ゲームプログラミング

▶　基礎編－第２回　デバッグ　◀

プログラミングの最終段階に行われるのがデバッグという作業です。デバッグとは、プログラムのミス(バグ)を取り除いていく作業です。エラーや警告のないプログラムでも、同じ場所で繰り返し強制終了したり、API呼び出しで必ずエラーになったり、さらに悪いことにはプログラムが何の規則もなく突然終了する場合があります。プログラムが正しく動作せず、その原因がわからないときには、デバッガを使って実行中のプログラムの内部状態をのぞいてみることになります。Visual C++には、高機能なデバッガ機能が備えられています。

ビルドエラー

ソースコードに誤りがあってビルドが失敗するとコンパイラやリンカがエラーメッセージを出力します。エラーメッセージはアウトプットウィンドウに出力されます。エラーメッセージは次のフォーマットで出力されます。

<ソースファイル名>(行番号) : <エラー(警告)コード> : <コメント>

コンパイルエラーと警告は、F4キーで発生した行を表示することができます。ビルド時に指摘されるエラーには、警告、コンパイルエラー、リンクエラーといったものがあります。

○警告

警告はエラーではありませんが、ときに致命的なバグを生む場合があります。たいていの警告は、浮動小数点を整数型の変数に代入した、符号付き変数と符号なし変数を比較した、といったものです。これらの警告はキャストすることによりなくすことができますが、本当にキャストが必要かどうかをよく考えないと、原因不明のバグに悩まされることになります。

○コンパイルエラー

コンパイル時に指摘されるエラーは、ほとんどが文法エラーです。セミコロンがない、変数名が違う、大文字と小文字を間違えた、といったものです。また、コンパイルエラーがないからといって、正しく動作するプログラムとは限りません。アルゴリズムが間違っている、関数の使用方法が間違っているといったものは指摘されません。これらはデバッガを利用しながら解決していきます。

○リンクエラー

ビルド作業の最後に行われるのが「リンク」です。プログラムはさまざまな関数を呼び出しながら処理を進めます。プログラムの実行時に、呼び出される関数がどこあるか知らないと困ってしまいます。そこで、実行する前にどのような関数が必要かを検知し、その場所を記録しておくか、またはその関数そのものをプログラムに取り込んでおく必要があります。リンクとは、このような作業のことを指し、いくつかのコンパイルされたファイルを連結する処理です。リンクを行うプログラムのことを「リンカ」と呼びます。

リンク処理のときに発生するエラーがリンクエラーです。たいていのリンクエラーは「外部ｼﾝﾎﾞﾙ～が未解決です」というものです。
コンパイラは、ソースコードをコンパイルするときに、他のソースコードからも参照することのできる変数や関数の名前を「シンボル」として記録しておきます。リンカは、コンパイラが記録したシンボルを見てリンク作業を行います。このときいずれかのファイルにも実体が存在しないエラーを報告します。このようなエラーを「シンボル未解決エラー」といい、以下のような場合に発生します。

- 関数名や変数名のスペルミスがある
  プロトタイプ宣言と関数本体はまったく同じ名前でなければなりません。特に、大文字と小文字は間違いやすいので気を付けましょう。また、関数本体が定義されていても、プロトタイプ宣言がない(ヘッダファイルがインクルードされていない)場合もエラーになります。
- プロジェクトにソースファイルを追加し忘れている
  プロトタイプ宣言が正しく行われていても、関数本体が別のソースファイルで定義されている場合は、そのファイルをプロジェクトに追加しないとリンクされません。
- 標準でないライブラリを使用している
  DirectXのようなあとからインストールしたライブラリや自作ライブラリなど、標準的にリンクさ

