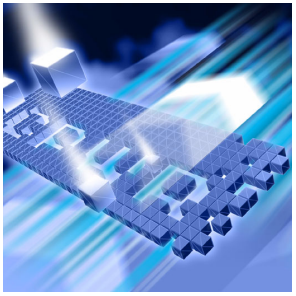
図 1-1.　構成ファイル管理設定

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：**すべての変更を破棄して、現在の構成ファイルを再びロードします。この操作では、最後に保存された状態の構成ファイルがロードされます。
- **ロード:[ロード元]**ダイアログ ボックスを開きます。**[内部ユーザー デフォルト]**を選択して、ユーザーのデフォルト設定をロードします。**[構成ファイル]**を選択すると、**[構成ファイルのロード]**ダイアログ ボックスが開きます。このダイアログ ボックスで、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の変更内容をすべて保存します。
- **名前を付けて保存：[構成ファイルの保存]**ダイアログ ボックスを開きます。このダイアログ ボックスから、現在の構成オプションを別のファイル名で保存します。
- **リセット：**プログラム プロパティ設定をすべて出荷時のデフォルト設定に戻します。
- **デフォルトの保存：**現在の設定をユーザーのデフォルトとして保存します。新しいプロジェクトのすべてでこの設定が使用されます。
- **デフォルトの削除：**ユーザーのデフォルトの構成設定を削除して、出荷時の設定に戻します。新しいプロジェクトのすべてで出荷時の設定が使用されます。

第２章
プログラムのチェックおよび分析



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

通常、DevPartner エラー検出には以下のようなタスクがあります。

- メモリ リーク、リソース リーク、インターフェイス リークを検索する
- ポインタ エラーとメモリ エラーを検索する
- メモリ破壊を検索する
- .NET アプリケーションでレガシー コードを分析する
- Win32 API コールを検証する
- アプリケーションのデッドロックを検索する

リークを検索する

DevPartner エラー検出は、メモリ リーク、リソース リーク、インターフェイス リークの検索に優れています。デフォルトでは、DevPartner エラー検出ではメモリ リークとリソース リークが検索され、インターフェイス リークは検索されません。インターフェイス リークを検索するには、**[COM オブジェクトの追跡]** 設定で **[COM オブジェクトの追跡を有効にする]** を選択します。

DevPartnerエラー検出には、ActiveCheckとFinalCheckの2つのメモリ リークを検出する方法があります。ActiveCheckでは、すべてのWindowsアプリケーションのメモリ リークを検索します。リークはアプリケーションのシャット ダウン時にレポートされます。FinalCheckでは、メモリ リークがアプリケーションで発生すると、実行時に包括的にレポートされます。この例には、ローカル変数が範囲外になった場合、メモリのブロックへの最後のポインタが再割り当てされた場合、ダングリング ポインタの使用、およびその他の検索の困難なエラーがあります。

ポインタ エラーとメモリ エラーを検索する

DevPartnerエラー検出では、ActiveCheckテクノロジとFinalCheckテクノロジの両方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、DevPartnerエラー検出ではWindowsコールに渡されたポインタを監視してエラーを検索します。**[コール バリデーション]**と**[メモリの追跡]**の設定を変更して、DevPartnerエラー検出で実行されるチェックの量を設定します。

FinalCheckを使用してプログラムを再コンパイルすると、DevPartnerエラー検出はプログラム内のすべてのポインタ参照で使用法が正しいかどうかをチェックします。FinalCheckでは、プログラムの非常に詳細な分析を行い、未初期化変数、ダングリング ポインタ、非関連ポインタの比較、配列のインデックス エラーなどの検出が困難な問題を特定できます。

メモリ破壊を検索する

DevPartnerエラー検出は、以下のような問題によって発生するメモリ破壊の問題を検索する場合に役立ちます。

- オーバーランを割り当てられたバッファ
- 解放後のメモリへの継続的なアクセス
- リソースの複数回の解放（二重解放など）

DevPartnerエラー検出ではActiveCheckモードでこれらのエラーの多くを検出できますが、FinalCheckでは最も詳細な分析を実行できます。

メモリ オーバーラン エラーが発生し、ActiveCheckしか使用できない場合は、**[メモリの追跡]**設定の**[実行時のヒープ ブロック チェック]**についての詳細情報を含むオンライン マニュアルを参照してください。

.NETアプリケーションでのレガシー コードへの移行を分析する

DevPartnerエラー検出では、ネイティブ アプリケーション プログラミングからマネージ アプリケーション プログラミングに移行する場合に役立つ以下のような分析を実行できます。

- Windowsアプリケーションのネイティブ部分の詳細な分析
- 混合コードを使用するアプリケーションのネイティブ部分とマネージ部分の間の移行レイヤの分析

- マネージ アプリケーションでのファイナライザの分析

これらのタイプの分析では、以下の対象を監視できます。

- ネイティブ アプリケーションからスローされマネージ コードに渡された未処理の例外
- パフォーマンスの問題を引き起こす可能性のあるガーベジ コレクタの動作
- 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 コードからネイティブ Windows ライブラリへの P/Invoke のコール
- マネージとネイティブの境界にわたるコールの頻度

この情報を使用して、アプリケーションの移行プロセスを計画し、監視できます。

ネイティブ コードから混合コードまたはマネージ コードへ移行する

移行プロセスでは以下の手順を実行します。

1. ネイティブ アプリケーションの COM 使用を分析して、使用されているオブジェクトを確認します。
2. P/Invoke と COM を使用してアプリケーションの部分を書き換え、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 **[PInvoke 相互運用性の監視]** を選択して、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. 必要な変更を加えます。
5. **[.NET 分析]** で、**[COM 相互運用性の監視]** と **[PInvoke 相互運用性の監視]** を選択して、マネージ コードとネイティブ コード間で行われたコールの回数を監視します。パフォーマンス データを使用すると、以下の追加の変更について決定する際に役立ちます。
   a. マネージ コードに移植する必要がある COM オブジェクトを判別する。
   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 コード間のコールの数を削減する必要があるかどうかを決定する。たとえば、データ レコードを一度に 1 項目ではなく 10 ～ 20 項目要求するメソッドを追加できます。
   c. ネイティブ API（Windows API など）へのコールが効率的に行われているかどうかを判別する。

ネイティブからマネージの境界にわたってスローされる未処理の例外をチェックすることもできます。そのためには、**[.NET 分析]** で **[例外の監視]** を選択します。ネイティブ コードで記述されたアプリケーションでは、例外によって呼び出し側にコールまたはメソッドが失敗したことが通知されます。アプリケーションの部分がマネージ コードで書き換えられたときに、例外の使用を監視して例外がマネージ コードに移行する前にキャッチします。

Win32 API コールを検証する

DevPartner エラー検出では、多数の Windows コールが認識されます。DevPartner エラー検出では、この機能を使用して、ポインタ、フラグ、列挙、ハンドル、リターン コードを検証できます。**[コール バリデーションを有効にする]** を選択して、アプリケーションで Windows コールが適切に使用されていることを確認します。

以下の**コール バリデーション**機能を設定できます。

- 監視対象の Windows コールを選択する。
- フラグ、範囲、列挙のチェックなどのさまざまなタイプの検証を選択的に無効にする。

これらの機能を使用して、ハンドルやポインタなどの重要なパラメータを検証し、現在のタスクに関係ないエラーのレポートが減少するように DevPartner エラー検出を設定できます。

アプリケーションのデッドロックを検索する

DevPartner エラー検出では、アプリケーションで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、デッドロックを検索します。追加のコントロールを使用すると、デッドロック分析を微調整できます。

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補助
- 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 テスト用ツール

複雑なアプリケーションを理解する

DevPartner エラー検出には、大規模で複雑なアプリケーションの理解に役立つ複数のツールがあります。以下の 3 つのシナリオを考えてみます。

- 既存のチームに新しい開発者が加わり、各種 DLL の対話方法を理解する必要があります。
- 問題（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が呼び寄せられ、厳しいエンジニアリングのスケジュールを前提に最も多くのリソースを投入する場所を理解する必要があります。
- 開発者がサードパーティ製ライブラリの使用を開始し、ライブラリで Windows リソースがリークしている理由を知る必要があります。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。

以下の DevPartner エラー検出機能を使用して、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、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 オブジェクトが使用されます。これらの COM オブジェクトが正しく使用されない場合、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 リークによってメモリ リークとリソース リークが引き起こされます。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、そのためそれらのオブジェクトによって割り当てられたメモリが適切に解放されません。

[COM オブジェクトの追跡] を使用して、リークした COM オブジェクトを表示できます。この情報は、アプリケーションの **`AddRef`** に対応する欠落した **`Release`** コールを行う場所を特定する場合に役立ちます。

デッドロック アナライザ

デュアル プロセッサが普及する前に記述された多くのレガシー アプリケーションは、現在の高パフォーマンスなコンピュータ システム上で実行されると予期しない動作をすることがあります。たとえば、アプリケーションは同期オブジェクトの不適切な使用によってデッドロック状態になることがあります。

DevPartner エラー検出の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の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も特定できることに注意してください。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一連の望ましくない状態が進展した場合、いつ発生してもおかしくないデッドロックのことです。DevPartner エラー検出では、これらの潜在的なデッドロックが実稼働環境で発生する前に特定できます。

モジュールとファイル

複雑なアプリケーションは、多くの場合複数の組織にわたって開発され、外部ベンダから購入したライブラリを含んでいます。デフォルトで、DevPartner エラー検出ではシステム以外の DLL のエラーがレポートされます。**[モジュールとファイル]** 設定を使用して、DevPartner エラー検出のエラー 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部分に制限します。その結果、複雑な問題の解決に使用できるより有効なエラー レポートが生成されます。

> **メモ：** **[モジュールとファイル]** 設定内のすべてのモジュールを無効にした場合は、エラーの種類によって報告されるものとされないものがあります。エラー検出では、任意のモジュール内のメモリ オーバーランと、`MFC`*`xxxx`*`.dll` ライブラリが原因で発生したその他のイベントは必ずレポートされます。

[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）および関連する詳細ペインには、プログラムへのビューが表示されます。このビューには、プログラムの実行中にロードされているDLLが表示されます。このレポートをよく確認すると、以下の質問に回答し、トレードオフを行う必要がある場合により的確な決定を下すことができます。

- このモジュールはインストゥルメントされているか。また、その方法。
- 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。

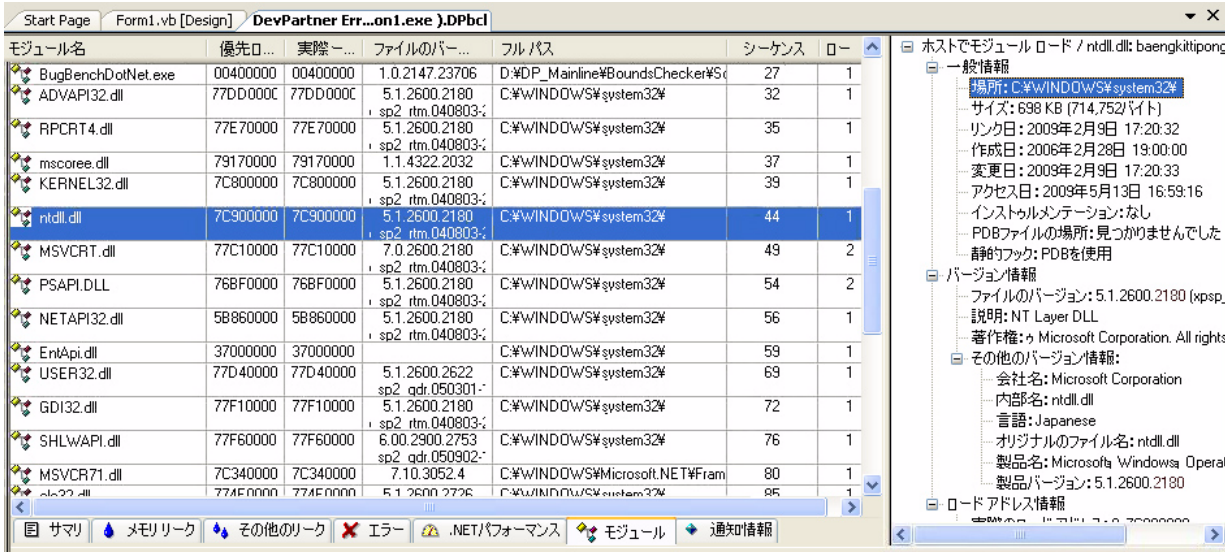


図 2-1. [モジュール]タブと詳細ペイン

検証結果ペインでの表示と並べ替え

DevPartnerエラー検出には、アプリケーションについて収集されたデータを表示するさまざまな方法があります。最初に、DevPartnerエラー検出では、**検索結果**ペインに高レベル レポートの**[サマリ]**タブが表示されます。**[サマリ]**タブを確認し、エントリをダブルクリックすると、詳細が表示されます。

情報の複数のレイヤを移動するこの機能では、データのさまざまなビューが表示されます。次に例を示します。

- 技術リーダーは、ある期間におけるメモリ リークがより多いかより少ないかなどの傾向を探してデータを確認することがあります。
- 開発者は、メモリ オーバーラン エラーやダングリング ポインタなどを収集する必要がある場合があります。

このマルチレベルのビューでは、最も関連の深いデータを特定し、**検索結果**ペインのいずれかのタブ（**[メモリ リーク]**、**[その他のリーク]**、**[エラー]**、**[.NET パフォーマンス]**、または**[モジュール]**）でより詳細なビューにアクセスできます。いずれかのタブでデータを表示するときに、カラム見出しをクリックすると、データをサイズ、発生回数、場所などでさらにソートできます。

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。このセクションで説明するような設定がある構成を作成すると、DevPartner エラー検出を使用して Windows アプリケーションによって実行される処理を監視し、レポートできます。

データ収集

[コール パラメータのデータ表示の深さ] パラメータを大きくして、より詳細な API パラメータ情報を生成します。データ表示の深さを大きくすると、処理速度が遅くなり、ログ ファイルのサイズが増加することがあります。

API コール レポーティング

[API コール レポーティングを有効にする] を選択して、API コールと戻り値をログに記録します。DevPartner エラー検出がパラメータとパラメータとして渡されるクラスについて収集する詳細の量は、**[データ収集]** 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。このオプションを選択すると、アプリケーションがマウス クリックやイベントの再ペイントなどのさまざまなウィンドウ イベントに応答する方法のビューが表示されます。

