

## ゲームプログラミングⅢ

### ○評価要件

- ☑敵の配置
- ☑敵管理クラスの作成
- ☑メモリリークの確認

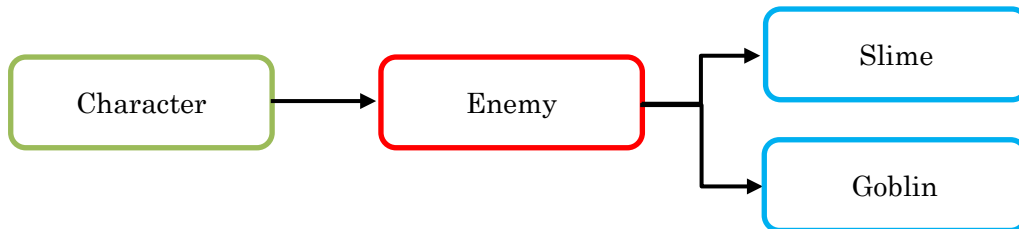
### ○概要

今回は敵を配置します。

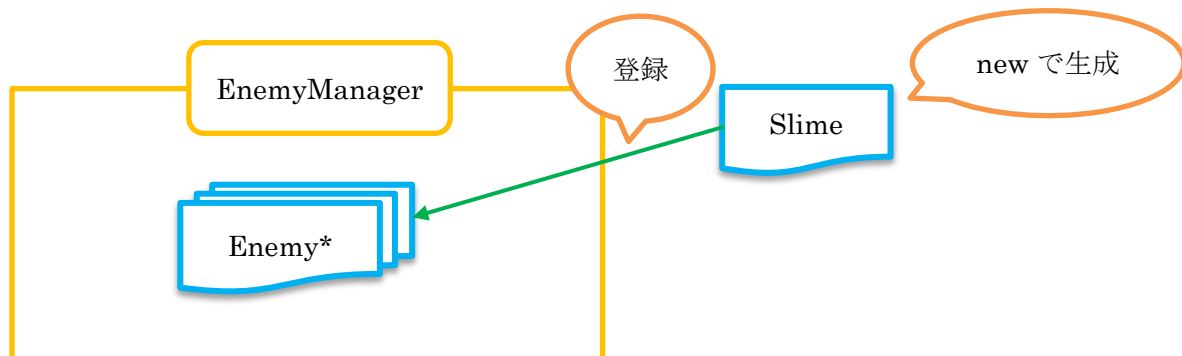
敵の行動はまだ実装しませんが、複数の敵をシーンに配置できるようにします。

まずは簡単に敵のクラス設計を考えましょう。

今回は下図のような継承でのクラスを作成していきます。



共通の「Enemy」クラスを継承して「Slime」や「Goblin」などがそれぞれ個別の処理を実装できるようにし、全ての「Enemy」を管理する「EnemyManager」クラスを作成しましょう。



## ゲームプログラミングⅢ

### ○エネミークラス

まずは全ての敵の基底となるエネミークラスを作成しましょう。

エネミークラスはエネミーマネージャーで管理され、エネミーマネージャーで更新処理や描画処理が呼び出されるようにします。

Enemy.h を作成し、下記プログラムコードを記述しましょう。

#### Enemy.h

```
#pragma once

#include "System/ModelRenderer.h"
#include "Character.h"

// エネミー
class Enemy : public Character
{
public:
    Enemy() {}
    ~Enemy() override {}

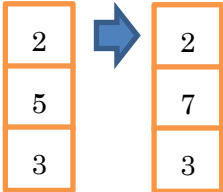
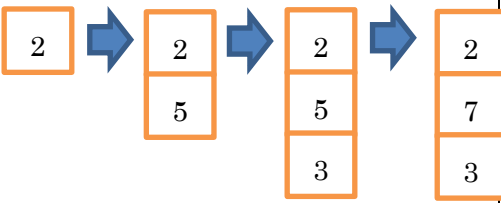
    // 更新処理
    virtual void Update(float elapsedTime) = 0;

    // 描画処理
    virtual void Render(const RenderContext& rc, ModelRenderer* renderer) = 0;
};
```

継承先で必ず実装させるように  
純粋仮想関数にする。

### ○std::vector について

std::vector とは C++ が標準で搭載している STL (Standard Template Library) の機能の一つで、可変長の配列を表現するためのものです。

普通の配列	可変長配列
<pre>int values[3]; values[0] = 2; values[1] = 5; values[2] = 3; values[1] = 7; int size = _countof(values);</pre>	<pre>std::vector&lt;int&gt; values; values.push_back(2); values.push_back(5); values.push_back(3); values[1] = 7; ←通常の配列と同じように値を設定できる int size = values.size(); ←配列の要素数を取得できる values.clear(); ←配列の要素数を 0 にできる</pre>
	

