[OpenGL 2D 2018 第12回]



# Actor構造体をファイルに分離する

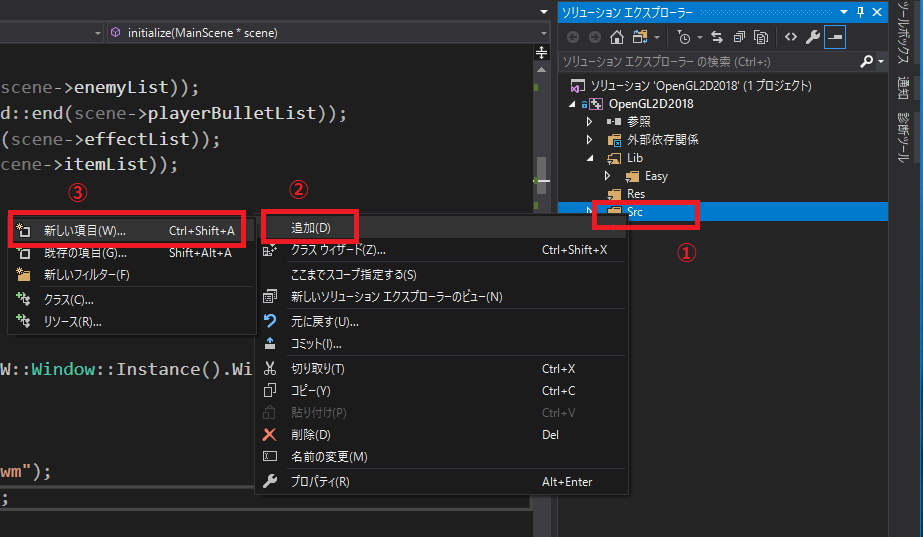
## 機能ごとにファイルを分ける理由

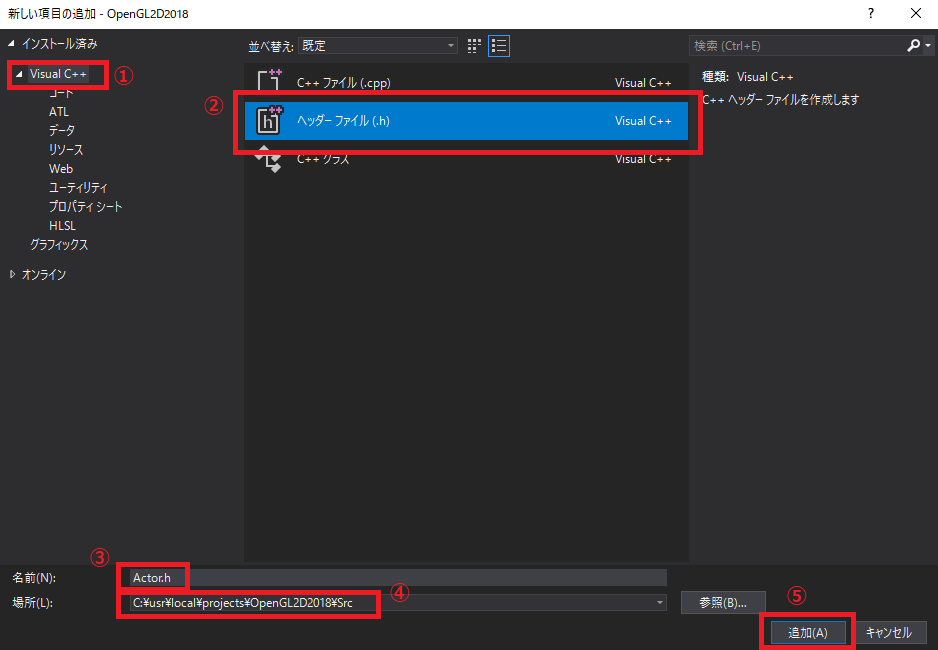
現在のMain.cppにはあらゆるプログラムが詰め込まれています。  
その結果、プログラムを変更する部分を見つけることが徐々に難しくなってきていると思います。  
今のところはなんとかなっているかもしれませんが、今後もプログラムが増えていくことを考えると、このあたりで対策をしておく必要があるでしょう。

一般的な対策としては、機能ごとにファイルを分割することがよく行われます。しかし、みなさんはどういった基準でプログラムを分ければいいかがまだ分からないと思います。そこで、簡単な部分からファイル分割を行い、分割する際の基本的な考え方を学んでいって貰おうと思います。

## Actor.hを追加する

最初のターゲットはActor構造体と、これに関係する関数です。  
まず、分割先となるファイルを追加します。

  
Visual StudioのソリューションエクスプローラーのSrcフィルターの上で右クリック(①)し、「追加(②)→新しい項目(③)」の順で選択して左クリックしてください。  
すると、「新しい項目の追加」というウィンドウが開きます。

  
左側のリストから「Visual C++(①)」を選択し、次に中央のリストから「ヘッダーファイル(.h)(②)」を選択します。選択したら「名前(③)」欄に「Actor.h」と入力してください。入力したら「場所(④)」欄の末尾に「\Src」という文字列を追加して、みなさんのプロジェクトフォルダの中にあるSrcフォルダを指すようにしてください(画像は筆者の環境のプロジェクトフォルダの例なので、そのまま書き写すことはしないでください。「D:\プロジェクトのあるフォルダ\Src」のような形になっていれば大丈夫です)。  
名前欄と場所欄を書き終えたら、「追加(⑤)」ボタンをクリックすると、場所欄で指定したフォルダに、名前欄で指定した名前のファイルが作成されます。

**[補足]**場所を間違えてしまうと、その間違えた場所にファイルが追加されてしまいます。ファイルがどの場所にあるかは、ビルドの際に非常に重要となります。間違えないよう注意深く書いてください。

## Actor.hをインクルードする

とりあえず作成したヘッダーファイルをMain.cppにインクルードします。  
Main.cppに次のプログラムを追加してください。

/\*\*  
\* @file Main.cpp  
\*/  
**+**#include "Actor.h"  
 #include "GLFWEW.h"  
 #include "Texture.h"  
 #include "Sprite.h"

## Actor構造体を移動する

Actor.hをインクルードしたら、Actor構造体をMain.cppからActor.hに移動しましょう。  
Main.cppのActor構造体の定義をカット(Ctrl+X)し、

const char title[] = "OpenGL2D 2018"; // ウィンドウタイトル.  
 const int windowWidth = 800; // ウィンドウの幅.  
 const int windowHeight = 600; // ウィンドウの高さ.  
**-**  
**-**/\*\*  
**-**\* ゲームキャラクター構造体.  
**-**\*/  
**-**struct Actor  
**-**{  
**-** Sprite spr; // スプライト.  
**-** Rect collisionShape; // 衝突判定の位置と大きさ.  
**-** int health; // 耐久力.  
**-**};  
  
 SpriteRenderer renderer; // スプライト描画用変数.  
 FontRenderer fontRenderer; // フォント描画用変数.

