[OpenGL 2D 2018 第10回]

Beyond the Copy&Paste

コピー＆ペーストの宇宙を越えて

# 重複するプログラムを関数にする

## コピペは間違いもコピペする

これまで、何度も「コピー&ペースト(コピペ)」を使ってプログラムを書いてもらいました。  
コピペは便利ですが、もしコピー元に間違った部分があった場合、修正する手間はペーストした回数に応じて2倍、3倍になってしまいます。

せっかく楽ができると思ったのに、手間が増えてしまっては本末転倒です。コピペの便利さをたもちながら、修正の手間も1回で済むような都合のいい方法があればいいのですが…。  
それがですね、実はあるんですよ。  
「関数(かんすう)」っていうんですけども。

## 関数>>>コピペ

「関数」は「大きいプログラムを作るための部品(小さいプログラム)」です。  
プログラムの機能の一部分をコピーする代わりに、その部分を関数にして名前をつけます。機能が必要とされる場所には、ペーストする代わりにその名前を書きます。すると、名前をつけられた関数が実行される、という仕組みです。

プログラムは関数の一箇所だけなので、間違った部分があったとしても修正は容易です。  
また、Visual Studioは関数の名前を知っているので、名前の最初の数文字を書くだけで、候補として関数の名前を挙げてくれます。プログラマーは矢印キーで書きたい名前を選んでエンター(Enter)キーを押すだけです。コピペのようにコピー元を探す手間もかかりませんし、うっかり単語を書き間違えることもありません。  
さらに、適切な名前をつけることで、その部分が実際には何をしているのか、ということが分かりやすくなり、プログラム全体の見通しがよくなります。

例えば、運動会のとある競技について、運動会プログラムに

「棒とカゴを2組用意する。赤と白の玉をそれぞれ数十個用意する。カゴを棒の先に取り付け、2本を離れた位置に垂直に立てる。赤と白の玉を立てた棒の周囲にまく。競技者は自分のチームの色の玉を取り、自分のチーム用のカゴに玉を投げ入れる。競技時間は60秒とし、この時間内により多くの玉をカゴに入れたチームの勝利とする。」

などと書かれているより、

「玉入れ競争」

のほうが分かりやすいですよね。  
また、玉入れが1回だけでなく、例えば1年生、3年生、5年生の3回行われる場合、最初の書き方を3回コピペしても、読みやすい運動会プログラムになるとは思えません。

このように、関数は様々な点でコピペよりも便利なのです。

# Actor配列の初期化を関数にする

## プロトタイプ宣言

手始めに、Actor型の配列を初期化するプログラムを関数にしてみましょう。これは次のプログラムの2つのfor文に当たります。

// 敵の配列を初期化.  
for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
 i->health = 0;  
}  
enemyGenerationTimer = 2;  
// 自機の弾の配列を初期化.  
for (Actor\* i = std::begin(playerBulletList);  
 i != std::end(playerBulletList); ++i) {  
 i->health = 0;  
}

プログラムを関数にする場合、最初に関数の名前を考えます。関数の名前は、そのブログラムが行うことを明確に表すような名前にするべきです。そうすることで、プログラムが読みやすくなります。今回作るのは「Actorの配列を初期化する関数」ですから、「initializeActorList(いにしゃらいず・あくたー・りすと」)」という名前にします。プロトタイプ宣言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);  
 bool detectCollision(const Rect\* lhs, const Rect\* rhs);  
**+**initializeActorList;

次に、関数のパラメータを決めます。  
これには、そのプログラムのどの部分がコピペするだけで動き、どの部分がコピペしたあとで変更しなければならなかったか、を考えるといいでしょう。変更しなければならなかった部分は、関数のパラメータになりえます。現在のところ、初期化は「敵」と「自機の弾」の2つの配列で実行されています。この2つのプログラムは、配列の名前以外は全く同一です。ということは、配列の名前をパラメータにすればよさそうです。

ただし、C言語とC++言語の仕様により、配列をそのままパラメータにすることはできません。  
どういうことかというと、CおよびC++言語では、配列の名前をパラメータに設定すると、それは「配列の先頭の要素」を示すポインタとみなされるのです。  
「？　先頭が分かっていればいいんじゃない？」と思うかもしれません。しかし、ポインタに変換されると「配列の長さを示す情報が消え失せてしまう」のです。そのため、配列の名前をパラメータに設定するだけでは、配列の終端の位置がわかりません。

そこで、配列の名前ではなく、配列の先頭と終端の2つをパラメータとします。初期化プログラムで言うと、std::begin(配列名)、std::end(配列名)の2つです。for文を見るとこの2はActor\*(あくたー・ぽいんた)型のようですから、パラメーターは「Actor\*を2つ」となります。先程書いたinitializeActorListの行を次のように変更してください。

/\*  
 \* プロトタイプ宣言.  
 \*/  
 void processInput(GLFWEW::WindowRef);  
 void update(GLFWEW::WindowRef);  
 void render(GLFWEW::WindowRef);  
 bool detectCollision(const Rect\* lhs, const Rect\* rhs);  
**-**initializeActorList;  
**+**initializeActorList(Actor\*, Actor\*);

関数は結果を返すことができますが、今回の場合、配列の先頭と終端が正しく設定されている限り失敗はありえませんし、他に特に返すような結果はありません。ですから、結果の型はvoid型でいいでしょう。先程書いたinitializeActorListの行を、さらに次のように変更してください。

