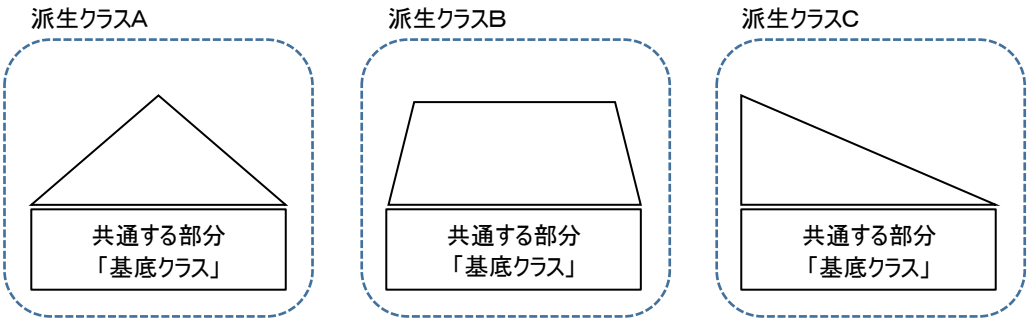
ゲームプログラミング応用／ゲームプログラミング実践

10日で分かる C++言語 LEVEL②

■継承

オブジェクト指向言語の重要な特性の一つに「継承（インヘリタンス）」があります。複数のオブジェクトを作る際に共通する部分が出て来た場合、その共通部分を基本のクラスとして設定しておき、実際のそれぞれのクラスは基本クラスの性質に独自の機能を拡張していくイメージになります。

その時の継承のもととなるクラスを「基底クラス（スーパークラス）」と言い。その基底クラスを継承して独自の機能を実装したクラスを「派生クラス（サブクラス）」と言います。



定義の仕方は、派生クラスを定義する時にベースとなる基底クラスを指定する形になります。

```
class 派生クラス名 : public 基底クラス名
```

CHARA_BASE.h　※ベースになるクラスは大文字などで表現すると分かり易い

```
class ACTOR_BASE
{
public:
	int mPosX;
	int mPosY;
};
```

Player.h

```
#include"ACTOR_BASE.h"
class Player : public ACTOR_BASE
{

};
```

Enemy.h

```
#include"CAHRA_BASE.h"
class Enemy : public ACTOR_BASE
{

};
```

ACTOR_BASEクラスを基底クラスとして、それぞれPlayerとEnemyの派生クラスを作成します。

今回の場合は、Player、Enemyのクラスから基底クラスで定義したint型変数mPosX、mPosYが使用できます。

■オーバーロード

ポリモーフィズムの一つで、コンストラクタを含め、引数や戻り値が違うものであれば「同じ名称で全く違う複数のメンバ関数を定義する事ができる」というものです。引数がない場合の初期化や、引数を渡す場合の初期化など、用途に合わせた処理を同じ名称の関数で実行できる便利な機能となります。

Enemy.h

```
class Enemy  :  public ACTOR_BASE
{
public:
    int mX;
    int mY;
    int mType;

    Enemy();                  // ①引数のないコンストラクタ(関数)
    Enemy(int, int);          // ②引数が 2 個のコンストラクタ(関数)
    Enemy(int, int, int);     // ③引数が 3 個のコンストラクタ(関数)
};
```

Enemy.cpp

```
Enemy::Enemy()                // ①
{
    mX = 0;
    mY = 0;
    type = 0;
}

Enemy::Enemy(int a, int b)    // ②
{
    mX = a;
    mY = b;
    mType = 0;
}

Enemy::Enemy(int a, int b, int t)     // ③
{
    mX = a;
    mY = b;
    mType = t;
}
```

main.cpp

```
void main()
{
    Enemy* e;

    // ①    e = new Enemy();
    // ②    e = new Enemy(100, 200);
    // ③    e = new Enemy(100, 200, 5);
}
```

デフォルトコンストラクタ

同じ関数名でも、引数の中身に応じたそれぞれの関数を呼び出す事ができる。

プログラム例の様に、オブジェクト生成する場合のコンストラクタに引数の渡し方を変えると、その引数に応じたメンバ関数が呼ばれる様になり、それぞれの処理を実行させる事ができます。（初期化やパラメータ数に応じた機能分けなどをする事ができます）。