> **メモ：** **これらのオプションのいずれかを選択すると、ログ ファイルのサイズが増加し、DevPartner エラー検出のパフォーマンスが遅くなりことがあります。**

API コール レポーティングのオーバーヘッドを最小限に抑えるには、現在のタスクに最も関連の深いシステム DLL のみを選択します。

COM コール レポーティング

[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]をオンにして、COM メソッド コールの収集を有効にします。

COM コール レポーティング情報を管理可能にするには、最も関連の深いインターフェイスのみを選択し、**[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 レポーティング

[.NET メソッド コール レポーティングを有効にする]をオンにして、.NET メソッド コールの収集を有効にします。.NET コール レポーティングを管理可能にするには、.NET ユーザー アセンブリのみを選択します（デフォルト）。

.NET 分析

ネイティブとマネージの混合コードのアプリケーションを記述する場合は、**.NET 分析**機能を使用して、以下のことを行います。

- ネイティブ コードからマネージ コードにスローされた未処理の例外を監視します。
- マネージ コードからネイティブ コードへ行われたコール（P/Invoke コールまたは COM メソッド コール）を監視します。
- **[例外の監視]**を選択して例外を監視します。

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には、**[COM 相互運用性の監視]**または**[PInvoke 相互運用性の監視]**を選択してから、**[相互運用性レポートのしきい値]**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場合は、レポートのしきい値に十分大きい値を選択し、**[モジュールとファイル]**設定を使用して不要な情報を減らします。

オフにするリバース エンジニアリングの機能グループ

ヒント: リバース エンジニアリング セッションの終了後は、これらの機能を必ず選択してください。

DevPartner エラー検出には、Windows アプリケーションでさまざまなタイプの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 セッション中は、DevPartner エラー検出のエラーとリーク検出のロジックをオフにすることをお勧めします。これらの機能を**[プログラムの設定]**ダイアログ ボックス（DevPartner エラー検出スタンドアロンの場合）または**[オプション]**ダイアログ ボックス（Visual Studio IDE の場合）で無効にするには、以下の手順を実行します。

1. **[コール バリデーション]**で、**[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]**で、**[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]**で、**[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]**で、**[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]**で、**[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、調べているコードのバグを特定するためのものです。これらの機能をオフにすると、コンポーネントまたはAPIのコードが機能する方法を理解するのに役立つ情報に集中できます。

モジュールとファイル

デフォルトで、DevPartnerエラー検出では**[システム ディレクトリ]**除外リストに表示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します。

リバース エンジニアリングを行う場合、通常は除外されるDLLを監視できます。DLLを監視すると、そのDLLを追跡して動作方法を確認できます。

たとえば、特定の共通コントロールでWIN32 APIコールが使用される方法を理解するには、**COMCTL32.DLL**を明示的に追加し、**[APIコール レポーティング]**を有効にします。

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、希望のDLLを追加します。

構成ファイル管理

[構成ファイル管理]を使すると、開発サイクルで特別なタスク用に設計された設定を作成し、保存できます。

次に例を示します。

- メモリ リーク、リソース リーク、およびCOMリークの検出
- メモリとバリデーションのみ
- リバース エンジニアリング
- 上記のいずれか。ただし、カスタム**[モジュールとファイル]**設定を使用して限定されたDLLのセットあり。

DevPartnerエラー検出でアプリケーションのビジネスにきわめて重要な部分（パスワードのチェックなど）が監視され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、DevPartnerエラー検出ログを選択的に無効にできます。詳細については、以下のファイルのイベント レポートに関するコメントを参照してください。

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

メモ： 64ビット バージョンのWindowsでは、このファイルは以下の場所にインストールされます。

¥Program Files (x86)¥Compuware¥DevPartner Studio¥BoundsChecker¥ErptApi¥NmApiLib.h

DevPartnerエラー検出の実行に伴い、大きい負荷がかかっている状態でのみ発生する多数の予期しない状況をアプリケーションで処理するという副作用が発生します。

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が何らかの値に初期化されるという誤った想定のもとに記述されています。DevPartnerエラー検出では、未初期化データ アクセスの検索を割り当てられると、さまざまなタイプのメモリ上に既知の充てんパターンを書き込みます。この例には、ローカル変数、および**new**、**malloc**、**HeapAlloc**、または**LocalAlloc**によって割り当てられたメモリがあります。

未初期化メモリがゼロになると想定してアプリケーションが記述されている場合、DevPartnerエラー検出で実行するとプログラムがクラッシュしたり、予期しない動作をすることがあります。そのような場合は、アプリケーションをFinalCheckでインストゥルメントし、DevPartnerエラー検出で再度チェックしてエラーを特定します。

> **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は、UserAllocators.datファイルにルーチンのエントリを追加します。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

DevPartnerエラー検出では、動的に割り当てられたメモリが解放されたあと、そのメモリに既知のパターンを書き込みます。これにより、解放された構造を参照しようとするアプリケーションではエラーが生成されます。多くの場合、ダングリング ポインタ エラーは診断と修復が困難です。アプリケーションをFinalCheckでインストゥルメントし、DevPartnerエラー検出で再度チェックしてエラーを特定します。

> **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は、UserAllocators.datファイルにルーチンのエントリを追加します。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、DevPartnerエラー検出では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 コードでのプログラムの検出

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 サーバーを利用するように記述されています。マルチスレッドのアプリケーションが綿密に設計されていない場合、プログラムが負荷のかかる状況に置かれたとき、デッドロックとリソース喪失の問題が発生することがあります。

マルチスレッドのアプリケーションを DevPartner エラー検出で実行すると、各種のスレッドのパフォーマンスが低下し、プログラムによってタイミング関連の問題が表示されることがあります。通常、このような問題の多くは、実稼働状況でプログラムに負荷がかけられた場合に発生します。DevPartner エラー検出を使用すると、開発プロセスで問題を発見し、実稼働に移行する前に修正できます。

DevPartner エラー検出でデッドロック分析を有効にしてアプリケーションを実行し、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チェックします。

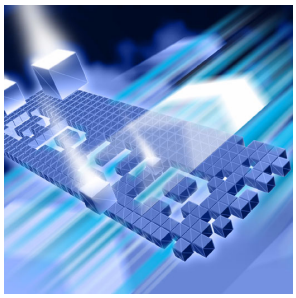
メモリ エラーとポインタ再利用エラーの検出

アプリケーションが複雑になるにつれて、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は大幅に増加します。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、リークの検索とプラグはタスクの一部に過ぎません。メモリが解放されると、ブロックに対するすべての未処理のポインタは「ダングリング」として宣言される必要があります。ダングリング ポインタを参照しようとすると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、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 ポインタは、プログラムで、解放されたブロック、または解放され、システムのその他の部分で再利用されたブロックが参照される原因となります。単純なデバッグ環境で実行されるプログラムでは、失敗の兆候が示されないことがあります。ただし、同じプログラムが実稼働環境に移されると、ランダムにクラッシュしたり、データが破損したり、予期しない結果を生成したりすることがあります。

第 3 章

複雑なアプリケーションの分析



この章では、複雑なアプリケーションをチェックするときに DevPartner エラー検出をより効果的に使用する方法について説明します。

複雑なアプリケーションについて

典型的なWindowsアプリケーションをデバッグする場合、デフォルトのDevPartnerエラー検出設定では、最も一般的なプログラミング問題を解決するのに十分なデータが収集されます。

複雑なアプリケーションをデバッグする場合、エラー検出設定をカスタマイズすると効果的です。

複雑なアプリケーションは次の2つのグループに分類できます。

- 多数の複雑なコンポーネントを含む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コンポーネント、COMコンポーネント、ISAPIフィルタなどの非従来型のアプリケーション

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、サイズのために監視が困難であるという点のみが例外的です。DevPartnerエラー検出を使用すると、大規模なアプリケーション全体を一度に分析するのではなく、論理的で管理可能な区分で分析できます。たとえば、大規模なアプリケーションの1つのDLLを作成する場合、以下のことができます。

- アプリケーションの部分を分析から除外する。
- アプリケーションの特定の部分だけを監視する。
- 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する。

非従来型のアプリケーション

非従来型のアプリケーションでは、複雑なスタートアップや設定の問題により、さまざまなエラー検出方法が必要な場合があります。DevPartnerエラー検出を設定して、このようなアプリケーションの監視に必要な特別なデバッグや分析の操作を実行できます。

DevPartnerエラー検出機能と複雑なアプリケーション

以下のエラー検出機能は複雑なアプリケーションを分析する場合に役立ちます。

- 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限定する機能
- 実行時にエラー検出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] を使用している場合は、起動するアプリケーションのフル パス名が、エラー検出で検索されるアプリケーションのフル パス名と完全に一致する必要があります。**

メモ： **このオプションは、BoundsChecker や DevPartner エラー検出の以前のリリースにおける Image File Execution Options の代わりに使用する機能です。**

このオプションは、DevPartner エラー検出スタンドアロン アプリケーション（BC.EXE）を使用している場合にのみ利用できます。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 エラー検出を構成し、興味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。

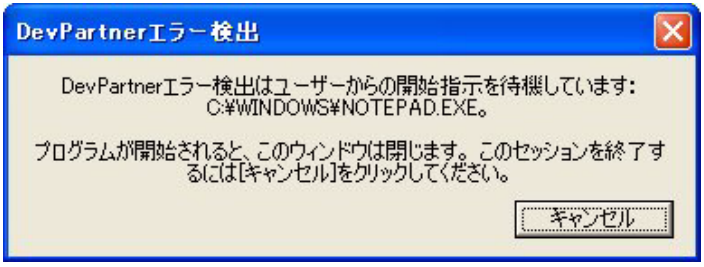


図 3-1. **[プロセスを待機] ダイアログ ボックス**

4 通常どおり、アプリケーションを起動します。

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起動している場合は、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、プログラムの限定的な部分を分析する場合に役立つ以下の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、分析からプログラムの部分を除外します。
- **[抑制]**と**[フィルタ]**を使用して、不要な情報が記録されたり表示されたりするのを防ぎます。
- **[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、エラー検出のログを切り替えます。
- アプリケーションに条件付きコードを追加して、**`StartEvtReporting`**と**`StopEvtReporting`**を呼び出します。

  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エラー検出がアクティブでない場合、これらのコールはすぐに返されます。

モジュールとファイル

大規模なアプリケーションを使用している場合は、**[モジュールとファイル]**設定を使用して、アプリケーションの部分が分析されるのを防げます。これにより、分析時間を短縮し、不要なエラー メッセージの数を減らせます。以下に除外できる部分の一部を示します。

- サードパーティ製DLLを含む不要な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 コードが使用できない場合は、エラーを除外します。

  メモ： **[モジュールとファイル]**設定内のすべてのモジュールを無効にした場合は、エラーの種類によって報告されるものとされないものがあります。エラー検出では、任意のモジュール内のメモリ オーバーランと、`MFC`*`xxxx`*`.dll`ライブラリが原因で発生したその他のイベントは必ずレポートされます。

「[モジュールとファイル]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エラー検出によってレポートされるエラーとイベントを非表示にするには、以下の2通りの方法があります。

- **[抑制]**では、指定したタイプのエラーやイベントがエラー検出ログに記録されなくなります。抑制されたエラーを表示するには、抑制のインストラクションを削除し、DevPartnerエラー検出でアプリケーションを再度実行する必要があります。
- **[フィルタ]**では、すでにログに記録されているエラーやイベントが非表示になります。フィルタしたエラーの表示と非表示を切り替えられます。

選択的イベント ログ

大規模なアプリケーションの小さい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とオフを切り替えます。この手法は以下の設定を選択した場合に特に役立つことがあります。

- APIまたはCOMコール ログ
- コール バリデーション

選択的イベント ログをいずれかのリーク検出機能（メモリ追跡、リソース追跡、COMインターフェイス追跡など）と共に使用する場合は、多くのリークがプログラムの終了時まで検出されないことに注意してください。プログラムの終了時にログが無効になっていると、検索しようとしているリークの多くはレポートされません。