/\*  
 \* プロトタイプ宣言.  
 \*/  
 void processInput(GLFWEW::WindowRef);  
 void update(GLFWEW::WindowRef);  
 void render(GLFWEW::WindowRef);  
 bool detectCollision(const Rect\* lhs, const Rect\* rhs);  
**-**initializeActorList(Actor\*, Actor\*);  
**+**void initializeActorList(Actor\*, Actor\*);

これでプロトタイプ宣言は完成です。

## 関数定義

続いて関数の定義を書いていきます。  
detectCollision関数の定義の下に、次のプログラムを追加してください。

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.  
**+**\*/  
**+**void initializeActorList(Actor\*, Actor\*)  
**+**{  
**+**}

これはまだ雛形を書いただけの状態ですが、これから内容を追加していきます。  
まずは関数のもとになるプログラムをコピーします。下記の最初のfor文をコピーしてください。

// 使用する画像を用意.  
 sprBackground = Sprite("Res/UnknownPlanet.png");  
 sprPlayer = Sprite("Res/Objects.png", glm::vec3(0, 0, 0), Rect(0, 0, 64, 32));  
  
 // 敵の配列を初期化.  
 for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
 i->health = 0;  
 }  
 enemyGenerationTimer = 2;

コピーしたら、それをinitializeActorList関数の定義にペーストしましょう。

/\*\*  
 \* Actorの配列を初期化する.  
\*/  
void initializeActorList(Actor\*, Actor\*)  
{  
**+** for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
**+** i->health = 0;  
**+** }  
 }

さて、このままでもエラーは出ませんが、関数の目的は「パラメータで設定された配列を初期化する」ことで、「enemyListを初期化する」ことではありません。  
そこで、パラメータで指定された値を使うように書き換えます。上記のプログラムを次のように変更してください。

/\*\*  
 \* Actorの配列を初期化する.  
**+**\*  
**+**\* @param first 初期化する配列の先頭.  
**+**\* @param last 初期化する配列の終端.  
\*/  
**-**void initializeActorList(Actor\*, Actor\*)  
**+**void initializeActorList(Actor\* first, Actor\* last)  
{  
**-** for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
**+** for (Actor\* i = first; i != last; ++i) {  
 i->health = 0;  
 }  
 }

これで関数定義も完了です。

## 初期化プログラムを関数で置き換える

それでは、完成した関数を使って初期化プログラムを置き換えていきましょう。  
Actor配列の初期化プログラムを次のように変更してください。

// 使用する画像を用意.  
 sprBackground = Sprite("Res/UnknownPlanet.png");  
 sprPlayer = Sprite("Res/Objects.png", glm::vec3(0, 0, 0), Rect(0, 0, 64, 32));  
  
**-**// 敵の配列を初期化.  
**-**for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
**-** i->health = 0;  
**-**}  
 enemyGenerationTimer = 2;  
**-**// 自機の弾の配列を初期化.  
**-**for (Actor\* i = std::begin(playerBulletList);  
**-** i != std::end(playerBulletList); ++i) {  
**-** i->health = 0;  
**-**}  
**+**initializeActorList(std::begin(enemyList), std::end(enemyList)); **+**initializeActorList(std::begin(playerBulletList),std::end(playerBulletList));

プログラムが書けたら、ビルドして実行してください。  
今回の変更ではプログラムの動作は何も変更していませんので、変更前と同じように動いていれば関数による置き換えは成功です。

# Actor配列の更新を関数にする

## プロトタイプ宣言

初期化の次は、更新も関数にしましょう。Actorの更新は以下の部分にあたります。

// 敵の更新.  
 for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
 if (i->health > 0) {  
 i->spr.Update(deltaTime);  
 if (i->spr.Tweener()->IsFinished()) {  
 i->health = 0;  
 }  
 }  
 }  
  
 // 自機の弾の更新.  
 for (Actor\* i = std::begin(playerBulletList);  
 i != std::end(playerBulletList); ++i) {  
 if (i->health > 0) {  
 i->spr.Update(deltaTime);  
 if (i->spr.Tweener()->IsFinished()) {  
 i->health = 0;  
 }  
 }  
 }  
 }

初期化のときと同じくプロトタイプ宣言を書くことから始めましょう。  
まずは名前を決めます。Actorの配列を更新するのですから「updateActorList(あっぷでーと・あくたー・りすと)」でいいでしょう。  
次にパラメーターですが、2つのfor文で違うところは初期化と同じく配列の名前くらいです。ということは今回も「Actor\*(あくたー・ぽいんたー)を2つ」でいいはずです。…おっと、deltaTimeのことを忘れていました。deltaTime変数はupdate関数の中で定義されています。このような変数は他の関数から見ることはできません。ですから、これもパラメーターとして渡す必要があります。  
最後は結果の型です。これも初期化と同じく、パラメーターさえ正しければ失敗する要素はなさそうなのでvoidでいいでしょう。  
全てを決めたらプロトタイプ宣言を書きます。プロトタイプ宣言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);  
 bool detectCollision(const Rect\* lhs, const Rect\* rhs);  
void initializeActorList(Actor\*, Actor\*);  
**+**void updateActorList(Actor\*, Actor\*, float);

