ゲームプログラミング応用／ゲームプログラミング実践

10日で分かる C++言語 LEVEL④

■デザインパターン（シングルトンパターン）

C++のデザインパターンの1つで、クラスからインスタンス（実体）を1つしか生成しない（生成できない）事です。

ゲームの中で1つしか存在しない様なクラスの実体を作る時に、誤って複数作ったりしないように使われる事が多く、唯一無二の存在としてゲームの中で君臨するストイックな存在です。

1. インスタンス（実体）化とは？

クラスのコンストラクタが実行されて「オブジェクトが作られた」となる時が実体化の瞬間です。

※ヘッダーファイルでのクラスの定義は、まだ実体がない状態です。

```
class GameTask
{
    // 何かの定義
};
```

```
#include “GameTask. h”

void main(void)
{
    GameTask    gTask1;        // ①実体化されている
    GameTask* gTask2;        // ②実体化されていない（入れ物を作っただけ）
    GameTask* gTask3 = new GameTask();      // ③実体化されている
}
```

2. インスタンス（実体）が一つという事とは？

1つのクラスのオブジェクトを1つしか作ってはいけないというルールになっています。

```
GameTask    gTask1;        // GameTask ｸﾗｽの実体が 1 個目だから OK
GameTask    gTask2;        // GameTask ｸﾗｽの実体が 2 個目だから NG!
GameTask* gTask3 = new GameTask();     // new しても既に実体があると NG!
```

上記の手順だと、2行目以降は実体が2個以上になるのでシングルトンのルールから外れてしまいます。

このルールをプログラム的に「1つしか作れない」様に強制する事でシングルトンが達成される事になります。

3. ではどうするのか？

結論から言うと「コンストラクタをprivateにする」です。

通常、privateに設定しているメンバ関数、メンバ変数は、呼び出す他のクラスからは見えません。
つまりprivateに設定している関数は呼び出せないという事から、コンストラクタも関数なのでprivateにしてしまえば呼び出されなくなり、実体を作る事ができなくなる訳です。

```
class GameTask
{
private:
    GameTask();          // private のコンストラクタ
};
```

```
#include “GameTask.h”

void main(void)
{
    GameTask    gTask1();          // ①エラー  ※実体が作れないので怒られる

    GameTask* gTask2();
    gTask2 = new GameTask();     // ②エラー  ※実体が作れないので怒られる
}
```

4. 全く作れなくなりましたが、どうしましょう？

結論としては「実体（インスタンス）をクラス自身で作る」です。

privateのコンストラクタでも、自分のクラスでは見える訳なのでそこにお任せをしてしまえば良いのです。
ですので、自分自身で実体を作ってその場所を返す様な関数を作れば良いのです。

しかしながら、その関数自体が何度も呼ばれると、その度に実体が作られて無意味では？となります。
ですので、「static変数」と「static関数」を駆使して任務を遂行するのです。

static 変数

メンバ変数に static にすると、そのメンバ変数はオブジェクトではなくクラスが保持する事になります。もともとローカル変数に static を付けた場合は、スコープを抜けても変数の値が保持されるものとなるので関数を抜けてもそのまま残り、再度その関数が呼ばれても初期化されません。

意味合い的には「グローバル変数と同じ」になります。
ただ、グローバル変数との違いは、グローバル変数は main()が呼ばれると初期化されるのに対して、static なローカル変数は、その関数が呼ばれた時に初期化される事にあります。

static 関数

static なメンバ関数はクラス固有のものとなり、各オブジェクトの通常のメンバ変数にアクセスできません。メンバ変数にアクセスできないとはなんて不便だと感じますが、その代わりに「オブジェクトを生成しなくても呼び出しができる」というメリットがあるのです。

5. 結果的にどの様な手法になるのか？
   1. コンストラクタをprivateにする。
   2. 作成した自分自身の実体を保存する専用のstaticな変数準備する。
   3. そのstaticな変数を返す受け渡し用の関数をpublicで作成する。(オブジェクトが無くても使える)
   4. publicな受け渡し用の関数から「staticな変数の参照」を受け取り利用する。