リークを検出する場合は、**[モジュールとファイル]**や**[抑制]**を使用して、不要な情報を除外します。

条件付きコード

プログラムを変更してDevPartnerエラー検出データ収集エンジンを呼び出し、エラー検出ログを有効または無効にできます。以下のサンプル コードは、不要な領域周囲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 [ インストール ディレクトリ ]
¥ERptApi¥NMApiLib.lib
// Include file is located in [ インストール ディレクトリ ]¥ErptApi
#include "nmapilib.h"
... [ 監視対象のコード ]
StopEvtReporting()
... [ 監視対象外のコード ]
StartEvtReporting()
... [ 監視対象のコード ]
```

StartEvtReporting または **StopEvtReporting** API コールを使用して、アプリケーションのビジネスにきわめて重要な部分が DevPartner エラー検出でログに記録されないようにすることもできます。この例には、パスワード検証や暗号化ルーチンがあります。DevPartner エラー検出がアクティブでない場合、API コールはすぐに返されます。

[モジュールとファイル]設定の使用

アプリケーションから除外する対象を決定するには、以下の手順を実行します。

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。
   - ◇ 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 レポーティング]**、**[コール バリデーション]**、**[COM コール レポーティング]**、**[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]** の設定をクリアします。
3. DevPartner エラー検出でプログラムを実行します。

   エラー検出では、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。すべての DLL がロードされるように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 **[設定]** または **[オプション]** ダイアログ ボックスを開いて、データ収集設定を選択します。
5. **[設定]** または **[オプション]** ダイアログ ボックスで、**[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 ディレクトリにあるファイルを除き、アプリケーションで使用されるすべての実行可能ファイルと DLL のリストが自動的に表示されます。
6. モジュールとファイルのリストを確認します。リストに表示された DLL で現在のタスクに関係ないものをすべてクリアします。不要な DLL をクリアしたリストから、各 DLL を展開し、監視対象のソース ファイルを選択します。
7. 特定のディレクトリの 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ディレクトリを追加します。システム ディレクトリから特定のファイルを追加する場合は、**[モジュールの追加]** をクリックし、監視対象の DLL のリストに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、1 つのフォルダから複数のフォルダに切り替えることができます。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]** ダイアログにも表示されてる場合を除く）。
	選択したディレクトリとすべてのサブディレクトリがテストから除外されます。

8 プログラムの部分のリークとエラーをソース コードなしで除外するには、**[ソース コードが利用できる場合にのみ、エラーとリークを表示する]** を選択します。

9 アプリケーションの論理サブセットを作成したら、**[構成ファイル管理]** を使用して設定を保存します。

ヒント: 複数の設定ファイルを作成する場合は、設定ファイルの 1 つに **基本設定** という名前を付けることができます。**基本設定** の設定を開始点として使用して、その他の設定ファイルを作成できます。

表 3-2 に、**[モジュールとファイル]** 設定を使用する方法のリストを示します。

表 3-2. [モジュールとファイル] 設定の使用法

デバッグの対象	エラー検出を設定して除外する対象
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 コントロールを含む 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部分に直接関連付けられていないすべてのモジュール
ISAPI フィルタ	ISAPI フィルタ以外の IIS または W3WP 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に当てはまらないアプリケーションの部分
プロセス外の COM オブジェクト	DLLHOST.exe や MTX.exe などの、DLL に直接関連付けられていないすべてのモジュール

メモ: ユーザーのコード以外のすべてを除外した場合、アプリケーションの部分によって間接的に引き起こされるメモリ リークやリソース リークを確認できないことがあります。

監視対象を決定する

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監視対象の部分を確認することが重要です。何を監視し、何を無視するかの決定は、リークやエラーを追跡する場合の成功に影響します。

監視対象を確認するには、アプリケーションに関する以下の質問について検討してください。

- アプリケーションはどのように起動しますか。
  - 直接起動しますか。
  - 別のプログラムを実行して起動しますか。
  - コントロール ペインから起動しますか。
  - アプリケーションは間接的に起動されますか。
- アプリケーションにはモジュールとファイルがいくつありますか。
  - アプリケーション内のすべてのモジュール（システム モジュール以外）を所有していますか。
  - すべてのモジュールのソースがありますか。
- アプリケーション全体に関心がありますか。それとも一部だけですか。
  - コントロールしないモジュールのエラーに関心がありますか。
  - アプリケーションはトランザクションのアプリケーションですか。その場合、アプリケーション全体を監視しますか。それともいくつかのトランザクションだけですか。
  - アプリケーションでは、ユーザーがコントロールしないコードから渡されたリソースが利用されていますか。

これらの質問に回答したら、DevPartner エラー検出を設定してアプリケーションを監視できます。

監視対象を確認するとき、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場合があることに注意してください。焦点を絞りすぎると、選択した分析サブセットとアプリケーションのそれ以外の部分の間で渡されるリソースを失う可能性があることに注意してください。

たとえば、ActiveX コントロールを記述し、テスト コンテナで実行している場合、DLL 内の動作を確認できます。ただし、オブジェクトを誤って呼び出した場合、リソースとインターフェイスの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、エラーは見つかりますが、コントロールの誤った使用法によって発生したエラーは見つかりません。

アプリケーションの起動方法

コンソールまたはWindowsアプリケーションを使用している場合、**[ファイル] > [開く]**を選択してエラー検出を設定し、アプリケーションを監視できます。DevPartnerエラー検出ではアプリケーションを開き、アプリケーションに直接リンクされたすべてのDLLを分析します。

非従来型のアプリケーションを使用している場合、アプリケーションは以下の2つのカテゴリのいずれかに当てはまります。

- 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には、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が含まれます。たとえば、ActiveXコントロールを作成したら、Visual Studioに付属のテスト 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起動を待機できます（「プロセスを待機」（29ページ）を参照）。以下にこの例を示します。

- Windows NTサービス
- プロセス外のCOMサーバー

サービスやCOMサーバーなどの多くの専門アプリケーションは、タイム クリティカルです。アプリケーションがタイム クリティカルな場合、最良の結果を得るには、DevPartnerエラー検出の使用時にタイム アウト ロジックを無効にします。

サービスを分析する

DevPartnerエラー検出ではWindows NTサービスを監視できます。サービスを監視する場合は、以下の点を考慮してください。

- サービスはブート時間に起動しますか。オンデマンドで起動しますか。
- サービスに特定のセキュリティ コンテキストは必要ですか。
- サービスを対話的に実行できますか。
- サービスはサービスとしてでなくても実行できますか。
- サービスにタイミングの問題はありますか。

DevPartnerエラー検出では、システムの稼働後に起動でき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 プロセスをとおしてサービスの起動や停止を手動で行うことができる必要があります。

要件とガイドライン

DevPartnerエラー検出でサービスを監視するには、実行に使用されるアカウントに管理者権限が必要です。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。

サービスを分析する

DevPartnerエラー検出でサービスを分析するには、以下の手順を実行します。

1. サービスを停止します。
2. 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）サービスのデバッグ構成をビルドします。
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

サービスの開始に失敗した場合、または開始したがほとんどすぐに終了した場合は、**SetServiceStatus**に渡された**ServiceStatus**ブロックで**dwWait**パラメータを変更する必要があります。サービスで指定されている値が小さすぎると、Windows NT**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場合は、**dwWait**を4,000,000などの大きい値に設定します。

> **メモ：** DevPartnerエラー検出での作業が終了したあと、**dwWait**の通常の値を復元します。

代替方式：ワーカー スレッドからのコントロール ロジックの分離

サービスをモジュール方式で作成した場合、ワーカー スレッドからサービス コントロール ロジックを分離できることがあります。1つの方法として、ワーカー スレッド ロジックに関連した単純なコンソール アプリケーションをラップする方法があります。このようにして、DevPartnerエラー検出を使用して、サービス ワーカー スレッドをWindowsコンソール プログラムであるかのようにチェックできます。

DevPartner エラー検出ログのオンとオフを切り替えるカスタム コード

対話的でないサービスを処理する場合、カスタム コードを作成して、サービスの実行中に DevPartner エラー検出ログのオンとオフを切り替えることができます。**dwControl** パラメータから **ControlService** に渡すコントロール コードに応答するようにカスタム コードを作成します。

サービス コントロール ロジックで起動および停止のイベント レポート API を呼び出せます。「条件付きコード」（31 ページ）を参照してください。

共通のサービス関連の問題

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ていることを確認します。管理者権限を取得できない場合は、上記の代替方式を行ってください。

サービスが起動するとすぐに終了する

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終了されたことが原因である可能性があります。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。

DevPartner エラー検出に有効な作業ディレクトリがあることも確認する必要があります。**[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。

引き続きこの問題が発生する場合は、上記の代替方式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。

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。エラー検出設定の **[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ゥルメントし、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。

引き続きこの問題が発生する場合は、上記の代替方式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シャットダウン時には、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することによるメモリ オーバーランの検出などの数多くのチェックが実行されます。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。

引き続きこの問題が発生する場合は、上記の代替方式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 コンテナを使用してActiveXコントロールを分析する

DevPartnerエラー検出をVisual Studioに付属のテスト コンテナ ユーティリティと組み合わせて使用して、ActiveXコントロール、およびテスト コンテナと共に使用できるその他のCOMオブジェクトを監視できます。

DevPartnerエラー検出をテスト コンテナと組み合わせて使用するには、以下の手順を実行します。

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 8¥Common7¥Tools¥TestCon32.exe
   ```

   ```
   C:¥Program Files¥Microsoft Visual Studio 9¥Common7¥Tools¥TestCon32.exe
   ```
3. **[モジュールとファイル]**設定を呼び出します。
4. **[TestCon32.exe]**を選択解除します。
5. **[モジュールの追加]**をクリックします。
6. ActiveXまたはCOMコントロールを含む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その他の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 アプリケーションが起動したら、以下の手順を実行します。

1 ツールバーの**[新規コントロール]**をクリックします。

2 表示されたリストからコントロールを追加します（Calendar Control 8.0 など）。

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 コントロールの実行を終了したら、テスト コンテナを終了します。

実行中、エラーが検出されると DevPartner エラー検出によってレポートされます。テスト コンテナの終了時に、実行中にレポートされなかったメモリ、リソース、インターフェイスのリークが DevPartner エラー検出によってレポートされます。

一般的なテスト コンテナの問題

DevPartner エラー検出で *TestCon32.exe* のエラーがレポートされる

デフォルトで、DevPartner エラー検出では、**[モジュールとファイル]**または**[システム ディレクトリ]**を使用して DLL と EXE が明示的に除外されていないかぎり、実行ファイル、およびプロセスに関連付けられたすべての DLL 内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。

DevPartner エラー検出 COM コール レポーティングでオブジェクトへのコールがログに記録されない

DevPartner エラー検出では、認識対象として指定された COM インターフェイスのメソッドのみが記録されます。DevPartner エラー検出に ActiveX コントロールについて知らせるには、**[COM コール レポーティング]**設定で、**[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]**を選択して、メソッドのログ記録を有効にします。