れないライブラリの中にある関数を使用しているのに、それを明示的にリンクしていない場合もリンクエラーになります。
リンクするライブラリを追加するには、メニューから「ﾌﾟﾛｼﾞｪｸﾄ(P)→設定(S)」でﾌﾟﾛｼﾞｪｸﾄの設定ダイアログを呼び出し、ﾘﾝｸﾀﾌﾞの「ｵﾌﾞｼﾞｪｸﾄ/ﾗｲﾌﾞﾗﾘ ﾓｼﾞｭｰﾙ(L)」に使用するライブラリを指定するか、ソースファイルの冒頭でプリプロセッサ「#pragma comment(lib, "ライブラリ名")」と宣言し、明示的にリンクするライブラリを指定します。

DebugモードとReleaseモード

VisualC++にはビルド時における設定をまとめた「プロジェクトの構成」というものがあり、これを切り替えることで必要に応じた実行ファイルが作成できるようになっています。ほとんどのプロジェクトではDebugモードとReleaseモードという２つのプロジェクト構成が作成されています。この２つのプロジェクト構成は、開発中はDebugモードを利用し、完成後にはReleaseモードを利用するというように使い分けると、目的に応じた実行ファイルが作成されるというわけです。DebugモードとReleaseモードの異なる点を以下の表にまとめます。

設定項目	Debugモード	Releaseモード
出力フォルダ	Debug	Release
ﾌﾟﾘﾌﾟﾛｾｯｻのﾏｸﾛ定義	_DEBUG	NDEBUG
デバッグ情報	生成する	生成しない
最適化	できない	ﾃﾞﾌｫﾙﾄは実行速度優先

「出力フォルダ」は、ビルド時に作成される.objファイルや.resファイルといった中間ファイル、最終的にリンクされる実行ファイルの置き場所として使われるフォルダです。これらのファイルはプロジェクト構成ごとに異なる内容で作られるため、別々のフォルダに保存されます。
「プリプロセッサのマクロ定義」は条件コンパイルで参照することにより、ソースコード中でどのモードでビルドされているのかを判断することができます。この定義は、すべてのソースコードの先頭で

```
Debugモード……#define _DEBUG
Releaseモード…#define NDEBUG
```

とするのと同等です。
「デバッグ情報」は、デバッガを使ってソースファイルを参照しながらデバッグするときに必要な情報で、Debugモードのときに生成されます。このファイルはサイズがかなり大きく、実行ファイルのサイズも大きくなってしまいます。拡張子が.pdbのファイルがデバッグ情報ファイルです。
「最適化」はリンク手前に行われ、実行速度を高めたり、実行ファイルのサイズを小さくするための処理です。最適化は非常に複雑な処理が行われており時間がかかります。最適化を行うと、動作は同じでありながらソースコードより質の高いコードが生成されます。

プログラムのデバッグ

プログラムにバグが発生したときは、バグを取り除くデバッグという作業を行います。ソースファイルを確認してもバグの原因が特定できない場合は、デバッガなどのような、デバッグをサポートするツールを使いながらデバッグをおいこないます。

○Debugモードデバッグ

Debugモードでは、デバッグ情報が作成されるためソースファイルを参照しながらデバッグすることができます。

デバッガの起動
デバッガはF5キー(またはﾋﾞﾙﾄﾞ(B)→ﾃﾞﾊﾞｯｸﾞの開始(D)→実行(G))で起動します。デバッガ起動時は、ブレークポイントの使用、ステップ実行、変数とメモリの内容の確認・書き換えといったことが行えます。

ブレークポイント
ブレークポイントとは、プログラムを特定の場所や条件で一時停止させる機能です。一時停止したプログラムは、変数の内容を確認したり、ステップ実行により実行順序を確認することができます。
VisualC++では、プログラムの位置に基づくブレークポイントとプログラムのデータに基づくブレークポイントの２種類のブレークポイントを設定できます。

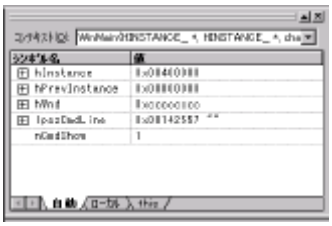
位置に基づくブレークポイントとは、プログラム中の命令に設定されたマーカーです。マークしておくと、その場所が実行されようとする直前にプログラムが停止します。
データに基づくブレークポイントとは、変数の値が間違って変更されているのに場所を特定できない、という場合に使用します。データブレークポイントを設定しておけば、変数の値が条件を満たしたときにプログラムが停止します。