Actor.hにペースト(Ctrl+V)します。

#pragma once  
  
**+**/\*\*  
**+**\* ゲームキャラクター構造体.  
**+**\*/  
**+**struct Actor  
**+**{  
**+** Sprite spr; // スプライト.  
**+** Rect collisionShape; // 衝突判定の位置と大きさ.  
**+** int health; // 耐久力.  
**+** int type; // 種類.  
**+**};

ファイルの先頭にある「#pragma once(ぷらぐま・わんす)」は、Visual Studioが自動的に書き込んだプログラムです。先頭にシャープ記号があることから、これもプリプロセッサ指令の一種だと分かりますね。なお、pragma(ぷらぐま)はpragmatic(ぷらぐまてぃっく)を省略したもので「実用的、実際的」という意味があります。そして、once(わんす)は「一回、一度」という意味の英単語です。

C++言語において、#pragma指令はビルド環境(＝実際の環境)固有の機能を提供できるようにするためのものです。直後に指定した文字列に応じて様々な働きをします。onceの場合は「このヘッダーファイルが同じファイルに複数回インクルードされた場合、最初の1ヶ所にだけインクルードを行う」という指令になります。同じファイルを何度もインクルードするのは無駄ですし、ヘッダーファイルの内容によっては1ファイルにつき1度しか宣言できないデータが存在することもあるため、この指令によってそういった問題が起きないようにしておくのです。

さて、ペーストしただけだとSpriteやRectの定義が見つからない状態になっていますので、これらが定義されているSprite.hをインクルードします。ついでにファイルの説明コメントも追加しておきます。  
Actor.hに次のプログラムを追加してください。

#pragma once  
**+**/\*\*  
**+**\* @file Actor.h  
**+**\*  
**+**\* Actor構造体およびActor構造体に関連する関数を宣言するヘッダファイル.  
**+**\*/  
**+**#include "Sprite.h"  
  
/\*\*  
\* ゲームキャラクター構造体.  
\*/

これでActor構造体をファイルに分離することができました。  
プログラムが書けたら、ビルドしてエラーがないことを確認してください。  
エラーが発生した場合は分離手順を見直してください。

# Actor構造体に関連する関数をファイルに分離する

## プロトタイプ宣言をActor.hに移動する

次は、initializeActorList関数などのActor構造体に関連する関数を分離します。

**[補足]**  
「Actor構造体に関連する関数」というのは、パラメーターとしてActor構造体(またはActor構造体のポインタ)を受け取るか、結果としてActor構造体(またはActor構造体のポインタ)を返すような関数のことです。また、関連する関数の中でしか使用されていない関数も含まれます。

最初にプロトタイプ宣言をActor.hにカット&ペーストしましょう。  
Main.cppのプロトタイプ宣言をカット(Ctrl+X)して、

/\*  
 \* プロトタイプ宣言.  
 \*/  
 void processInput(GLFWEW::WindowRef);  
 void update(GLFWEW::WindowRef);  
 void render(GLFWEW::WindowRef);  
**-**bool detectCollision(const Rect\*, const Rect\*);  
**-**void initializeActorList(Actor\*, Actor\*);  
**-**Actor\* findAvailableActor(Actor\*, Actor\*);  
**-**void updateActorList(Actor\*, Actor\*, float);  
**-**void renderActorList(const Actor\*, const Actor\*);  
**-**  
**-**using CollisionHandlerType = void(\*)(Actor\*, Actor\*);  
**-**void detectCollision(Actor\*, Actor\*, Actor\*, Actor\*, CollisionHandlerType);  
void playerBulletAndEnemyContactHandler(Actor \* bullet, Actor \* enemy);  
void playerAndEnemyContactHandler(Actor \* player, Actor \* enemy);

Actor.hにペースト(Ctrl+V)します。

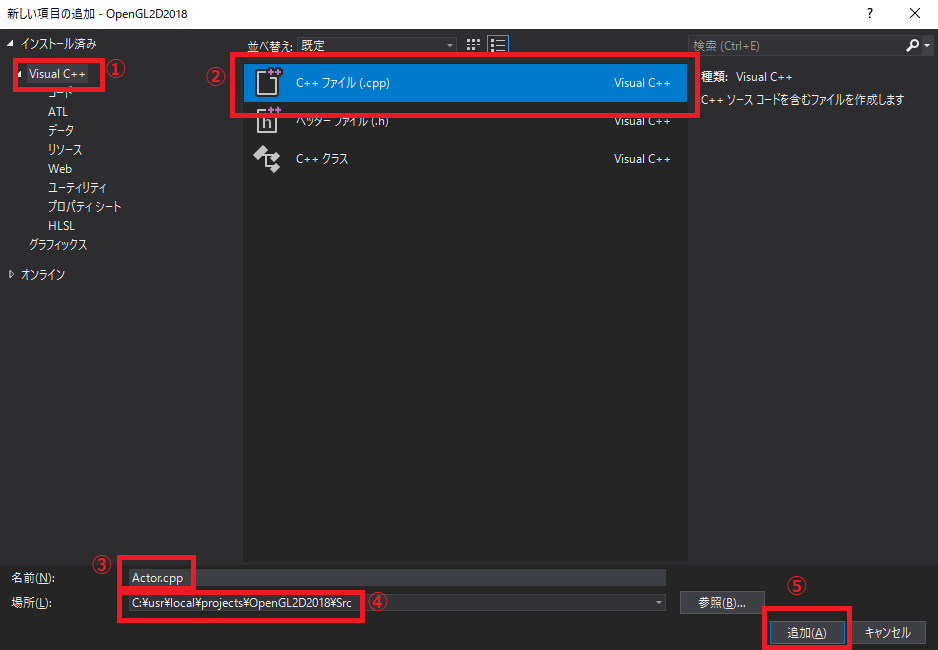
/\*\*  
\* ゲームキャラクター構造体.  
\*/  
struct Actor  
{  
 Sprite spr; // スプライト.  
 Rect collisionShape; // 衝突判定の位置と大きさ.  
 int health; // 耐久力.  
 int type; // 種類.  
};  
**+**bool detectCollision(const Rect\*, const Rect\*);  
**+**void initializeActorList(Actor\*, Actor\*);  
**+**Actor\* findAvailableActor(Actor\*, Actor\*);  
**+**void updateActorList(Actor\*, Actor\*, float);  
**+**void renderActorList(const Actor\*, const Actor\*);  
**+**  
**+**using CollisionHandlerType = void(\*)(Actor\*, Actor\*);  
**+**void detectCollision(Actor\*, Actor\*, Actor\*, Actor\*, CollisionHandlerType);