尚、引数に何もないコンストラクタの事を「デフォルトコンストラクタ」と言います。

■オーバーライド

こちらもポリモーフィズムの一つで、親子関係にあるクラスにおいて基底クラスと派生クラスに同名・同型の関数がある場合、派生クラスの関数が基底クラスの関数を上書きするイメージになります。

※基底クラスの関数はそのまま残ります。基底クラス＋派生クラスの2つの関数になります。

ACTOR_BASE.h

```
class ACTOR_BASE
{
public:
	void Init()
	{
		// 処理①
	}
};
```

Player.h

```
class Player  :  public ACTOR_BASE
{
public:
	void Init()
	{
		// 処理②
	}
};
```

main.cpp

```
void main()
{
	Player* player = new Player();
	ACTOR_BASE* aActor = new ACTOR_BASE();
	ACTOR_BASE* pActor = new Player();

	player->Init();		// 処理②を実行
	aActor->Init();		// 処理①を実行
	player->Init();		// 処理①を実行　※Player を new したのに ACTOR_BASE 扱いになっている
}
```

オーバーライドされた関数は**原則的に派生クラスの方の関数を実行**します。

注意点！
- メンバ変数の名前を統一して記述ミスを減らす。
- 原則的に同じ機能には同じ名前を付けて処理に統一感を持たせる。

□派生クラスのコンストラクタとデストラクタの順番は？

オブジェクトが生成されるとコンストラクタが呼ばれ、オブジェクトが削除されるとデストラクタが呼ばれますが、親クラスのコンストラクタとデストラクタの呼ばれるタイミングはどうなるのでしょう？

順番としては

・コンストラクタ・・・・・基底クラスのコンストラクタ　→　派生クラスのコンストラクタ

・デストラクタ・・・・・・派生クラスのデストラクタ　→　基底クラスのデストラクタ

となります。

が、デストラクタには一部注意点があります。

```
class BaseClass
{
public:
        BaseClass() { }   //  基礎クラスのコンストラクタ
        ~BaseClass() { }   //  基底クラスのデストラクタ
};

class SubClass   :   public BaseClass
{
public:
         SubClass() { }   //  派生クラスのコンストラクタ
         ~SubClass() { }   //  派生クラスのデストラクタ
};

void main()
{
        SubClass* sub = new SubClass();    // SubClass のポインタに Sub オブジェクトを構築
        delete sub;

        BaseClass* base = new SubClass();    // BaseClass のポインタに Sub オブジェクトを構築
        delete base;
}
```

実行時のコンストラクタ・デストラクタの動き

```
------ SubClass のポインタに Sub オブジェクトを構築
①基底クラス(base)のコンストラクタを実行
②派生クラス(sub)のコンストラクタを実行
③派生クラス(sub)のデストラクタを実行
④基底クラス(base)のデストラクタを実行

------ BaseClass のポインタに Sub オブジェクトを構築
①基底クラス(base)のコンストラクタを実行
②派生クラス(sub)のコンストラクタを実行
※派生クラスのデストラクタが呼ばれない
④基底クラス(base)のデストラクタを実行
```

2回目の例では、基底クラスのポインタに派生クラスをインスタンスしているので、デストラクタ時の「自身」とは「基底クラスになっています。その為、派生クラスのデストラクタが呼ばれなくなってしまいます。

ですので、そうならない為にも、継承（ポリモーフィズム）をする場合は「基底クラスのデストラクタを virtual(バーチャル)修飾子」にして、デストラクタ時に派生クラスが呼ばれる様に「基底クラスの関数を仮想にしておく」と良いです。

```
class BaseClass
{
public:
	BaseClass() { }			//  基底クラスのコンストラクタ
	virtual ~BaseClass() { }	//  基底クラスのデストラクタ
};

class SubClass  :  public BaseClass
{
	 SubClass() { }			//  派生クラスのコンストラクタ
	 ~SubClass() { }		//  派生クラスのデストラクタ
};


void main()
{
	SubClass* sub = new SubClass();	// Sub クラスのポインタに Sub クラスを代入
	delete sub;

	BaseClass* base = new SubClass();	// Base クラスのポインタに Sub クラスを代入
	delete base;
}
```