ブレークポイントの設定

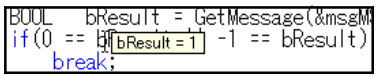
位置ブレークポイントのマークは、マークしたい命令の行にカーソルを移動して、F9キー(または をクリック）を押します。マークするとその行の左端に"●"が表示されます。ブレークポイントを解除するのもF9キーです(Ctrl+Alt+F9で全ブレークポイント解除)。
データブレークポイントを設定する場合は、ﾌﾞﾚｰｸﾎﾟｲﾝﾄダイアログを使用します。メニューから「編集(E)→ﾌﾞﾚｰｸﾎﾟｲﾝﾄ(K)」で呼び出すことができます。ﾃﾞｰﾀﾀﾌﾞをクリックし、デバッガに監視させたい変数名を式で入力します。式は"i == 100"や"nCount > 25"のように、if文などの条件式と同じように入力します。

デバッグウィンドウ

デバッガを起動すると、デバッグウィンドウを表示することができます。デバッグウィンドウには、変数の値を表示する「変数ウィンドウ」、メモリの内容を表示する「メモリウィンドウ」など、デバッグに有効なさまざまなウィンドウがあります。変数の値を表示するだけでなく、書き換えることもできます。変数の値を知りたいだけなら、「クイックウォッチ」という機能が便利です。これは、調べたい変数の上にマウスカーソルを合わせるだけです。
デバッグウィンドウが表示されない場合には、メニューから「表示(V)→ﾃﾞﾊﾞｯｸﾞｳｨﾝﾄﾞｳ(d)」を選択し、表示したいウィンドウを選びます。



▲変数ウィンドウ



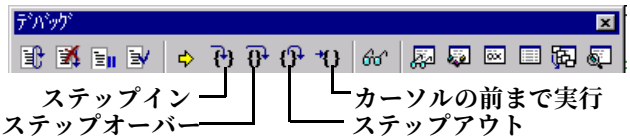
▲クイックウォッチ

▲コールスタックウィンドウ

デバッグウィンドウにはこのほかにも「コールスタック」というウィンドウがあります。コールスタックウィンドウでは、どのような関数呼び出しを経て現在の位置にプログラムが進んできたのかを知ることができます。このウィンドウでは、下に表示される関数が上にある関数を呼び出したことになります。ある関数を呼び出すコードが１か所だけであればあまり重要な情報ではありませんが、複数の場所から呼び出される場合は、どのような条件でバグが発覚するのかを特定するための有益な情報となることがあります。

ステップ実行

ステップ実行とは、ソースコードを１行実行するたびに実行を停止する機能です。ステップ実行を一回実行すると変数ウィンドウの内容が変化します。このとき値が書き換えられた変数があれば、赤い文字で表示されるようになります。ステップ実行には３種類あり、通常はステップオーバーを実行し、必要に応じてステップイン、ステップアウトを実行するという使い方をします。これらの操作は「デバッグツールバー」から行います。



「ステップオーバー」(F10)と「ステップイン」(F11)はプログラムを１行ずつ実行します。ステップオーバーは関数呼び出しも１行として扱い、ステップインは関数内に実行位置が移ります。
「ステップアウト」は現在実行中の関数から抜け出し、「カーソルの前まで実行」は現在のカーソル位置の直前までプログラムを実行して停止します。デバッグの中止はShift+F5キーです。

○Releaseモードデバッグ

最終的にプログラムはReleaseモードでビルドします。Debugモードで十分にデバッグしたプログラムでも、Releaseモードでのみ発現するバグが多々あります。
VisualC++では、Releaseモードでビルドしてあるということは、ソースコードデバッグを行うことができず、アセンブラレベルでのデバッグになる場合がほとんどです。このような場合には、以下のよう