プログラムをファイルを分割すると、それぞれのファイルの中でプロトタイプ宣言が必要となります。そんなとき、必要なプロトタイプ宣言がヘッダファイルに書いてあれば、そのヘッダファイルをインクルードするだけで、その場にプロトタイプ宣言を書いたのと同じ効果が得られるという仕組みです。

プログラムが書けたら、ビルドしてエラーがないことを確認してください。  
エラーが発生した場合は分離手順を見直してください。

## Actor.cppを追加する

次に関数の定義を分離します。しかし、関数の定義は基本的にcppファイルに書かれますので、Actor.hには移動できません。そこで、Actor用のcppファイルを追加することにしましょう。

Actor.hを追加したときと同様に、Visual StudioのソリューションエクスプローラーのSrcフィルターの上で右クリックし、「追加→新しい項目」の順で選択して左クリックしてください。  
すると、「新しい項目の追加」というウィンドウが開きます。  


左側のリストから「Visual C++(①)」を選択し、次に中央のリストから「C++ファイル(.cpp)(②)」を選択します。選択したら「名前(③)」欄に「Actor.cpp」と入力してください。入力したら「場所(④)」欄がみなさんのプロジェクトのSrcフォルダを指していることを確認します。もし違うフォルダ名になっていたら、Srcフォルダを指すように書き換えてください(画像は筆者の環境のプロジェクトフォルダの例なので、そのまま書き写すことはしないでくださいね)。  
名前欄と場所欄を書き終えたら、「追加(⑤)」ボタンをクリックすると、場所欄で指定したフォルダに、名前欄で指定した名前のファイルが作成されます。

## Actor.hをインクルードする

作成したActor.cppはActor構造体に関連する関数の移動先となります。そして、移動してきた関数をビルドするにはActor構造体の定義やプロトタイプ宣言が必要となります。これらは既にActor.hに移動してありますから、Actor.hをインクルードするだけでO.K.です。  
Actor.cppに次のプログラムを追加してください。

**+**/\*\*  
**+**\* @file Actor.cpp  
**+**\*/  
**+**#include "Actor.h"

## 関数の定義を移動する

インクルード文を追加したら、関数の定義を移動しましょう。  
Main.cppから関数定義をカット(Ctrl+X)し、

fontRenderer.AddString(glm::vec2(-64 , 300), str);  
 fontRenderer.EndUpdate();  
 fontRenderer.Draw();  
  
 window.SwapBuffers();  
 }  
**-**  
**-**/\*\*  
**-**\* 2つの長方形の衝突状態を調べる.  
**-**\*  
**-**\* @param lhs 長方形その1.  
**-**\* @param rhs 長方形その2.  
**-**\*  
**-**\* @retval true 衝突している.  
**-**\* @retval false 衝突していない.  
**-**\*/  
**-**bool detectCollision(const Rect\* lhs, const Rect\* rhs)  
**-**{  
**-** return  
**-** lhs->origin.x < rhs->origin.x + rhs->size.x &&  
**-** lhs->origin.x + lhs->size.x > rhs->origin.x &&  
**-** lhs->origin.y < rhs->origin.y + rhs->size.y &&  
**-** lhs->origin.y + lhs->size.y > rhs->origin.y;  
**-**}  
**-**  
**-**/\*\*  
**-**\* Actorの配列を初期化する.  
**-**\*  
**-**\* @param first 初期化対象の先頭要素のポインタ.  
**-**\* @param last 初期化対象の終端要素のポインタ.  
**-**\*/  
**-**void initializeActorList(Actor\* first, Actor\* last)  
**-**{  
**-** for (Actor\* i = first; i != last; ++i) {  
**-** i->health = 0;  
**-** }  
**-**}  
**-**  
**-**/\*\*  
**-**\* 利用可能なのActorを取得する.  
**-**\*  
**-**\* @param first 検索対象の先頭要素のポインタ.  
**-**\* @param last 検索対象の終端要素のポインタ.  
**-**\*  
**-**\* @return 利用可能なActorのポインタ.  
**-**\* 利用可能なActorが見つからなければnullptr.  
**-**\*  
**-**\* [first, last)の範囲から、利用可能な(healthが0以下の)Actorを検索する.  
**-**\*/  
**-**Actor\* findAvailableActor(Actor\* first, Actor\* last)  
**-**{  
**-** for (Actor\* i = first; i != last; ++i) {  
**-** if (i->health <= 0) {  
**-** return i;  
**-** }  
**-** }  
**-** return nullptr;  
**-**}  
**-**  
**-**/\*\*  
**-**\* Actorの配列を更新する.  
**-**\*  
**-**\* @param first 更新対象の先頭要素のポインタ.  
**-**\* @param last 更新対象の終端要素のポインタ.  
**-**\* @param deltaTime 前回の更新からの経過時間.  
**-**\*/  
**-**void updateActorList(Actor\* first, Actor\* last, float deltaTime)  
**-**{  
**-** for (Actor\* i = first; i != last; ++i) {  
**-** if (i->health > 0) {  
**-** i->spr.Update(deltaTime);  
**-** if (i->spr.Tweener()->IsFinished()) {  
**-** i->health = 0;  
**-** }  
**-** }  
**-** }  
**-**}  
**-**  
**-**/\*\*  
**-**\* Actorの配列を描画する.  
**-**\*  
**-**\* @param first 描画対象の先頭要素のポインタ.  
**-**\* @param last 描画対象の終端要素のポインタ.  
**-**\*/  
**-**void renderActorList(const Actor\* first, const Actor\* last)  
**-**{  
**-** for (const Actor\* i = first; i != last; ++i) {  
**-** if (i->health > 0) {  
**-** renderer.AddVertices(i->spr);  
**-** }  
**-** }  
**-**}  
**-**  
**-**/\*\*  
**-**\* 衝突を検出する.  
**-**\*  
**-**\* @param firstA 衝突させる配列Aの先頭ポインタ.  
**-**\* @param lastA 衝突させる配列Aの終端ポインタ.  
**-**\* @param firstB 衝突させる配列Bの先頭ポインタ.  
**-**\* @param lastB 衝突させる配列Bの終端ポインタ.  
**-**\* @param function A-B間で衝突が検出されたときに実行する関数.  
**-**\*/  
**-**void detectCollision(Actor\* firstA, Actor\* lastA,  
**-** Actor\* firstB, Actor\* lastB, CollisionHandlerType function)  
**-**{  
**-** for (Actor\* a = firstA; a != lastA; ++a) {  
**-** if (a->health <= 0) {  
**-** continue;  
**-** }  
**-** Rect rectA = a->collisionShape;  
**-** rectA.origin += glm::vec2(a->spr.Position());  
**-** for (Actor\* b = firstB; b != lastB; ++b) {  
**-** if (b->health <= 0) {  
**-** continue;  
**-** }  
**-** Rect rectB = b->collisionShape;  
**-** rectB.origin += glm::vec2(b->spr.Position());  
**-** if (detectCollision(&rectA, &rectB)) {  
**-** function(a, b);  
**-** if (a->health <= 0) {  
**-** break;  
**-** }  
**-** }  
**-** }  
**-** }  
**-**}  
  
