

# オブジェクト指向と ゲームプログラミング

## 基礎編 - 第2回 デバッグ

### 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、デバッグ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。VisualC++には、高機能なデバッグ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。これらはデバッグ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「外部シンボルが未解決です」というものです。

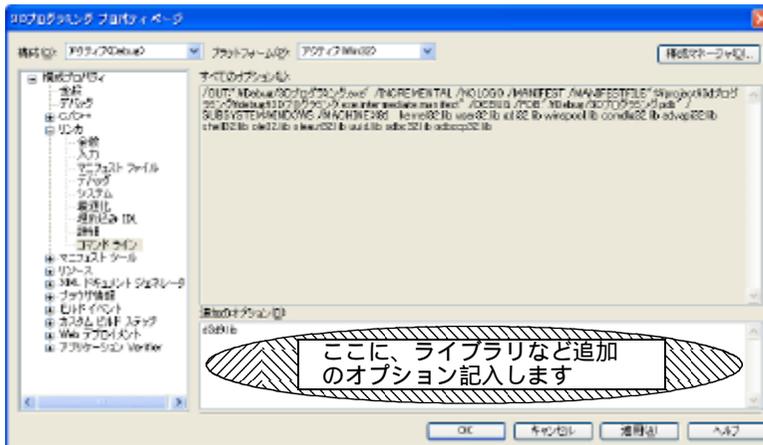
コンパイラは、ソースコードをコンパイルするときに、他のソースコードからも参照することのできる変数や関数の名前を「シンボル」として記録しておきます。リンカは、コンパイラが記録したシンボルを見てリンク作業を行います。このときいずれかのファイルにも実体が存在しないエラーを報告します。このようなエラーを「シンボル未解決エラー」といい、以下のような場合に発生します。

- ・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あればなりません。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「プロジェクト(P) (プロジェクト名)プロパティ(P)」でプロジェクトのプロパティページダイアログを呼び出してリンクの「コマンドライン」を選択し、「追加のオプション」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、リンクするライブラリを明示します。



## 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
プリプ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	デ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
```

とするのと同様です。

「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行います。

### 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。以下のような機能を用い、バグが発生している場所を特定していきます。

### デバッガの起動

デバッガはF5キー(またはデバッグ(D) 開始(S))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、あるアドレスの値が間違ってい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、設定したアドレスの値が変更され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

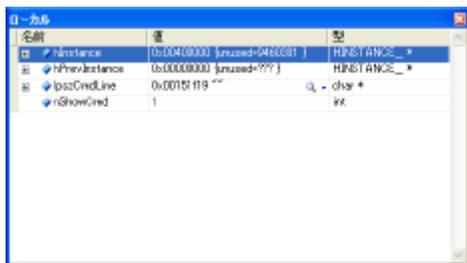
位置ブレークポイントのマークは、マークしたい命令の行にカーソルを移動して、F9キーを押します。マークするとその行の左端に" **B** "が表示されます。ブレークポイントを解除するのもF9キーです(Ctrl+Alt+F9ですべてのブレークポイントを解除)。

データブレークポイントを設定する場合は、メニューから「デバッグ(D) ブレークポイントの作成(B) メモリの変更時にブレーク(M)」を選択します。ダイアログに、アドレス、バイト数を設定すれば、そのアドレスの値が変更されたときに、プログラムが一時停止します。

### 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。

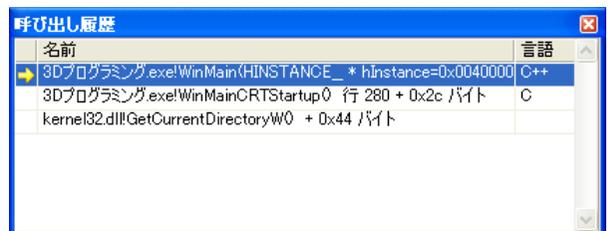
デバッグウィンドウが表示されない場合には、メニューから「表示(V) デバッグウィンドウ(d)」を選択し、表示したいウィンドウを選びます。



変数ウィンドウ



クイックウォッチ



呼び出し履歴ウィンドウ

デバッグウィンドウにはこのほかにも「呼び出し履歴」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ます。デバッグの中止はShift+F5キーです。

VisualC++では、初期化されない自動変数は0xccという値で初期化されます。実際には、関数の入口で自動変数が使用する領域(スタックフレーム)を0xccで埋め尽くしています。自動変数が0xccccccccで初期化されることで、未初期化データを容易に検出することができます。WindowsNT/2000/XPでは、0xccccccccというアドレスは一般アプリケーションからは読むことも書き込むこともできないので、アクセスした瞬間に例外が発生します。そのため、未初期化ポインタによるバグは、かなりの確率でアクセスした瞬間に補足できるはずですが、

このほか、Debugモードでは、変数の内容が以下のように設定されます。

値	内容
0xfdfdfdfd	ヒープ領域の末尾
0xdddddddd	解放されたメモリ領域
0xcdcdcdcd	未使用のメモリ領域
0xcccccccc	初期化されない自動変数

デバッグビルドでの初期値

## 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素要素にprintf関数において、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。

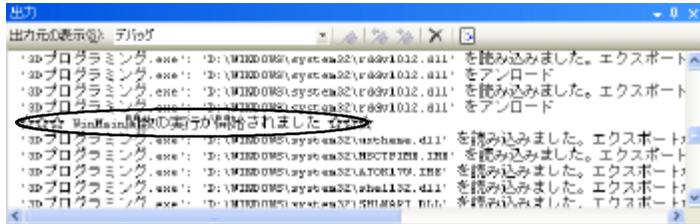
MAPファイルを生成するには、プロジェクトのプロパティページダイアログを呼び出し、リンクのデバッグ選択し、「マップ ファイルを生成」を「はい」に設定します。

## 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にはデバッグ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関数の実行が開始されました %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ごとに、この関数でメッセージを出力しておくことで、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。その返し方には、以下のような種類があります。

- APIの戻り値として返す(API)
- GetLastError関数を使って返す(API)
- HRESULT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。