な方法で場所を特定しながらデバッグすることになります。

printfデバッグ
これは、バグの原因と思われる要所要所にprintf関数をおいて、現在の状態を出力というもっとも原始的なデバッグ方法です。一般的なWindowsプログラムでは、printf関数を実行してもメッセージを出力できないので、以下のような方法で出力します。

・fprintf関数でファイルに書き込む
・sprintf関数、wsprintf関数で整形したメッセージをMessageBox関数で出力する

MAPファイルから強制終了した関数を特定する
Releaseモードのプログラムが何らかの理由で強制終了した場合はデバッグすることができません。こうした場合に得ることのできる唯一の情報が、エラーがおきたアドレスです。このアドレスがどの関数の中なのかを特定する簡単な方法が、MAPファイルを利用することです。MAPファイルには関数のアドレスが出力されており、エラーがおきたアドレスと比べることで強制終了した関数を特定することができます。
MAPファイルを生成するには、「ﾌﾟﾛｼﾞｪｸﾄ(P)→設定(S)」でﾌﾟﾛｼﾞｪｸﾄの設定ダイアログを呼び出し、ﾘﾝｸﾀﾌﾞの「MAP ﾌｧｲﾙを生成する(M)」にチェックを入れます。

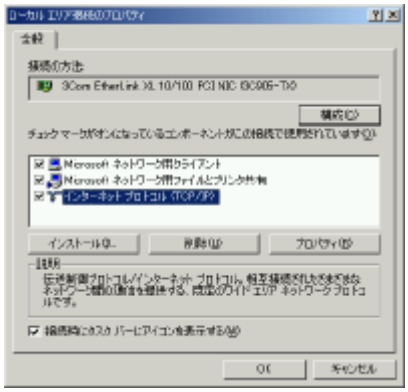
リモートデバッグ

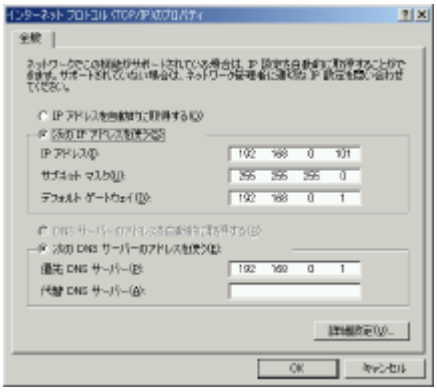
リモートデバッグとは、２台のマシンをＬＡＮで接続し、１台のマシンでプログラムを実行し、もう１台のマシンでプログラムをデバッグするというものです。ディスプレイを占有するゲームのようなアプリケーションでもデバッガを使いながらデバッグすることができます。
リモートデバッグでは、デバッガを起動し、プログラムを監視する側を「ホストマシン」、実際にプログラムを走らせる側を「リモートマシン」と呼びます。

２台のマシンにはTCP/IPプロトコルが必要です。仮にホストマシンのIPアドレスを"192.168.0.101"、リモートマシンのIPアドレスを"192.168.0.102"とします。IPアドレスの確認は、Windows98の場合は「ﾈｯﾄﾜｰｸｺﾝﾋﾟｭｰﾀで右クリック→ﾌﾟﾛﾊﾟﾃｨ」、Windows2000の場合は「ﾏｲﾈｯﾄﾜｰｸで右クリック→ﾛｰｶﾙｴﾘｱ接続で右クリック→ﾌﾟﾛﾊﾟﾃｨ」の順に選択していき、ネットワークのプロパティダイアログを開きます。次に、リストの中から使用するネットワークアダプタとプロトコル(TCP/IP)をダブルクリックし、TCP/IPのダイアログを開きます。

