スマートポインタ

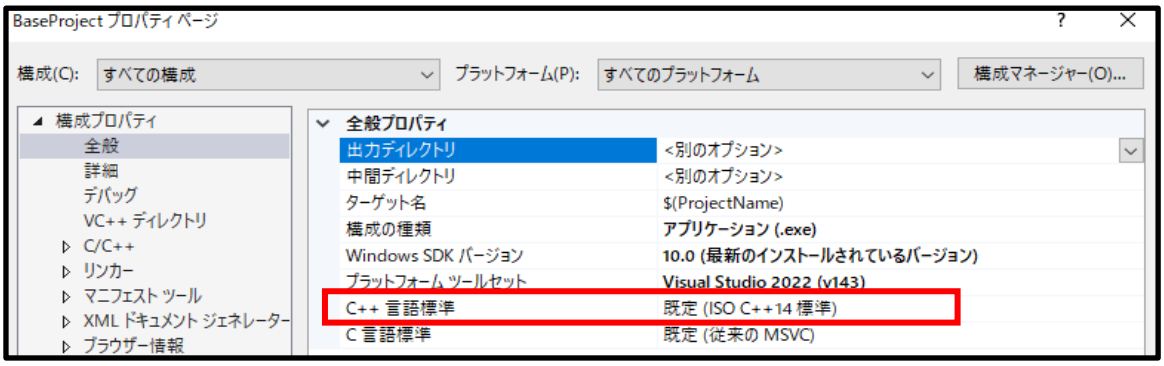
メモリ制御について、前回お話ししましたが、プログラムが複雑になると、
メモリの解放忘れや、重複解放が発生しやすくなります。
重複解放とは、

```
Player* p = new Player();

delete p;

～

delete p;
```

このように delete を重ねて実行してしまうことで、
Ｃ＋＋言語の場合、「 動作を保証しない 」という扱いになります。
もしかしたら、動作し続けるかもしれませんし、他のメモリが破壊されている
かもしれませんが、その場で強制終了となることも多いでしょう。

- 解放し忘れによるメモリリーク
- 重複解放による強制終了

この問題が何とかならないかということで、
Ｃ＋＋１１から、スマートポインタという機能が追加されました。
その後、言語バージョンが上がるごとに改良されていっているのですが、
ここではＣ＋＋１４時点での機能を中心に解説していきます。



大きなゲーム会社様を狙っている方は、しっかり１７、２０と新しい言語機能の
学習を欠かさないようにしてください。
面接時に質問され、普段からの学習意欲を試されます。

今まで使用してきた、
生ポインタから、スマートポインタに切り替えることで、

- 解放し忘れによるメモリリーク
  → 不要となる条件を満たすと、自動的に解放してくれるようになる。
- 重複解放による強制終了
  → 自分で解放しないので、重複解放になることがない。