実行時のコンストラクタ・デストラクタの動き

```
----- SubClass のポインタに Sub クラスを構築
①基底クラスのコンストラクタを実行
②派生クラスのコンストラクタを実行
③派生クラスのデストラクタを実行
④基底クラスのデストラクタを実行

----- BaseClass のポインタに Sub クラスを構築
①基底クラスのコンストラクタを実行
②派生クラスのコンストラクタを実行
③派生クラスのデストラクタを実行
④基底クラスのデストラクタを実行
```

■virtual（仮想関数）

仮想関数とも呼ばれ、継承したクラスで同名の関数を作った際、基底の関数に virtual キーワードを付けておくと、継承したオブジェクトを基底クラスとして扱っても、その際に呼び出される関数は派生クラスの関数が呼び出される様になるというものです。文字だと分かり辛いので図にしてみます。

virtual 関数の有無の違い

```
class BASE
{
public:
    void Action(){ ①処理 }
};

class Player : public BASE
{
public:
    void Action(){ ②処理 }
};

void main()
{
    BASE* p = new Player();
    p->Action();   // 基底の①が実行される

    Player* p = new Player();
    p->Action();   // 派生の②が実行される
}
```

```
class BASE
{
public:
    virtual void Action(){ ①処理 }
};

class Player : public BASE
{
public:
    void Action(){ ②処理 }
};

void main()
{
    BASE* p = new Player();
    p->Action();   // 派生の②が実行される

    Player* p = new Player();
    p->Action();   // 派生の②が実行される
}
```

最初の頃は違いが分かり辛いと思いますが、今後「ポリモーフィズム」や「参照渡し」などが登場するにつれ、基底クラスと派生クラスの明確な使い分けを行う必要が出てきますので重要度が増してくると思います。

□virtual にしたメンバ関数は呼び出せないの？

派生クラスを実装した場合に基底クラスの関数を呼び出す場合は、「基底クラス名::関数名」で呼び出す事ができます。ついでに言うと、基底クラスのポインタで派生クラスを生成した場合は、あくまでの基底クラスのオブジェクトなので派生クラスの変数は見る事ができません。

```
class BASE
{
public:
    int mBasePos;
    virtual void Action(){ ①処理 }
};

class Player : public BASE
{
public:
    int mPlayerPos;
    void Action(){ ②処理 }
};
```

```
void main()
{
    BASE* p = new Player();
    p->Action();   // 派生②を実行
    p->BASE::Action();  // 基底①を実行

    p->mBasePos = 0;  // 使える
    p->mPlayerPos = 0;  // 使えない
        ※Player クラスの変数は見えません
}
```

□純粋仮想関数

「継承先で必ず実装する関数なので基底オブジェクトでは定義しなくてよい（定義しない）」と、始めからはっきりしているものは、基底オブジェクトの virtual 定義時に、純粋仮想関数として定義しておきます。この関数が「一つでもあるクラス」は、純粋仮想クラスと言い、関数の中身がない為、単品でのオブジェクト化ができなくなります。