## 関数定義

続いて関数の定義を書きます。  
初期化と同様に雛形から始めます。  
initializeActorList関数の定義の下に、次のプログラムを追加してください。

/\*\*  
 \* Actorの配列を初期化する.  
\*  
\* @param first 初期化する配列の先頭.  
\* @param last 初期化する配列の終端.  
\*/  
void initializeActorList(Actor\* first, Actor\* last)  
{  
 for (Actor\* i = first; i != last; ++i) {  
 i->health = 0;  
 }  
 }  
**+  
+**/\*\*  
**+**\* Actorの配列を更新する.  
**+**\*  
**+**\* @param first 更新する配列の先頭.  
**+**\* @param last 更新する配列の終端.  
**+**\* @param deltaTime 前回の更新からの経過時間.  
**+**\*/  
**+**void updateActorList(Actor\* first, Actor\* last, float deltaTime)  
**+**{  
**+**}

雛形が書けたら、関数の元になるプログラムをコピペします。  
3.1の最初にあるfor文と同じものをコピーして、上記のプログラムにペーストしてください。

/\*\*  
\* Actorの配列を更新する.  
\*  
\* @param first 更新する配列の先頭.  
\* @param last 更新する配列の終端.  
\* @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 = std::begin(enemyList); i != std::end(enemyList); ++i) {  
**+** if (i->health > 0) {  
**+** i->spr.Update(deltaTime);  
**+** if (i->spr.Tweener()->IsFinished()) {  
**+** i->health = 0;  
**+** }  
**+** }  
**+** }  
}

最後に、パラメーターの値を使うようにfor文を変更します。

void updateActorList(Actor\* first, Actor\* last, float deltaTime)  
{  
**-**  for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
**+**  for (Actor\* i = first; i != last; ++i) {  
 if (i->health > 0) {  
 i->spr.Update(deltaTime);  
 if (i->spr.Tweener()->IsFinished()) {  
 i->health = 0;  
 }  
 }  
 }  
}

Actor配列の更新関数ができました！

## 更新プログラムを関数で置き換える

完成した関数で更新プログラムを置き換えましょう。  
Actor配列の更新プログラムを、次のように変更してください。

**-**// 敵の更新.  
**-**for (Actor\* i = std::begin(enemyList); i != std::end(enemyList); ++i) {  
**-** if (i->health > 0) {  
**-** i->spr.Update(deltaTime);  
**-** if (i->spr.Tweener()->IsFinished()) {  
**-** i->health = 0;  
**-** }  
**-** }  
**-**}  
**-**  
**-**// 自機の弾の更新.  
**-**for (Actor\* i = std::begin(playerBulletList);  
**-**  i != std::end(playerBulletList); ++i) {  
**-**  if (i->health > 0) {  
**-**  i->spr.Update(deltaTime);  
**-**  if (i->spr.Tweener()->IsFinished()) {  
**-**  i->health = 0;  
**-**  }  
**-**  }  
**-**}  
**+**updateActorList(std::begin(enemyList), std::end(enemyList), deltaTime); **+**updateActorList(std::begin(playerBulletList), std::end(playerBulletList),  
**+** deltaTime);

プログラムが書けたら、ビルドして実行してください。  
この変更も、プログラムの動作は何も変更していません。ですから、変更前と同じように動いていれば関数による置き換えは成功です。

**[課題01(必須)]**  
Actor配列を描画するプログラムを関数で置き換えてください。  
なお、描画は英語で「render(れんだー)」といいます。

# 空いている構造体の検索を関数にする

## プロトタイプ宣言

今度は「空いている構造体を検索する」プログラムを関数にしてみます。  
これは、自機の弾の発射の以下の部分などに当たります。

// 空いている弾の構造体を検索.Actor\* bullet = nullptr;  
for (Actor\* i = std::begin(playerBulletList);  
 i != std::end(playerBulletList); ++i) {  
 if (i->health <= 0) {  
 bullet = i;  
 break;  
 }  
}

敵の場合も同様のプログラムが使われていますから、これらの置き換えを目指します。

まずは関数名を決めます。検索に対応する英語はいくつかあるのですが、今回は「find(ふぁいんど)」としましょう。また、今回作りたい関数の目的から「空いている構造体」というのは、より正確には「利用できる構造体」という意味だと考えられます。「利用できる」に該当する英単語は「Available(あべいらぶる)」ですから、「findAvailableActor(ふぁいんど・あべいらぶる・あくたー)」という名前にしましょう。

次にパラメーターです。これは初期化や更新と同様に配列の先頭と終端があればいいでしょう。つまり「Actor\*(あくたー・ぽいんたー)が2つ」です。

最後に関数の結果ですが、空いている構造体を見つけるのが目的なのですから、見つかった構造体のポインタを返す必要がありそうです。つまり、結果の型は「Actor\*」になります。

上記を踏まえて、プロトタイプ宣言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);  
 bool detectCollision(const Rect\* lhs, const Rect\* rhs);  
void initializeActorList(Actor\*, Actor\*);  
void updateActorList(Actor\*, Actor\*, float);  
void renderActorList(const Actor\*, const Actor\*);  
**+**Actor\* findAvailableActor(Actor\*, Actor\*);