 /\*\*  
 \* 自機の弾と敵の衝突を処理する.  
 \*  
 \* @param bullet 自機の弾のポインタ.  
 \* @param enemy 敵のポインタ.  
 \*/  
 void playerBulletAndEnemyContactHandler(Actor\* bullet, Actor\* enemy)

そして、Actor.cppにペースト(Ctrl+V)します。

/\*\*  
\* @file Actor.cpp  
\*/  
#include "Actor.h"  
**+**  
**+**/\*\*  
**+**\* 2つの長方形の衝突状態を調べる.  
**+**\*  
**+**\* @param lhs 長方形その1.  
**+**\* @param rhs 長方形その2.  
**+**\*  
**+**\* @retval true 衝突している.  
**+**\* @retval false 衝突していない.  
**+**\*/  
**+**bool detectCollision(const Rect\* lhs, const Rect\* rhs)  
**+**{  
**+** return  
**+** lhs->origin.x < rhs->origin.x + rhs->size.x &&  
**+** lhs->origin.x + lhs->size.x > rhs->origin.x &&  
**+** lhs->origin.y < rhs->origin.y + rhs->size.y &&  
**+** lhs->origin.y + lhs->size.y > rhs->origin.y;  
**+**}  
**+**  
**+**/\*\*  
**+**\* Actorの配列を初期化する.  
**+**\*  
**+**\* @param first 初期化対象の先頭要素のポインタ.  
**+**\* @param last 初期化対象の終端要素のポインタ.  
**+**\*/  
**+**void initializeActorList(Actor\* first, Actor\* last)  
**+**{  
**+** for (Actor\* i = first; i != last; ++i) {  
**+** i->health = 0;  
**+** }  
**+**}  
**+**  
**+**/\*\*  
**+**\* 利用可能なのActorを取得する.  
**+**\*  
**+**\* @param first 検索対象の先頭要素のポインタ.  
**+**\* @param last 検索対象の終端要素のポインタ.  
**+**\*  
**+**\* @return 利用可能なActorのポインタ.  
**+**\* 利用可能なActorが見つからなければnullptr.  
**+**\*  
**+**\* [first, last)の範囲から、利用可能な(healthが0以下の)Actorを検索する.  
**+**\*/  
**+**Actor\* findAvailableActor(Actor\* first, Actor\* last)  
**+**{  
**+** for (Actor\* i = first; i != last; ++i) {  
**+** if (i->health <= 0) {  
**+** return i;  
**+** }  
**+** }  
**+** return nullptr;  
**+**}  
**+**  
**+**/\*\*  
**+**\* Actorの配列を更新する.  
**+**\*  
**+**\* @param first 更新対象の先頭要素のポインタ.  
**+**\* @param last 更新対象の終端要素のポインタ.  
**+**\* @param deltaTime 前回の更新からの経過時間.  
**+**\*/  
**+**void updateActorList(Actor\* first, Actor\* last, float deltaTime)  
**+**{  
**+** for (Actor\* i = first; i != last; ++i) {  
**+** if (i->health > 0) {  
**+** i->spr.Update(deltaTime);  
**+** if (i->spr.Tweener()->IsFinished()) {  
**+** i->health = 0;  
**+** }  
**+** }  
**+** }  
**+**}  
**+**  
**+**/\*\*  
**+**\* Actorの配列を描画する.  
**+**\*  
**+**\* @param first 描画対象の先頭要素のポインタ.  
**+**\* @param last 描画対象の終端要素のポインタ.  
**+**\*/  
**+**void renderActorList(const Actor\* first, const Actor\* last)  
**+**{  
**+** for (const Actor\* i = first; i != last; ++i) {  
**+** if (i->health > 0) {  
**+** renderer.AddVertices(i->spr);  
**+** }  
**+** }  
**+**}  
**+**  
**+**/\*\*  
**+**\* 衝突を検出する.  
**+**\*  
**+**\* @param firstA 衝突させる配列Aの先頭ポインタ.  
**+**\* @param lastA 衝突させる配列Aの終端ポインタ.  
**+**\* @param firstB 衝突させる配列Bの先頭ポインタ.  
**+**\* @param lastB 衝突させる配列Bの終端ポインタ.  
**+**\* @param function A-B間で衝突が検出されたときに実行する関数.  
**+**\*/  
**+**void detectCollision(Actor\* firstA, Actor\* lastA,  
**+** Actor\* firstB, Actor\* lastB, CollisionHandlerType function)  
**+**{  
**+** for (Actor\* a = firstA; a != lastA; ++a) {  
**+** if (a->health <= 0) {  
**+** continue;  
**+** }  
**+** Rect rectA = a->collisionShape;  
**+** rectA.origin += glm::vec2(a->spr.Position());  
**+** for (Actor\* b = firstB; b != lastB; ++b) {  
**+** if (b->health <= 0) {  
**+** continue;  
**+** }  
**+** Rect rectB = b->collisionShape;  
**+** rectB.origin += glm::vec2(b->spr.Position());  
**+** if (detectCollision(&rectA, &rectB)) {  
**+** function(a, b);  
**+** if (a->health <= 0) {  
**+** break;  
**+** }  
**+** }  
**+** }  
**+** }  
**+**}