ですので、誰かに継承して貰うだけの役割りとしてのクラスを作る事になります。

```
virtual　戻り値　関数名(引数の型) = 0;
```

仮想関数

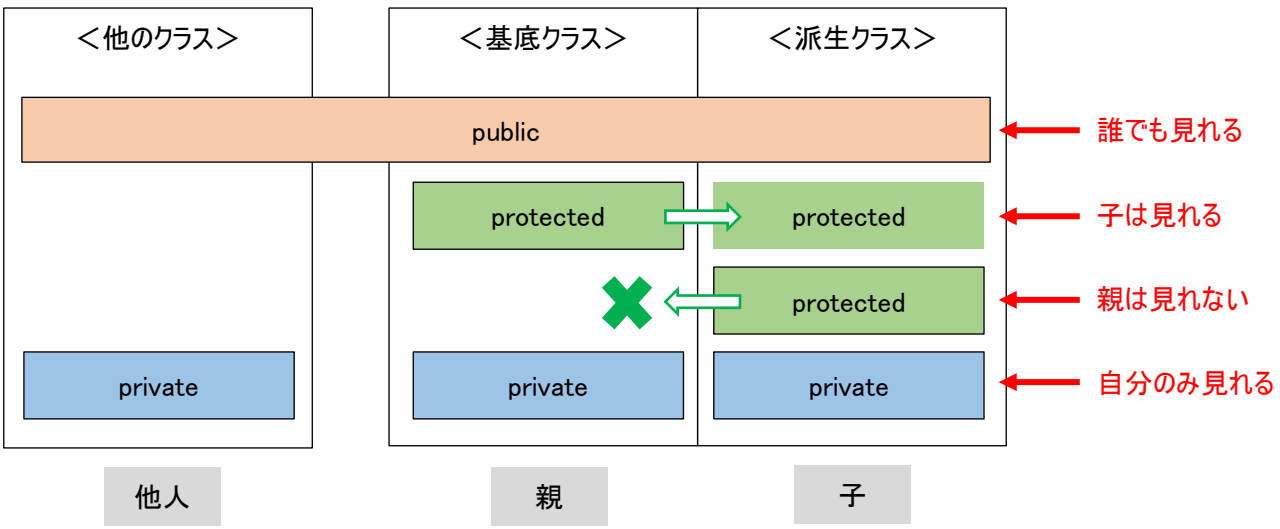
純粋仮想関数

■アクセス指定子 part.2

継承が出て来たので、アクセス指定子の残りの protected について説明していきます。

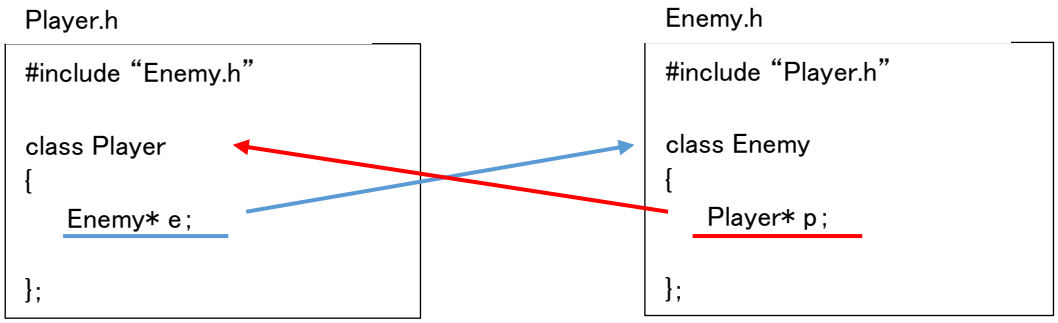
アクセス指定子	意味
public	全ての範囲からアクセス（読み込み・書き込み）可能
private	同一クラス、インスタンス内でのみアクセス可能
protected	同一クラス、インスタンスに加え、継承された派生クラスでもアクセス可能

protected メンバは private メンバ同様、クラス外からのアクセスはできません。private との違いは継承した派生クラスからは public 同様に扱えるという事で、派生クラスのみにアクセスをさせたい時に protected を付けます。
逆に派生クラスで protected で定義したものは基底クラスではアクセスできません。継承した先で有効となります。



■相互参照

クラスの数が増えてある程度規模が大きくなってくると、クラス同士で互いに参照する様になります。その事を相互参照といいますが、その相互参照自体には問題はありません。しかしながらプログラミングをする上で、特に#includeのヘッダーファイルに問題が発生します。



Playerクラスを使用する時にPlayer.hではEnemy.hをインクルードしますが、同時に
Enemyクラスを使用する時にEnemy.hではPlayer.hをインクルードしますので、
いつまでたっても#include処理が終わらない事になってしまいます。

そこで解決策です！

ヘッダーファイルで定義する「クラスのポインター」は「型の指定」であり「実体」ではありません。
ですので定義時は「使用するクラス名を記述するだけ」で OK なのです。
このような処理を「前方宣言」といいいます。

例）

Player.h

```
#include “Enemy.h”
class Enemy;    // Enemy の前方宣言

class Player
{
    Enemy* e;  // Enemy*を認識

};
```

