オブジェクト指向とゲームプログラミング

▷　フレームワーク編 － 第８回　シングルトン　◁

シングルトン

シングルトン(Singleton)とは、「あるクラスのインスタンスがたった１つになることを保証し、それに対するグローバルなアクセス方法を提供する」というものです。
ディスプレイ、キーボード、システムタイマといった、アプリケーションの中でインスタンスが概念的に唯一となるクラスをモデル化する際に、Singletonを利用します。こういったクラスのオブジェクトが複数生成できてしまうのは、不自然なことであり、場合によっては危険なことになるのです。

Singletonはグローバル変数を改良したものです。グローバル変数を用いれば、ほかのソースファイルからも容易にアクセスできるようになりますが、それだけではインスタンスが複数生成されないようにすることはできません。
より良い解決策は、クラス自体がその唯一のインスタンスを管理する責任を持つようにすることです。クラスは、新しいオブジェクトの生成要求を制御してインスタンスが１つしか生成されないことを保証し、また、そのインスタンスにアクセスする方法も提供することができます。これがSingletonです。

Singletonは、以下のように、staticなメンバ関数とstaticなメンバ変数を用いれば、オブジェクトがいくつ生成されても各メンバは共有されるため、実現できるように思えます。

```
// アプリケーションクラス
class App {
public:
    static bool Init();     // 初期化
    static void Release();  // 解放
    static int  Main();     // メイン

private:
    static HWND m_hWnd;     // ウィンドウハンドル
};
```

このようなクラスの設計方法(デザインパターン)は、モノステート(Monostate)パターンと呼ばれるものです。monoは「１つの」、stateは「状態」という意味です。確かに「実質的にそのクラスのインスタンスが１つしか存在しない」のと同じ状態が実現されています。インスタンスがいくつ存在しても、すべてのメンバ変数がstaticなので、実質的に状態が１つしか取りえません。
しかし、この方法は、ある種の状況下ではいくつものデメリットを抱えているのです。たとえば、staticな関数は仮想関数にすることができません。App::Main関数のコードを修正しなければ、その動作を変更することが難しいという大きな問題があるのです。

Singletonは、通常以下のようなコードで実装します。

```
// ヘッダファイル Singleton.h
class Singleton {
public:
    static Singleton* GetInstance()     // 唯一のアクセスポイント
    {
        if(m_pInstance == NULL)
            m_pInstance = new Singleton;
        return m_pInstance;
    }

    … 通常のメンバ関数の定義 …

private:
    Singleton();                        // クラス外部でのオブジェクト生成を抑止する
    Singleton(const Singleton&);        // コピー生成の抑止

    static Singleton*   m_pInstance;    // 唯一のインスタンス格納域
};
```

```
// ソースファイルSingleton.cpp
Singleton*   Singleton::m_pInstance = NULL;
```

すべてのコンストラクタはprivateとなっているため、クラスの外ではオブジェクトを生成することはできません。Singletonクラス自身のメンバ関数、つまりGetInstance関数だけがオブジェクトの生成を許されています。このため、Singletonオブジェクトの唯一性をコンパイル時点で保証できるようになります。このようなクラスのデザインパターンを、Singletonパターンと呼びます。

GetInstance関数がまったく呼び出されない場合、Singletonオブジェクトは生成されません。このデザインパターンに掛かるコストは、GetInstance関数の先頭で行われる判定です。最初の要求があった時点で生成を行うという方法は、オブジェクトの生成に時間が掛かり、かつ滅多に使われないものである場合に有効となります。

これをポインタでなく、オブジェクトそのもので置き換えれば、判定がなくなり、実装を単純化できると考えられますが、その方法にはいくつかの問題があります。

```
// ヘッダファイル Singleton.h
class Singleton {
public:
    // 唯一のアクセスポイント
    static Singleton* GetInstance() { return &m_Instance; }

    … 通常のメンバ関数の定義 …

private:
    Singleton();                        // クラス外部でのオブジェクト生成を抑止する
    Singleton(const Singleton&);        // コピー生成の抑止

    static Singleton   m_Instance;      // 唯一のインスタンス格納域
};

// ソースファイルSingleton.cpp
Singleton   Singleton::m_Instance;
```