ホストマシン
IPｱﾄﾞﾚｽ：192.168.0.101

リモートマシン
IPｱﾄﾞﾚｽ：192.168.0.102



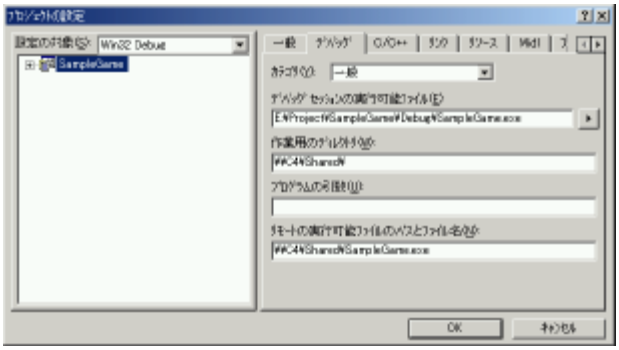


「次のIPアドレスを使う」が選択されており、IPアドレス欄にIPアドレスが入力されていることを確認します。

リモートマシンでは、ホストマシンが自由に読み書きできる「共有フォルダ」を設定しておきます。また、「VisualC++ ﾃﾞﾊﾞｯｸﾞﾓﾆﾀ」(MSVCMON.EXE)を起動し、正しく設定しておく必要があります。MSVCMON.EXEは、VisualC++をインストールしたフォルダ→COMMON→MSDEV98→BINにあります。

設定(S)でホストマシンのIPアドレスを入力し、接続(C)をクリックすれば、リモートマシンの設定は終了です。

ホストマシンでは、VisualC++を起動してプロジェクトを読み込み、メニューからﾌﾟﾛｼﾞｪｸﾄ(P)→設定(S)を選択し、リモートデバッグ用のプロジェクト設定を行います。



「設定の対象(S)」は必ず"Win32 Debug"を選択します。まず、ﾃﾞﾊﾞｯｸﾞタブから設定します。「ﾘﾓｰﾄの実行可能ﾌｧｲﾙのﾊﾟｽとﾌｧｲﾙ名(N)」に、リモートマシンで共有されているフォルダの共有名と実行ファイル名を、マシン名から正確に入力します。「作業用のﾃﾞｨﾚｸﾄﾘ(W)」は、リモートマシンのマシン名と共有フォルダ名を入力します。次に、ﾘﾝｸタブを選択し「出力ﾌｧｲﾙ名(N)」を「ﾘﾓｰﾄの実行可能ﾌｧｲﾙのﾊﾟｽとﾌｧｲﾙ名(N)」と同じものにしておきます。たとえばマシン名C4、共有フォルダ名Shared、実行ファイル名SampleGame.exeの場合は、それぞれ以下のように入力します。

作業用のﾃﾞｨﾚｸﾄﾘ(W)：\\C4\Shared\
ﾘﾓｰﾄの実行可能ﾌｧｲﾙのﾊﾟｽとﾌｧｲﾙ名(N)：\\C4\Shared\SampleGame.exe
出力ﾌｧｲﾙ名(N)：\\C4\Shared\SampleGame.exe
※共有フォルダにドライブを割り当てると、もう少し楽に設定できます。

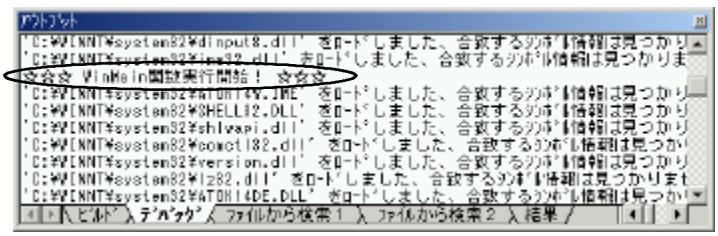
最後にメニューから「ﾋﾞﾙﾄﾞ(B)→ﾃﾞﾊﾞｯｶﾞﾘﾓｰﾄ接続(N)」を選択し、F5(またはF10/F11)キーでデバッガを起動します。このときダイアログが開くので、「この他のＤＬＬもデバッグ対象とする」チェックを外し、OKボタンを押すとリモートデバッグが開始されます。
なお、画像や音声をファイルから読み込む場合は、リモートマシンに正しくコピーしないと読み込まれません。