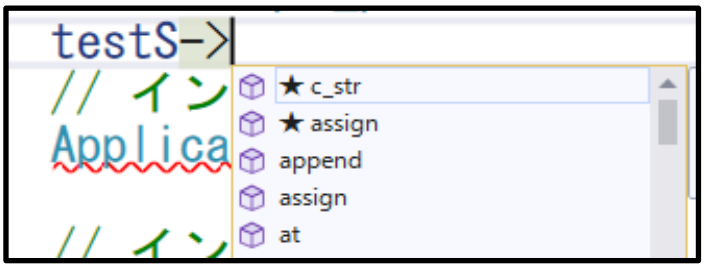
先ほど問題に上がった 2 点が解決されます。

試しにユニークポインタという、スマートポインタの 1 種を使用してみましょう。

main.cpp

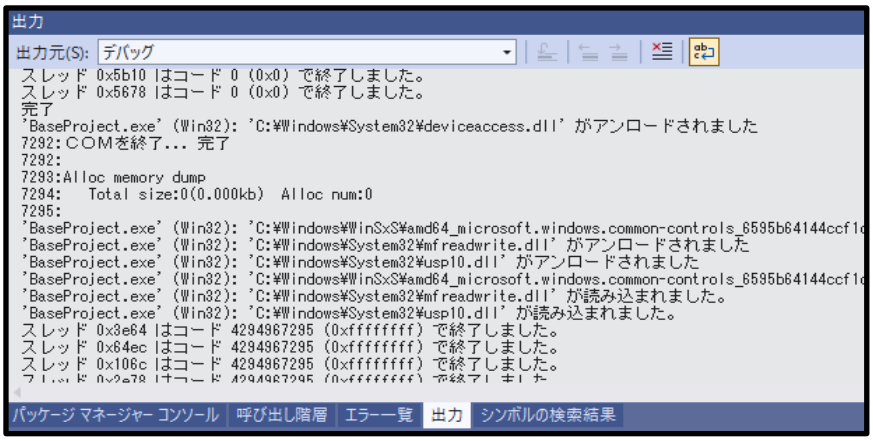
```
～
#include <memory>
～

int WINAPI WinMain(
    _In_ HINSTANCE hInstance, _In_opt_ HINSTANCE hPrevInstance,
    _In_ LPSTR lpCmdLine, _In_ int nCmdShow)
{

    // メモリリーク検出
    _CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);

    // ①生ポインタ
    //int* test = new int();
    //delete test;

    // ②ユニークポインタ
    std::unique_ptr<int> testI = std::make_unique<int>();
    std::unique_ptr<std::string> testS = std::make_unique<std::string>();
```

それぞれ int 型 と string 型 のポインタを作成しました。
本当にポインタなのか確認を取るため、testS 変数の後に、
アロー演算子(->)を付けて、string クラスの関数候補が出てくるか確認してみましょう。
※生ポインタと異なり、ドット(.)では、候補が出てきませんので注意



string クラスの関数に余り見覚えが無く、納得ができない方は、
Playerクラスなどで、試してみましょう。

そして、実行後、VisualStudioの出力タブを確認してみてください。



動的にメモリを生成して、メモリ解放の delete 文を打っていないにも
関わらず、

```
Detected memory leaks!
```

この忌まわしいメッセージが表示されておりません。
メモリが不要となる条件が満たされたタイミングで、メモリがキチンと
解放されていることがわかります。
今回は、ローカル変数になりますので、WinMain関数終了時に解放されています。

ユニークポインタの型宣言

```
std::unique_ptr<型名> 変数名_;
```

ユニークポインタの実体生成

```
std::unique_ptr<型名> 変数名 =
    std::make_unique<型名>(コンストラクタの引数１, 引数２);
```

先ほどは、ユニークポインタを使用しましたが、
他にも２種類のスマートポインタがあります。

所有権という考え方が出てくるのですが、もし、よくわからなければ、
最後にメモリを解放する責任者、と理解してください。

とあるサイトのまとめ表になりますが、

	コピー	ムーブ	アクセス	所有権	特徴
unique_ptr	×	○	○	単一	軽量・シンプル
shared_ptr	○	○	○	共有	コピー可能だが、 オーバーヘッドが存在
weak_ptr	○	○	△	なし	所有権を持たず shared_ptrへの参照を 保持

簡潔な説明になっていますが、実際に使ってみないと、なかなか理解・納得が
難しいかと思います。なぜなら、この３種類の使い分けに関しては、
クラス設計が大きく関わってくるからです。

