敵の生成

敵の実装のためにこれまで同様、ENEMYクラスを作成する
また、敵の数はランダムに生成されるようにする

ENEMYクラスもPLAYERクラスを継承する。

ENEMYクラス

```
class ENEMY : public PLAYER {
private:
	SDV accele; //加速度

public:
	//コンストラクタ
	ENEMY() {
		point.Set(0, 0);
		vellocity.Set(0, 0);
		length.Set(16, 40);
		accele.Set(0, 0);
		angle = 0;
	}
```

↓続く

・ENEMYクラス②

[basic.h]

・・・

```
#define ENEMY_X_1 600          //敵のX座標①
#define ENEMY_X_2 540          //敵のX座標②
#define ENEMY_Y_UP 80          //敵のY座標上限
#define ENEMY_Y_DOWN 360       //敵のY座標下限
```

ENEMYクラス③

[class_def.h]

・・・

```
//コンストラクタ２
ENEMY(int n, int total) {
        //nが偶数の時は手前側に、奇数の時は後側に配置する
        point.X = n % 2 == 0 ? ENEMY_X_1 : ENEMY_X_2;
        point.Y = ((double)n / total) * (ENEMY_Y_DOWN -
        ENEMY_Y_UP) + ENEMY_Y_UP;
        vellocity.Set(2, 2);
        length.Set(16, 40);
        accele.Set(0, 0);
        angle = 0;
}
```

↓続く

・補足

[三項演算子]

条件式？真の時になる値 : 偽の時になる値;

[例]

X = a < b？3 : 5;

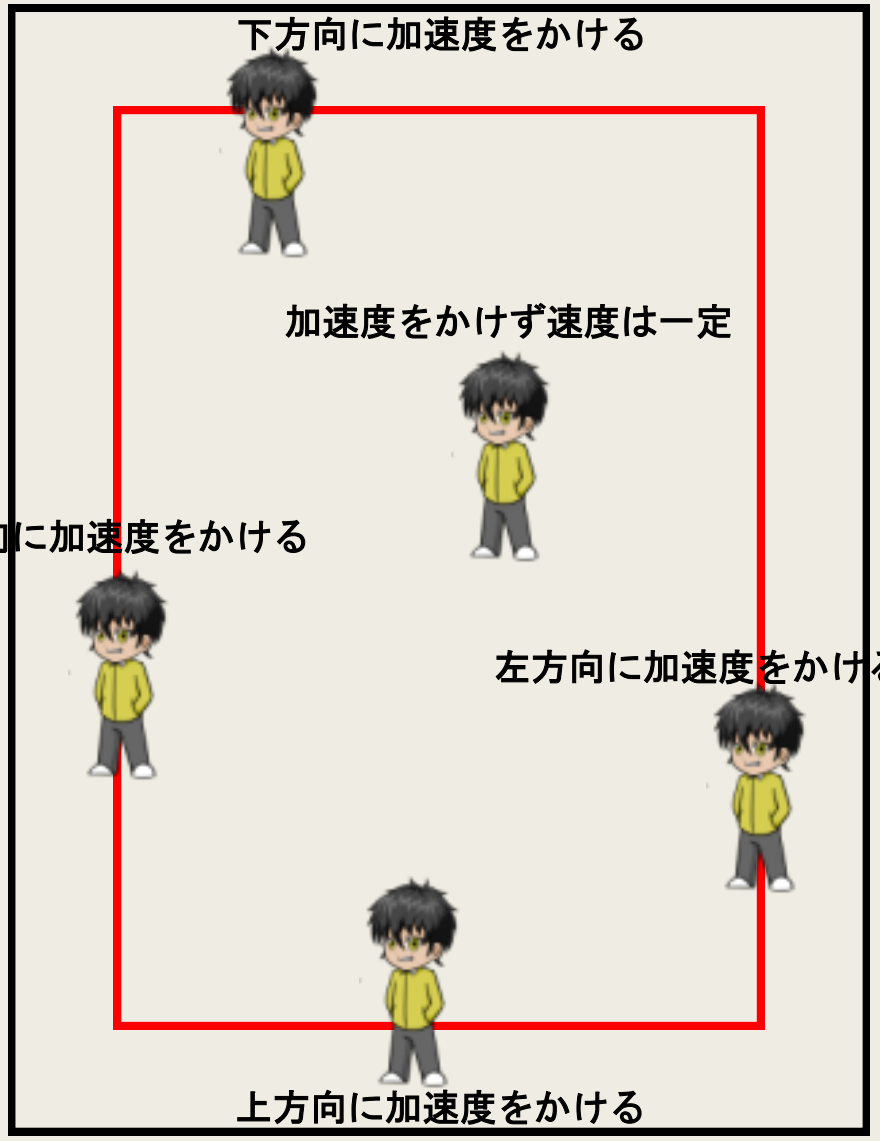
a = 1, b = 3のとき、条件式が真となるためXに3が代入される。
a = 8, b = 4のとき、条件式が偽となるためXに5が代入される。