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

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。

COM を使用するアプリケーションを分析する

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ガとして設定する必要があります。

DevPartner エラー検出を COM コンポーネントのデバッガとして設定するには、以下の手順を実行します。

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]** の順に選択します。
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、**[デバッガ内で実行する]** を選択します。
7. **デバッガのパス**を変更して、**bc.exe** を示します。フル パスを入力します。DevPartner エラー検出のインストール時にデフォルトの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
   ```

   メモ： 64 ビット バージョンの Windows では、DevPartner エラー検出は以下の場所にインストールされます。**¥Program Files (x86)¥Compuware¥DevPartner Studio¥BoundsChecker¥bc.exe**.

ヒント： **dllhost.exe** が削除されるのを避けるには、**[参照]** をクリックするのではなく、パスを切り取って貼り付けるか、入力します。

注意： デバッガのパスの最後から **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を保存します。

DevPartner エラー検出をコンポーネントのデバッガとして設定したあと、以下の手順を実行します。

1. 以下のいずれかの方法で、コンポーネントを起動します。
   - ◇ 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起動します。
     - — 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、**[スタート]** 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、エラー検出の実行を開始します。

メモ： COMコンポーネントのエラーとイベントだけを表示するには、DevPartnerエラー検出の[モジュールとファイル]設定でモジュールのリストから`dllhost.exe`とその他のDLLを削除します。

3 コンポーネントの実行を終了したら、コンポーネントをシャットダウンします。以下の手順に従ってください。

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検出と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。

一般的なCOMの問題

DevPartner エラー検出で `dllhost.exe` のエラーがレポートされる

デフォルトで、DevPartnerエラー検出では、[モジュールとファイル]または[システム ディレクトリ]を使用してDLLとEXEが明示的に除外されていないかぎり、実行ファイル、およびプロセスに関連付けられたすべてのDLL内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。

DevPartner エラー検出の COM コール レポーティングでコンポーネントへのコールがログに記録されない

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 コールのみが記録されます。[COMコール レポーティング]設定で、[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]を選択して、メソッドのログ記録を有効にします。

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

DevPartner エラー検出では、**[COM オブジェクトの追跡]** 設定で **[COM オブジェクトの追跡を有効にする]** を選択した場合にのみ COM インターフェイス リーク情報がレポートされます。また、監視対象の COM クラスも指定する必要があります。

ユーザーのインターフェイスのみを追跡するには、**[COM オブジェクトの追跡]** 設定の COM クラスのリストを確認し、ユーザーのクラスを選択してその他をすべてクリアします。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したあと、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、DevPartner エラー検出ではメモリ リーク、リソース リーク、およびインターフェイス リークの最終的な検出が行われます。

dllhost.exe がタイム アウトする前に終了するには、**[コンポーネント サービス]** ウィンドウでコンポーネントを見つけ、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。

[プロセスを待機] を使用して dllhost.exe をデバッグするには

dllhost.exe をデバッグする場合は、**[プロセスを待機]** を使用しないでください。Windows 2000 システムと Windows XP システムで作成されるコンポーネントの数を考慮すると、コンポーネント サービス デバッグ オプションを使用して、COM によって提供されるサポートされているメカニズムを使用するほうが安全です。

サポートされているメカニズムを使用しないと、その他の COM コンポーネントが要求された場合に予想外の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を DevPartner エラー検出に関連付けるため、コンポーネントが正常に起動しない可能性があります。

IIS 5.0 の ISAPI フィルタを分析する

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、以下の手順を実行します。

1. 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）デバッグ構成がある ISAPI フィルタをビルドします。
2. Internet Information Server（IIS）サービスを停止します。

3 **inetinfo.exe** の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]** の下の **[モジュールとファイル]** を開いて、すべての EXE と DLL をクリアします。

c **[モジュールの追加]** をクリックして、モジュールのリストに ISAPI フィルタを追加します。

d 残りの設定を ISAPI フィルタに適した設定に更新します。

4 [アプリケーション保護] モードの [高（分離プロセス）] を使用するためにテストする ISAPI 拡張機能が含まれる仮想ディレクトリを設定します。

a Internet Information Services マネージャを開きます。

b 仮想ディレクトリまで移動します。

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 **[アプリケーション保護]** を **[高（分離プロセス）]** 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。

エラー検出が、IIS を使用するように初期化されて、IIS の開始を待機します。

6 **[サービス]** コントロール パネルから IIS Admin サービスを開始します。

7 IIS サーバーへの一連の HTTP 要求を生成し、ISAPI フィルタを実行します。

8 ISAPI フィルタの実行を終了したあと、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検出とリーク検出が実行されます。

一般的な ISAPI フィルタの問題

ISAPI フィルタのデバッグに関連する一般的な問題の多くについては、すでにサービスの一般的な問題で説明しました。

以下の問題は、IIS と ISAPI のフィルタのデバッグに固有のものです。

IIS が起動してすぐにハングする

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。

DevPartner エラー検出ログに非常に多くの不要な情報が含まれている

[モジュールとファイル]設定を使用して、**inetinfo.exe**およびISAPIフィルタ以外のすべてのDLLを実行します。

inetinfo.exeを最初に実行するとき、DevPartnerエラー検出では、プロセスに動的にロードされたすべてのDLLが自動的にモジュールとファイルのリストに追加されます。**[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をすべてクリアします。不要なDLLをリストから削除しないでください。リストから削除すると、単純に再びリストに追加され、次回の実行中に有効になります。

デバッグ サービスと ISAPI のフィルタに関するその他の情報源

- IISとISAPIのフィルタのデバッグ手法について説明している多数の優れた記事をMSDNで参照できます。
- コンピュウェアのWebサイトでは多数のサポート技術情報を参照できます。

IIS を DevPartner エラー検出と対話的にデバッグするためのヒント

- 管理者権限でアカウントにログインする必要があります。
- この提案で問題が解決されない場合は、Microsoftテクニカル ノート 63:「ISAPIアプリケーションのデバッグ」を確認してください。
  http://msdn.microsoft.com/ja-jp/library/bewb5yw3.aspx

IIS 6.0 の ISAPI フィルタを分析する

IIS 6.0を以下のいずれかの方法で設定している場合、DevPartnerエラー検出を使用してISAPIフィルタを分析できます。

- IIS 5.0プロセス分離モード
- IIS 6.0デフォルト構成

ISAPIフィルタをDevPartnerエラー検出で分析するには、まず最適化しないで（オプションでFinalCheckを使用）[デバッグ]でISAPIフィルタをビルドします。次に、使用しているIIS構成の手順に従います。

IIS 5.0 プロセス分離モード

IIS 6.0をIIS 5.0プロセス分離モード構成で実行する場合、DevPartnerエラー検出を**inetinfo.exe**実行可能ファイルに対して実行します。

ISAPIフィルタを分析するには、以下の手順を実行します。

1 **inetinfo.exe**の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]の下の[モジュールとファイル]を開いて、すべてのEXEとDLLをクリアします。

c [モジュールの追加]をクリックして、モジュールのリストにISAPIフィルタを追加します。

d 残りの設定をISAPIフィルタに適した設定に更新します。

2 [アプリケーション保護]モードの[高（分離プロセス）]を使用するためにテストするISAPI拡張機能が含まれる仮想ディレクトリを設定します。

a IIS Adminユーティリティを開きます。

b 仮想ディレクトリまで移動します。

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]タブの[アプリケーション保護]を[高（分離プロセス）]に設定します。

3 [コントロール パネル]の[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。
エラー検出が、IISを使用するように初期化されて、IISの開始を待機します。

5 **IIS Admin**サービスと**World Wide Web Publishing**サービスを開始します。

DevPartnerエラー検出では、**inetinfo.exe**プロセスとISAPIフィルタの監視が自動的に開始されます。

6 Webサーバーへの一連のHTTP要求を生成し、ISAPIフィルタを実行します。

7 ISAPIフィルタの実行を終了したあと、IISマネージャを使用して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。これにより、**inetinfo.exe**プロセスが停止されます。

メモ： DevPartnerエラー検出で[停止]ボタンをクリックすると、DevPartnerエラー検出と**inetinfo.exe**プロセスの両方が終了し、収集したデータが消失します。

8 エラー検出によって、プロセス終了時のエラー検出とリーク検出が実行されます。

IIS 6.0 デフォルト構成

IIS 6.0をデフォルト構成モードで実行する場合、DevPartnerエラー検出を**W3WP.exe**実行可能ファイルに対して実行します。

ISAPIフィルタを分析するには、以下の手順を実行します。

1 **W3WP.exe**の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**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]の下の[モジュールとファイル]を開いて、すべてのEXEとDLLをクリアします。

c [モジュールの追加]をクリックして、モジュールのリストにISAPIフィルタを追加します。

d 残りの設定をISAPIフィルタに適した設定に更新します。

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 **Internet Information Server（IIS）マネージャ**を開きます。

b 仮想ディレクトリまで移動します。

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。

エラー検出が、IISを使用するように初期化されて、IISの開始を待機します。

4 **IIS Admin**サービスと**World Wide Web Publishing**サービスを開始します。

エラー検出では、**W3WP.exe**プロセスとISAPIフィルタの監視が自動的に開始されます。

5 Webサーバーへの一連のHTTP要求を生成し、ISAPIフィルタを実行します。

6 ISAPIフィルタの実行を終了したあと、IISマネージャを使用して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。これにより、**W3WP.exe**プロセスが停止されます。

メモ： DevPartnerエラー検出で**[停止]**ボタンをクリックすると、エラー検出と**W3WP.exe**プロセスの両方が終了し、収集したデータが消失します。

7 エラー検出によって、プロセス終了時のエラー検出とリーク検出が実行されます。

一般的なIIS 6.0 ISAPIフィルタの問題

IIS 5.0一般的なISAPIフィルタの問題のセクションのすべての項目が当てはまります。いくつかの例では、**inetinfo.exe**を**W3WP.exe**に置き換える必要があります。

「一般的なISAPIフィルタの問題」（43ページ）を参照してください。

以下の新しい問題はIIS 6.0に当てはまります。

- Microsoftでは、IIS 6.0のデフォルト構成をより安全なものに再設計しました。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 の起動、停止、再起動を行います。これらの操作を実行するには、ツリーで **<MachineName>** ノードを右クリックし、**All Tasks->Restart IIS** を選択します。これによってダイアログ ボックスが開き、そこから IIS の起動と停止を制御できます。
- 最良の結果を得るには、IIS を監視する前に API コール ログなどのログ機能をオフにします。ログ機能をオンにすると、DevPartner エラー検出では非常に大きいログ（.DPBcl）ファイルが作成されるため、IIS サーバーのパフォーマンスに影響を与えます。

  メモ： 全般ダイアログの [イベントをログに記録] をオフ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かぎり、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 ボタンからイベント ログを明確に有効にするまですべてのレポートを抑制する場合にのみ使用します。

よく寄せられる質問（FAQ）

DevPartner エラー検出の ActiveCheck と FinalCheck またはテクノロジの違い

DevPartner エラー検出には、以下の 2 つの操作モードがあります。

- **ActiveCheck** － このモードでは、DevPartner エラー検出は 32 ビット Windows プログラムで動作し、オペレーティング システムと 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て、メモリ リーク、リソース リーク、および無効な（または解放された）関数に渡されたポインタの使用を探します。
- **FinalCheck** － このモードでは、DevPartner エラー検出 FinalCheck インストゥルメンテーション ロジックを使用して C または 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、**[DevPartner]>[ネイティブ C/C++ インストゥルメンテーション マネージャ]** をクリックします。

  FinalCheck インストゥルメンテーションでは、DevPartner エラー検出ですべてのポインタのフェッチ、保存、またはインストゥルメントするモジュール内での間接的な発生を監視できます。DevPartner エラー検出では範囲に出入りする変数も監視できます。

注意： 評価順序が正しく定義されていないコードをインストゥルメントすると、不正なデータ、ハング、クラッシュなどの予期しない結果が生じることがあります。

「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、オブジェクトへの値の格納など、「副作用」がある場合の評価順が明確に定義されていません。たとえば、以下のステートメントの評価順序はあいまいです。`i = ++i + 2;`

このステートメントには、変数 *i* に値を保存する部分が 2 か所ありますが、この言語ではそれらの実行順序が定義されていません。このようなコードをインストゥルメントした場合は、評価順序が固定ではないため、結果が予測できない可能性があります。

FinalCheck モードで実行する場合も、すべての ActiveCheck 分析は拡張 FinalCheck 分析と共に実行されることに注意してください。

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、割り当てられていないメモリへの読み取り／書き込みの検出に特化しています。

コール バリデーションを有効にするタイミング

コール バリデーションを有効にすると、DevPartner エラー検出によってプログラムのより多くのメモリ エラーとポインタ エラーが検出されます。検出されるイベントが大量になる可能性があるため、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] 機能を選択した場合の DevPartner エラー検出の動作

[メモリ ブロック チェックを有効にする]（デフォルトでは無効）を選択すると、DevPartner エラー検出ではより詳細な ActiveCheck 分析が実行されます。この機能によって DevPartner エラー検出の実行速度が 20% 遅くなる場合があることに注意してください。

[メモリの追跡] の DevPartner エラー検出 [保護バイト] 設定の使用方法

[メモリの追跡]設定の[保護バイト]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ます。

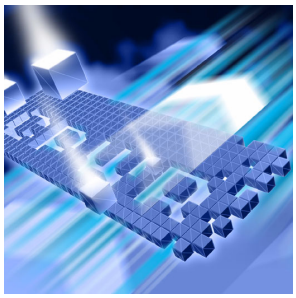
[回数]を4から8や16の大きい値に増やします。

保護バイトの数を多くすると、ヒープ ブロック間の間隔が広くなり、DevPartner エラー検出でオーバーランを監視するためのブロック間の領域が提供されます。

[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。

このオプションでは、メモリ機能を呼び出すたびにDevPartnerエラー検出で各ヒープ ブロックを検証するように指定します。これによりプログラムの実行速度は非常に遅くなりますが、ヒープ破壊がプログラムの限定的な領域に分離されるため、追跡が容易になります。

第 4 章
ユーザーが作成したアロケータの使用



この章では、ユーザー作成のメモリ アロケータを実装する方法について説明します。

概要

DevPartner エラー検出では、ユーザー作成のメモリ アロケータに対するメモリ分析を実行できます。これを行うには、メモリ アロケータの記述を `UserAllocators.dat` という名前のテキスト ファイルに追加します。このファイルは、DevPartner エラー検出インストールの data サブディレクトリにインストールされています。ユーザー作成のアロケータをこのファイルに追加すると、DevPartner エラー検出によって、オペレーティング システムで提供されるメモリ割り当てルーチンと同様に、アロケータが処理されます。DevPartner エラー検出によって、`UserAllocators.dat` 内に記述されたユーザー作成のアロケータが原因で発生したリークが検出された場合は、そのユーザー作成のアロケータ内の低水準アロケータではなく、ユーザー作成のアロケータ自体が [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。

必要な情報を収集する

ユーザー作成のアロケータを 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. ユーザー作成のアロケータの割り当て関数、解放関数、再割り当て関数、およびサイズ関数の正確な名前。
2. ユーザー作成のメモリ アロケータが静的にリンクされているか、または別のモジュール（DLL）で提供されているかを確認します。
3. ユーザー作成のアロケータを含むモジュール（DLL）の名前。
4. ルーチンに対するパラメータを調べて、ブロックのサイズ、およびメモリ ブロックに関連付けられたポインタが渡されるか、呼び出し側に返されるかを確認します。
5. 解放されたブロックにデータを格納する割り当て、またはユーザー作成のアロケータでメモリをゼロにするなどの、アロケータにおける特殊な前提。

ユーザーが作成したアロケータの名前を検索する

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

ルーチンの名前を検索するには、以下の手順を実行します。

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 や recalloc など）
   - ◇ メモリ ブロック サイズ関数（_msize）など
2. 考慮する状況として、以下の 2 つの基本的なケースがあります。
   - ◇ 定義している関数を含むモジュールのシンボル（pdb ファイル）があります。pdb ファイルを使用できる場合は、内部シンボルを使用できます。短縮または展開された関数名を検索するには、モジュールのコンパイル時にリンカ／デバッガ オプションを使用して、マップ ファイルを作成します。次に、マップ ファイルの **Publics by Value** セクションを調べて、関数の名前を確認します。この方法では、userAllocator 関数定義に Static キーワードが必須となります。
   - ◇ シンボルがない、またはシンボルが無効です。
     pdb ファイルを使用できない場合は、Visual Studio のコマンド 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。

割り当て関数の名前は、使用される呼び出しルールと言語の選択によって、短縮されている場合と短縮されていない場合があります。C++ を使用している場合、通常、名前は短縮されます。以下の小さい C++ プログラムについて検討してください。

```
#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ビルドする前に、Visual Studio の [プロジェクト] 設定で Generate マップ ファイルを選択します。アプリケーションをビルドしたら、マップ ファイルを開き、new 演算子と delete 演算子のメソッドを検索します。メソッドは以下のようなものです。

```
global operator new:        ??2SampleClass@@SAPAXI@Z
global operator delete:     ??3SampleClass@@SAXPAX@Z
```

これらの演算子は、UserAllocators.datに以下のように記述できます。

```
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
```

ユーザー作成のアロケータによるメモリに関する特殊な前提

通常、DevPartnerエラー検出では、ユーザー プログラムにポインタを返す前に、割り当てられたメモリに充てんパターンがフィルされ、メモリのブロックが解放されたあと、そのブロックに無効データがフィルされます。

メモリ アロケータによってブロックが特殊なデータで初期化される場合は、ブロックを上書きしてデータを失うことがないように、NOFILLフラグを使用する必要があります。

メモリ アロケータで、ブロックが解放されたあとにそのブロックから読み取りできることが前提となっている場合は、NOPOISONフラグを使用する必要があります。「無効データのフィル」が発生しないようにする理由には、以下のようなさまざまなものがあります。

- 割り当てステータスを追跡する、別の解放されたブロックにリンクする、などの目的で、メモリ アロケータによってデータが解放されたブロックに保存される。
- アプリケーションで、解放されたブロックは再割り当てされるまで引き続き参照可能であるとみなされている。解放されたブロックを参照し続けるのは危険ですが、解放されたブロックを参照する多くのアプリケーションが現在も使用されています。

UserAllocators.dat でエントリを作成する

「必要な情報を収集する」（52ページ）で説明した情報を収集したら、UserAllocators.dat でアロケータを記述できます。

関数を記述するには、関数ごとに1つのレコードをファイル内に作成します。各レコードは以下の形式に従います。

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

*Record_Type*は、作成している関数のタイプを示します。あとに続くパラメータでは、アロケータ関数についての詳細情報を提供します。

レコードを作成する場合、各フィールドを1つまたは複数のスペースかタブ文字で区切ります。レコードは複数行にわたって記述できます。

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	以前アロケータ関数によって割り当てられた特定のメモリ ブロックのサイズをクエリできる関数。
Reallocator	以前アロケータ関数によって割り当てられたメモリ ブロックのサイズを調整できる関数。 リアロケータ関数では、メモリの同じブロックが返される場合と返されない場合があることに注意してください。
Ignore	DevPartner エラー検出で無視する（メモリを追跡しない）アロケータ関数またはリアロケータ関数。

モジュール

各 UserAllocator レコード タイプで、記述されている関数を含むモジュールを指定する必要があります。以下の3種類のモジュールを記述できます。

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（foo.dll など）を含む、明示的に名前が付けられたモジュール（DLL）。 **メモ**：モジュール名では、ワイルドカードは使用できません。

