後期課題１

２日で分かるC++言語応用編②

■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ｵﾌﾞｼﾞｪｸﾄがなくても使える static 関数
    void Update(void){}
};
```

＜cpp ファイル＞　参照バージョン

```
#include “Singleton.h”

// ③ｵﾌﾞｼﾞｪｸﾄがなくても使える static 関数
Singleton& Singleton::getInstance()
{
    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
}
```

使い方

```
#include “Singleton.h”

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

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ｵﾌﾞｼﾞｪｸﾄがなくても使える static 関数
    void Update(void){}
};
```

＜cpp ファイル＞　ポインターバージョン

```
#include “Singleton.h”

// ③ｵﾌﾞｼﾞｪｸﾄがなくても使える static 関数
Singleton* Singleton::getInstance()
{
    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
}
```

使い方

```
#include “Singleton.h”

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

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ｺﾋﾟｰ禁止
```

※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(){}     // ﾃﾞ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){}     // ｺﾋﾟｰｺﾝｽﾄﾗｸﾀを private 化
   Singleton& operator=(const Singleton& s){} // 代入演算子(ｵｰﾊﾞｰﾗｲﾄﾞ)private 化
   virtual ~Singleton(){}     // ﾃﾞｽﾄﾗｸﾀを 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 instance;     // 実体の生成
      return instance;               // 参照で自分のｲﾝｽﾀﾝｽを返す
   };

   // ----- ｸﾗｽ関数(任意)
   void SystemInit(void);   // ①
   void Init(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ｸﾞ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)となりﾒﾝﾊﾞ関数などを使用できる
   Singleton::GetInstance().SystemInit();  // ① この時にｲﾝｽﾀﾝｽが作られる
   Singleton::GetInstance().Init();        // 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（画面遷移）

では早速、ゲームループ部分をシングルトンに変更していきましょう。

GameTask.h（例）

```
#pragma once

class CHARA_BASE;
class Player;
class Enemy;
class Shot;
class CollisionCheck;

enum GAME_MODE {
    GAME_INIT,
    GAME_TITLE,
    GAME_MAIN,
};

class GameTask
{
private:
    int SystemInit(void);    // ｼｽﾃﾑ初期化
    int GameInit(void);
    int GameTitle(void);
    int GameMain(void);

    GAME_MODE  gameMode;    // 画面遷移管理用
    int newKey;
    int oldKey;
    int trgKey;

    Player* p; // ﾌﾟﾚｲﾔｰｵﾌﾞｼﾞｪｸﾄ
    Enemy* e[10];  // 敵ｵﾌﾞｼﾞｪｸﾄ
    Shot* s;
    CollisionCheck* cCheck;
public:
    GameTask();    // ｺﾝｽﾄﾗｸﾀ
    ~GameTask();   // ﾃﾞｽﾄﾗｸﾀ
    int GameUpdate(void);   // ｹﾞｰﾑﾙｰﾌﾟﾒｲﾝ関数
};
```

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(){} // ﾃﾞ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&){}    //ｺﾋﾟｰｺﾝｽﾄﾗｸﾀを private 化
    GameTask& operator=(const GameTask&){} // 代入演算子のｵｰﾊﾞｰﾗｲﾄﾞを private 化
   ~GameTask(){} // ﾃﾞｽﾄﾗｸﾀ
```

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 GameTask 型のインスタンスを参照で返す」という意味
    static GameTask& GetInstance(void){
        static GameTask gInstance; // GameTask の実体を生成し gInstance に保存する
        return gInstance;  // その実体を返す
    }
```

static 関数である事に注目！実体なしで使える。

3. 手順3　最初に呼び出したい関数をpublicに置いておく。

```
private:
    void SystemInit(void);
```

⇩

```
public:
    void SystemInit(void);
```

4. cpp側で実際に使用していく。

main.cpp

```
#include "DxLib.h"
#include "GameTask.h"