Enemy.h

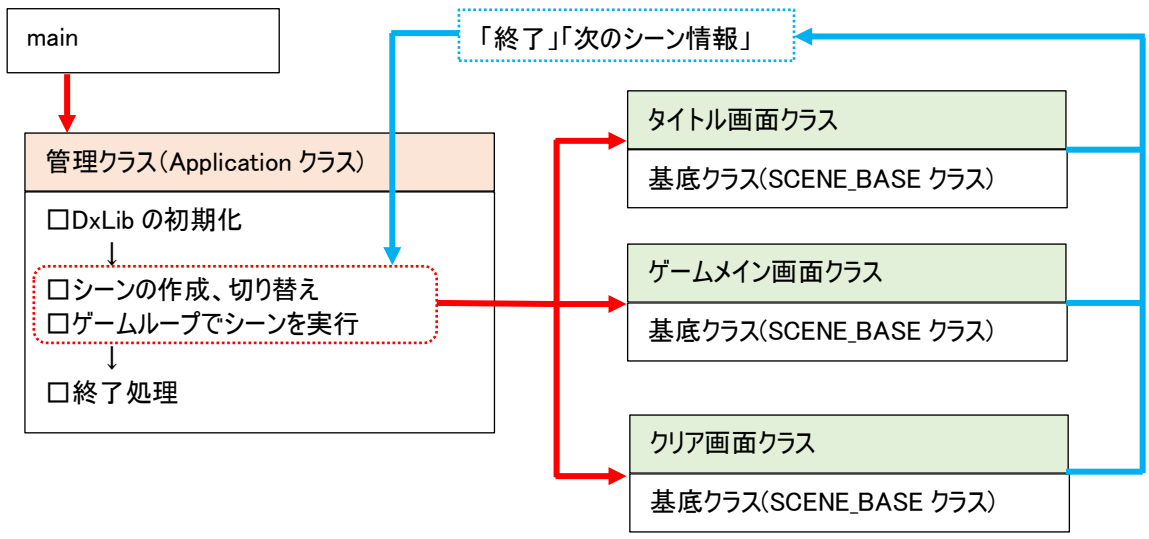
```
#include “Player.h”
class Player;    // Player の前方宣言

class Enemy
{
    Player* p;  // Player*を認識

};
```

■継承を使用してゲームループの画面遷移を作成

それでは、タイトル画面やゲーム画面の処理を共通するベースとなるクラスから継承し、それらを切り替えながらシーン変更ができる様なゲームループの仕組みを作成していきましょう。

まず設計

どんな構成になるかのイメージをしておきます。



基本的に、クラスは別のクラスを干渉しないような処理を考えていきます。あくまでも自分のクラスだけの処理に専念するイメージです。しかしながら、どうしても情報を渡す必要がある場合もありますので、その時は関数などを使用して対応していきましょう。

今回は、管理クラスがシーンを切り替えながら実行していく方式なので、各シーンを共通のクラス（の型）として扱えるように、シーンの基底クラスを作成しそれを継承した派生クラスとして構築していきます。

SCENE_BASE.h

```
#pragma once

class SCENE_BASE
{
public:
    // シーンチェンジで使用(親クラスの場所)
    class Application* mApplication;

public:
    SCENE_BASE(class Application* application);
    virtual ~SCENE_BASE();
    virtual bool Run();
};
```

親クラスのポインターを保持する為
※前方宣言をする所をこのように直接型名を明記する事もできます

派生クラスの Run()関数を使用していくので virtual を付けて仮想化しておきます。

SceneTitle.h

```
#pragma once

#include "SCENE_BASE.h"

class SceneTitle : public SCENE_BASE
{
private:
    // SceneTitleでのみ使用する変数を定義する

public:
    SceneTitle(class Application* application);
    ~SceneTitle();
    bool Run();
};
```

SceneGame.h

```
#pragma once

#include "SCENE_BASE.h"

class SceneGame : public SCENE_BASE
{
private:
    // SceneGameでのみ使用する変数を定義する

public:
    SceneGame(class Application* application);
    ~SceneGame();
    bool Run();
};
```

SceneTitle.cpp

```
#include "DxLib.h"
#include "../Application.h"
#include "SceneTitle.h"

SceneTitle::SceneTitle(Application* application) : SCENE_BASE(application)
{
    // タイトルシーンの初期化
}

SceneTitle::~SceneTitle()
{
    // タイトルシーン終了の後始末
}

bool SceneTitle::Run()
{
    DrawString(0, 0, "GameTitleLoop", 0xffff00);

    return true;    // 継続:true／終了:false
}
```