カット&ペーストしたら、ビルドしてください。  
おそらく「renderer: 定義されていない識別子です。」といったエラーが発生することでしょう。  
このエラーは、コンピューターが「Actor.cppをビルドしたところ、rendererという名前が使われるまでにrendererの定義がどこにも見つからなかったんだ。だから、どうしていいか分からないよ。」と言っているのです。解決策は、その名前がどういうものなのかを書き加えることです。今回の場合はrendererがSpriteRenderer型の変数であることを書き加えればいい、ということになります。

## externキーワード

ところで、rendererは既にMain.cppに定義されています。そのため、Actor.cppに同じ名前の変数を定義しようとすると、コンピューターはどちらが本物なのかわからなくなって「1つ以上の複数回定義されているシンボルが見つかりました。」のようなエラーを出力してしまいます。

このエラーは「extern(えくすたーん)」という機能を使うことで解決可能です。externは「external(えくすたーなる)」を省略したもので「外部の」という意味の英単語です。externキーワードは、「変数が外部で定義されている」ことをコンピューターに知らせます。これによって、コンピューターはその変数がどういう型なのかに加えて、定義が別のファイルに存在することも分かるので、本物かどうかで悩むこともなくなる、というわけです。

実際にexternキーワードを使ってみましょう。  
Actor.cppに次のプログラムを追加してください。

/\*\*  
 \* @file Actor.cpp  
 \*/  
 #include "Actor.h"  
**+**  
**+**extern SpriteRenderer renderer;  
   
 /\*\*  
 \* 2つの長方形の衝突状態を調べる.  
 \*  
 \* @param lhs 長方形その1.  
 \* @param rhs 長方形その2.  
 \*  
 \* @retval true 衝突している.  
 \* @retval false 衝突していない.  
 \*/

プログラムが書けたらビルドしてください。今度はエラーが出なくなったと思います。

## externをヘッダーファイルに移す

これからさらにプログラムを分割していきますが、その際にもrendererなどの変数定義が必要になってきます。その都度externを使うのも手間がかかりますので、必要なものをまとめてヘッダーファイルに書いておくことにしましょう。

Actor.hを追加したのと同じ要領で、Srcフォルダに「GameData.h」というヘッダーファイルを追加してください。追加したら、GameData.hに次のプログラムを追加してください。

#pragma once  
**+**/\*\*  
**+**\* @file GameData.h  
**+**\*/  
**+**#include "Sprite.h"  
**+**#include "Font.h"  
**+**  
**+**extern SpriteRenderer renderer; // スプライト描画用変数.  
**+**extern FontRenderer fontRenderer; // フォント描画用変数.

次に、Actor.cppを以下のように変更してください。

/\*\*  
 \* @file Actor.cpp  
 \*/  
 #include "Actor.h"  
**+**#include "GameData.h"  
**-**  
**-**extern SpriteRenderer renderer;  
   
 /\*\*  
 \* 2つの長方形の衝突状態を調べる.  
 \*  
 \* @param lhs 長方形その1.  
 \* @param rhs 長方形その2.

プログラムが書けたらビルドしてエラーが出ないことを確認してください。

# ゲームオーバー画面をファイルに分離する

## GameOverScene.hを追加する

Actorの次は、ゲームオーバー画面を分離していきましょう。  
Actor.hやGameData.hを追加したのと同じやり方で、Srcフォルダに「GameOverScene.h」というヘッダーファイルを追加してください。  
追加したら、Main.cppにGameOverScene.hをインクルードします。  
Main.cppに次のプログラムを追加してください。

/\*\*  
\* @file Main.cpp  
\*/  
**+**#include "GameOverScene.h"  
#include "Actor.h"  
 #include "GLFWEW.h"  
 #include "Texture.h"  
 #include "Sprite.h"

インクルードすることによって、GameOverScene構造体を移動させたあとも、コンピューターに定義を見つけてもらえるようにしています。

## GameOverScene構造体とプロトタイプ宣言を移動する

追加したヘッダーファイルに、構造体とプロトタイプ宣言を移動します。  
次のように、Main.cppからGameOverScene構造体と関連するプロトタイプ宣言をカット(Ctrl+X)し、

bool initialize(TitleScene\*);  
 void finalize(TitleScene\*);  
 void processInput(GLFWEW::WindowRef, TitleScene\*);  
 void update(GLFWEW::WindowRef, TitleScene\*);  
 void render(GLFWEW::WindowRef, TitleScene\*);  
  
 TitleScene titleScene;  
**-**  
**-**/\*\*  
**-**\* ゲームオーバー画面で使用する構造体.  
**-**\*/  
**-**struct GameOverScene  
**-**{  
**-** Sprite bg;  
**-** float timer;  
**-**};  
**-**bool initialize(GameOverScene\*);  
**-**void finalize(GameOverScene\*);  
**-**void processInput(GLFWEW::WindowRef, GameOverScene\*);  
**-**void update(GLFWEW::WindowRef, GameOverScene\*);  
**-**void render(GLFWEW::WindowRef, GameOverScene\*);  
  
 GameOverScene gameOverScene;

GameOverScene.hにペーストしてください。

#pragma once  
**+**  
**+**/\*\*  
**+**\* ゲームオーバー画面で使用する構造体.  
**+**\*/  
**+**struct GameOverScene  
**+**{  
**+** Sprite bg;  
**+** float timer;  
**+**};  
**+**bool initialize(GameOverScene\*);  
**+**void finalize(GameOverScene\*);  
**+**void processInput(GLFWEW::WindowRef, GameOverScene\*);  
**+**void update(GLFWEW::WindowRef, GameOverScene\*);  
**+**void render(GLFWEW::WindowRef, GameOverScene\*);

Actor.hでもそうでしたが、このファイルにはSpriteの定義がありません。さらにGLFWEW::WindowRefの定義もないようです。そのため、このままではビルドに失敗してしまうでしょう。そこで、これらが定義されているヘッダーファイルをインクルードします。ついでにファイルの説明も追加しましょうか。

GameOverScene.hに次のプログラムを追加してください。

#pragma once  
**+**/\*\*  
**+**\* @file GameOverScene.h  
**+**\*  
**+**\* ゲームーオーバー画面のための構造体および関数を宣言するヘッダファイル.  
**+**\*/  
**+**#include "GLFWEW.h"  
**+**#include "Sprite.h"  
  