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
   // ----- ｼﾝｸﾞﾙﾄﾝの GameTask を作成し SystemInit()を呼び出し初期化を行う
   // ※以降、GameTask は唯一存在するｲﾝｽﾀﾝｽ(実体)として画面遷移を管理する
   GameTask::GetInstance().SystemInit();

   // ---------- ｹﾞｰﾑﾙｰﾌﾟ
   while (ProcessMessage() == 0 && CheckHitKey(KEY_INPUT_ESCAPE) == 0)
   {
      ClsDrawScreen();

      // 以降 Update()関数の中でタイトル～ゲームメイン～クリアなどの分岐を行う
      GameTask::GetInstance().Update();

      ScreenFlip();
   }
   DxLib_End();
   return 0;
}
```

GameTask::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。

5. 関数ポインタで画面の分岐を行う場合・・

GameTask.h

```
private:
   // ｹﾞｰﾑﾙｰﾌﾟ用関数ﾎﾟｲﾝﾀｰを定義
   int (GameTask::*gLoopPtr)(void);
```

GameTask.cpp

```
int GameTask::SystemInit()
{
   ※DxLib の初期化など

   // ｹﾞｰﾑﾙｰﾌﾟ(関数ﾎﾟｲﾝﾀ)を GameInit に変更
   gLoopPtr = &GameTask::GameInit;
}

int GameTask::Update()
{
   // 関数ﾎﾟｲﾝﾀｰを利用してそれぞれの画面の関数を実行する
   int rtnID = (this->*gLoopPtr)();  // ｹﾞｰﾑﾙｰﾌﾟ(関数ﾎﾟｲﾝﾀｰ)
   return rtnID;
}
```

画面を切り替えたい時

```
gLoopPtr = &GameTask::実行させたい画面の関数;

gLoopPtr = &GameTask::GameInit;
gLoopPtr = &GameTask::GameTitle;
gLoopPtr = &GameTask::GameOver;

など・・
```

■シングルトン活用（キー入力クラス）

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。

KeyMng.h

```
#pragma once

enum KEY_MODE {
   // ----- P1
   P1_UP = 0,
   P1_RIGHT,
   P1_DOWN,
   P1_LEFT,
   P1_A,
   P1_B,
   P1_PAUSE,
   KEY_MAX
};

// ----- ｼﾝｸﾞﾙﾄﾝの KeyCheck ｸﾗｽを定義する
class KeyMng
{
private:
   // ----- ｺﾝｽﾄﾗｸﾀ群
   KeyMng();   // ﾃﾞ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&){}   // ｺﾋﾟｰｺﾝｽﾄﾗｸﾀを private 化
   KeyMng& operator=(const KeyMng& g){}   // 代入演算子のｵｰﾊﾞｰﾗｲﾄﾞを private 化
   ~KeyMng(){}   // ﾃﾞｽﾄﾗｸﾀ
public:
   // ----- KeyMng ｵﾌﾞｼﾞｪｸﾄの実体を返す(ｼﾝｸﾞﾙﾄﾝ)
   static KeyMng& GetInstance() {
      static KeyMng keyInstance;   // KeyMng の実体を生成。keyInstance に保持
      return keyInstance;
   }
   void Init(void);
   bool Update();   // ｷｰ状態更新(毎ﾙｰﾌﾟ更新)

   // 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 trgKey[KEY_MAX];   // ﾄﾘｶﾞ状態
   int upKey[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
};
```

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。

シングルトン以外は、毎ループキー入力状態を確認して配列に保存する仕組みは今まで通り。

```
#include  “KeyMng.h”
void main()
{
   //KeyMng& k = KeyMng::GetInstance(); // ｲﾝｽﾀﾝｽを参照で受け取りそれを使う方法も可
   //k.Update();
   KeyMng::GetInstance().Update();   // 最初の呼び出しでｼﾝｸﾞﾙﾄﾝになり毎ﾙｰﾌﾟで呼び出される。
```

■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（画面遷移）
・シングルトン活用（キー入力）