表 4-2. モジュール タイプ （続き）

モジュール タイプ	説明
静的にリンクされた ユーザー アロケータ	ユーザー割り当て関数またはメソッドを含む、明示的に名前が付けられたモジュール（DLL または実行ファイル）。ただし、この場合、関数またはメソッドは元はライブラリ（.lib ファイル）の一部でした。モジュールにリンクすると、カスタマ コードで関数やメソッドを参照できますが、これらの関数やメソッドは外部からは見えません。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、デバッグ シンボル内の関数やメソッドを検索するように DevPartner エラー検出を変更します。 **メモ**：レコードのオプションのパラメータに STATIC キーワードを含めない場合、DevPartner エラー検出によってユーザー作成の割り当て関数やメソッドが正しく監視されません。
*CRT	これは、アプリケーションで関数が見つかった場合は常に参照できる特殊なケースです。 **メモ**：*CRT の * はワイルドカード文字ではありません。 *CRT は、以下の 3 つのケースを対象とします。 • Microsoft C ランタイム ライブラリ • 静的にリンクされた C ランタイム ライブラリ • ユーザーがパッチしているすべてのものと同じ短縮名を持つすべてのユーザー関数（**`global operator new`** など）

アロケータ レコード

アロケータ レコードを作成して、メモリを割り当てる関数を記述します。以下の形式に従います。

```
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、指定された関数の戻り値は、割り当てられたブロックのアドレスになります。BufferLocパラメータを指定すると、この動作を変更して、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ように指定できます。（`UserAllocators.dat`内の`MAPIAllocateBuffer`を参照してください）。

DevPartnerエラー検出では、割り当てられたブロックのアドレスが見つかると予想した場所がコール後にNULLである場合（戻り値または`BufferLoc`で指定されたパラメータの値）、割り当ては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`という単語にして、割り当てルーチンを記述していることを示す必要があります。
module_name	ユーザー作成のアロケータを含むモジュール（実行ファイルまたはDLL）の名前。 **メモ：**モジュール名では、ワイルドカードは使用できません。
func_name	ユーザー作成のアロケータ内のメモリのブロックを割り当てる関数の名前。C++を使用している場合は、関数の「短縮」名にする必要があります。 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	このパラメータは、割り当てるメモリの種類を記述する場合に使用します。現在の DevPartner エラー検出では、以下のメモリの種類が定義されています。 • MEM_MALLOC malloc、calloc、strdup などのルーチンから返されるメモリのブロック。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 new 演算子から返され、delete 演算子によって解放されるメモリのブロック。 • MEM_CUSTOM1 ～ MEM_CUSTOM9 特定のデアロケータと組み合わせる必要のあるメモリのブロック。この種類を使えば、開発者は、上述した標準メモリ アロケータに影響を及ぼさない独自のカスタム メモリ アロケータを宣言することができます。 DevPartner エラー検出では、特定のメモリの種類で割り当てられたメモリのブロックが同じ種類の関数で解放されるかどうかが検証されます。種類が一致しない場合、DevPartner エラー検出では、実行時にメモリ競合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 にする必要があります。 ユーザー作成の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。正しい数を指定しなかった場合は、予期せぬ結果が生じる可能性があります。
size_value	割り当てるブロックのサイズを含むパラメータの番号。最初のパラメータの番号は 1 です。
count_num	このオプション パラメータは、size パラメータと count パラメータを受け取る calloc のような関数の記述に使用します。割り当てるサイズに必要なブロック数を指定します。このパラメータを省略した場合は、1 とみなされます。
buffer_loc	割り当てるブロックのアドレスを保持するアドレスを含むパラメータの番号。最初のパラメータの番号は 1 です。

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
[Optional Parameters]	レコードの最後に、以下のオプションのパラメータを指定できます。 • DEBUG このパラメータを指定した場合、DevPartner エラー検出では、フックについての詳細情報が表示されます。出力ウィンドウ（または dbgview）に、フックを設定しようとして発生したエラーに関する情報が表示されます。また、通知情報ペインに、フックが成功した関数やフックされた関数が呼び出された回数などの各フックに関する統計値が表示されます。 • NODISPLAY このパラメータを指定した場合、DevPartner エラー検出では、要求された各フックの詳細情報が通知情報ペインの上部に表示されません。 • NOFILL このパラメータを指定した場合、DevPartner エラー検出では、返されたバッファが DevPartner エラー検出の「fill」文字列でフィルされません。 メモ：calloc のように、ユーザー作成のアロケータによってブロックがデータで初期化される場合には、NOFILL を指定してデータが破損されないようにします。 • NOGUARD このパラメータを指定した場合、DevPartner エラー検出では、この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 DevPartner エラー検出によって、ユーザー作成のアロケータが静的にパッチされます。ユーザー作成のアロケータをアプリケーションにリンクして、インターフェイスをエクスポートした別の DLL の中で使用しない場合は、static オプションを指定します。

アロケータ レコードのサンプル

以下は、仮定のアロケータ レコード関数の例です。

例 1　この例では、関数 mallocXX は、MyAlloc.dll というライブラリにあります。この関数は、最初のパラメータでサイズを渡される 1 つのパラメータを含む malloc タイプの演算子であると想定されます。DevPartner エラー検出では、アプリケーション プログラムに返す前にメモリ ブロックをフィルできません。すべての MEM_MALLOC タイプの関数は、この関数によって割り当てられたブロックを解放できます。

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

例 2　この例は、Microsoft iostream コードのカスタム グローバル演算子 new の追跡に使用されるファイルのものです。この関数は C ランタイム ライブラリにあることに注意してください。レコードではモジュール名として *CRT が指定されます。このため、DevPartner エラー検出では、関数が Microsoft C または C++ ランタイム ライブラリのいずれかにあると想定されます。

この関数には、サイズが最初のパラメータに保存された 4 つのパラメータがあります。DevPartner エラー検出では、メモリを要求するプログラムに返す前にブロックをフィルできます。

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

例 3　この例は、サイズを最初のパラメータで渡される 1 つのパラメータがある CustomAllocXX という関数を示しています。

DevPartner エラー検出では、アプリケーション プログラム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 が指定されていることに注意してください。DevPartner エラー検出では、この関数で割り当てられたメモリが、同じ MEM_CUSTOM1 タイプのルーチンによって解放されるかどうかが検証されます。別の解放ルーチンを使用すると、メモリの解放後に割り当て競合メッセージが生成されます。

```
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 ファイルとしてビルドされ、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ます。MyAlloc には 4 つのパラメータがあります。最初のパラメータはデータ レコードのサイズ、2 番めのパラメータは 1 つのブロックで割り当てられるレコードの数です。3 番めのパラメータは、割り当てるブロックのアドレスを保持する場所のアドレスを含みます。アプリケーションから取得されたメモリはあらかじめ初期化されているため、DevPartner エラー検出では、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ていない場合は、STATIC を指定する必要があります。

デアロケータ レコード

デアロケータ レコードを作成して、メモリを解放する関数を記述します。以下の形式に従います。

```
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という単語にして、解放ルーチンを記述していることを示す必要があります。
module_name	ユーザー作成のアロケータを含むモジュール（実行ファイルまたはDLL）の名前。 **メモ**：モジュール名では、ワイルドカードは使用できません。
func_name	ユーザー作成のアロケータ内のメモリのブロックを解放する関数の名前。C++ を使用している場合は、関数の「短縮」名にする必要があります。 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	このパラメータは、解放されるメモリの種類を記述するメカニズムを提供します。現在のDevPartnerエラー検出では、以下のメモリの種類が定義されています。 • MEM_MALLOC malloc、calloc、strdupなどのルーチンから返されるメモリの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 new演算子から返され、delete演算子によって解放されるメモリの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のあるメモリのブロックを記述します。この種類を使えば、開発者は、上述した標準メモリ アロケータに影響を及ぼさない独自のカスタム メモリ アロケータを宣言することができます。 DevPartnerエラー検出では、特定のメモリの種類で割り当てられたメモリのブロックが同じ種類の関数で解放されるかどうかが検証されます。種類が一致しない場合、DevPartnerエラー検出では、実行時にメモリ競合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にする必要があります。 ユーザー作成の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。正しい数を指定しなかった場合は、予期せぬ結果が生じる可能性があります。

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	解放されるブロックへのポインタを含むパラメータの番号。最初のパラメータの番号は 1 です。
[Optional Parameters]	レコードの最後に、以下のオプションのパラメータを指定できます。 • DEBUG このパラメータを指定した場合、DevPartner エラー検出では、フックについての詳細情報が表示されます。出力ウィンドウ（または dbgview）に、フックを設定しようとして発生したエラーに関する情報が表示されます。また、通知情報ペインに、フックが成功した関数やフックされた関数が呼び出された回数などの各フックに関する統計値が表示されます。 • NODISPLAY このパラメータを指定した場合、DevPartner エラー検出では、要求された各フックの詳細情報が通知情報ペインの上部に表示されません。 • NOPOSION NOPOISON を指定した場合は、メモリ ブロックが解放されても DevPartner エラー検出による無効データの上書きが行われません。 ユーザー作成のアロケータによって、解放済みのブロックにデータが保存される場合、または、アプリケーションによって、解放済みのブロック内のデータが引き続き使用される場合は、NOPOSION を指定してデータの破損を回避します。 • STATIC DevPartner エラー検出によって、ユーザー作成のアロケータが静的にパッチされます。ユーザー作成のアロケータをアプリケーションにリンクして、インターフェイスをエクスポートした別の DLL の中で使用しない場合は、static オプションを指定します。

DevPartner エラー検出では、解放関数からの戻り値はチェックされません。

デアロケータ レコードのサンプル

以下は、仮定のデアロケータ レコードの例です。

例 1　この例では、freeXX という関数が MyAlloc.dll というライブラリにあります。この関数には、解放されるブロックへのポインタが最初のパラメータで渡される 1 つのパラメータがあります。DevPartner エラー検出では、アプリケーション プログラムに返す前にメモリ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 内の MyFree という関数を示しています。この関数には、解放されるブロックへのポインタが最初のパラメータで渡される 1 つのパラメータがあります。DevPartner エラー検出では、アプリケーション プログラムに返す前にメモリに無効データをフィルする必要があります。ブロックが解放されると、DevPartner エラー検出では、ブロックが MEM_CUSTOM1 タイプのアロケータによって割り当てられたかどうかが検証されます。ブロックがこのグループからのものでない場合、エラーが生成されます。

```
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 ファイルとしてビルドされ、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ます。MyFree には 3 つのパラメータがあります。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 番めのパラメータには、解放されるアドレスが含まれています。また、プライベート解放ルーチンでは、ブロックが解放されると、解放されたブロックにプライベート情報が保持されます。

```
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ていない場合は、STATIC を指定します。

QuerySize レコード

QuerySize レコードを作成して、割り当てられたメモリのブロックのサイズを返す関数を記述します。以下の形式に従います。

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

メモ： ユーザー定義のアロケータの QuerySize レコードを省略すると、DevPartner エラー検出によって、その関数の誤ったブロック サイズが返されます。

表 4-5. QuerySize 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 という単語にして、サイズ ルーチンを記述していることを示す必要があります。
module_name	ユーザー作成のアロケータを含むモジュール（実行ファイルまたは DLL）の名前。 **メモ：** モジュール名では、ワイルドカードは使用できません。
func_name	ユーザー作成のアロケータ内の割り当てられたメモリのブロックのサイズを返す関数の名前。C++ を使用している場合は、関数の「短縮」名にする必要があります。 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	このパラメータは、クエリされるメモリの種類を記述するメカニズムを提供します。現在の DevPartner エラー検出では、以下のメモリの種類が定義されています。 • MEM_MALLOC malloc、calloc、strdup などのルーチンから返されるメモリの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 new 演算子から返され、delete 演算子によって解放されるメモリの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のあるメモリのブロックを記述します。この種類を使えば、開発者は、上述した標準メモリ アロケータに影響を及ぼさない独自のカスタム メモリ アロケータを宣言することができます。 DevPartner エラー検出では、特定のメモリの種類で割り当てられたメモリのブロックが同じ種類の関数で解放されるかどうかが検証されます。種類が一致しない場合、DevPartner エラー検出では、実行時にメモリ競合エラーが表示されます。

表 4-5. QuerySize 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 にする必要があります。 ユーザー作成の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。正しい数を指定しなかった場合は、予期せぬ結果が生じる可能性があります。
address_value	クエリされるブロックへのポインタを含むパラメータの番号。最初のパラメータの番号は 1 です。
[*Optional Parameters*]	レコードの最後に、以下のオプションのパラメータを指定できます。 • DEBUG このパラメータを指定した場合、DevPartner エラー検出では、フックについての詳細情報が表示されます。出力ウィンドウ（または dbgview）に、フックを設定しようとして発生したエラーに関する情報が表示されます。また、通知情報ペインに、フックが成功した関数やフックされた関数が呼び出された回数などの各フックに関する統計値が表示されます。 • NODISPLAY このパラメータを指定した場合、DevPartner エラー検出では、要求された各フックの詳細情報が通知情報ペインの上部に表示されません。 • STATIC DevPartner エラー検出によって、ユーザー作成のアロケータが静的にパッチされます。ユーザー作成のアロケータをアプリケーションにリンクして、インターフェイスをエクスポートした別の DLL の中で使用しない場合は、static オプションを指定します。

この関数からの戻り値として、size_t でブロックのサイズが提供されるとみなされます。

QuerySize レコードのサンプル

以下は、仮定の QuerySize レコードの例です。

例 1 この例では、`MySize` という関数が `foo.dll` というライブラリにあります。この関数には、ブロックへのポインタが最初のパラメータでクエリされる 1 つのパラメータがあります。

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

例 2　この例では、MySize 関数が 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ます。MySize 関数には 2 つの関数があり、クエリされるアドレスは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ていない場合は、STATIC を指定します。

リアロケータ レコード

リアロケータ レコードを作成して、メモリを再割り当てする関数を記述します。以下の形式に従います。

```
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、指定された関数の戻り値は、割り当てられたブロックのアドレスになります。BufferLoc パラメータを指定すると、この動作を変更して、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ように指定できます（`UserAllocators.dat` ファイル内の `MAPIAllocateBuffer` を参照してください）。