## 関数定義

つぎは関数定義です。Actor配列を描画する関数の定義の下に、次のプログラムを追加してください。

}  
  
**+**/\*\*  
**+**\* 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;  
**+**}

雛形が書けたら、上記のプログラムに元にあるプログラムをコピペして、パラメーターを使うように調整してください。

/\*\*  
 \* 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)  
{  
**+** Actor\* bullet = nullptr;  
**+** for (Actor\* i = first; i != last; ++i) {  
**+** if (i->health <= 0) {  
**+** bullet = I; **+** break;  
**+** }  
**+** }  
}

最後に、関数が結果を返すようにします。上記のプログラムを次のように変更してください。

Actor\* findAvailableActor(Actor\* first, Actor\* last)  
{  
**-** Actor\* bullet = nullptr;  
**+** Actor\* result = nullptr;  
 for (Actor\* i = first; i != last; ++i) {  
 if (i->health <= 0) {  
**-** bullet = I; **+** result = I; break;  
 }  
 }  
**+** return result;  
}

変数の名前をより汎用的な「result(りざると)」に変更し、関数の最後にreturn(りたーん)文を追加して結果を返すようにしました。なお、「result(りざると)」というのは「結果」や「成果」という意味の英単語です。

## 検索プログラムを関数で置き換える

それでは検索プログラムを関数で置き換えましょう。  
自機の弾を発射するプログラムを、次のように変更してください。

if (gamepad.buttonDown & GamePad::A) {  
**-** // 空いている弾の構造体を検索. **-** Actor\* bullet = nullptr;  
**-** for (Actor\* i = std::begin(playerBulletList);  
**-** i != std::end(playerBulletList); ++i) {  
**-** if (i->health <= 0) {  
**-** bullet = i;  
**-** break;  
**-** }  
**-** }  
**+** Actor\* bullet = fundAvailableActor(  
**+** std::begin(playerBulletList), std::end(playerBulletList));  
// 空いている構造体が見つかったら、それを使って弾を発射する. if (bullet != nullptr) {  
 bullet->spr = Sprite("Res/Objects.png",  
 sprPlayer.Position(), Rect(64, 0, 32, 16));  
 bullet->spr.Tweener(TweenAnimation::Animate::Create(  
TweenAnimation::MoveBy::Create(1, glm::vec3(1200, 0, 0),  
TweenAnimation::EasingType::Linear)));  
 bullet->collisionShape = Rect(-8, -4, 16, 8);  
 bullet->health = 1;  
 }  
 }

プログラムが書けたら、ビルドして実行してください。ここまでの関数と同じく、この変更もプログラムの動作は何も変えてていません。ですから、変更前と同じように動いていれば、置き換えは成功です。

**[課題02(必須)]**  
空いている敵構造体の検索プログラムをfindAvailableActor関数で置き換えてください。

# 衝突判定を関数にする

## プロトタイプ宣言

置き換えの最後は、衝突判定です。  
関数名は「detectCollision(でぃてくと・こりじょん)」とします。detect(でぃてくと)は「検出する」という意味の動詞で、collision(こりじょん)は「衝突」という意味の名詞です。つまり、detectCollisionは「衝突を検出する」という意味になります。

**[補足]**  
この名前は、2つのRect型の間で衝突を検出する関数でも使われています。C++言語では、パラメーターが異なっていれば、関数に同じ名前を付けることが許されています。これは「オーバーロード(overload)」と呼ばれていて、「意味するところは同じだがその処理対象が異なる」場合などに使われます。

さて、衝突判定は2つの配列の間で判定を行います。ですから、パラメーターは2つではなく、少なくとも4つ必要です。さらに、衝突判定関数が結果を返すかどうかを考えます。衝突が起こったら衝突した構造体のポインターを返す、という方法もありそうですが、その場合、ループ中で最初に見つかった衝突だけしか分かりません。

そこで今回は、「衝突が起こったときに実行するプログラム」もパラメーターにしようと思います。このパラメーターが必要なのは、あるデータAとBが衝突したときになにが起こるかは、AとBの種類によって異なるはずだからです。

プロトタイプ宣言に、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);  
 bool detectCollision(const Rect\* lhs, const Rect\* rhs);  
void initializeActorList(Actor\*, Actor\*);  
void updateActorList(Actor\*, Actor\*, float);  
void renderActorList(const Actor\*, const Actor\*);  
Actor\* findAvailableActor(Actor\*, Actor\*);  
**+**void detectCollision(Actor\*, Actor\*, Actor\*, Actor\*,  
**+** /\* ここに、衝突が起こったときに実行するプログラムを渡したいけれど… \*/);

一応書けました…が、「衝突が起こったときに実行するプログラム」はどんなパラメーターにすればいいのでしょう。実はC++言語には、こんなときに使える機能がいくつか備わっています。今回はそのうちのひとつ「関数のポインター」を使ってみることにしましょう。

「関数のポインター」を使うためには、まず「関数の型」を決めます。これには「using(ゆーじんぐ)宣言」というものを使います。  
プロトタイプ宣言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);  
 bool detectCollision(const Rect\* lhs, const Rect\* rhs);  
void initializeActorList(Actor\*, Actor\*);  
void updateActorList(Actor\*, Actor\*);  
void renderActorList(const Actor\*, const Actor\*);  
Actor\* findAvailableActor(Actor\*, Actor\*);  
**+**using CollisionHandlerType = void (\*)(Actor\*, Actor\*); // 衝突処理関数の型.  
void detectCollision(Actor\*, Actor\*, Actor\*, Actor\*,  
 /\* ここに、衝突が起こったときに実行するプログラムを渡したいけれど… \*/);