6-1. とりあえず雛型を作ってみる

＜h ファイル＞　参照バージョン

```
class Singleton
{
private:
	Singleton();		// ①コンストラクタを private にする

public:
	static Singleton& getInstance();	// ③オブジェクトがなくても使える static 関数
	{
		static Singleton	obj;		// ②自分自身を作成して static 変数に保存
		return	obj;				// ③static な変数を返す(返り値が参照になっている)
	}

	void Update(void){}
};
```

使い方

```
#include“Singleton. h”

void main(void)
{
	Singleton& s = Singleton: :getInstance();	// インスタンスをゲットして保存
	s. Update();		// s の public 関数を使用できる(メンバ変数も同様)
}
```

この瞬間 Singleton クラス内の static 変数に Singleton オブジェクトが作られて保存されている。

6-2. ポインターで記述しても問題ない。

＜h ファイル＞　ポインターバージョン

```
class Singleton
{
private:
	Singleton();		// ①コンストラクタを private にする
public:
	static Singleton* getInstance();	// ③オブジェクトがなくても使える static 関数
	{
		static Singleton	obj;		// ②自分自身を作成して static 変数に保存
		return &	obj;			// ③static な変数を返す
	}

	void Update(void){}
};
```

使い方

```
#include“Singleton.h”

void main(void)
{
	Singleton* s = Singleton::getInstance();	// インスタンスをゲットして保存
	s->Update();		// s の public 関数を使用できる(メンバ変数も同様)
}
```

この瞬間 Singleton クラス内の static 変数に
Singleton オブジェクトが作られて保存されている。

□シングルトンまだダメなんです！

この状態で正攻法ではインスタンスを1つしか作れないので大丈夫ですが、実は他のやり方で複数作成ができてしまいます。それは、作成したインスタンス自体がコピーされてしまう事で複製ができてしまう事です。

1. コピーコンストラクタをprivateにしてコピーが出来ない様にする。

```
private:
	Singleton(const Singleton& s){};		// ①コピー禁止
```

※インスタンスのコピーができてしまう例

```
Singleton s1;
Singleton s2 = s1;	// 初期化時に他の Singleton 型で初期化する
```

※こうするとコピーコンストラクタが呼ばれる。→コピーコンストラクタを private にする。

2. 代入演算子をprivateにして別の入れ物に代入出来ない様にする。

```
private:
        Singleton& operator=(const Singleton&);    // ②代入禁止
```

※インスタンスの代入ができてしまう例

```
Singleton s1;
Singleton s2;
s1 = s2;     // 代入もできる。
```

※代入が出来ない様に指定した型のイコール演算子を private にして無効にしておく。

□シングルトンクラスの標準形

クラスの定義

```
class Singleton
{
private:
    Singleton(){}       // デフォルトコンストラクタ(外部から生成できない様に private に)
    Singleton(const Singleton& s){}       // コピーコンストラクタを private 化
    Singleton& operator=(const Singleton& s){} // 代入演算子(オーバーライド)private 化
    virtual ~Singleton(){}       // デストラクタを virtual にしておく
public:
    // Singleton::GetInstance()を使って Singleton のインスタンスを取得する事で
    // クラスへのアクセスが可能になる。
    static Singleton& GetInstance()   // 参照渡しの関数
    {
        static Singleton mInstance;       // 実体の生成
        return mInstance;                 // 参照で自分のインスタンスを返す
    };

    // ----- クラス関数(任意)
    void Init(void);                  // ①
    void Update(void);                // ②
};
```

利用の仕方

```
#include “Singleton. h”

void main()
{
    // ----- クラスを呼び出すと、最初のアクセス時にシングルトンインスタンスを作成する
    // ※以降、そのクラスは唯一存在するインスタンス(実体)となりメンバ関数などを使用できる
    Singleton::GetInstance(). Init();        // ① この時にインスタンスが作られる
    Singleton::GetInstance(). Update();    // ② その後は唯一のインスタンスを活用
}
```