/\*\*  
\* ゲームオーバー画面で使用する構造体.  
\*/

書けたらビルドして、エラーが発生しないことを確認してください。  
エラーが発生した場合は手順を見直してください。

## 関数の定義を移動する

続いて、関数の定義を移動します。まず移動先となるファイルを追加しましょう。  
Actor.cppのときと同じ方法で、「GameOverScene.cpp」というC++ファイルを追加してください。  
追加したファイルを開き、次のプログラムを追加してください。

**+**/\*\*  
**+**\* @file GameOverScene.cpp  
**+**\*/  
**+**#include "GameOverScene.h"

これで、GameOverScene構造体とプロトタイプ宣言がGameOverScene.cppに取り込まれます。  
次に関数の定義を移動しますが、同じことは既にActor.cppでやりましたので、課題とします。

**[課題01]**  
GameOverScene.hでプロトタイプ宣言されているすべての関数の定義を、Main.cppからGameOverScene.cppへカット&ペーストしてください。ビルドするとエラーが表示されますが、この時点では気にしなくて構いません。。

## gamestate変数をexternする

ゲームオーバー画面では、タイトル画面に移動するためにgamestate変数を使っています。  
しかしこの変数はMain.cppの中に書かれていて、GameOverScene.cppからは見えません。  
こんなときはextern(えくすたーん)キーワードの出番です。

Main.cppからgamestateに関するプログラムをコピー(Ctrl+C)し、GameData.hにペースト(Ctrl+V)したら、externキーワードを書き足してください。

#include "Sprite.h"  
 #include "Font.h"  
  
 extern SpriteRenderer renderer; // スプライト描画用変数.  
 extern FontRenderer fontRenderer; // フォント描画用変数.  
**+**  
**+**const int gamestateTitle = 0;  
**+**const int gamestateMain = 1;  
**+**const int gamestateGameover = 2;  
**+**extern int gamestate; // ゲームの状態.

なお、C++言語のconst変数はC++ファイルごとに別の定義として扱われるため、externキーワードは使いません。ただし、このままではMain.cppの定義と重複してしまうので、Main.cpp側のconst変数を消しておきましょう。  
Main.cppを次のように変更してください。

bool initialize(TitleScene\*);  
 void finalize(TitleScene\*);  
 void processInput(GLFWEW::WindowRef, TitleScene\*);  
 void update(GLFWEW::WindowRef, TitleScene\*);  
 void render(GLFWEW::WindowRef, TitleScene\*);  
  
 TitleScene titleScene;  
 GameOverScene gameOverScene;  
  
**-**const int gamestateTitle = 0;  
**-**const int gamestateMain = 1;  
**-**const int gamestateGameover = 2;  
int gamestate; // ゲームの状態.

それでは、externしたgamestate変数をGameOverScene.cppに取り込みましょう。  
GameOverScene.cppに次のプログラムを追加してください。

/\*\*  
\* @file GameOverScene.cpp  
\*/  
#include "GameOverScene.h"  
**+**#include "GameData.h"  
  
 /\*\*  
 \* ゲームオーバー画面の初期設定を行う.  
 \*  
 \* @param scene ゲームオーバー画面用構造体のポインタ.  
 \*  
 \* @retval true 初期化成功.  
 \* @retval false 初期化失敗.  
 \*/  
 bool initialize(GameOverScene\* scene)

書けたらビルドしてください。  
…うーん、まだエラーがなくなりませんね。  
コンピューターは、titleSceneやTitleSceneという名前、initialize(GLFWEW::Window&, TitleScene\*)という関数が何なのか分からないようです。

## シーン変数をexternする

先ほどのエラーの原因は、ゲームオーバー画面のプログラムではタイトル画面用の構造体や関数を使っているのに、コンピューターから見えるところにそれらの定義がなかったからです。  
ここではやることが2つあります。ひとつめはtitleScene変数を他のファイルから見えるようにすることです。そして、もうひとつはTitleScene構造体とinitialize関数を他のファイルから見えるようにする、つまりファイルに移動することです。

それでは、titleScene変数をexternすることから済ませましょう。ついでなので、gameOverScene変数もexternしておきます。GameData.hに次のプログラムを追加してください。

#pragma once  
 /\*\*  
 \* @file GameData.h  
 \*/  
 #include "Sprite.h"  
 #include "Font.h"  
  
 extern SpriteRenderer renderer; // スプライト描画用変数.  
 extern FontRenderer fontRenderer; // フォント描画用変数.  
   
 const int gamestateTitle = 0;  
 const int gamestateMain = 1;  
 const int gamestateGameover = 2;  
 extern int gamestate; // ゲームの状態.  
**+**  
**+**// 先行宣言.  
**+**struct TitleScene;  
**+**struct GameOverScene;  
**+**  
**+**extern TitleScene titleScene;  
**+**extern GameOverScene gameOverScene;

GameData.hは既にGameOverScene.cppへインクルード済みですから、titleScene変数が見えるようになったはずです。

## windowWidth, windowHeigthを移動する

次に、windowWidthとwindowHeightの定義をGameData.hに移動します。  
Main.cppから以下のプログラムをカット(Ctrl+X)して、

#include "Sprite.h"  
 #include "Font.h"  
 const char title[] = "OpenGL2D 2018"; // ウィンドウタイトル. **-**const int windowWidth = 800; // ウィンドウの幅.  
**-**const int windowHeigh = 600; // ウィンドウの高さ.  
  
 SpriteRenderer renderer; // スプライト描画用変数.  
 FontRenderer fontRenderer; // フォント描画用変数.

GameData.hに貼り付け(Ctrl+V)てください。

#include "Sprite.h"  
 #include "Font.h"  
 **+**const int windowWidth = 800; // ウィンドウの幅.  
**+**const int windowHeigh = 600; // ウィンドウの高さ.  
  
 extern SpriteRenderer renderer; // スプライト描画用変数.  
 extern FontRenderer fontRenderer; // フォント描画用変数.