if else文を一行の演算子にまとめてしまったようなもの

・敵の動き

・・・移動範囲のライン

・・・加速度をかけるライン



赤ラインを超えた場合、移動範囲限界が近づくため、反対方向へ加速度をかける。
これにより、擬似的に自動移動が表現できる。

ENEMYクラス③

```
void Move() {
        /*範囲外が近づいたら動いてる方向と反対方向に加速度をかける処理*/
        if (point.X < 5 * WINDOW_WIDTH / 8)
                accele.X = 0.1;
        else if (point.X > 7 * WINDOW_WIDTH / 8)
                accele.X = -0.1;
        else
                accele.X = 0;

        if (point.Y < GAME_UNDER / 5)
                accele.Y = 0.1;
        else if (point.Y > 7 * GAME_UNDER / 8)
                accele.Y = -0.1;
        else
                accele.Y = 0;

        vellocity += accele;
        point += vellocity;
```

↓続く

・ENEMYクラス④

```
		//一応移動範囲外の処理
		if (point.Y - length.Y < 0)
			point.Y = length.Y;
		if (point.Y + length.Y > WINDOW_HEIGHT)
			point.Y = WINDOW_HEIGHT - length.Y;
		if (point.X + length.X > WINDOW_WIDTH)
			point.X = WINDOW_WIDTH - length.X;
		if (point.X - length.X < WINDOW_WIDTH / 2)
			point.X = WINDOW_WIDTH / 2 + length.X;
	}

	void Draw(int graph) {
		int sizeX, sizeY;
		const int line = 55;
		GetGraphSize(graph, &sizeX, &sizeY);
		DrawRectRotaGraphF(point.X, point.Y, line, 0, sizeX - line,
		sizeY, 1.0, angle, graph, TRUE, FALSE);
	}
};
```

・vector

敵の数はランダムで決めるようにしたので、敵用のオブジェク
変数の数は出現によって変わる。
そのためENEMYクラスのオブジェクト変数にはvectorを使う。

vector・・・動的配列クラス。

[vector宣言テンプレ]

vector <型名>　vector変数名;

[basic.h]

・・・

#include <vector>←.hはつけないの注意

using namespace std;

・・・

[main.cpp]

/**変数宣言**/

・・・

//オブジェクト

・・・

vector <ENEMY> enemy;

・vectorの要素の挿入と参照

vectorのメンバ関数「push_back」を使う

vector変数名.push_back(挿入したいデータが格納された変数or値);

[例(int型のvector)]

```
vector<int> test;
int T = 8;
//挿入
test.push_back(3);   ←１番目に3を挿入する。
test.push_back(T);   ←２番目にTの値(8)を挿入する。
test.push_back(T + 3);   ←３番目にT + 3(11)を挿入する。

//参照（通常の配列と同じ）
test[0]   →   3
test[1]   →   8
test[2]   →   11
```