関数型のusing宣言は次のような形式になっています。

型の名前 この型が関数であることを示す記号  
 │ │  
using CollisionHandlerType = void (\*)(Actor\*, Actor\*);  
 │ 　　　　　　　│  
 結果の型　 パラメーターのリスト

このusing宣言を読み解くと「2つのActor\*をパラメーターとし、結果を返さない関数の型に、CollisionHandlerType(こりじょん・はんどらー・たいぷ)という名前を付けてください」という命令になります。なお、Handler(はんどらー)というのは「～を扱うもの」という名詞です。そしてType(たいぷ)は「型」という意味です。ですからCollisionHandlerTypeは「衝突を扱う関数の型」という意味になります。

早速、この型を使ってdetectCollision関数のプロトタイプ宣言を書き直してみましょう。  
上記のプログラムを次のように変更してください。

using CollisionHandlerType = void (\*)(Actor\*, Actor\*); // 衝突処理関数の型.  
**-**void detectCollision(Actor\*, Actor\*, Actor\*, Actor\*,  
**-** /\* ここに、衝突が起こったときに実行するプログラムを渡したいけれど… \*/);  
**+**void detectCollision(Actor\*, Actor\*, Actor\*, Actor\*, CollisionHandlerType);

これでO.K.です！

## 関数定義

続いて関数の定義を書いていきます。  
findAvailableActor関数の定義の下に、次のプログラムを追加してください。

Actor\* findAvailableActor(Actor\* first, Actor\* last)  
{  
Actor\* result = nullptr;  
 for (Actor\* i = first; i != last; ++i) {  
 if (i->health <= 0) {  
result = I; break;  
 }  
 }  
return result;  
}  
**+  
+**/\*\*  
**+**\* 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 handler 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 handler)  
**+**{  
**+**}

雛形が書けたら、元になるプログラムをコピーし、detectCollision関数の中にペーストします。  
上記のプログラムに次のプログラムを追加してください。

void detectCollision(Actor\* firstA, Actor\* lastA,  
 Actor\* firstB, Actor\* lastB, CollisionHandlerType handler)  
{  
**+** // 自機の弾と敵の衝突判定.  
**+** for (Actor\* bullet = std::begin(playerBulletList);  
**+** bullet != std::end(playerBulletList); ++bullet) {  
**+** if (bullet->health <= 0) {  
**+** continue;  
**+** }  
**+** Rect shotRect = bullet->collisionShape;  
**+** shotRect.origin += glm::vec2(bullet->spr.Position());  
**+** for (Actor\* enemy = std::begin(enemyList);  
**+** enemy != std::end(enemyList); ++enemy) {  
**+** if (enemy->health <= 0) {  
**+** continue;  
**+** }  
**+** Rect enemyRect = enemy->collisionShape;  
**+** enemyRect.origin += glm::vec2(enemy->spr.Position());  
**+** if (detectCollision(&shotRect, &enemyRect)) {  
**+** bullet->health -= 1; //  
**+** enemy->health -= 1; // この部分は課題の進み具合によって  
**+** score += 100; //　異なっていると思います.  
**+** break; //  
**+** }  
**+** }  
**+** }  
 }

そして、パラメーターを使うように修正しましょう。  
上記のプログラムを次のように変更してください。

void detectCollision(Actor\* firstA, Actor\* lastA,  
 Actor\* firstB, Actor\* lastB, CollisionHandlerType handler)  
{  
**-** // 自機の弾と敵の衝突判定.  
**-** for (Actor\* bullet = std::begin(playerBulletList);  
**-** bullet != std::end(playerBulletList); ++bullet) {  
**-** if (bullet->health <= 0) {  
**+** for (Actor\* a = firstA; a != lastA; ++a) {  
**+** if (a->health <= 0) {  
 continue;  
 }  
**-** Rect shotRect = bullet->collisionShape;  
**-** shotRect.origin += glm::vec2(bullet->spr.Position());  
**-** for (Actor\* enemy = std::begin(enemyList);  
**-** enemy != std::end(enemyList); ++enemy) {  
**-** if (enemy->health <= 0) {  
**+** Rect rectA = a->collisionShape;  
**+** rectA.origin += glm::vec2(a->spr.Position());  
**+** for (Actor\* b = firstB; b != lastB; ++b) {  
**+** if (b->health <= 0) {  
 continue;  
 }  
**-** Rect enemyRect = enemy->collisionShape;  
**-** enemyRect.origin += glm::vec2(enemy->spr.Position());  
**-** if (detectCollision(&shotRect, &enemyRect)) {  
**-** bullet->health -= 1;  
**-** enemy->health -= 1;  
**-** score += 100;  
**-**  break;  
**+** Rect rectB = b->collisionShape;  
**+** rectB.origin += glm::vec2(b->spr.Position());  
**+** if (detectCollision(&rectA, &rectB)) {  
**+** handler(a, b);  
**+** if (a->health <= 0) { **+**  break;  
**+**  }  
 }  
 }  
 }  
 }

変数名はAとBという２つのグループによるループであることが分かるような名前に変更しています。そして、detectCollision関数がtrueを返した場合はhandlerを関数として実行するようにしました。パラメーターには、衝突を起こしたaおよびbを設定しています。