## GameData.hをインクルードする

いくつかの変数をGameData.hに移動したので、Main.cppでその変数を使えなくなってしまいました。再び使えるようにするには、それらの変数が書いてあるGameData.hをインクルードします。  
Main.cppに次のプログラムを追加してください。

/\*\*  
\* @file Main.cpp  
\*/  
#include "GameOverScene.h"  
**+**#include "GameData.h"  
#include "Actor.h"  
 #include "GLFWEW.h"  
 #include "Texture.h"  
 #include "Sprite.h"

これで問題の半分ほどは解決したはずです。  
あとひとつ作業が残っていますが、それは次章で行うことにしましょう。

# タイトル画面をファイルに分離する

## TitleScene.hを追加し、Main.cppにインクルードする

GameOverScene.cppを完成させるために、TitleScene構造体とそれに関連する関数のプロトタイプ宣言をファイルに分離します。

Actor.hやGameOverScene.hを作成したときと同じ手順で、Srcフォルダに「TitleScene.h」というヘッダーファイルを追加してください。追加したら、TitleScene.hをMain.cppにインクルードしましょう。  
Main.cppに次のプログラムを追加してください。

/\*\*  
\* @file Main.cpp  
\*/  
**+**#include "TitleScene.h"  
 #include "GameOverScene.h"  
#include "Actor.h"  
 #include "GLFWEW.h"  
 #include "Texture.h"  
 #include "Sprite.h"

TitleScene構造体と、それに関連するプロトタイプ宣言の移動は課題とします。

**[課題02]**  
TitleScene構造体と、それに関連するプロトタイプ宣言を、Main.cppからTitleScene.hにカット&ペーストしてください。その後、TitleScene.hに必要なヘッダーファイルをインクルードしてください(どのファイルが必要なのかについてはGameOverScene.hを参考にしてください)。

## TitleScene.hをGameOverScene.cppにインクルードする

TitleScene.hのプログラムができたら、GameOverScene.cppにインクルードしましょう。  
GameOverScene.cppに次のプログラムを追加してください。

/\*\*  
\* @file GameOverScene.cpp  
\*/  
#include "GameOverScene.h"  
**+**#include "TitleScene.h"  
#include "GameData.h"  
  
 /\*\*  
 \* ゲームオーバー画面の初期設定を行う.  
 \*  
 \* @param scene ゲームオーバー画面用構造体のポインタ.  
 \*  
 \* @retval true 初期化成功.  
 \* @retval false 初期化失敗.  
 \*/  
 bool initialize(GameOverScene\* scene)

これでようやく、GameOverScene.cppがビルドできるようになりました。  
ビルドしてエラーが出ないことを確認してください。

## 関数の定義を移動する

ゲームオーバー画面のファイル分割がうまくいったことが確認できたら、タイトル画面の分割を続けましょう。最初の作業は分割先となるC++ファイルを追加することです。Actor.cppやGameOverScene.cppを追加したのと同じ方法で、皆さんのプロジェクトのSrcフォルダに「TitleScene.cpp」という名前のファイルを追加してください。

追加したファイルを開き、次のプログラムを追加してください。

**+**/\*\*  
**+**\* @file TitleScene.cpp  
**+**\*/  
**+**#include "TitleScene.h"

これで、TitleScene構造体とプロトタイプ宣言がTitleScene.cppに取り込まれます。  
プロトタイプ宣言された関数の移動は課題とします。

**[課題03]**  
TitleScene.hにプロトタイプ宣言されている関数の定義を、Main.cppからTitleScene.cppにカット&ペーストしてください。ビルドするとエラーが表示されますが、この時点では気にしなくて構いません。。

## ヘッダーファイルのインクルードを追加する

コンピューターはいろいろな名前が分からずに困っているようですから、エラーをなくすためにすこしずつ修正していきましょう。  
タイトル画面では、画面の描画にrenderer変数とfontRenderer変数を使っています。  
この2つはGameData.hにexternしてありますから、GameData.hをインクルードすれば良さそうです。また、音声を再生するためにAudio::Engineを使っています。これはAudio.hで定義されているのでした。まずはこの2つのファイルをインクルードします。  
TitleScene.cppに次のプログラムを追加してください。

/\*\*  
\* @file TitleScene.cpp  
\*/  
#include "TitleScene.h"  
**+**#include "GameData.h"  
**+**#include "Audio.h"

## メイン画面の初期化を関数にする

これで少しはエラーが減りました。残ったエラーも多いですが、実はそれは全てupdate関数の中で発生しています。タイトル画面用のupdate関数にはメイン画面用の初期化を行うプログラムがあります。しかし、この部分で使っている変数はMain.cppの中で定義されていますから、TitleScene.cppをビルドするときはコンピューターからは見えません。

さて、変数を別のファイルから見えるようにする方法として、externキーワードを使うというものがありましたね。しかし今回は別の方法を取ることにします。問題になっているのは「メイン画面用の初期化を行うプログラム」でした。しかし、そもそも今回作っているのはタイトル画面用のC++ファイルなのですから、そこにメイン画面用のプログラムがあること自体がおかしいと考えられます。

## extern VS 関数

このような場合、メイン画面用の初期化を行うプログラムを関数にして、タイトル画面ではその関数を呼び出す形するほうが適切でしょう。externに対する関数の利点は、たくさんの変数をexternしなくても、関数のプロトタイプ宣言をひとつ作るだけで済むことです。  
欠点は、関数を作らなければならないことです。しかし、特定の目的を持った機能を関数として定義するのは、多くの場合でよりよいプログラミング作法です。ですから、これが本当に欠点だと感じることはほとんどないでしょう。

## MainScene.hを追加する

複数のファイルで簡単にプロトタイプ宣言を行うには、ヘッダーファイルを作成してそこに書くのが一番お手軽です。Actor.hやGameData.hを追加したときと同じ方法で、Srcフォルダに「MainScene.h」という名前のヘッダーファイルを作成してください。  
追加したら、MainScene.hに次のプログラムを追加してください。

#pragma once  
**+**/\*\*  
**+**\* @file MainScene.h  
**+**\*/  
**+**  
**+**/\*\*  
**+**\* メイン画面で使用する構造体.  
**+**\*/  
**+**struct MainScene  
**+**{  
**+**};  
**+**bool initialize(MainScene\*);

初期化関数のパラメータはTitleSceneやGameOverSceneと同じ形式になっている方が間違えにくくなると思います。そこで、中身のないMainScene構造体を追加しています。理想的にはメイン画面で利用する全ての変数をこの構造体の中に詰め込むべきなのでしょうが、大きな変更になりそうなので今回は見送りました。

## MainScene.hをインクルードする

メイン画面の初期化関数はMain.cppに追加します。なぜなら、メイン画面の初期化で必要な変数は全てMain.cppの中に書かれているからです。

まずはヘッダーファイルをインクルードします。  
Main.cppに次のプログラムを追加してください。

/\*\*  
\* @file Main.cpp  
\*/  
**+**#include "MainScene.h"  
#include "TitleScene.h"  
 #include "GameOverScene.h"  
#include "Actor.h"  
 #include "GLFWEW.h"  
 #include "Texture.h"  
 #include "Sprite.h"

## MainScene型の変数を追加する

次に、MainScene型の変数を追加します。  
Main.cppに次のプログラムを追加してください。

int gamestate; // ゲームの状態.  
  