最悪、生ポインタを全部 shared_ptr に変えてしまっても動作はしますが、
ゲーム業界の受験でコードチェックされた時に、良い印象にはならないでしょう。

これから使い分けを解説していきますが、
設計は考え方によって、大きく変わり、絶対的な正解もありませんので、
一例として、参考して貰えたらと思います。

設計のテーマは、" なるべくシンプルに、早く "、です。

TitleSceneのヘッダーファイルを見てください。

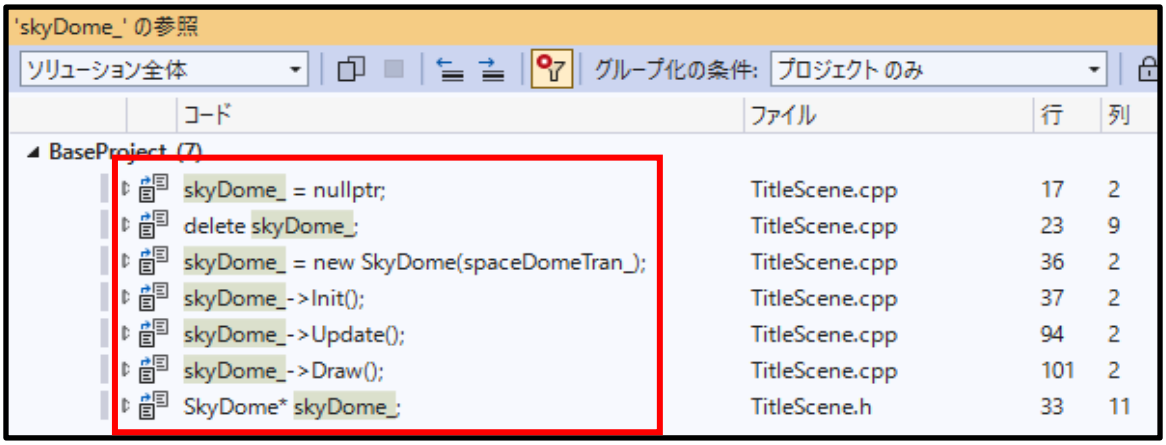
TitleScene.h

```
    // スカイドーム(背景)
    SkyDome* skyDome_;

    // アニメーション
    AnimationController* animationController_;
```

生ポインタが２か所あります。
まずは、これらをスマートポインタにリファクタリング(最適化)して
いきましょう。

skyDome_変数をダブルクリックして、〝すべての参照を検索〟してください。

この生ポインタを使用している箇所が一覧で出てきます。



確認すると、他クラスにポインタ、もしくは、参照渡しをしている箇所が、
ありません。(※今回、対象のメンバ変数はゲッターがありませんが、
あればその使用箇所もチェックしてください)

ということは、TitleScene の中だけで使用箇所が完結している、
TitleScene でしか使用されていない
→ TitleScene が解放されれば解放して良い
→ 責任者は、TitleScene のみ = 単一の所有者
　となりますので、unique_ptr に置き換えるのが好ましいでしょう。

TitleScene.h

```
#include <memory>

    ～

    // スカイドーム(背景)
    std::unique_ptr<SkyDome> skyDome_;
```

TitleScene.cpp

```
TitleScene::~TitleScene(void)
{
    delete skyDome_;                ←　削除
    delete animationController_;
}

void TitleScene::Init(void)
{

    ～

    // 背景
    spaceDomeTran_.pos = AsoUtility::VECTOR_ZERO;
    skyDome_ = std::make_unique<SkyDome>(spaceDomeTran_);
    skyDome_->Init();

    ～

}
```

実行して、キチンと動作すること、メモリリークが発生していないことを
確認します。
同じ手順で、animationController_ もリファクタリングしましょう。

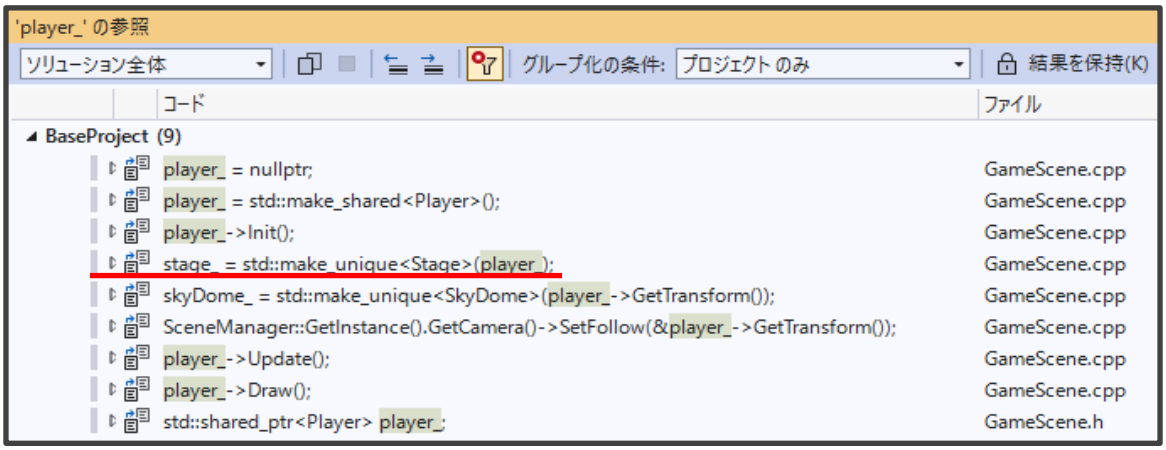
次は、GameScene のリファクタリングに移ります。

GameScene には、３つの生ポインタがあります。

GameScene.h

```
// ステージ
Stage* stage_;

// スカイドーム
SkyDome* skyDome_;

// プレイヤー
Player* player_;
```