この方法では、プログラム起動時にSingletonオブジェクトが生成されます。プログラム起動時に生成されるオブジェクトの初期化順は、同じソースファイル内なら記述した順番に行われますが、異なるソースファイル間における初期化順はC++の規格では定義されていないため、複数のオブジェクトが相互に依存し合っている場合に、初期化の順序が問題となる場合があります。

これに対し、ポインタ版Singletonでは、最初のGetInstance関数呼び出し時にSingletonオブジェクトが生成されます。これは、オブジェクトの生成順を明確に指定できることを表しています。

シングルトンの破棄

ポインタ版Singletonでは、GetInstance関数が最初に呼び出された際にインスタンスが生成されますが、そのインスタンスの破棄について問題が起きる場合があります。

Singletonのインスタンスを誰が、どこで、いつ破棄をするのかという問題です。たいていの場合、プログラムの終了時にdeleteを用いて破棄すれば問題ありません。deleteを忘れてしまったとしても、WindowsなどほとんどのＯＳでは、アプリケーションが使用していたすべてのメモリを解放してくれるので、メモリリークは起きません。しかし、別のリークが発生するのです。それはリソースリークというものです。ウィンドウやフォント、ネットワークなどの資源(リソース)は、ＯＳによって完全に解放されることはありません。

Singletonの破棄に対する最も単純な解決策は、以下のように静的ローカル変数を用いることです。

```
Singleton& GetInstance()    // 唯一のアクセスポイント
{
    static Singleton   theInstance;
    return theInstance;
}
```

メンバ関数内のstaticな変数は、プログラムの実行が初めてそこに差し掛かった際に生成されます。この場合は、最初のGetInstance関数の呼び出し時に、変数theInstnaceが生成されることになります。破棄に関しても、後入れ先出し順(LIFO順：生成順の逆順で破棄される。最初に生成されたオブジェクトが最後に破棄される)で行われます。

この方法を用いれば、プログラム終了時に確実にデストラクタが呼び出され、その順序も指定できます。ただし、引数を取らないコンストラクタしか使用できない(引数を取るコンストラクタや=演算子で初期値を与えてしまうと、プログラム起動時にインスタンスが生成される)という問題があります。

課　題

CGameAppクラスをシングルトン化しましょう。

CGameAppクラスはアプリケーション全体を表すクラスです。その性質上、このクラスのインスタンスは１つのアプリケーションにつき１つあれば十分です。複数のインスタンスが存在してしまうと、資源の取り合いなど、相互に影響し合って思いがけないバグを生み出してしまう可能性があります。しかし、インスタンスが１つしかないという保証があれば、その前提条件の下でプログラミングできます。

(１)CGameAppクラスのすべてのコンストラクタをprivateに変更しましょう。

コンストラクタをprivateにすると、CGameAppクラスの外からはオブジェクトを生成することができなくなります。オブジェクトを生成している部分がエラーになるか、ビルドして確認しましょう。

(２)CGameAppクラスの唯一のオブジェクトを生成し、そのインスタンスを返却する関数を作成します。以下のプログラムを適切な場所に追加しましょう。

```
// シングルトンインスタンスの取得
static CGameApp& GetInstance()
{
    static CGameApp   theGameApp;
    return theGameApp;
}
```

CGameApp::GetInstance関数が初めて呼ばれたときに、CGameAppクラスのオブジェクトtheGameAppが生成されます。この関数は、CGameAppクラスの外で各メンバにアクセスする場合に使用します。以下のように記述すると、メンバにアクセスできます。

```
CGameApp::GetInstance().メンバ;
```

(３)インスタンスへのアクセスを容易にするため、以下のインライン関数を適切な場所に追加しましょう。

```
/*****************************************************************************************************/
/*                                          インスタンス取得                                         */
/*****************************************************************************************************/
inline CGameApp& App() { return CGameApp::GetInstance(); }
```

インライン関数Appにより、CGameAppクラスのメンバへのアクセスは、

```
App().メンバ;
```

と記述することができます。また、Releaseモードではインライン展開されるため、関数呼び出しのオーバーヘッドもなくなります。

(４)WinMain関数を以下のように変更しましょう。

```
int ?????? WinMain(????????? hInstance, ????????? hPrevInstance, LPTSTR lpCmdLine, ??? nShowCmd)
{
    if(App().Initialize(hInstance, true) == false)
        return -1;

    return App().Main();
}
```