■シングルトン活用（イメージマネージャー）

では早速、画像関係の管理クラスをシングルトンに変更していきましょう。

（例）ImageManager．h

```
#pragma once

// ----- シングルトンクラスを定義する
class ImageManager
{
private:
	// ----- コンストラクタ群
	ImageManager() {}		// デフォルトコンストラクタを private にする
	ImageManager(const ImageManager&){}	// コピーコンストラクタを private にする
	ImageManager& operator=(const ImageManager&){}	// 代入演算子のオーバーライドを private
	~ImageManager();		// デストラクタ
public:
	// ----- ImageMng オブジェクトの実体を返す(シングルトン)
	static ImageManager& GetInstance()
	{
		static	ImageManager	mImgInst;	// ImageMng の実体を生成。mImgInst に保存する
		return	mImgInst;	// 保持した mImgInst を返す
	}

	// ----- 変数（今回は読み込みなどを考慮せずに手抜きしています）
	//		本来は private 変数に読み込んで ID を返すなどの処置が必要です！
	int mSkyDomeModel = 0;		// スカイドーム

	// ----- 関数
	void LoadModelData(void);	// 3D モデル読み込み
};
```

1．手順1　コンストラクタをprivateにする（コピー、代入もできない様にする）

```
private:
	// ----- コンストラクタ群
	GameTask(){} // デフォルトコンストラクタを private にして外部から生成できない様にする
	GameTask(const GameTask&){}	// コピーコンストラクタを private 化
	GameTask& operator=(const GameTask&){} // 代入演算子のオーバーライドを private 化
   ~GameTask(){} // デストラクタ
```

2．手順2　自分のインスタンスを返すpublicなアクセサ(static Instance()関数)を作る

```
public:
	// 「static で定義された ImageManager 型のインスタンスを参照で返す」という意味
	static ImageManager& GetInstance()
	{
		static	ImageManager	mImgInst;	// ImageMng の実体を生成。mImgInst に保存する
		return	mImgInst;	// 保持した mImgInst を返す
	}
```

3. その他の関数は普通にcppに処理を書いていく

ImageManager. cpp

```
#include <DxLib.h>
#include "ImageManager. h"

void ImageManager::LoadModelData()
{
	mSkyDomeModel = MV1LoadModel("model/sky/BG_Sky1.x");
}

ImageManager::~ImageManager()
{
	MV1DeleteModel(mSkyDomeModel);
}
```

4. cpp側で実際に使用していく。

main.cpp

```
#include "DxLib. h"
#include "ImageManager. h"

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    // シングルトンの ImageManager を作成し LoadModelData()を呼び出す
    // ※以降、ImageManager は唯一存在するインスタンス(実体)として画像関係を管理する
    ImageManager::GetInstance();
    ImageManager::GetInstance(). LoadModelData();

    // ---------- ゲームループ
    while (ProcessMessage() == 0 && CheckHitKey(KEY_INPUT_ESCAPE) == 0)
    {
        ImageManager::GetInstance(). mSkyDomeModel = 0;   // 変数も普通にアクセスできる
    }
    return 0;
}
```