まずは、stage_ と skyDome_ をリファクタリングしてください。
終わったら player_ の方を見てみます。



ステージクラスのコンストラクタで、Playerクラスの生ポインタを渡しています。

試しに、 player_ を unique_ptr で宣言してみます。

Stage クラスのコンストラクタの引数の型も合わせて修正します。

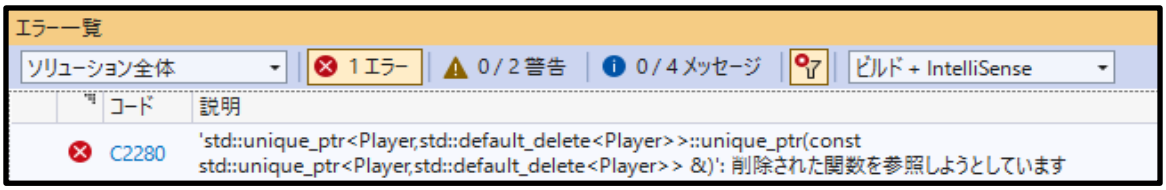
Stage.h

```
// コンストラクタ
Stage(std::unique_ptr<Player> player);
```

Stage.cpp

```
Stage::Stage(std::unique_ptr<Player> player)
    : resMng_(ResourceManager::GetInstance())
{
    //player_ = player;                    ←型が異なるので一旦、コメント
    activeName_ = NAME::MAIN_PLANET;
    step_ = 0.0f;
}
```

実行してみると、以下のようなエラーが発生します。

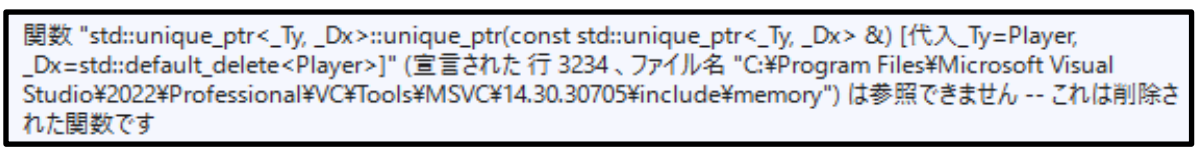


削除された関数を参照しているようです。
どういうことかというと、

GameScene.cpp

```
void GameScene::Init(void)
{

    // プレイヤー
    player_ = std::make_unique<Player>();
    player_->Init();

    // ユニークポインタコピー不可例
    std::unique_ptr<Player> player2 = player_;
```

お試しコードを追加してみると、ここでも同じようなエラーメッセージが
表示されます。



削除された関数です。

unique_ptr は所有権が単一であると定められており、
責任者が１人である際に、使われるスマートポインタになります。

```
// ユニークポインタコピー不可例
std::unique_ptr<Player> player2 = player_;
```

このコードでは、代入式によって、自分自身(player_)を左辺にコピーしようと
しているので、責任者が２人(player_ と player2)になってしまいます。
そうならないように、あらかじめコピーコンストラクタが削除されているのが、
unique_ptr となります。

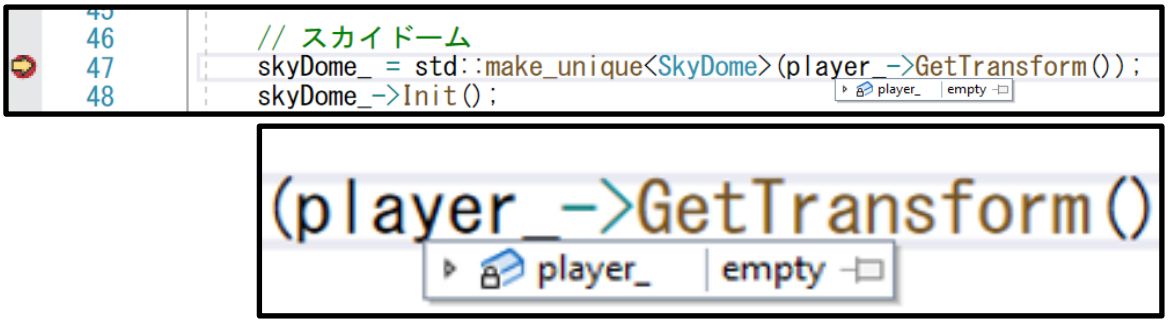
Stageのコンストラクタでも同じことがおきており、スマートポインタの値渡し
(コピーを行う)引数の渡し方になっているのでエラーになっています。

```
Stage::Stage(std::unique_ptr<Player> player)
    : resMng_(ResourceManager::GetInstance())
```