 TitleScene titleScene;  
**+**MainScene mainScene;  
 GameOverScene gameOverScene;  
  
 /\*\*  
 \* プログラムのエントリーポイント.  
 \*/

この変数は他のファイルからも見えるようにしたいので、GameData.hにexternを追加しましょう。  
GameData.hに次のプログラムを追加してください。

const int gamestateTitle = 0;  
 const int gamestateMain = 1;  
 const int gamestateGameover = 2;  
 extern int gamestate; // ゲームの状態.  
  
// 先行宣言.  
struct TitleScene;  
**+**struct MainScene;  
struct GameOverScene;  
  
extern TitleScene titleScene;  
**+**extern MainScene mainScene;  
extern GameOverScene gameOverScene;

## メイン画面用の初期化関数を追加する

続いて初期化関数を追加していきます。  
Main.cppに次のプログラムを追加してください。

/\*  
 \* プロトタイプ宣言.  
 \*/  
 void processInput(GLFWEW::WindowRef);  
 void update(GLFWEW::WindowRef);  
 void render(GLFWEW::WindowRef);  
void playerBulletAndEnemyContactHandler(Actor \* bullet, Actor \* enemy);  
 void playerAndEnemyContactHandler(Actor \* player, Actor \* enemy);  
**+**  
**+**/\*\*  
**+**\* メイン画面用の構造体の初期設定を行う.  
**+**\*  
**+**\* @param scene メイン画面用構造体のポインタ.  
**+**\*  
**+**\* @retval true 初期化成功.  
**+**\* @retval false 初期化失敗.  
**+**\*/  
**+**bool initialize(MainScene\* scene)  
**+**{  
**+** return true;  
**+**}  
  
 /\*\*  
 \* プログラムのエントリーポイント.  
 \*/

そして、TitleScene.cppから次のプログラムをカット(Ctrl+X)し、

} else if (scene->mode == scene->modeNextState) {  
 finalize(scene);  
 gameData.gamestate = gameData.gamestateMain;  
**-  
-** sprBackground = Sprite("Res/UnknownPlanet.png");  
**-** sprPlayer.spr =  
**-** Sprite("Res/Objects.png", glm::vec3(0, 0, 0), Rect(0, 0, 64, 32));  
**-** sprPlayer.collisionShape = Rect(-24, -8, 48, 16);  
**-** sprPlayer.health = 1;  
**-**  
**-** initializeActorList(std::begin(enemyList), std::end(enemyList));  
**-** initializeActorList(std::begin(playerBulletList),  
**-** std::end(playerBulletList));  
**-** initializeActorList(std::begin(effectList), std::end(effectList));  
**-**  
**-** score = 0;  
**-**  
**-** enemyMap.Load("Res/EnemyMap.json");  
**-** mapCurrentPosX = mapProcessedX = windowWidth;  
**-**  
**-** Audio::EngineRef audio = Audio::Engine::Instance();  
**-** seBlast = audio.Prepare("Res/Audio/Blast.xwm");  
**-** sePlayerShot = audio.Prepare("Res/Audio/PlayerShot.xwm");  
**-** bgm = audio.Prepare(L"Res/Audio/Neolith.xwm");  
**-** bgm->Play(Audio::Flag\_Loop);  
}

先ほど作成したinitialize関数に貼り付け(Ctrl+V)てください。

int gamestate; // ゲームの状態.  
  
/\*\*  
\* メイン画面用の構造体の初期設定を行う.  
\*  
\* @param scene メイン画面用構造体のポインタ.  
\*  
\* @retval true 初期化成功.  
\* @retval false 初期化失敗.  
\*/  
bool initialize(MainScene\* scene)  
{  
**+** sprBackground = Sprite("Res/UnknownPlanet.png");  
**+** sprPlayer.spr =  
**+** Sprite("Res/Objects.png", glm::vec3(0, 0, 0), Rect(0, 0, 64, 32));  
**+** sprPlayer.collisionShape = Rect(-24, -8, 48, 16);  
**+** sprPlayer.health = 1;  
**+**  
**+** initializeActorList(std::begin(enemyList), std::end(enemyList));  
**+** initializeActorList(std::begin(playerBulletList),  
**+** std::end(playerBulletList));  
**+** initializeActorList(std::begin(effectList), std::end(effectList));  
**+**  
**+** score = 0;  
**+**  
**+** enemyMap.Load("Res/EnemyMap.json");  
**+** mapCurrentPosX = mapProcessedX = windowWidth;  
**+**  
**+** Audio::EngineRef audio = Audio::Engine::Instance();  
**+** seBlast = audio.Prepare("Res/Audio/Blast.xwm");  
**+** sePlayerShot = audio.Prepare("Res/Audio/PlayerShot.xwm");  
**+** bgm = audio.Prepare(L"Res/Audio/Neolith.xwm");  
**+** bgm->Play(Audio::Flag\_Loop);  
**+**  
 return true;  
}  
  
 /\*\*  
 \* プログラムのエントリーポイント.  
 \*/

## initialize関数を使う

それでは、作成した初期化関数をタイトル画面用のupdate関数の中で呼び出しましょう。  
まずMainScene.hをインクルードします。  
TitleScene.cppに次のプログラムを追加してください。

/\*\*  
\* @file TitleScene.cpp  
\*/  
#include "TitleScene.h"  
**+**#include "MainScene.h"  
 #include "GameData.h"  
#include "Audio.h"

インクルードしたら、TitleScene.cppのupdate関数に次のプログラムを追加してください。

void update(GLFWEW::WindowRef window, TitleScene\* scene)  
 {  
 const float deltaTime = window.DeltaTime();  
  
 scene->bg.Update(deltaTime);  
 scene->logo.Update(deltaTime);  
  
 if (scene->timer > 0) {  
 scene->timer -= deltaTime;  
 return;  
 }  
 if (scene->mode == scene->modeStart) {  
 scene->mode = scene->modeTitle;  
 } else if (scene->mode == scene->modeNextState) {  
 finalize(scene);  
 gamestate = gamestateMain;  
**+** initialize(&mainScene);  
 }  
 }

これでタイトル画面のエラーはなくなるはずです。  
ビルドしてエラーがなくなったことを確認してください。