デバッグメッセージの出力

プログラムのデバッグ中は、変数の内容やデバッグ用のメッセージを出力しながら作業をしたい場合があります。「Releaseモードデバッグ」で紹介したような方法で出力することもできますが、APIにはデバッガ起動時に限ってアウトプットウィンドウにメッセージを出力する関数があります。
OutputDebugString関数は、出力したい文字列を引数に渡すとアウトプットウィンドウにその文字列を出力します。'\n'などの制御文字も使用できます。たとえば、

```
OutputDebugString("☆☆☆ WinMain関数実行開始！ ☆☆☆\n");
```

とすると、アウトプットウィンドウには以下のように出力されます。



プログラムのポイントにごとにOutputDebugString関数でメッセージを出力しておくと、プログラムがどこまで実行されたのか、どのような順序でプログラムが実行されているのかを確認することができます。変数の内容を出力したい場合は、sprintf関数やwsprintf関数を使って変数の内容を文字列に整形したあと、OutputDebugString関数に渡します。たとえば、変数iの内容を出力する場合は以下のようにします。

```
TCHAR   szDbgMsg[256];
wsprintf(szDbgMsg, "*** 変数iの内容:%d\n", i);
OutputDebugString(szDbgMsg);
```

なお、デバッガなしでプログラムを起動したときはOutputDebugString関数は無視されます。

エラーコードの取得とエラーメッセージの表示

プログラムの作成で非常に面倒なのがエラーの処理です。ファイルがない、変な値の入った変数を使用してしまった…など、予想しなければならないことが多いので大変です。かといって、エラー処理をおろそかにすると原因不明のアプリケーションエラーやプログラムの妙な動作に悩まされてしまうことになります。
特にAPIやDirectXでは、ファイルの処理をはじめ、直接ユーザや周辺機器とやり取りすることが多いため、エラーが発生しやすく、さまざまな種類のエラーが発生します。これらの正常でない状態が発生しても異常終了しないようにプログラムを作成しなければなりません。そのため、APIとDirectXのエラーの取り扱いをきちんと押さえておくことはよりよいプログラムを書き、ひいてはデバッグの楽なプログラムを書くために必須の事項といえます。

○エラーコード

エラーが発生した際に正しく後始末を行うためにはまず、正常に終了したのか、それとも何らかの理由で失敗したのかを知る必要があります。APIやDirectXでは、正常に処理が完了できたかどうかをエラーコードという形で返します。その返し方は以下のような種類があります。

①APIの戻り値として返す(API)
②GetLastError関数を使って返す(API)
③HRESULT型の戻り値を返す(DirectX)

いずれの方法にしても、返す値や詳細なエラー情報の返し方は関数によって異なるので、関数のリファレンスを調べて、その関数にあったエラーチェックを行う必要があります。

○エラーメッセージの取得

エラーコードは32ビットの数値で、それぞれのコードに対応したエラーが存在します。数値はプログラムからは扱いやすいのですが、エラーの原因を調べるときには不親切です。いちいちAPI32JPN.HLP(目次→エラーコード)などで調べなければならないからです。できれば、
「正常に終了しました」　「パラーメータが異常です」
と出力したいものです。APIとDirect3DXには、エラーコードからエラーメッセージを生成するための関数があります。これらの関数を使って、エラーコードを引数に与えると対応するエラーメッセージをアウトプットウィンドウに出力する関数を作成してみましょう。

・API編
APIのエラーコードからエラーメッセージを生成するのは非常に簡単です。APIでエラーが発生したら、ただちにGetLastError関数でエラーコードを取得します。取得したエラーコードをFormatMessage関数に渡すことにより、エラーメッセージを生成することができます。

```
/*****************************************************************************************************/
/*                                          エラーメッセージ出力                                     */
/*****************************************************************************************************/
void APIOutputErrorString(const DWORD dwErr)
{
    // エラーメッセージ生成
    LPVOID   lpBuffer;
    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
                  NULL, dwErr, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                  (LPTSTR)&lpBuffer, 0, NULL);

    // エラーメッセージ出力
    OutputDebugString("*** API ERROR - ");
    OutputDebugString((LPCTSTR)lpBuffer);

    // 生成したエラーメッセージを解放
    LocalFree(lpBuffer);
}
```