今回は複数のエネミーの管理をするために利用します。

## ゲームプログラミングⅢ

### ○エネミーマネージャークラス

全ての敵を管理するためのエネミーマネージャークラスを作成しましょう。

まずは全ての敵の更新処理と描画処理を一括で行う関数を実装します。

EnemyManager.cpp と EnemyManager.h を作成し、下記プログラムコードを記述しましょう。

#### EnemyManager.h

```
#pragma once

#include <vector>
#include "Enemy.h"

// エネミーマネージャークラス
class EnemyManager
{
private:
    EnemyManager() {}
    ~EnemyManager() {}

public:
    // 唯一のインスタンス取得
    static EnemyManager& Instance()
    {
        static EnemyManager instance;
        return instance;
    }

    // 更新処理
    void Update(float elapsedTime);

    // 描画処理
    void Render(const RenderContext& rc, ModelRenderer* renderer);

private:
    std::vector<Enemy*> enemies;
};
```

エネミーマネージャークラスはゲームで  
唯一のものとして扱いたいので  
シングルトンにする

複数のエネミーを  
管理するため、  
エネミーのポインタを  
std::vector で管理する

#### EnemyManager.cpp

```
#include "EnemyManager.h"

// 更新処理
void EnemyManager::Update(float elapsedTime)
{
    for (Enemy* enemy : enemies)
    {
        enemy->Update(elapsedTime);
    }
}

// 描画処理
void EnemyManager::Render(const RenderContext& rc, ModelRenderer* renderer)
{
}
```

範囲 for 文で  
管理しているエネミーの  
更新処理を一括実行

## ゲームプログラミングⅢ

```
for (Enemy* enemy : enemies)
{
    enemy->Render(rc, renderer);
}
```

これでひとまずエネミーの更新処理と描画処理をするための枠組みができました。

続いてエネミークラスを継承したスライムクラスを作成しましょう。

スライムクラスではとりあえず、3D モデルを読み込み、描画するだけのクラスを実装しましょう。

EnemySlime.cpp と EnemySlime.h を作成し、下記プログラムコードを記述しましょう。

### EnemySlime.h

```
#pragma once

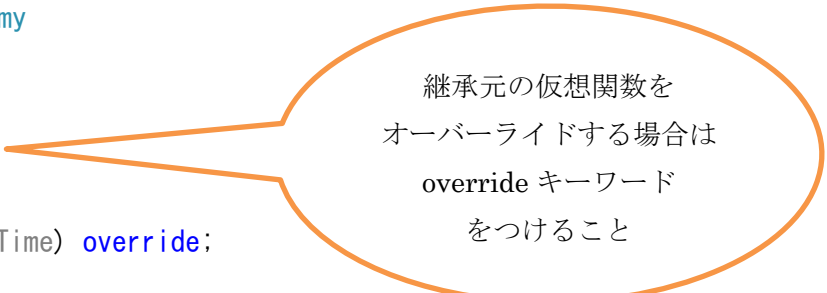
#include "System/Model.h"
#include "Enemy.h"

// スライム
class EnemySlime : public Enemy
{
public:
    EnemySlime();
    ~EnemySlime() override;

    // 更新処理
    void Update(float elapsedTime) override;

    // 描画処理
    void Render(const RenderContext& rc, ModelRenderer* renderer) override;

private:
    Model* model = nullptr;
};
```



### EnemySlime.cpp

```
#include "EnemySlime.h"

// コンストラクタ
EnemySlime::EnemySlime()
{
    model = new Model("Data/Model/Slime/Slime.mdl");

    // モデルが大きいのでスケーリング
    scale.x = scale.y = scale.z = 0.01f;
}

// デストラクタ
EnemySlime::~EnemySlime()
```

## ゲームプログラミングⅢ

```
{
    delete model;
}

// 更新処理
void EnemySlime::Update(float elapsedTime)
{
    // オブジェクト行列を更新
    UpdateTransform();

    // モデル行列更新
    model->UpdateTransform();
}

// 描画処理
void EnemySlime::Render(const RenderContext& rc, ModelRenderer* renderer)
{
    renderer->Render(rc, transform, model, ShaderId::Lambert);
}
```

実装が終わったら次はエネミーマネージャーを拡張します。

現状のエネミーマネージャーはエネミーを登録することができません。

EnemyManager.h と EnemyManager.cpp を開き下記プログラムコードを追記しましょう。

### EnemyManager.h

```
---省略---
// エネミーマネージャー
class EnemyManager
{
public:
    ---省略---

    // エネミー登録
    void Register(Enemy* enemy);

    ---省略---
};
```

### EnemyManager.cpp

```
---省略---

// エネミー登録
void EnemyManager::Register(Enemy* enemy)
{
    enemies.emplace_back(enemy);
}
```