※ヘッダーファイルの場所は、自分のプロジェクトに合わせます

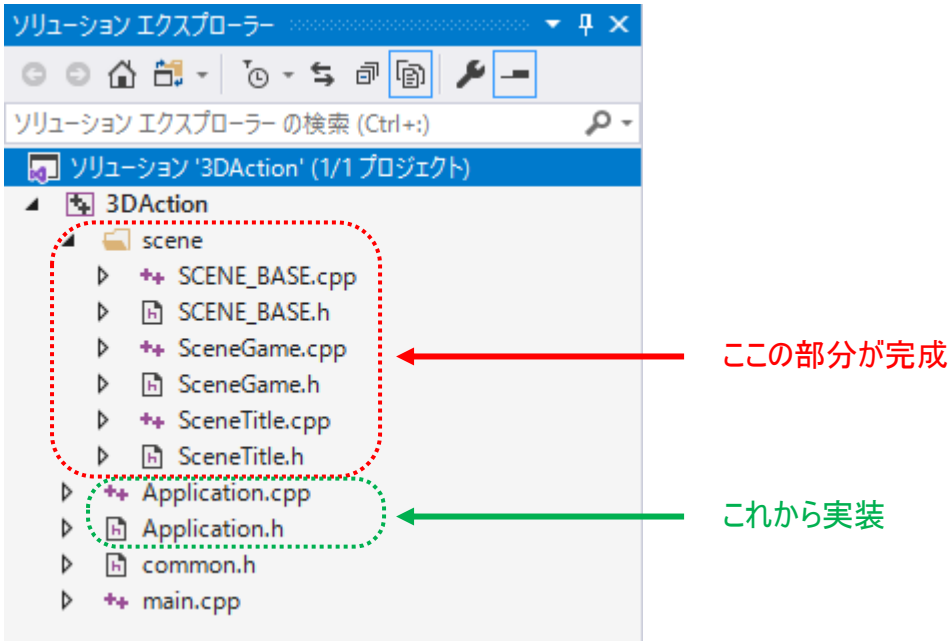
そのまま受け渡す

※シーンを終了したい時は false

SceneGame.cpp も、基本的に SceneTitle と同様です。

```
#include "DxLib.h"
#include "../Application.h"
#include "SceneGame.h"

SceneGame::SceneGame(Application* application) : SCENE_BASE(application)
{
    // ゲームシーンの初期化
}

SceneGame::~SceneGame()
{
    // ゲームシーン終了の後始末
}

bool SceneGame::Run()
{
    DrawString(0, 0, "GameMainLoop", 0xffff00);

    return true;    // 継続:true／終了:false
}
```

シーン関連は一旦終了。次にシーン管理（Application クラス）の実装を行います。



Application.h （ゲームシーン管理クラス）

```
#pragma once

enum class SCENE_ID
{
    NOT_SELECT = -1,
    TITLE = 0,
    GAME,
    CLEAR
};

class Application
{
public:
    class SCENE_BASE* mScene = nullptr;      // 再生するシーン

    SCENE_ID mNextScene = SCENE_ID::NOT_SELECT;
    SCENE_ID mNowScene = SCENE_ID::NOT_SELECT;
public:
    Application() {}    // コンストラクタ
    ~Application(); // デストラクタ

    int SystemInit(void);   // DxLib初期化
    bool GameCreate(void);  // ゲーム生成処理
    void Shutdown(void);    // 終了処理
    void Run(void);         // ループ実行
};
```