```

void APIOutputErrorString(const DWORD inErr)
{
    // エラーメッセージ生成
    void* pBuffer;
    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER | FORMAT_MESSAGE_FROM_SYSTEM,
        NULL, inErr, MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPTSTR)&pBuffer, 0, NULL);

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

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

```

- 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 -

inErr...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(hWnd == NULL)
    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。

```

void DXOutputErrorString(const HRESULT inErr)
{
    // エラーメッセージ生成
    TCHAR ErrStr[256];
    wprintf(ErrStr, "*** Error String - %s(%s)%n",
        DXGetErrorDescription9(inErr), DXGetErrorString9(inErr));

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

```

- 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 -

inErr...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(pD3DDevice7, FALSE);
if(hr != D3D_OK)
    DXOutputErrorString(hr); // エラーの原因を出力

```

## 条件コンパイル

プログラムの開発中は、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(もしくは#endif) 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した。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(pd3D9 == NULL) {
        OutputDebugString("*** Error - Direct3D未初期化\n");
        return false;
    }
#endif
```

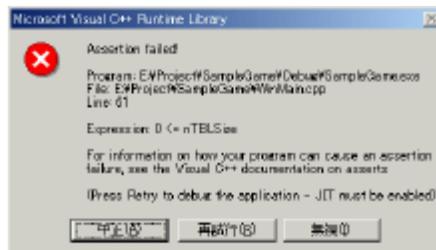
## アサーション

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* pDest, const COLORREF inColor)
{
    assert(pDest != NULL); // ポインタのチェック。NULLなら中断
    :
}

// 配列初期化関数
bool InitTbl(int tbl[], const int inTblSize)
{
    assert(inTblSize <= 0); // 引数チェック。配列サイズが0以下なら中断
    :
}
```

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。

## メモリリーク

newやmalloc関数で確保したメモリ領域を解放しないかったり、不正に解放することをメモリリークと呼びます。メモリリークは、プログラムを突然異常終了させ、どこで発生しているのが非常にわかりにくいという、非常に深刻な状態を引き起こします。

VisualC++では、メモリリークをレポートする機能が提供されています。レポートは、WinMain関数のあるソースファイルで<crtdbg.h>をインクルードし、以下のプログラムをWinMain関数の最初に記述すれば、アウトプットウィンドウに表示されるようになります。ただし、ダブルクリックしても該当箇所が表示されないので、発生箇所を見つけるのは少し困難です。

```
_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF|_CRTDBG_LEAK_CHECK_DF);
```

## 課 題

以下のプログラムは、エラー処理を行う関数を定義したものです。Error.hとError.cppの足りない部分を補ってプログラムを完成させましょう。

- Error.h -

```
/*
=====
                        オブジェクト指向ゲームプログラミング
Programmed by Hibikino software. Copyright (c) 2005 Hibikino software. All rights reserved.
=====
【対象OS】
Microsoft Windows2000/XP

【コンパイラ】
Microsoft Visual C++ 2005

【プログラム】
Error.h
        エラー処理ヘッダ

【履歴】
* Version    1.00    2005/02/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 ERROutputErrorStringAPI()
{
    APIOutputErrorString(GetLastError());
}
```

- Error.cpp -

```
/*
=====
                          オブジェクト指向ゲームプログラミング
Programmed by Hibikino software. Copyright (c) 2005 Hibikino software. All rights reserved.
=====
【対象OS】
  Microsoft Windows2000/XP
【コンパイラ】
  Microsoft Visual C++ 2005
【プログラム】
  Error.cpp          エラー処理
【履歴】
  * Version   1.00   2005/02/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 ERROutputErrorStringAPI(const DWORD inErr)
{
    関数の内容は各自考えましょう
}

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