## ゲームプログラミングⅢ

これでエネミーマネージャーにエネミーを登録できるようになりました。

敵を配置する準備ができたのでシーンに敵を配置しましょう。

SceneGame.cpp を開き、下記プログラムコードを追記しましょう。

SceneGame.cpp

```
---省略---
#include "EnemyManager.h"
#include "EnemySlime.h"

// 初期化
void SceneGame::Initialize()
{
    ---省略---

    // エネミー初期化
    
}

---省略---

// 更新処理
void SceneGame::Update(float elapsedTime)
{
    ---省略---

    // エネミー更新処理
    
}

// 描画処理
void SceneGame::Render()
{
    ---省略---

    // 3Dモデル描画
    {
        ---省略---

        // エネミー描画
        

        ---省略---
    }

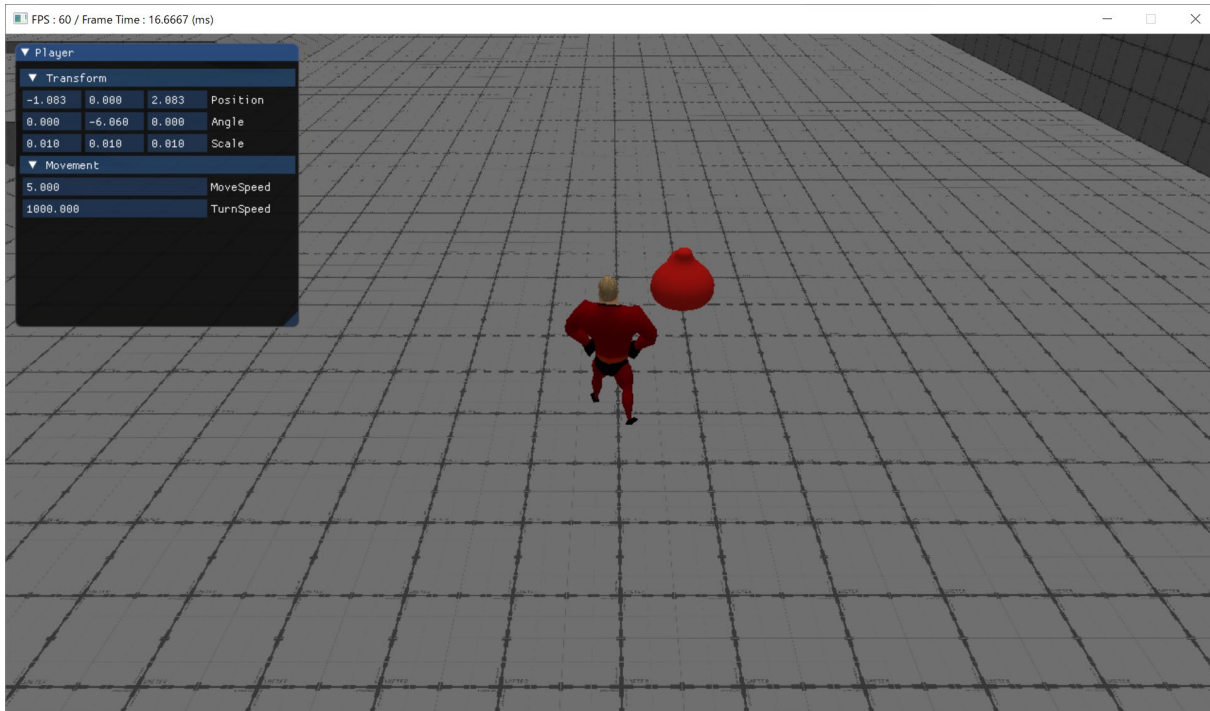
    ---省略---
}
```