それでは、unique_ptr は引数で渡せないのかというと、そういうわけではなく、
所有権の移動や参照を使用すれば、引数で渡すことができます。
試しに所有権の移動を行ってみましょう。

```
stage_ = std::make_unique<Stage>(std::move(player_));
stage_->Init();

// ステージの初期設定
//stage_->ChangeStage(Stage::NAME::MAIN_PLANET);          ←一旦、コメント
```

コンパイルは通るはずですので、ブレークポイントを貼って、
挙動を確認します。



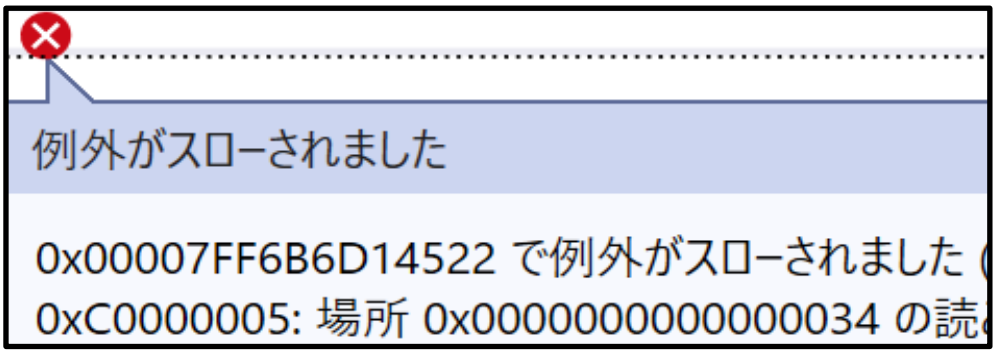
player_ が empty(空)になっています。

コードをそのまま追っていって、

```
106     void SkyDome::ChangeStateFollow(void)
107     {
108         transform_.pos = syncTransform_.pos;  ≤1ミリ秒経過
109         transform_.Update();
110     }
```

syncTransform_ {modelId=??? scl={x=??? y=??? z=??? } rot={x=??? y=??? z=??? } ...}

不定値の箇所で例外エラーが発生します。



所有権の移動により、 GameScene の player_ メンバ変数から、

```
Stage::Stage(std::unique_ptr<Player> player)
    : resMng_(ResourceManager::GetInstance())
{
    //player_ = player;
    activeName_ = NAME::MAIN_PLANET;
    step_ = 0.0f;
}
```

Stageのコンストラクタのローカル変数に所有権が渡り、
そのままコンストラクタ終了時に破棄され、Playerの実体は消滅して
しまったのが原因です。

不用意に所有権を移動させると危ないです。

ということで、コピーが許されている shared_ptr を使ってみましょう。
※この時点で、shared_ptr の使用に反対の方もいらっしゃるかと思いますが、
　順を追って解説していくため、shared_ptr で進めます

シェアードポインタの型宣言

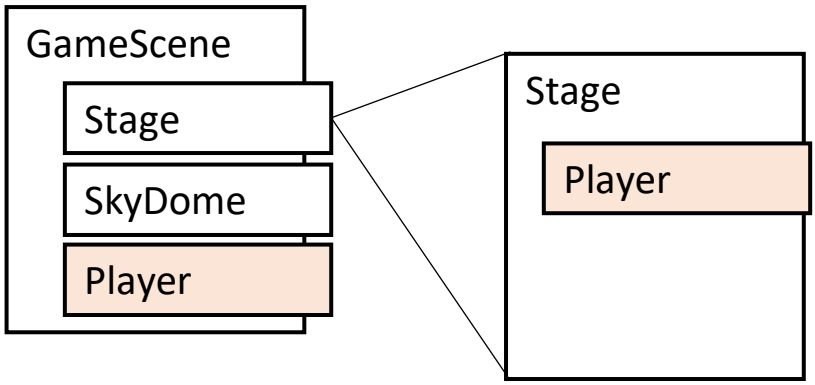
```
std::shared_ptr<型名> 変数名_;
```

シェアードポインタの実体生成

```
std::shared_ptr<型名> 変数名 =
    std::make_shared<型名>(コンストラクタの引数１, 引数２);
```