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 という単語にして、再割り当てルーチンを記述していることを示す必要があります。
module_name	ユーザー作成のアロケータを含むモジュール（実行ファイルまたは DLL）の名前。 **メモ：** モジュール名では、ワイルドカードは使用できません。
func_name	ユーザー作成のアロケータ内のメモリのブロックを再割り当てする関数の名前。C++ を使用している場合は、関数の「短縮」名にする必要があります。 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	このパラメータは、再割り当てされるメモリの種類を記述するメカニズムを提供します。現在の DevPartner エラー検出では、以下のメモリの種類が定義されています。 • MEM_MALLOC malloc、calloc、strdup などのルーチンから返されるメモリの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 new 演算子から返され、delete 演算子によって解放されるメモリの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のあるメモリのブロックを記述します。この種類を使えば、開発者は、上述した標準メモリ アロケータに影響を及ぼさない独自のカスタム メモリ アロケータを宣言することができます。 DevPartner エラー検出では、特定のメモリの種類で割り当てられたメモリのブロックが同じ種類の関数で解放されるかどうかが検証されます。種類が一致しない場合、DevPartner エラー検出では、実行時にメモリ競合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 にする必要があります。 ユーザー作成の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。正しい数を指定しなかった場合は、予期せぬ結果が生じる可能性があります。
address_value	再割り当てされるブロックへのポインタを含むパラメータの番号。最初のパラメータの番号は 1 です。
size_value	再割り当てするブロックのサイズを含むパラメータ番号。最初のパラメータの番号は 1 です。
count_num	このオプション パラメータは、size パラメータと count パラメータを受け取る calloc のような関数の記述に使用します。割り当てるサイズに必要なブロック数を指定します。このパラメータを省略した場合は、1 とみなされます。
buffer_loc	再割り当てするブロックのアドレスを保持するアドレスを含むパラメータの番号。最初のパラメータの番号は 1 です。

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
[*Optional Parameters*]	レコードの最後に、以下のオプションのパラメータを指定できます。 • DEBUG このパラメータを指定した場合、DevPartner エラー検出では、フックについての詳細情報が表示されます。出力ウィンドウ（または dbgview）に、フックを設定しようとして発生したエラーに関する情報が表示されます。また、通知情報ペインに、フックが成功した関数やフックされた関数が呼び出された回数などの各フックに関する統計値が表示されます。 • NODISPLAY このパラメータを指定した場合、DevPartner エラー検出では、要求された各フックの詳細情報が通知情報ペインの上部に表示されません。 • NOFILL このパラメータを指定した場合、DevPartner エラー検出では、前の割り当ての最後に追加されたその他のバイトが DevPartner エラー検出の「fill」文字列でフィルされません。 メモ：calloc のように、ユーザー作成のアロケータによってブロックがデータで初期化される場合には、NOFILL を指定してデータが破損されないようにします。 • NOGUARD このパラメータを指定した場合、DevPartner エラー検出では、この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 DevPartner エラー検出によって、ユーザー作成のアロケータが静的にパッチされます。ユーザー作成のアロケータをアプリケーションにリンクして、インターフェイスをエクスポートした別の DLL の中で使用しない場合は、static オプションを指定します。

DevPartner エラー検出では、再割り当て関数からの戻り値がチェックされ、NULL 値はエラーを示すとみなされます。NULL 以外のアドレスは、新たに割り当てられたメモリのブロックであるとみなされます。

リアロケータ レコードのサンプル

以下は、仮定のリアロケータ レコードの例です。

例 1　reallocXX という関数が foo.dll というモジュールで宣言されます。この関数には 2 つのパラメータがあります。最初のパラメータは既存のメモリ ブロックのアドレス、2 番めのパラメータは要求されたブロックのサイズです。オプションのパラメータが指定されていないため、DevPartner エラー検出では、アプリケーション プログラムにコントロールを返す前に、新しいメモリに（ブロックがより大きいことを前提として）充てんパターンがフィルされます。

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

例 2　reallocClear という関数が foo.dll というモジュールで宣言されます。この関数には 3 つのパラメータがあります。最初のパラメータは既存のメモリ ブロックのアドレス、3 番め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では、新しいブロックで割り当てられた追加のメモリに対して独自のフィル処理が実行されます。このため、DevPartner エラー検出では、新しいブロックの追加のメモリをフィルできません。

メモ： DevPartner エラー検出では、パラメータ 2 の内容は無関係であるため無視されます。

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

例 3　MyRealloc という関数が .LIB ファイルにビルドされ、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ます。MyRealloc には、4 つのパラメータがあります。最初と 4 番めのパラメータは、DevPartner エラー検出には関係ありません。2 番めのパラメータには既存のブロックのアドレスが含まれており、3 番めのパラメータにはブロックの新しいサイズが含まれています。データ収集ルーチンでは、新しいデータが再割り当てのブロックにあらかじめロードされます。

```
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ていない場合は、STATIC を指定します。

Ignore レコード

Ignore レコードを作成して、DevPartner エラー検出メモリ追跡システムで無視する関数を記述します。以下の形式に従います。

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

Ignore レコードを使用して、DevPartner エラー検出でユーザー作成のアロケータを無視するか、ユーザー作成のアロケータで使用される下位アクセス ルーチンを無視するかを指定します。Ignore レコードでは、DevPartner エラー検出メモリ追跡システムで、通常は監視される API が追跡されないように指定できます。

注意： Ignore レコードを作成した場合、DevPartner エラー検出メモリ追跡システムでは、それらの API から割り当てられたメモリや解放されたメモリが監視されなくなります。その結果、DevPartner エラー検出では、このメモリが認識されなくなります。これにより、コール バリデーション モジュールと FinalCheck 分析モジュールで、不正なエラー メッセージや不完全なエラー メッセージが生成されることがあります。この関数の使用法についてご不明な点は、テクニカル サポートにお問い合わせください。

表 4-7. Ignore レコード 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 という単語にして、API を無視するように記述してい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（実行ファイルまたは DLL）の名前 **メモ：**モジュール名では、ワイルドカードは使用できません。
func_name	メモリ追跡システムで無視する関数の名前。C++ を使用している場合は、関数の「短縮」名にする必要があります。 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]	• DEBUG このパラメータを指定した場合、DevPartner エラー検出では、フックについての詳細情報が表示されます。出力ウィンドウ（または dbgview）に、フックを設定しようとして発生したエラーに関する情報が表示されます。また、通知情報ペインに、フックが成功した関数やフックされた関数が呼び出された回数などの各フックに関する統計値が表示されます。 • NODISPLAY このパラメータを指定した場合、DevPartner エラー検出では、要求された各フックの詳細情報が通知情報ペインの上部に表示されません。 • STATIC DevPartner エラー検出によって、ユーザー作成のアロケータが静的にパッチされます。ユーザー作成のアロケータをアプリケーションにリンクして、インターフェイスをエクスポートした別の DLL の中で使用しない場合は、static オプションを指定します。

Ignore レコードのサンプル

以下は、仮定の Ignore レコードの例です。

例 1　この例では、Ignore レコードが作成され、DevPartner エラー検出では、GlobalAlloc によって割り当てられたメモリは監視されますが、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無視されます。

メモ：これにより、DevPartner エラー検出によって多数の誤ったメモリ リークがレポートされます。

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

例 2　この例では、コールの GlobalAlloc ファミリによって操作されたメモリが DevPartner エラー検出で無視されます。

メモ：これにより、DevPartner エラー検出によって多数の誤ったコール バリデーション エラーと FinalCheck エラーがレポートされます。

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

メモ：これらの 3 行を UserAllocators.dat に追加することはお勧めできません。

例 3　この例では、指定したモジュール内の静的にリンクされた関数によって操作されたメモリが DevPartner エラー検出で無視されます。独自の代替メモリ割り当てライブラリを作成して標準の C ランタイム ライブラリと同じ名前を使用し、DevPartner エラー検出でライブラリの使用を監視しない場合は、以下のようなリンク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：このようなリンクを UserAllocators.dat ファイルに追加する前に、テクニカル サポートにご連絡ください。

UserAllocator フック要求をコーディングする

UserAllocator フック要求を作成する場合に、考慮する必要のある注意事項を以下に示します。

- MEM_CUSTOM1 ～ MEM_CUSTOM9 を使って、個別のメモリ割り当てをシステムの割り当てから分離します。
- アロケータのタイプごとに一意のカスタム アロケータのタイプを指定します。
- 必ず、アロケータに対応するタイプを使っ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は、同様に MEM_CUSTOM1 タイプでフックされたデアロケータを使ってメモリを解放する必要があります。デアロケータが一致しない場合は、割り当て競合エラーが報告されます。

UserAllocators に関するコード要件

UserAllocator のメモリ割り当てフックを利用するには、アプリケーション コードに、メモリの割り当てと解放を制御する関数を含める必要があります。

アロケータ関数フック

- 呼び出すアロケータ関数には、必要なメモリのバイト数を示すパラメータを含める必要があります。
- この関数は、割り当てられたメモリの場所を返すか、その場所を指しているアドレスをパラメータで返す必要があります。
- その他のパラメータを関数に含めることもできます。

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);
```

// 上の例では、メモリ アロケータによって、割り当てられたメモリの場所が pVoid ポインタに格納されます。

```
Allocator
    Module=bcheap.dll
    Function=GetMemoryAgain
    MemoryType=MEM_CUSTOM1
    NumParams=2
    Size=1
    BufferLoc=2
    Static
    Noguard
    NoFill
    Debug
```