―説明―
APIOutputErrorString関数は、エラーコードからエラーメッセージを生成し、アウトプットウィンドウに出力します。

―パラメータ―
dwErr…エラーコード。GetLastError関数で取得したエラーコードを指定します。

―戻り値―
なし

―使用例―
```
    HWND    hWnd = CreateWindowEx(省略);
    if(NULL == hWnd)
        OutputErrorString(GetLastError());  // エラーの原因を出力
```

・DirectX編
DirectX9には、DirectXやGetLastError関数が返したエラーコードからエラーメッセージを生成するDXGetErrorDescription9関数と、関連づけられている名前を取り出すDXGetErrorString9関数があります。これらの関数を使用するには、ヘッダファイル<dxerr9.h>とライブラリ"dxerr9.lib"が必要になります。

```
/*****************************************************************************************************/
/*                                          エラーメッセージ出力                                     */
/*****************************************************************************************************/
void DXOutputErrorString(const HRESULT hrErr)
{
    // エラーメッセージ生成
    TCHAR   szErrStr[256];
    wsprintf(szErrStr, "*** Error String - %s(%s)\n",
            DXGetErrorDescription9(hrErr), DXGetErrorString9(hrErr));

    // エラーメッセージ出力
    OutputDebugString(szErrStr);
}
```

―説明―
DXOutputErrorString関数は、DirectXまたはGetLastError関数が返したエラーコードからエラーメッセージを生成し、アウトプットウィンドウに出力します(メッセージは英語です)。

―パラメータ―
hrErr…エラーコード。DirectXまたはGetLastError関数が返したエラーコードを指定します。

―戻り値―
なし

―使用例―
```
    HRESULT   hr = D3DXPrepareDeviceForSprite(g_lpD3DDevice7, FALSE);
    if(D3D_OK != hr)
        DXOutputErrorString(hr);   // エラーの原因を出力
```

条件コンパイル

プログラム開発中は、不安定なプログラムのためエラーチェックを厳重に行う必要があります。しかし、安定して動作するReleaseモードでは厳重なエラーチェックは必要ない場合があります。このように、DebugモードとReleaseモードで動作を変えたいときなどに使用するのが条件コンパイルです。条件コンパイルは、あるマクロが定義されているときとそうでないときで、コンパイルするプログラムを変えることができます。
条件コンパイルを行うには、条件コンパイルを行いたい部分をプリプロセッサ「#ifdef(#ifndef) マクロ名～#else～#endif」で囲みます。#ifdefは「もし～というマクロが定義されていたら」、#ifndefは「もし～というマクロが定義されていないなら」という意味になります。#elseは「そうでないなら」、#endifは「条件コンパイルの終了」を表します。条件コンパイルを乱用するとリストが見にくくなるだけでなく、処理が把握しづらくなるので多様は禁物です。
Debugモードではマクロ"_DEBUG"、Releaseモードではマクロ"NDEBUG"がすべてのソースファイルで自動的に定義されるようになっています。これを利用してそれぞれのモードに合わせた処理をさせることができます。たとえば、次のようにするとDebugモードのときのみチェックが行われます。

```
#ifdef _DEBUG
    if(NULL == g_lpDDraw7) {
        OutputDebugString("*** Error - DirectDraw未初期化\n");
        return false;
    }
#endif
```

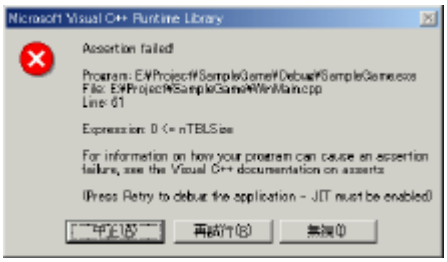
アサーション

アサーションは、プログラムが正しいかどうかのチェックで、違反している場合は強制的に実行を中断させます。適切なアサーションを使用したプログラムは、多くのバグや落とし穴を避けることができます。