エラーが発生しないように、リファクタリングしてください。

責任者が複数いますので、



それぞれ、メンバ変数(クラスが破棄されるまで保持される)で宣言されていますので、
上の図だと、GameScene と Stage の両方の実体が破棄(デストラクタ)されないと、
Player は破棄されないようになります。

shared_ptr では、参照カウントという仕組みを使用して、
メモリ解放のタイミングを見計らっています。

参照カウントは、use_count()関数で確認することができます。
上の図だと 2 になります。

では、問題です。

main.cpp

```
std::shared_ptr<int> testSha_;
void TestShared(std::shared_ptr<int> a)
{
    int cnt2 = testSha_.use_count();
    std::shared_ptr<int> b = a;

    // シェアードポインタの参照カウント
    int cnt3 = testSha_.use_count();
}

int WINAPI WinMain(
    _In_ HINSTANCE hInstance, _In_opt_ HINSTANCE hPrevInstance,
    _In_ LPSTR lpCmdLine, _In_ int nCmdShow)
{

    ～

    // ③シェアードポインタ
    testSha_ = std::make_shared<int>(5);
    // シェアードポインタの参照カウント
    int cnt1 = testSha_.use_count();
    // 最初は実装しない
    TestShared(testSha_);
    int cnt4 = testSha_.use_count();
```

それぞれ、cnt1,2,3,4の参照カウントはいくつになるでしょうか。

スコープ・メモリの良い勉強になるかと思いますので、
気になったパターンはコードで起こして、自分で確認していきましょう。

冒頭でもお伝えしましたが、全ての生ポインタを shared_ptr に変更すれば、
一応、動作はします。
これが、なぜ良くないかというと、

- 所有権について、まるで理解していない

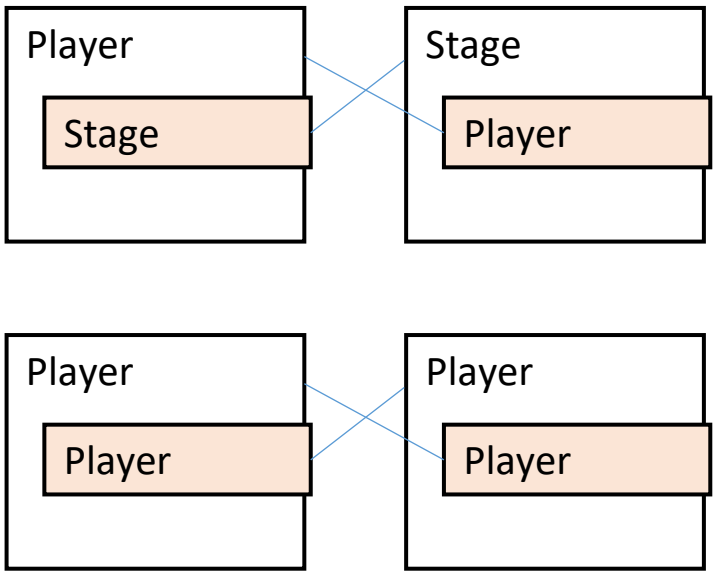
- shared_ptr のオーバーヘッドにより、
  処理速度をが向上させる余地が明らかに存在する

shared_ptr のオーバーヘッドとは、先ほどの参照カウントで、
各クラスが破棄されていないか監視する必要がありますので、
監視対象(責任者)が多ければ多いほど、処理時間がかかってしまいます。

速度面においては、できる限り、shared_ptr よりも、
unique_ptr の方が好ましいということになります。

所有権が単一ではなく、でも shared_ptr のオーバーヘッドも気になる。
更には、shared_ptr 特有の循環参照も解消したい。
そんな要望を満たすために、第３のスマートポインタ weak_ptr の登場です。

循環参照



お互いがお互いを参照しているため、メモリ解放できず、
メモリリークになります。

そこで、弱参照と呼ばれる weak_ptr を使用します。
weak_ptr は、shared_ptr が所有権を持つメモリを管理します。
(shared_ptr ありきのスマートポインタ)
weak_ptr を使用することで、オーバーヘッドと、メモリリークを無くします。