デアロケータ関数フック

- 呼び出すデアロケータ関数には、解放するメモリのアドレスを示すパラメータを含める必要があります。
- その他のパラメータを関数に含めることもできます。

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
```

リアロケータ関数フック

- 呼び出すリアロケータ関数には、必要なメモリのバイト数を示す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「古い」メモリのアドレスを示す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、その場所を指しているアドレスをパラメータで返す必要があります。
- その他のパラメータを関数に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 フックをデバッグする

エラー検出では、UserAllocator フックに関する情報の表示方法を変更して、そのデバッグを容易にするために使用可能な 2 つのキーワードが提供されます。

NoDisplay

デフォルトで、エラー検出の通知情報ペインの上部に、ファイル要求から解釈されたフックの詳細が表示されます。以下に例を示します。

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

フック情報の詳細を通知情報ペインの上部に表示したくない場合は、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、フック要求に追加することによって拡張詳細が表示される Debug キーワードも提供されます。

通知情報ペイン

フック要求内に Debug キーワードがあれば、エラー検出の実行が完了してから、通知情報ペインの下部に拡張詳細が表示されます。通知情報ペインの下部に表示される詳細には、フックが成功した関数やフックされた関数が呼び出された回数などの各フックに関する統計値が含まれます。

エラーの詳細

通知情報ペインの下部に、関数がフックされなかったというメッセージが表示された場合は、その失敗したフックのデバッグを支援する詳細がエラー検出に表示されます。

Visual Studio で作業している場合は、出力ウィンドウにエラーの詳細が表示されます。

スタンドアロン バージョンのエラー検出を実行している場合は、**DbgView** ツールにエラーの詳細が表示されます。

UserAllocators.dat でエラーを診断する方法

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 エラー検出インストール ディレクトリの Data サブディレクトリに格納されたテキスト ファイルです。このファイルが削除されたり、読み取り不可にされたりすると、DevPartner エラー検出によってエラーがレポートされます。

- ファイル書き込み エラー

  DevPartner エラー検出セッションが開始されると、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]** でディレクトリの指定を変更するか、ディレクトリのプロパティを変更して、書き込み可能にします。

- 解析エラー

  DevPartner エラー検出で UserAllocators.dat ファイルの解析中にエラーが発生した場合、DevPartner エラー検出では [エラー] タブにエラーが記録されます。UserAllocators.dat エラーの発生時に [メモリの追跡] または [リソースの追跡] が有効な場合、これらの機能は無効になります。

トークン解析エラー

DevPartner エラー検出では、以下のルールを使用して、ファイルが一度に 1 行ずつ解析されます。

- 空白行、およびセミコロン（;）やダブルスラッシュ（//）で始まる行は無視されます。
- 追加される各 UserAllocator 定義は有効なレコード タイプで始まる必要があります。
- 各定義のすべてのパラメータは、1 つまたは複数のスペースかタブで区切り、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。パラメータでは、大文字と小文字が区別されることがあり、場合によっては、有効範囲内にある必要があります（たとえば、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したエントリがある場合にも、レコードが相互に競合する場合、エラーが生成されることがあります。UserAllocators.dat は高度な機能とみなされるため、広範な照合は行われません。

UserAllocators.dat の変更後に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 エラー検出では、ユーザー作成のアロケータ間のコールが監視されます。DevPartner エラー検出に対して API を正しく記述しなかった場合、アプリケーションがクラッシュしたり、予期しない動作をすることがあります。このような問題の最も一般的な原因の 1 つは、関数のパラメータの数を誤って指定することです。

一定のままのメモリの内容に依存し、記述に NOFILL オプションまたは NOPOSION オプションを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。お問い合わせの際には、以下の情報をお知らせください。

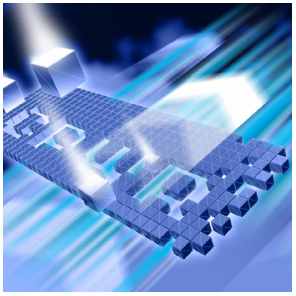
- 実行している DevPartner のバージョン
- UserAllocators.dat ファイルのコピー
- 発生している問題の詳細

場合によっては、ユーザー アロケータ関数を含む DLL のコピーと DLL へのリンクに使用されるマップ ファイルをご提供いただくこともあります。

可能な場合は、Ignore レコードは使用しないでください。Ignore レコードによって、アプリケーションの分析時に DevPartner エラー検出が予期しない応答をすることがあります。

第５章

デッドロック分析



- バックグラウンド：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方法
- 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、カスタマのアプリケーションで、デッドロック、潜在的なデッドロック、およびその他の同期エラーを検索する自動化された方法です。

この章では、以下の内容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
- デッドロックと潜在的なデッドロックの例
- 同期トピックの詳細情報のソース

バックグラウンド：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/C++ プログラムには、多数の関数を呼び出し、さまざまな処理を実行したあと終了する単純なメイン ルーチンがありました。これらのプログラムでは、シングル スレッドの実行が使用されました。つまり、プログラムでは一度に 1 つのインストラクションが実行されました。デバッガを使用してプログラムの手順を実行するとしたら、それぞれの処理が実行されていくのを映画の映像のように見ることができました。

スレッド

新しいアプリケーションは、マルチスレッドにできます。スレッドとは、コントロールのフローのことです。マルチスレッドのアプリケーションには、2つ以上のコントロールのフローがあります。Windows **CreateThread** 関数を使用すると、追加のスレッドを作成できます。**CreateThread** には、新たに作成したスレッドで実行される必要がある関数のアドレスを含む多数のパラメータがあります。**CreateThread** 関数が正常に実行されると、アプリケーションには追加の実行のスレッドが作成されます。

スレッドを暗黙的に作成するには、多数の方法があります。この例には、サードパーティ製ライブラリ、COM または DCOM、あるいは共通言語ランタイムを使用して、**_beginthread** を呼び出す方法があります。

プログラムで複数のスレッドを実行する場合、2つのスレッドで同じリソースに同時にアクセスを試みることができます。このリソースには、変数、ファイル、ハンドル、Windows リソースなどが含まれます。複数のスレッドが同じリソースに同時にアクセスしようとすると、同期化の問題が発生することがあります。たとえば、T1 と T2 という2つのスレッドで、どちらも1～100の数字を印刷しようとすると、各スレッドからの出力は以下のようになります。

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

両方のスレッドを同時に実行した場合、出力は以下の例のように不規則なものになります。スレッド T1 からの出力は標準体、スレッド T2 からの出力は太字の斜体です。

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

クリティカル セクション

このような問題を防ぐには、スレッド間の相互作用を調整する必要があります。最新のオペレーティング システムには一連の同期関数があり、これらの関数を呼び出して共有リソースへのアクセスを調整できます。最も使いやすく一般的な同期オブジェクトは、クリティカル セクションと呼ばれます。クリティカル セクションは、一度に1つのスレッドだけがリソースにアクセスできるようにする単純な関数です。

上記のように、1～100の数を印刷するように作成されたスレッド T1 と T2 の例について考えてみます。クリティカル セクション C1 を定義すると、両方のスレッドを実行する場合に不規則な出力を防ぐことができます。このクリティカル セクションでは、出力ストリームへのアクセスが制御されます。スレッド T1 と T2 によって実行される機能は、以下のように変更する必要があります。

1. いずれかのスレッドでクリティカル セクション C1 を作成します。
2. 次に、各スレッドで以下の手順を実行します。
   a. クリティカル セクション C1 を要求します。
   b. 1～100の数のリストを印刷します。
   c. クリティカル セクション C1 を解放します。

3 このあとスレッドは解放され、他のスレッドに影響を及ぼさない、要求されたその他の処理を実行します。

手順 2- a は、クリティカル セクション C1 へのスレッドの排他的アクセスを許可するようオペレーティング 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 セクションが使用できない場合、オペレーティング システムではスレッドを一時停止し、C1 が使用可能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すると、C1 のクリティカル セクション ルールに従うその他のスレッドでは、出力を印刷できません。スレッドで 1 ～ 100 の数が印刷されたあと、手順 2- c で、オペレーティング システムに **LeaveCriticalSection** が通知されます。これにより、他のスレッドのクリティカル セクションが解放されます。

プログラムのすべてのスレッドで、クリティカル セクションを使用して端末への出力を印刷する必要があると定めたルールはありません。ただし、このルールに従った場合、出力は常に正確に表示されます。

この同じルールは、変数、構造、ファイル、その他の共有リソースに適用できます。

メモ：2 つの出力ストリームが相互に衝突するコードを作成しないかぎり、ほとんどの場合、クリティカル セクションのコンソール出力をラップする必要はありません。

デッドロック－基本定義

前述の例に基づき、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許可する非常に単純なメカニズムであるといえます。ただし、クリティカル セクションは、問題の原因となることがあります。

C1、C2、および C3 という名前の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す。これらの各クリティカル セクションは、スレッド間で共有される個別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（たとえば、C1）へのアクセスを許可され、次に別のクリティカル セクション（たとえば、C2）へアクセスしようとした場合、C2 がすでに別のスレッドに割り当てられていることもあり得ます。別のスレッドがすぐに C2 を解放すれば、問題はありません。最初のスレッドは C2 が使用可能になるまで待機したあと、C2 へのアクセスを許可され、処理が続行されます。

一方、C2 を保有するスレッドが、別の同期オブジェクトが使用可能になるのを待機する必要がある場合（C1 など）、両方のスレッドが必要なリソースへのアクセスを待機して停止します。2 つ以上のスレッドが、使用可能になることのないリソースを待機して停止した場合、その結果はデッドロックと呼ばれます。

デッドロックを回避する方法

デッドロックは、複数のスレッドが共有リソースを使用しようとしたが、そのリソースにアクセス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。

- 同期オブジェクトが必要な場合にのみ、それらのオブジェクトへのアクセスを要求します。オブジェクトにアクセスしたら、できるだけ早く使用し、他のスレッドが使用できるように、そのオブジェクトを解放します。
- 同時に複数の同期オブジェクトにアクセスして特定の処理を実行する必要がある場合は、最初のオブジェクトを要求し、次に2番めのオブジェクトへのアクセスを試みます。2番めのオブジェクトが使用できない場合、両方のオブジェクトを解放し、少しの間待機します。待機が終了したら、再度リソースへのアクセスを試みます。別のリソースの待機中にスレッドがブロックされた場合、リソースの所有権を解放することが非常に重要です。オブジェクトを解放しないと、「致命的な囲い込み」を引き起こし、デッドロック状況をさらに悪化させるだけの場合があります。
- 常に同じ順序でリソースを要求します。たとえば、C1、C2、およびC3にアクセスして処理を実行する必要がある場合、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。
- 処理を実行する必要がある同期オブジェクトをすべて取得したら、別のリソースの待機をブロックする可能性がある処理を実行しません。

同期オブジェクトを処理する方法は、その他にも数多くあります。「追加情報」（84ページ）に、同期オブジェクトについて説明しているMSDNリソースと書籍を示します。

潜在的なデッドロック

DevPartnerエラー検出では、ユーザーが安全でない方法でリソースにアクセスしていることが検出されると、潜在的なデッドロックがレポートされます。この例として、T1、T2、T3の3つのスレッドがあり、これらのすべてのスレッドでクリティカル 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利用するアプリケーションの例があります。

表5-1に、各スレッドで特定の処理を実行するために必要なクリティカル セクションを示します。

表 5-1. 潜在的なデッドロックの例：スレッドとそれらのスレッドに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、単独で動作し、必要なクリティカル セクションを取得して、指定されたタスクを実行できます。ただし、3つのスレッドがすべて同時にこれらの処理を実行しようとすると、問題が発生することがあります。

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。このサンプル コードは以下の場所にあります。