C++の標準ライブラリassert.hは、assertマクロを提供しており、次のように関数のように使用することができます。

```
assert(診断式);
```

「診断式」の評価が偽なら実行は中断されて診断結果が出力されます。真の場合は実行を続けます。



▲assertマクロによる診断結果

上のダイアログボックスのように、assertマクロは問題を起こしたソースを細かく特定できるように、ファイル名と行番号の両方を知らせてくれます。ここで「中止(A)」を選ぶと、ダイアログを閉じ、プログラムの実行は終了します。「無視(I)」を選ぶとプログラムを続行することができます。

assertマクロは以下のようにポインタや引数のチェックに使用します。こうすることにより、ポインタや引数が必ず正しい値であることを保証することができます。

```
// 塗りつぶし関数
bool ColorFill(const HDC hDC, RECT* prcDest, const COLORREF rgbColor)
{
    assert(NULL != prcDest);        // ポインタのチェック。NULLなら中断
                :

// 配列初期化関数
bool InitTBL(int nTBL[], int nTBLSize)
{
    assert(0 <= nTBLSize);          // 引数チェック。配列サイズがマイナスなら中断
                :
```

assertマクロは、マクロNDEBUGが定義されているとき(Releaseモード時)は無視されるので、エラーチェックのための条件コンパイルは必要なくなります。

○ 課題 ○

エラー処理を行う関数を定義するError.cppとError.hのたりない部分を補って完成させましょう。

－Error.cpp－

```
/*
=============================================================================
                        ゲームプログラミング
     Programmed by Hibikino software. Copyright (c) 2003 Hibikino software. All rights reserved.
=============================================================================

  【対象ＯＳ】
     Microsoft Windows98/2000以降
```

```
【コンパイラ】
   Microsoft VisualC++ 6.0J ServicePack5

【プログラム】
   Error.cpp
            エラー処理

【履歴】
   * Version    0.00      2003/04/dd  hh:mm:ss  初版

================================================================================
*/

/*******************************************************************************/
/*                                インクルードファイル                         */
/*******************************************************************************/
#include "Error.h"

/*******************************************************************************/
/*                                スタティックライブラリ                       */
/*******************************************************************************/
#pragma comment(lib, ※ここは各自考えましょう)

/*******************************************************************************/
/*                                ＡＰＩエラーメッセージ出力                   */
/*******************************************************************************/
void APIOutputErrorString(const DWORD dwErr)
{
    ※関数の内容は各自考えましょう
}

/*******************************************************************************/
/*                                エラーメッセージ出力                         */
/*******************************************************************************/
void DXOutputErrorString(const HRESULT hrErr)
{
    ※関数の内容は各自考えましょう
}
```

－Error.h－

```
/*
================================================================================
                                ゲームプログラミング
    Programmed by Hibikino software. Copyright (c) 2003 Hibikino software. All rights reserved.
================================================================================

【対象ＯＳ】
   Microsoft Windows98/2000以降

【コンパイラ】
   Microsoft VisualC++ 6.0J ServicePack5

【プログラム】
   Error.h
            エラー処理ヘッダ

【履歴】
   * Version    0.00      2003/04/dd  hh:mm:ss  初版

================================================================================
*/

#pragma once

/*******************************************************************************/
/*                                インクルードファイル                         */
/*******************************************************************************/
※ここは各自考えましょう
※ここは各自考えましょう

/*******************************************************************************/
/*                                プロトタイプ                                 */
/*******************************************************************************/
```

```
※ここは各自考えましょう
※ここは各自考えましょう

/*****************************************************************************************************/
/*                                          インライン関数                                           */
/*****************************************************************************************************/
inline void APIOutputErrorString()
{
    APIOutputErrorString(GetLastError());
}
```