それから、breakの条件を追加してみました。aはBグループの複数の構造体と衝突している可能性があるので、単純にbreakするのはまずいと考えたからです。しかし、aのhealthが0以下になったのなら、aについてはこれ以上衝突が起こることはありませんからbreakするべきでしょう。

これで衝突判定関数は完成です。

## 衝突処理関数のプロトタイプ宣言

出来上がった衝突判定関数を見直してみると、Actorのhealthを減らしたり、scoreを加算するプログラムがなくなっています。衝突判定プログラムを関数で置き換えるには、このプログラムも関数にする必要があります。まずはプロトタイプ宣言からやっていきましょう。  
プロトタイプ宣言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);  
 bool detectCollision(const Rect\* lhs, const Rect\* rhs);  
void initializeActorList(Actor\*, Actor\*);  
void updateActorList(Actor\*, Actor\*);  
void renderActorList(const Actor\*, const Actor\*);  
Actor\* findAvailableActor(Actor\*, Actor\*);  
using CollisionHandlerType = void (\*)(Actor\*, Actor\*); // 衝突処理関数の型.  
void detectCollision(Actor\*, Actor\*, Actor\*, Actor\*, CollisionHandlerType);  
**+**void playerBulletAndEnemyContactHandler(Actor\*, Actor\*);

playerBulletAndEnemyContactHandler(ぷれいやー・ばれっと・あんど・えねみー・こんたくと・はんどらー)が、今回作成する関数の名前です。やたら長い名前ですが、要するに「プレイヤーの弾と敵の接触を扱う関数」という意味です(contactは「接触」という意味の動詞・名詞です)。

**[補足]**  
このくらい長い名前でも、一度書いてしまえば、あとは数文字入力するだけでVisual Studioが候補に挙げてくれるので、困ることは少ないと思います。候補が出たら、矢印キーで選択してEnterを押すだけですから(「英語なので読みづらい」ということはあるでしょうけど)。

それから、この関数の宣言が、CollisionHandlerTypeと同じパラメーターと結果の型を持っている点に注目してください。この関数はdetectCollision関数のパラメーターとして使いたいので、型を揃えておかなくてはならないのです。

## 衝突処理関数の定義

続いて関数の定義を書きます。  
detectCollision関数の定義の下に、次のプログラムを追加してください。

if (detectCollision(&rectA, &rectB)) {  
 handler(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)  
**+**{  
**+** bullet->health -= 1;  
**+** enemy->health -= 1;  
**+** score += 100;  
**+**}

これで定義は完成です。

## 衝突判定プログラムを関数で置き換える

ようやく衝突判定プログラムを置き換えることができます。  
update関数の中の衝突判定プログラムを、次のように変更してください。

updateActorList(std::begin(enemyList), std::end(enemyList)); updateActorList(std::begin(playerBulletList), std::end(playerBulletList));  
  
// 自機の弾と敵の衝突判定.  
**-**for (Actor\* bullet = std::begin(playerBulletList);  
**-** bullet != std::end(playerBulletList); ++bullet) {  
**-** if (bullet->health <= 0) {  
**-** continue;  
**-** }  
**-** Rect shotRect = bullet->collisionShape;  
**-** shotRect.origin += glm::vec2(bullet->spr.Position());  
**-** for (Actor\* enemy = std::begin(enemyList);  
**-** enemy != std::end(enemyList); ++enemy) {  
**-** if (enemy->health <= 0) {  
**-** continue;  
**-** }  
**-** Rect enemyRect = enemy->collisionShape;  
**-** enemyRect.origin += glm::vec2(enemy->spr.Position());  
**-** if (detectCollision(&shotRect, &enemyRect)) {  
**-** bullet->health -= 1;  
**-** enemy->health -= 1;  
**-** break;  
**-** }  
**-** }  
**-**}  
**+**detectCollision(  
**+** std::begin(playerBulletList), std::end(playerBulletList)),  
**+** std::begin(enemyList), std::end(enemyList)),  
**+** playerBulletAndEnemyContactHandler);

プログラムが書けたら、ビルドして実行してください。  
ここまでの関数と同じく、この変更もプログラムの動作は何も変えてていません。ですから、変更前と同じように動いていれば、関数による置き換えは成功です。

# 爆発を表示する

## 爆発用のActor配列を追加する

作成した関数を使って、敵を破壊したときに爆発を発生させてみましょう。  
まずは爆発用のActor配列を追加します。

/\*  
 \* ゲームの表示に関する変数.  
 \*/  
 SpriteRenderer renderer; // スプライト描画用変数.  
 FontRenderer fontRenderer; // フォント描画用変数.  
 Sprite sprBackground; // 背景用スプライト.  
 Actor sprPlayer; // 自機用スプライト.  
 glm::vec3 playerVelocity; // 自機の移動速度.  
 Actor enemyList[128]; // 敵のリスト.  
 Actor playerBulletList[128]; // 自機の弾のリスト.  
**+**Actor effectList[128]; // 爆発などの特殊効果用スプライトのリスト.