```
...¥DevPartner Studio¥Examples¥DeadlockPhilosophers
```

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。

テーブルの周りに着席した哲学者は以下の3種類のことができます。

1. **休む**：休んでいる哲学者は座ったままで何もしません。休んでいる時間はランダムです。
2. **話す**：話をする哲学者は、興味を持って聞いてくれるテーブルの他のだれかに話しかけます。話をしている時間はランダムです。

3 **食べる：**空腹な哲学者は食べようとします。そのために、箸を取り上げようとします。最も単純な場合には、哲学者は常に左の箸を先に取り上げようとします。うまくいったら、次に右の箸を取り上げようとします。左と右の両方の箸を手にした哲学者は任意の期間食事します。それから箸を置き、休むか話し始めます。

最初の箸を取り上げられなかった哲学者は、数秒待ち、再度箸を取り上げようとします。うまくいった哲学者は、次に右の箸を取り上げようとします。右の箸が使用できない場合、哲学者は数秒待ち、再度右の箸を取り上げようとします。

すべての哲学者が同時に左の箸を取り上げた場合、問題が発生します。その場合、だれも左の箸を置かないため、全員が飢え死にしてしまいます（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。テーブルに哲学者と箸を追加すれば、実際のデッドロック数は減少する傾向があります。ただし、すべての哲学者が同時に空腹になる可能性はその場合にも存在します。これは、潜在的なデッドロックと呼ばれます。

潜在的なデッドロックは実稼動システムで発生することが多いため、多くの場合、追跡が最も困難なデッドロックです。これらの問題を開発システムで複製する試みは、通常、長時間を要し、多くの場合問題の核心が見つかりません。

潜在的なデッドロックが検出された場合、DevPartnerエラー検出によって、実際のデッドロックが発生するずっと前に通知されます。DevPartnerエラー検出では、実際のデッドロックが発生する方法を説明した詳細な情報が表示されます。これにより、より容易にコードを変更して問題の発生を防ぐことができます。

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すべての同期オブジェクトにおいても、エラーと以下のような問題のある用法が監視されます。

- ユーザーが指定した期間を超える待機
- すでに所有されているクリティカル セクションに再入するスレッド
- すでにスレッドによって所有されているMutexの待機
- 同期オブジェクトを解放せずにスレッドが終了する

また、DevPartnerエラー検出を設定して、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いるかどうかを検証できます。たとえば、プロセスの外部からアクセスできないように、同期オブジェクトを名前なしにする必要があるかどうかを判断できます。すべての名前付きオブジェクトは、潜在的なエラーとしてのフラグが付けられます。このリストを使用して、システムの他のプロセスによる不要なアクセスを防ぐために必要なセキュリティ記述子が、名前付きの同期オブジェクトに含まれているかどうかを確認できます。

すべての同期エラーのリストは、オンライン ヘルプで、検出されたエラーセクションのデッドロック エラーに表示されます。

その他の同期オブジェクト

Windows オペレーティング システムには、78 ページ で説明したクリティカル セクションの他にもさまざまな種類の同期オブジェクトがあります。同期オブジェクトのリストと、MSDN からの定義の抜粋を以下に示します。抜粋のテキストは斜体で示されています。それぞれの用語について、詳細な定義および関連するコードの例を示します。

クリティカル セクション

クリティカル セクション オブジェクト使用できるのがシングル プロセスのスレッドのみである点を除いて、クリティカル セクション オブジェクトでは、Mutex オブジェクトによって提供される同期と類似の同期が提供されます。

詳細な定義は、以下の URL を参照してください。

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/Critical_Section_Objects.asp

コードの例は、以下の URL を参照してください。

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using_critical_section_objects.asp

Event

イベント オブジェクトは、***SetEvent*** 関数を使用することで、状態が通知されるように明示的に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すシグナルをスレッドに送信する場合に役立ちます。

詳細な定義は、以下の URL を参照してください。

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/event_objects.asp

コードの例は、以下の URL を参照してください。

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using_event_objects.asp

Mutex

Mutex オブジェクトは、いずれかのスレッドによって所有されている場合は状態が通知され、所有されていない場合は状態が通知されないように設定される同期オブジェクトです。一度に 1 つのスレッドのみが Mutex オブジェクトを所有できます。

Mutex オブジェクトは、クリティカル セクションよりも大幅に速度が遅くなります。

詳細な定義は、以下の URL を参照してください。

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/mutex_objects.asp

Mutex オブジェクトの使用例は、以下の URL を参照してください。

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 オブジェクトとは、ゼロと指定した最大値間の数を保持する同期オブジェクトのことです。この数は、スレッドがセマフォ オブジェクトの待機を終了するたびに減少し、スレッドがセマフォを解放するたびに増加します。

詳細な定義は、以下の URL を参照してください。

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/semaphore_objects.asp

コードの例は、以下の URL を参照してください。

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using_semaphore_objects.asp

追加情報

MSDN リファレンス

同期オブジェクトの詳細については、MSDN で以下の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法：	**http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/using_synchronization.asp**
同期参照：	**http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/synchronization_reference.asp**

その他の参照

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 著

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

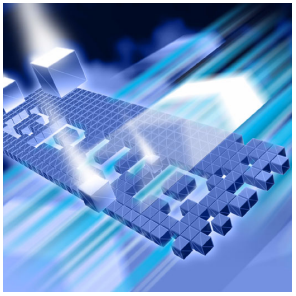
『Debugging Windows Applications, 1st Edition』John Robbins 著

『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、対応する対処法を実行してみてください。それでも問題が解決しない場合は、テクニカル サポートにお問い合わせください。

問題	対処法
エラー検出がメモリ上でステッピングする	• **[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をフィルする]** 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] ボタンで **[検出されたプログラム エラー]** ダイアログ ボックスを閉じると、デバッガでエラー検出が停止できなくなる	• **[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、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一時ブレークポイントが設定されます。一時ブレークポイントは、アプリケーションがデバッガ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、デバッガ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」エラーが表示され、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	• `new` と `delete` を使用していると、エラー検出から、ブロックが `free` によって解放されたことを通知されます。この場合、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++ コード生成]** プロパティ ページの **[ランタイム ライブラリ]** エントリで制御します。
エラー検出で、演算子 new と free の間での割り当ての不一致がレポートされる	• **`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。

問題	対処法
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	• マネージ デバッガを使ってマネージ C++ アプリケーション上でエラー検出を実行している場合は、アプリケーションではなく、`mscorwks.dll` または `mscorsrv.dll` から取得されたメモリ割り当てとリソース割り当てが表示されるため、エラー検出では無視されます。混合コードまたはマネージ コード アプリケーション上でエラー検出を使用する場合は、以下の手順に従います。 1 プロセスのデバッグに使用するバージョンの Visual Studio を開きます。 2 DevPartner エラー検出スタンドアロン 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 プロセスを開いて、**[プログラム]>[実行]** を選択し、そのプロセスを実行します。 4 **[ツール]>[プロセスにアタッチ]** をクリックして、ターゲット プロセスにデバッガをアタッチします。 **メモ**：**[マネージ コード デバッグ]**（**[混合]** や **[自動]** ではありません）を指定して、デバッガでマネージ コードのブレークポイントが正確にヒットするようにします。 この手順を使えば、マネージ コード内のブレークポイントを設定してヒットさせることができます。また、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。

問題	対処法

エラー検出下でアプリケーションを実行すると、「不正な文字列」エラーが表示される	• ASCII 文字列をワイド文字列にキャストして API パラメータとして使用すると、実際の問題と無関係なエラーが報告される可能性があります。キャストには、「私を信じてください。これがどのデータ型かはわかっています」という意味があり、エラー検出ではキャストが検出されません。以下に例を示します。
	
	`BSTR m_DSNName;`
	`m_DSNName=SysAllocString(BSTR(""));`
	
	コンパイラでは以下のように解釈されます。
	
	`m_DSNName=SysAllocString((wchar_t *)"");`
	
	エラー検出によって以下のようにレポートされます。
	
	不正な文字列：`SysAllocString` へのコールのアドレス `0x0FE2751F` で、ブロック `0x0FDF0000`（290816）が `NULL` で終了されていません。
	
	`SysAllocString` ステートメントでは、空の ASCII 文字列がワイド文字列にキャストされ、`SysAllocString` への入力パラメータとして使用されます。また、最初の 2 バイト NULL を検索することによって、`wchar_t` 文字列の最後が判断されます。さらに、この 2 バイト NULL の位置から割り出された長さを使って、新しい BSTR のサイズが特定されます。これによって、1 バイト ASCII 文字列の先頭とその後ろで偶然見つかった最初の 2 バイト NULL の間に存在するゴミのコピーが保存された `m_DSNName` の新しい BSTR が正常に作成されます。
	
	上の例では、エラー検出によって、入力文字列の `wchar_t`（実際には 1 バイトの空の ASCII 文字列）が検査され、不正な文字列（NULL 終了していない）が報告されます。このケースで使用したテスト コードでは、DevPartner エラー検出では、1 バイト ASCII 文字列の先頭と偶然見つかった 2 バイト NULL の間に有効なメモリ ブロックが存在することが認識されます。そのメモリが他の目的のために割り当てられていない場合にだけ、入力文字列のスキャンが行われます。そのため、このケースでは、数バイトがスキャンされただけで、そのメモリ領域に 2 バイト NULL が存在しないことが報告されます。この問題を避けるには、ワイド文字列を入力として使用しているコードを変更します。

問題	対処法
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が aspnet アカウントの制限下で実行される状況は 2 つあります。デフォルトでは、ASP.NET が有効な Web ページから DCOM または COM のアプリケーションやコンポーネントが起動されると、aspnet アカウント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 アカウントは制限付きアカウントになっています（これはユーザー グループのメンバで、同等の権限が設定されています）。このため、この状況では、COM コンポーネントにはエラー検出が正しく機能するために必要なセキュリティ権限がありません。この問題を回避するには、対話ユーザーのコンテキスト内で（dcomcnfg.exe を介して）実行されるように DCOM または COM アプリケーションやコンポーネントを設定する必要があります。以下の手順で、対話ユーザーのコンテキストで実行されるように DCOM または COM のアプリケーションやコンポーネントを設定します。 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 **[識別]** 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ないか、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 レポーティング] を無効にするか、**[すべてのタイプ]** ツリービューで選択するアセンブリの数を制限します。 • アプリケーション内のリークだけを検出したい場合は、リーク分析のみの有効化を検討します。**[メモリの追跡]** 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、リークの監視を除いて、メモリ追跡内のすべての機能が無効になります。メモリ追跡による、オーバーラン、未初期化メモリの使用、またはダグリング ポインタの監視が実施されなくなります。システム モジュールによって割り当てられたメモリの評価が行われないため、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
ログが大き過ぎる	• **[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 レポーティング]** を無効にするか、**[すべてのタイプ]** ツリー ビューで選択するアセンブリの数を制限します。 • **[API コール レポーティング]** の **[API 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、追跡するリソースを制限します。
エラー検出で `UserAllocators.log` ファイルの作成に失敗したという内容の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]** を設定して、`UserAllocators` ファイルを書き込むディレクトリを制御できます。
Windows Vista 上でエラー検出を使ってターゲット 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、ターゲット実行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、コール 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	• エラー検出が、実行ごとに、Microsoft Symbol Server からシンボルを取得することを試みています。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。 以下の Microsoft のシンボル ダウンロード ページを参照してください。 http://www.microsoft.com/japan/whdc/devtools/debugging/debugstart.mspx

問題	対処法
COM オブジェクトの追跡（または COM コール レポーティング）が正常に機能していないように見える	• **[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 レポーティング]** 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	• **[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をフィルする]** チェック ボックスをオフにします。アプリケーションで解放済みの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 バリデーション]** で、**[コール前に引数で指定されたバッファを規定値で埋める]** チェックボックスをオフにします。 • 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	• 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	• 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終了されたことが原因である可能性があります。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。 • エラー検出に有効な作業ディレクトリが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たつと、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。 • 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ゥルメントし、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	対処法
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シャットダウン時には、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 ブロックを再チェックすることによるメモリ オーバーランの検出などの数多くのチェックが実行されます。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の分析に失敗する	• エラー検出でサービスの分析に失敗する場合があります。一般的な原因は、プロセスの監視に使用されている 1 つ（または複数）のディレクトリが、作成するプロセスに対して書き込み可能になっていないことです。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。 • TMP と TEMP の環境変数は、プロセスに対して書き込み可能なディレクトリを参照している必要があります。LOCAL_SYSTEM コンテキストを実行するようにサービスを構成した場合は、こ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。
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	• 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。 • 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「プログラム設定を変更する」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	• FinalCheck でアプリケーションを完全にインストゥルメント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	• 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 リークが検出されずに、FinalCheck で検出される	• FinalCheck でインストゥルメントすると、より堅牢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タ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。

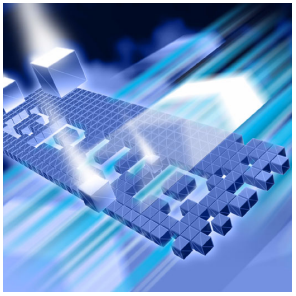
問題	対処法
エラー検出を指定して実行するとアプリケーションが失敗する（通常はメモリ追跡を有効にしている）	• **[オプション]**または**[設定]**ダイアログの**[メモリの追跡]**で、**[解放時に無効データをフィルする]**チェック ボックスをオフにします。 • **[オプション]**または**[設定]**ダイアログの**[メモリの追跡]**で、**[保護バイトを有効にする]**と**[確保時にフィルする]**のフィル パターンを入れ替えてみてください。 • 不具合のあるPDBファイルを使用している可能性があります。**NMShared¥4.7**ディレクトリで、**NMSYMDATA.DAT**という名前のASCIIテキスト ファイルを作成します。このファイルには、不具合のあるPDBファイルに関連付けされたモジュール名を含めます。後ろに「,0x0」を付加します。以下に例を示します。`ADVAPI32.DLL,0x0` • Visual Studio MFCアプリケーション ウィザードを使って生成されたアプリケーションのデバッグ中に、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た場合は、エラー検出によって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では、コンパイラが「最低限のデバッグ情報」を生成するあいまいなpragmaが設定されます。使用しているOSの構造体に余分なフィールドが追加されている場合は、エラー検出による構造体のサイズの計算時にデバッグ情報の中の誤ったサイズが使用されることによって、書き込み禁止メモリがフィルされる可能性があります。プロジェクトのC++プリプロセッサ設定ページでプリプロセッサ定義に`_AFX_FULLTYPEINFO`を追加して、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**という名前で`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名を追加します。1行に1つのモジュール名を記述します。以下に例を示します。**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	• Webアプリケーションのようなアプリケーションやサービス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（「サービスを分析する」（35ページ）を参照してください）。このオプションは、エラー検出アプリケーション（**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。

問題	対処法
デバッグ対象のモジュールを System files ディレクトリに配置したが、そのディレクトリが制限されているため、モジュールをデバッグできない	• **[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 `Unrestricted_modules.txt` というファイルを編集します。以下に例を示します。 **<*InstallationRoot*>¥Data¥Unrestricted_modules.dat** 含めるモジュール名を追加します。1行に1つのモジュール名を記述します。以下に例を示します。**MYCUSTOMDRIVER.DLL**
エラー検出で **dllhost.exe** や **TestCon32.exe** のエラーがレポートされる	• エラー検出で **dllhost.exe** や **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。
COM コール レポーティング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	• エラー検出では、認識対象として指定された COM インターフェイスのメソッドのみが記録されます。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 ボックスの **[COM コール レポーティング]** 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	• 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したあと、エラー検出がハングしたかのように長時間応答しない	• 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、エラー検出ではメモリ リーク、リソース リーク、およびインターフェイス リークの最終的な検出が行われます。
IIS が起動してすぐにハングする	• 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。
コマンド ライン バッチ プロセス（VCBUILD.EXE を使用）を通して自動コンパイルしたエラー検出用のコードがインストゥルメントできない	• コマンド ラインでコンパイラを呼び出してコードをインストゥルメントするには、DevPartner を使ってインストールした NMVCBUILD バリアントを使用する必要があります。NMVCBUILD によるコンパイル方法については、nmvcbuild を使用してネイティブ C/C++ コードをインストゥルメントするを参照してください。
Windows Vista 上でエラー検出用のインストゥルメンテーションを使ってターゲット アプリケーションを構築しようとするとエラーが表示される	• 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、ターゲット実行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。

問題	対処法
エラー検出下でアプリケーションを実行すると、不正なデータまたは予期せぬ結果が発生する	• コードに、未定義の評価順序に依存する式がないことを確認してください。評価順序が正しく定義されていないコードをインストゥルメントすると、不正なデータ、ハング、クラッシュなどの予期しない結果が生じることがあります。 C/C++ 標準では、オブジェクトへの値の格納など、「副作用」がある場合の評価順が明確に定義されていません。たとえば、以下のステートメントの評価順序はあいまいです。`i = ++i + 2;` このステートメントには、変数「i」に値を保存する部分が 2 か所ありますが、この言語ではそれらの実行順序が定義されていません。このようなコードをインストゥルメントした場合は、評価順序が固定ではないため、結果が予測できない可能性があります。
混合モード（マネージ/アンマネージ）環境でエラー検出から不正なリークとオーバーランが報告される	• マネージ デバッガを使ってマネージ C++ アプリケーション上でエラー検出を実行している場合は、アプリケーションではなく、`mscorwks.dll` または `mscorsrv.dll` から取得されたメモリ割り当てとリソース割り当てが表示されるため、エラー検出では無視されます。混合コードまたはマネージ コード アプリケーション上でエラー検出を使用する場合は、以下の手順に従います。 1 プロセスのデバッグに使用するバージョンの Visual Studio を開きます。 2 DevPartner エラー検出スタンドアロン 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 プロセスを開いて、**[プログラム]>[実行]** を選択し、そのプロセスを実行します。 4 **[ツール]>[プロセスにアタッチ]** をクリックして、ターゲット プロセスにデバッガをアタッチします。 **メモ**：**[マネージ コード デバッグ]**（**[混合]** や **[自動]** ではありません）を指定して、デバッガでマネージ コードのブレークポイントが正確にヒットするようにします。 この手順を使えば、マネージ コード内のブレークポイントを設定してヒットさせることができます。また、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



ファイルとその用途

セッション中の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下の表に示します。この表には、ファイルの場所、名前、目的、およびそのファイルをユーザーが変更可能かどうかが記載されています。

ファイル名とパス	用途
<*Program Root*>¥Data¥ctisafe.dat	ポインタの値を保存せずにポインタを受け取る既知の関数を指定します。MemTrack と FinalCheck では、このファイルを使って安全な関数の追跡が行われます。この情報によって、ポインタが未知の関数に渡されたときのエラーのトリガーが回避されます。ある関数がこのファイルに書き込まれると、MemTrack と FinalCheck では、その関数でポインタのコピーが保存されなかったと判断されます。 必要に応じて関数を追加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は、コンピュウェアのテクニカル サポートにご相談ください。
<*Program Root*>¥Data¥BCDefault.DPRul	エラー検出によってロードされる抑制ファイルとフィルタ ファイルのデフォルト セットをリストします。 エラー検出内の抑制とフィルタの編集用ダイアログを使って、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 ディレクトリでのみ有効です。追加した情報をシステム全体で有効にするには、手動でファイルを編集して、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 パス名を指定します。
<*Program Root*>¥Data¥*.DPFlt **<*Program Root*>¥Data¥*.DPSup**	エラー検出で使用されるフィルタと抑制を定義します。.DPFlt ファイルと .DPSup ファイルのそれぞれに、システム モジュール固有のフィルタと抑制が含まれています。 エラー検出の抑制とフィルタの編集用ダイアログを通して、抑制とフィルタを追加、変更、または削除します。これらのファイルは手動で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 ディレクトリに存在するかどうかに関係なく、無制限 DLL にするモジュールを指定します。システム ディレクトリに存在するモジュールは、エラーの検査から除外するようにマークする必要があります。デフォルトで、Unrestricted_modules.txt には、さまざまなバージョンの MFC モジュールが含まれています。 このファイルを手動で編集して、システム ディレクトリに存在する特定のモジュール名を追加すれば、エラー検出でそれらのモジュールのエラーが検査されます。
<*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.*	エラー検出のユーザー呼び出し可能インターフェイスへのアクセスを提供します。NMApiLib.h では、エラー検出に対するユーザー呼び出し可能インターフェイスが定義されます。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。ユーザー呼び出し可能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）除外するモジュールのリストが保存されています。モジュールは 1 行ごとにリストされます。例：MYCUSTOMDRIVER.DLL
<*Program Root*>¥DPSErrorDetection.xsd	セッション ファイル データを XML にエクスポートするときに使用されるスキーマ情報が保存されています。このファイルは編集しないでください。
<*NMShared Root*>¥NMSymData.dat	（ユーザー作成ファイル）不正な PDB ファイルに関連付けられたモジュール名（後ろに「,0x0 が付加される）が保存されています。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されるスキーマを指定します。

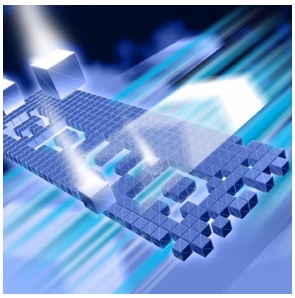
キー

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

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

メモ： 64 ビット バージョンの Windows では、DevPartner Studio は以下の場所にインストールされます。**¥Program Files (x86)¥Compuware¥DevPartner Studio¥** と **¥Program Files (x86)¥Common Files¥Compuware¥**

索引



ま

む

め

も

よ

り

ろ

わ

日本コンピュウェア株式会社
テクニカル・サポートのご案内

オンライン・サポート・サイト FrontLine Japan

コンピュウェアの製品およびサポートに関する追加情報は、FrontLine Japan で提供されています。

http://frontlinej.compuware.co.jp

FrontLine Japan のご利用には事前のユーザー登録が必要です。製品、サポートに関する重要な情報も配信されますので、是非この機会にご登録ください。

テクニカル・サービスデスク

営業時間　月～金　9:00～18:00（祝祭日、弊社休業日は除く）

コンピュウェア製品の利用に関しての技術的な質問やサポートに対するお問い合わせ窓口として、テクニカル・サービスデスクを設置しています。

お問い合わせの際は、FrontLine Japan のお問い合わせフォームをご利用ください。緊急時には、 フリーダイヤル 0120-188-540 までお電話ください。

テクニカル・サポートの詳しい内容は、FrontLine Japan をご覧ください。