スライムを  
生成して配置

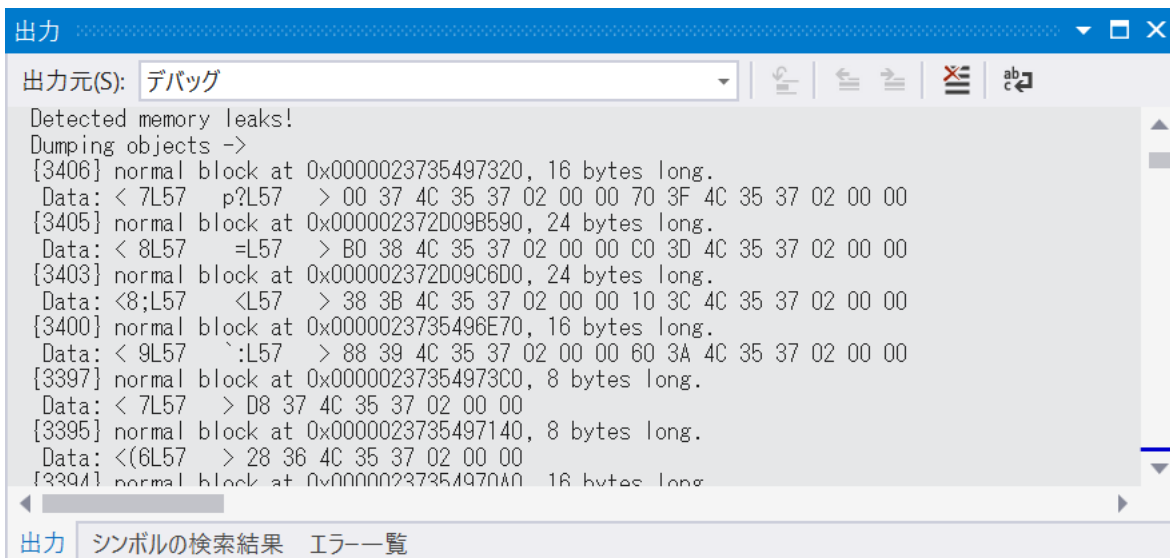
実装出来たら実行確認してみましょう。

下図のようにスライムが表示されていれば OK です。

## ゲームプログラミングⅢ

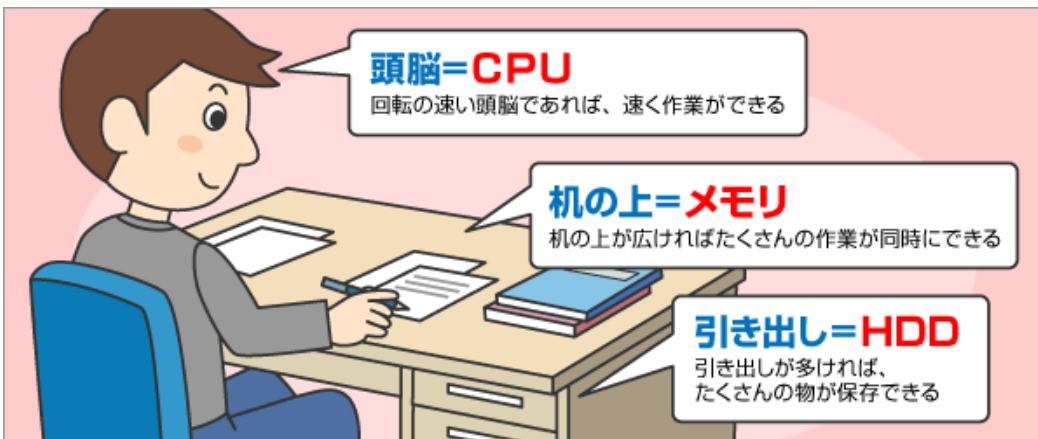


確認出来たらウインドウの×ボタンを押してゲームを終了してください。  
ゲーム終了後、VisualStudioの「出力」ウインドウを見てみましょう。  
下図のようにメモリリークが発生しているはずです。



メモリリークとは確保したメモリの解放を忘れている状態のことを指します。

## ゲームプログラミングⅢ



メモリを作業する机と例えると、**new** することで机の上にノートなどを出します。ノートを使って机の上で作業をし、必要なくなったら **delete** することでノートを片付けます。最終的には机の上には何もなくなっているはずですが、机の上になにか残っている状態で作業を終了してしまうことを「メモリリーク」と呼びます。

今回は `GameScene::Initialize()` でスライムを **new** しているのに **delete** をしていないため発生しています。

プログラムを組む人によってマネージャーの管理方法は様々ですが、今回は **EnemyManager** に登録された **Enemy** はマネージャーが破棄まで管理するようにします。

エネミーマネージャーに管理されているエネミーを全て削除する関数を作成します。**EnemyManager.h** と **EnemyManager.cpp** を開き、下記プログラムコードを追記しましょう。

### EnemyManager.h

```
---省略---  
  
// エネミーマネージャー  
class EnemyManager  
{  
public:  
    ---省略---  
  
    // エネミー全削除  
    void Clear();  
  
    ---省略---  
};
```

### EnemyManager.cpp

```
---省略---
```



## ゲームプログラミングⅢ

```
// エネミー全削除
void EnemyManager::Clear()
{
    
}
```

実装したら SceneGame の終了処理で呼び出すようにしましょう。

SceneGame.cpp

```
---省略---

// 終了化
void SceneGame::Finalize()
{
    // エネミー終了化
    

    ---省略---
}
```

実行確認をしてみてゲーム終了時に VisualStudio の出力ウィンドウにメモリリークのログが出力されていないなければ課題は完了です。

お疲れさまでした。