ImageManager::GetInstance().関数名　で呼び出せる訳です。

※ある意味グローバル変数です。

■シングルトン活用（キー入力マネージャー）

キー入力状態を管理するオブジェクトも唯一の存在で良いのでシングルトンクラスにしてみます。

1. 手順1　シングルトンの形でクラスを定義する

KeyMng. h

```
#pragma once

enum KEY_MODE {
    // キーの種類分
    KEY_UP = 0,
    KEY_RIGHT,
    KEY_DOWN,
    KEY_LEFT,
    KEY_A,
    KEY_B,
    KEY_PAUSE,
    KEY_MAX
};

// ----- シングルトンの KeyMng クラスを定義する
class KeyMng
{
private:
    // ----- コンストラクタ群
    KeyMng();     // デフォルトコンストラクタを private にして外部から生成できない様にする
    KeyMng(const KeyMng&){}     // コピーコンストラクタを private 化
    KeyMng& operator=(const KeyMng& g){}     // 代入演算子のオーバーライドを private 化
    ~KeyMng(){}     // デストラクタ
public:
    // ----- KeyMng オブジェクトの実体を返す（シングルトン）
    static KeyMng& GetInstance() {
        static KeyMng   mKeyInstance;     // KeyMng の実体を生成。mKeyInstance に保持
        return   mKeyInstance;
    }
    void Init(void);
    bool Update(void);       // キー状態を毎回ループで更新）
    void SetKeyConfig(int before, int after)

    // キー状態を保存
    int newKey[KEY_MAX];       // 今回フレームで押している状態
    int trgKeyDown[KEY_MAX];     // トリガ状態
    int  trgKeyUp[KEY_MAX];       // 離したキー状態
    int oldKey[KEY_MAX];          // 前フレームで押している状態
    int configKey[KEY_MAX];       // キーコンフィグ

};
```

この部分はキーチェックの仕組みを独自で作成しましょう。

2. その他の関数は普通にcppに処理を書いていく
　入力状況の確認などの関数はゲームループ毎に処理する方法で従来通りとなります。
ゲームパッド・多人数プレイヤー・アナログキー情報などを改良してみましょう。

KeyMng.cpp

```
#include “DxLib. h”
#include ”KeyMng. h”

KeyMng: :KeyMng()
{
	// ----- 初期化（全てのキーを押していない事にする）
	for(int i = 0;  i < KEY_MAX;  i++)
	{
		newKey[ i ] = 0;		// 今回の入力状況
		trgKeyDown[ i ] = 0;	// トリガキー
		trgKeyUp[ i ] = 0;	// アップキー
		oldKey[ i ] =  0;		// 前回の入力状況
	}
}

void KeyMng: :Update()
{
	// ----- 初期化（全てのキーを押していない事にする） ※oldKeyはクリアしない
	for(int i = 0;  i < KEY_MAX;  i++)
	{
		newKey[ i ] = 0;		// 今回の入力状況
		trgKeyDown[ i ] = 0;	// トリガキー
		trgKeyUp[ i ] = 0;	// アップキー
	}

	// ----- newKey
	if(CheckHitKey(KEY_INPUT_UP)) newKey[KEY_UP] = 1;
	if(CheckHitKey(KEY_INPUT_DOWN)) newKey[KEY_DOWN] = 1;
	if(CheckHitKey(KEY_INPUT_RIGHT)) newKey[KEY_RIGHT] = 1;
	if(CheckHitKey(KEY_INPUT_LEFT)) newKey[KEY_LEFT] = 1;
	if(CheckHitKey(KEY_INPUT_LSHIFT)) newKey[KEY_A] = 1;
	if(CheckHitKey(KEY_INPUT_LCONTROL)) newKey[KEY_B] = 1;

	// ----- trgKey
	for(int i = 0;  i < KEY_MAX;  i++)
	{
		trgKeyDown[ i ] = newKey[ i ] & ~oldKey[ i ];	// トリガキー
		trgKeyUp[ i ] = ~newKey[ i ] & oldKey[ i ];		// アップキー
		olfKey[ i ] = newKey[ i ];	// 次の入力チェックの為の準備
}

void KeyMng: :SetKeyConfig(int before, int after)
{
	configKey[ before ] = after;
}
```

3. cpp 側で実際に使用していく。

```
#include "DxLib.h"
#include "KeyMng.h"

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    // シングルトンの KeyMng を GetInstance で呼び出し生成しておく
    // ※以降、KeyMng は唯一存在するインスタンス(実体)としてキー入力を管理する
    KeyMng::GetInstance();

    // ---------- ゲームループ
    while (ProcessMessage() == 0 && CheckHitKey(KEY_INPUT_ESCAPE) == 0)
    {
        KeyMng::GetInstance().Update();    // キー状態を更新
        if(KeyMng::GetInstance().newKey[KEY_UP])
        {
            // UP キーが押された!
        }
    }
    return 0;
}
```