爆発だけのための配列、というのももったいないので、エフェクト全般のための配列ということにして、名前をeffectList(えふぇくと・りすと)としてみました。

次にeffectListを初期化します。  
自機の弾を初期化するプログラムの下に、次のプログラムを追加してください。

enemyGenerationTimer = 2;  
 initializeActorList(std::begin(enemyList), std::end(enemyList));  
 initializeActorList(std::begin(playerBulletList), std::end(playerBulletList));  
**+**initializeActorList(std::begin(effectList), std::end(effectList));

更新処理も追加します。  
自機の弾を更新するプログラムの下に、次のプログラムを追加してください。

updateActorList(std::begin(enemyList), std::end(enemyList), deltaTime);  
 updateActorList(std::begin(playerBulletList), std::end(playerBulletList),  
 deltaTime);  
**+**updateActorList(std::begin(effectList), std::end(effectList), deltaTime);

**[課題03(必須)]**  
effectListの描画処理を追加してください。

## 爆発アニメーションを追加する

爆発はフレーム・アニメーションで表現しようと思います。  
フレーム・アニメーションを表示するにはタイムラインを作る必要があるのでしたね。敵のアニメーション用のタイムライン変数の下に、次のプログラムを追加してください。

// 敵のアニメーション.  
 const FrameAnimation::KeyFrame enemyKeyFrames[] = {  
 { 0.000f, glm::vec2(480, 0), glm::vec2(32, 32) },  
 { 0.125f, glm::vec2(480, 96), glm::vec2(32, 32) },  
 { 0.250f, glm::vec2(480, 64), glm::vec2(32, 32) },  
 { 0.375f, glm::vec2(480, 32), glm::vec2(32, 32) },  
 { 0.500f, glm::vec2(480, 0), glm::vec2(32, 32) },  
 };  
 FrameAnimation::TimelinePtr tlEnemy;  
**+**  
**+**// 爆発アニメーション.  
**+**const FrameAnimation::KeyFrame blastKeyFrames[] = {  
**+** { 0 / 60.0f, glm::vec2(416, 0), glm::vec2(32, 32) },  
**+** { 5 / 60.0f, glm::vec2(416, 32), glm::vec2(32, 32) },  
**+** { 10 / 60.0f, glm::vec2(416, 64), glm::vec2(32, 32) },  
**+** { 15 / 60.0f, glm::vec2(416, 96), glm::vec2(32, 32) },  
**+** { 20 / 60.0f, glm::vec2(416, 96), glm::vec2(32, 32) },  
**+**};  
**+**FrameAnimation::TimelinePtr tlBlast;  
  
 // 敵の出現を制御するためのデータ.  
 TiledMap enemyMap;  
 float mapCurrentPosX;  
 float mapProcessedX;

続いてタイムラインを作成します。  
敵のタイムラインを作成するプログラムの下に、次のプログラムを追加してください。

random.seed(std::random\_device()()); // 乱数エンジンの初期化.  
   
tlEnemy = FrameAnimation::Timeline::Create(enemyKeyFrames);  
**+**tlBlast = FrameAnimation::Timeline::Create(blastKeyFrames);  
  
 // 使用する画像を用意.  
 sprBackground = Sprite("Res/UnknownPlanet.png");  
 sprPlayer = Sprite("Res/Objects.png", glm::vec3(0, 0, 0), Rect(0, 0, 64, 32));

## 爆発を発生させる

準備は整いました。いよいよ敵を爆発させるときです。  
playerBulletAndEnemyContactHandler関数に、次のプログラムを追加してください。

void playerBulletAndEnemyContactHandler(Actor \* bullet, Actor \* enemy)  
 {  
 bullet->health -= 1;  
 enemy->health -= 1;  
**-** score += 100;  
**+** if (enemy->health <= 0) {  
**+** score += 100;  
**+** Actor\* blast = findAvailableActor(  
**+** std::begin(effectList), std::end(effectList));  
**+** if (blast != nullptr) {  
**+** blast->spr = Sprite("Res/Objects.png", enemy->spr.Position());  
**+** blast->spr.Animator(FrameAnimation::Animate::Create(tlBlast));  
**+** namespace TA = TweenAnimation;  
**+** blast->spr.Tweener(TA::Animate::Create(  
**+** TA::Rotation::Create(20 / 60.0f, 1.5f)));  
**+** blast->health = 1;  
**+** }  
**+** }  
 }

プログラムが書けたら、ビルドして実行してください。  
敵を破壊したときに爆発が表示されたら成功です。

# 自機と敵の衝突

## 自機の型を変更する

自機に衝突判定を付けるには、自機の型をActorに変更する必要があります。  
この変更を以下に示します。

Sprite sprBackground; // 背景用スプライト.  
**-**Sprite sprPlayer; // 自機用スプライト.  
**+**Actor sprPlayer; // 自機用スプライト.  
 glm::vec3 playerVelocity; // 自機の移動速度.