どのシーンを実行する（実行している）かを管理するのに数値でなく名称を使用する為に、enum class を使用してシーンの ID を設定しておきます。

現在のシーンの実体をポインターで保存しておく

※mNextScene は次のシーン名を名称で保存しておく
※mNowScnene は現在のシーンを名称で保存しておく

Application.cpp

```
#include "DxLib.h"
#include "common.h"
#include "Application.h"

#include "scene/SCENE_BASE.h"
#include "scene/SceneTitle.h"
#include "scene/SceneGame.h"

int Application::SystemInit()
{
    // DxLib初期設定
    SetWindowText(WINDOW_NAME); // ウインドウ名
    SetGraphMode(SCREEN_SIZE_X, SCREEN_SIZE_Y, 16); // 画面サイズ設定
    ChangeWindowMode(true);     // ウインドウモード
    if (DxLib_Init() == -1)
    {
        return -1;  // DXLib初期化
    }
    SetDrawScreen(DX_SCREEN_BACK);

    if (!GameCreate())
    {
        return -1;  // ゲーム構築、初期化
    }
    return 0;   // SystemInit正常終了
}
```

```
#pragma once

constexpr auto WINDOW_NAME = "3DPG_ACTION";
constexpr auto SCREEN_SIZE_X = 1024;
constexpr auto SCREEN_SIZE_Y = 600;
```

スクリーンサイズなど色々な場面で使用する情報をマクロ定義して使用します

```cpp
bool Application::GameCreate()
{
    // ゲーム初期化
    mNowScene = SCENE_ID::TITLE;
    mNextScene = SCENE_ID::NOT_SELECT;
    mScene = new SceneTitle(this);

    return true;    // 正常終了
}

void Application::Shutdown()
{
    DxLib_End();
}

Application::~Application()
{
}

void Application::Run()
{
    // ----- ゲームループ
    while (ProcessMessage() == 0 && CheckHitKey(KEY_INPUT_ESCAPE) == 0)
    {
        ClsDrawScreen();
        switch (mNowScene)
        {
        case SCENE_ID::TITLE:
            break;
        case SCENE_ID::GAME:
            break;
        }
        ScreenFlip();
    }
    Shutdown(); // アプリケーション終了
}
```

最初に実行するシーン名を定義します

次に実行するシーン名を定義します

タイトルシーンの実体を生成

引数に自分自身(Application)のポインタを渡しておく

分岐は mNowScene の変数を使用して
とりあえず Switch 文で行います。
※当然それ以外でも可能

最後は main.cpp

main.cpp は管理クラスを静的にインスタンスして、システム初期化（DxLib)をした後にループを実行するだけのシンプルな構成になります。ゲームループ自体も application.Run()の中で処理されます。

```
#include "DxLib.h"
#include "Application.h"

int WINAPI WinMain(_In_ HINSTANCE hInstance, _In_opt_ HINSTANCE hPrevInstance,
                   _In_ LPSTR lpCmdLine, _In_ int nShowCmd)
{
    Application application;    // 生成
    if (application.SystemInit() == -1) return 0;   // DxLib初期化

    application.Run();  // ループ

    return 0;
}
```

実際に画面遷移をさせる場合（SceneTitle → SceneGame への遷移の場合）

SceneTitle.cpp

```
bool SceneTitle::Run()
{
    if (CheckHitKey(KEY_INPUT_SPACE))
    {
        mApplication->mNextScene = SCENE_ID::GAME;      // 次のシーンを指定
        return false;   // Titleシーン終了
    }

    DrawString(0, 0, "GameTitleLoop", 0xffff00);
    return true;    // 継続:true／終了:false
}
```

※シーン切り替えをするタイミングで、
シーン終了の返り値と次のシーンの
設定を行って終了させる。

Application.cpp

```
void Application::Run()
{
    // ----- ゲームループ
    while (ProcessMessage() == 0 && CheckHitKey(KEY_INPUT_ESCAPE) == 0)
    {
        ClsDrawScreen();
        switch (mNowScene)
        {
        case SCENE_ID::TITLE:
            if (mScene->Run() == false)
            {
                if (mNextScene == SCENE_ID::GAME)
                {
                    delete(mScene);
                    mScene = new SceneGame(this);
                }
            }
            break;
        case SCENE_ID::GAME:
            break;
        }
        ScreenFlip();
    }
    Shutdown(); // アプリケーション終了
}
```

シーンを実行した後、そのシーンが終了になるかをチェックする

次のシーンが何であるかをチェック

・今のシーンを削除
→そのシーンのデストラクタが働く
・新しいシーンをインスタンスする

■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。

・継承
・オーバーロード
・オーバーライド
・virtual（仮想関数）
・純粋仮想関数
・アクセス指定子 part.2（protected）
・相互参照
・継承したクラスを使った画面遷移