ウィークポインタの型宣言

```
std::weak_ptr<型名> 変数名_;
```

ウィークポインタの実体生成

```
std::shared_ptr<型名> A;
std::weak_ptr<型名> B = A;
```

ウィークポインタの使用方法

```
B.lock()->変数
B.lock()->関数()
```

→ lock関数によって、参照先を保持するshared_ptrを取得している
(使用中に解放されてしまうのを防止するため)

それでは、Stageクラス と WarpStarクラス のメンバ変数、player_ を、
weak_ptr に変更して、エラーが無くなるまでリファクタリングしましょう。

これまで制作してきた、ゲームアーキテクチャのゲームでは、
シーンに必要なものは、シーンの初期化でロードして、
最後にまとめてメモリ解放するようにしていますので、
この作り方だと、

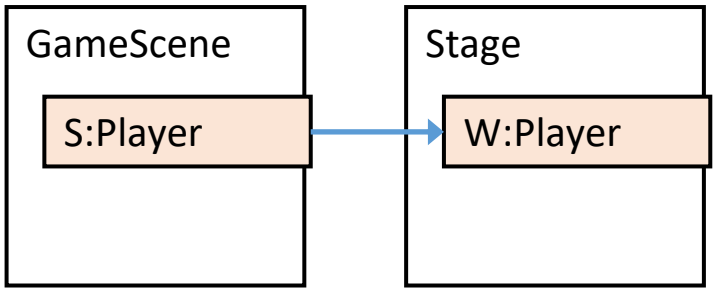
使用箇所が１つのクラス内に留まっている
　unique_ptr

複数クラスから、参照される
　shared_ptr

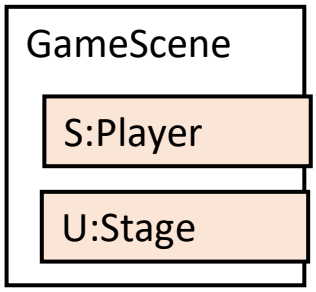
参照する側
　weak_ptr

という、シンプルな使い分けができます。

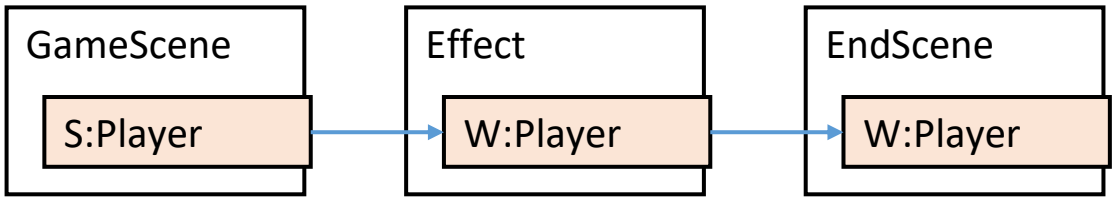
下図でいくと、Stageクラスで使用している、Playerクラスの weak_ptr は、
Player が解放された後に、使用しないように制御する必要があります。



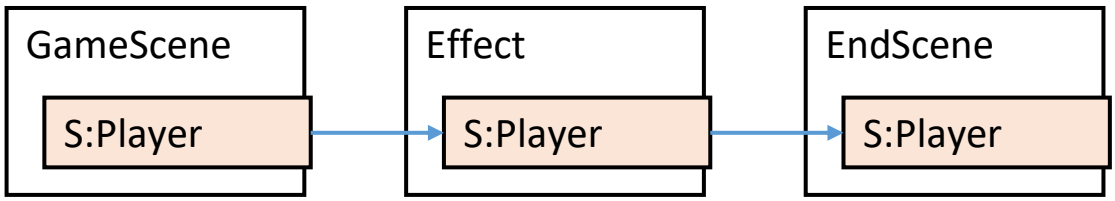
しかし、Player が 解放されるということは、
GameScene が解放されているということでもありますし、
Stage が解放されているということにもなりますので、
問題が発生しにくい作りになっています。



一方で、例えば、シーンを跨いで Player クラスを使いたい場合、



エフェクトが終了したら、
GameSceneを破棄する = Player も破棄される
ということになり、EndScene でエラーになります。



こういった場合は、複数の shared_ptr で管理する必要があるでしょう。