## 自機の初期化を変更する

Actor型になったことで、衝突判定用の範囲と耐久力を設定できるようになりました。  
自機の画像を設定するプログラムを、次のように変更してください。

// 使用する画像を用意.  
 sprBackground = Sprite("Res/UnknownPlanet.png");  
**-** sprPlayer =  
**+** 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;

## プレイヤーの入力を処理するプログラムの変更

自機が破壊されているのに移動したり弾を発射できるのは不自然ですよね。  
そこで、自機のhealthが0以下の場合は、入力を受け付けないようにします。  
processInput関数を次のように変更してください。

/\*\*  
 \* プレイヤーの入力を処理する.  
 \*  
 \* @param window ゲームを管理するウィンドウ.  
 \*/  
 void processInput(GLFWEW::WindowRef window)  
 {  
 window.Update();  
**+** if (sprPlayer.health <= 0) {  
**+** playerVelocity = glm::vec3(0, 0, 0);  
**+** } else {  
 // 自機の速度を設定する.  
 const GamePad gamepad = window.GetGamePad();  
 if (gamepad.buttons & GamePad::DPAD\_UP) {  
 playerVelocity.y = 1;  
 } else if (gamepad.buttons & GamePad::DPAD\_DOWN) {  
 playerVelocity.y = -1;  
 } else {  
 playerVelocity.y = 0;  
 }  
 if (gamepad.buttons & GamePad::DPAD\_RIGHT) {  
 playerVelocity.x = 1;  
 } else if (gamepad.buttons & GamePad::DPAD\_LEFT) {  
 playerVelocity.x = -1;  
 } else {  
 playerVelocity.x = 0;  
 }  
 if (playerVelocity.x || playerVelocity.y) {  
 playerVelocity = glm::normalize(playerVelocity) \* 400.0f;  
 }  
 // 弾の発射.  
 if (gamepad.buttonDown & GamePad::A) {  
 Actor\* bullet = findAvailableActor(  
 std::begin(playerBulletList), std::end(playerBulletList));  
 if (bullet != nullptr) {  
**-** bullet->spr = Sprite("Res/Objects.png",  
**-** sprPlayer.Position(), Rect(64, 0, 32, 16));  
**+** bullet->spr = Sprite("Res/Objects.png",  
**+** sprPlayer.spr.Position(), Rect(64, 0, 32, 16));  
 bullet->spr.Tweener(TweenAnimation::Animate::Create(  
 TweenAnimation::MoveBy::Create(1, glm::vec3(1200, 0, 0))));  
 bullet->collisionShape = Rect(-16, -8, 32, 16);  
 bullet->health = 1;  
 }  
 }  
**+** }  
 }  
  
 /\*\*  
 \* ゲームの状態を更新する.  
 \*  
 \* @param window ゲームを管理するウィンドウ.  
 \*/

## 自機の移動処理の変更

自機のhealthが0以下の場合は、update関数の移動処理も無効化します。  
update関数を次のように変更してください。

/\*\*  
 \* ゲームの状態を更新する.  
 \*  
 \* @param window ゲームを管理するウィンドウ.  
 \*/  
 void update(GLFWEW::WindowRef window)  
 {  
 const float deltaTime = window.DeltaTime();  
 // 自機の移動.  
**+** if (sprPlayer.health > 0) {  
 if (playerVelocity.x || playerVelocity.y) {  
**-** glm::vec3 newPos = sprPlayer.Position() + playerVelocity \* deltaTime;  
**-** const Rect playerRect = sprPlayer.Rectangle();  
**+** glm::vec3 newPos =  
**+** sprPlayer.spr.Position() + playerVelocity \* deltaTime;  
**+** const Rect playerRect = sprPlayer.spr.Rectangle();  
 if (newPos.x < -0.5f \* (windowWidth - playerRect.size.x)) {  
 newPos.x = -0.5f \* (windowWidth - playerRect.size.x);  
 } else if (newPos.x > 0.5f \* (windowWidth - playerRect.size.x)) {  
 newPos.x = 0.5f \* (windowWidth - playerRect.size.x);  
 }  
 if (newPos.y < -0.5f \* (windowHeight - playerRect.size.y)) {  
 newPos.y = -0.5f \* (windowHeight - playerRect.size.y);  
 } else if (newPos.y > 0.5f \* (windowHeight - playerRect.size.y)) {  
 newPos.y = 0.5f \* (windowHeight - playerRect.size.y);  
 }  
**-** sprPlayer.Position(newPos);  
**+** sprPlayer.spr.Position(newPos);  
 }  
**-** sprPlayer.Update(deltaTime);  
**+** sprPlayer.spr.Update(deltaTime);  
**+** }  
 // 敵の出現.

## 自機のhealthが0以下の場合は表示しない

破壊されているのに何事もなかったかのように表示され続けるのはおかしいので、自機のhealthが0以下の場合は表示しないようにしましょう。  
render関数を次のように変更してください。

renderer.BeginUpdate();  
 renderer.AddVertices(sprBackground);  
**-**renderer.AddVertices(sprPlayer);  
**+**if (sprPlayer.health > 0) {  
**+** renderer.AddVertices(sprPlayer.spr);  
**+**}  
 renderActorList(std::begin(enemyList), std::end(enemyList));  
 renderActorList(std::begin(playerBulletList), std::end(playerBulletList));  
 renderActorList(std::begin(effectList), std::end(effectList));  
 renderer.EndUpdate();  
 renderer.Draw({ windowWidth, windowHeight });

自機をアニメーションさせる課題を完了している場合は、その部分のsprPlayerもsprPlayer.sprに置き換えてください。

すべての変更が終わったらプログラムをビルドし、問題なく実行できることを確認してください。

**[課題04(必須)]**  
自機と敵の衝突判定を作成してください。  
なお、自機の変数sprPlayerは配列ではないのでstd::begin関数や std::end関数は使えません。そのかわり、「&sprPlayer」と書くことで、配列の先頭と同じポインターが得られます。また、「&sprPlayer + 1」と書くことで、配列の終端と同じポインターが得られます。