□ついでにキーコンフィグ

キー情報の番号を直接 newKey などの配列番号として利用するのでなく、番号毎に何番のキーが対応しているかを管理する変数(配列)を利用していきます。

KeyMng.cpp

```
#include "DxLib.h"
#include "KeyMng.h"

KeyMng::KeyMng()
{
    // ----- 初期化(全てのキーを押していない事にする)
    for(int i = 0; i < KEY_MAX; i++)
    {
        newKey[ i ] = 0;        // 今回の入力状況
        trgKeyDown[ i ] = 0;    // トリガキー
        trgKeyUp[ i ] = 0;      // アップキー
        oldKey[ i ] = 0;        // 前回の入力状況
    }

    // ----- デフォルトコンフィグ
    configKey[ KEY_UP ] = KEY_UP;
    configKey[ KEY_DOWN ] = KEY_DOWN;
    configKey[ KEY_RIGHT ] = KEY_RIGHT;
    configKey[ KEY_LEFT ] = KEY_LEFT;
    configKey[ KEY_A ] = KEY_A;
    configKey[ KEY_B ] = KEY_B;
    configKey[ KEY_PAUSE ] = KEY_PAUSE;
}
```

configKey[0]に何番のキーが登録されているかを保持して、そのキー番号を判定に使用します。

キー入力の際で登録されているキーが押されたかをチェックする為に configKey の配列を入れます。
又、どのキーを変更するかの関数を準備して、変更後のキーで上書きできる様にしておきます。

```
#include “DxLib. h”
#include ”KeyMng. h”

void KeyMng: :Update()
{
	～ 省略 ～

	// ----- newKey
	if(CheckHitKey(KEY_INPUT_UP)) newKey[ configKey[KEY_UP] ] = 1;
	if(CheckHitKey(KEY_INPUT_DOWN)) newKey[ configKey[KEY_DOWN] ] = 1;
	if(CheckHitKey(KEY_INPUT_RIGHT)) newKey[ configKey[KEY_RIGHT] ] = 1;
	if(CheckHitKey(KEY_INPUT_LEFT)) newKey[ configKey[KEY_LEFT] ] = 1;
	if(CheckHitKey(KEY_INPUT_LSHIFT)) newKey[ configKey[KEY_A] ] = 1;
	if(CheckHitKey(KEY_INPUT_LCONTROL)) newKey[ configKey[KEY_B] ] = 1;

	～ 省略 ～
}

void KeyMng: :SetKeyConfig(int before, int after)
{
	configKey[ before ] = after;
}
```

キーを変更したい場合は SetKeyConfig()関数で、どのキーをそのキーに変更したかをセットします。

```
#include ″DxLib. h″
#include ″KeyMng. h″

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    // KEY_A を KEY_B に変更
    KeyMng: :GetInstance(). SetKeyConfig(KEY_A, KEY_B);

}
```

```
1:デフォルト
    configKey[ KEY_A ] = KEY_B; なので
    if(KeyMng: :GetInstance(). newKey[ configKey[ KEY_A ] ])
    {
        // KEY_A を押した時
    }

2:KeyMng: :GetInstance(). SetKeyConfig(KEY_A, KEY_B)で変更後
    if(KeyMng: :GetInstance(). newKey[ configKey[ KEY_A ] ])
    {
        // KEY_A は KEY_B になっているので KEY_B を押した時となる
    }
```

■まとめ

このドキュメントで出て来た用語の一覧です。各キーワードの理解はいかがでしょうか。

・デザインパターン（シングルトン）
・シングルトン活用（イメージマネージャー）
・シングルトン活用（キー入力マネージャー）