・関数定義ヘッダーファイル

プロジェクト > 新しい項目の追加
↓
ヘッダーファイルを選択し、「function_def.h」と名前を付 けて追加ボタン

[function_def.h]

```
#include “basic.h”
#ifndef _FUNCTION_H_
#define _FUNCTION_H_
void EnemySet(vector<ENEMY> &enemy, int num) {
        //全要素を削除してリセット
        enemy.clear();
        for (int i = 0; i < num; i++) {
                //挿入用のオブジェクト変数生成
                ENEMY *data = new ENEMY(i, num - 1);
                //データの挿入
                enemy.push_back(*data);
                //オブジェクトの破棄
                delete data;
        }
}
#endif
```

・補足

[参照型変数]

```
void EnemySet(vector<ENEMY> &enemy, int num)
                                    ↑これ
```

別の変数と領域を共有する変数。初期化時に別の変数のアドレスと値をコピーする。

[例]

```
int a = 0;          //通常の変数aを宣言
int &b = a;         //参照型変数bを宣言しaで初期化

a = 3;              //この時bの値も3になる
b = 5;              //この時aの値も5になる
```

このように、片方の値を変えると、もう片方の値も変化するため関数の引数に使うことで、ローカル変数を関数外で操作できるようになる。

・補足②

[delete演算子]

newで生成したオブジェクトを消去する演算子。

```
for (int i = 0; i < num; i++) {
        ENEMY *data = new ENEMY(i, num - 1);   ←　生成
        enemy.push_back(*data);
}   ←　変数的には消滅するが、メモリ上にはデータが残っている
```

deleteを使用しないと、使わないデータがメモリ上に蓄えられ続け、
所謂処理落ちが発生する可能性がある。
そのため、使用しなくなった変数は宣言した{}の領域を抜け出す前に
deleteする必要がある。

[clear()]

vectorの全要素を削除するメンバ関数。

[size()]

vectorの要素数を取得するメンバ関数。

乱数関数の作成

DxLibに乱数を発生させる関数としてGetRand()があるが、最小値が0で固定されており、汎用性に若干欠ける。
そのため、引数で範囲を指定し、その間で乱数を発生させる関数を作成する。

[function_def.h]

```
//第一引数が最小値、第二引数が最大値
int IntRand(int MIN,int MAX){
        int temp;
        //最小値が最大値より大きかった場合入れ替える
        if(MIN > MAX){
                temp = MIN;
                MIN = MAX;
                MAX = temp;
        }
        return GetRand(MAX - MIN) + MIN;
}
```

・敵の実装

[main.cpp]

```
/**変数宣言**/
 ・・・
//オブジェクト
 ・・・
Vector<ENEMY> enemy;
EnemySet(enemy,IntRand(2,5));
 ・・・
//グラフィックハンドル
 ・・・
int enemy_graph = LoadGraph(“DATA/graph/enemy.png”);
 ・・・
```

・敵の実装②

・・・

```
/*Moveを呼び出して、動作の処理*/
player->Move();
for (int i = 0; i < NPC_NUM; i++)
        npc[i]->Move(player->getPoint());
for (int i = 0; i < enemy.size(); i++)
        enemy[i].Move();

/*Drawを呼び出して、描画の処理*/
player->Draw(player_graph);
for (int i = 0; i < NPC_NUM; i++)
        npc[i]->Draw(npc_graph);
for (int i = 0; i < enemy.size(); i++)
        enemy[i].Draw(enemy_graph);
```

・・・

・プロトタイプ宣言

basic.h
↓
class_def.h
↓
function_def.h
の順でincludeするので、basic.hにプロトタイプ宣言を記述し、function_def.hに定義を記述する。

[basic.h]

```
/**クラスのプロトタイプ宣言**/
class PLAYER;
class NPC;
class ENEMY;

/**関数プロトタイプ宣言**/
bool InitSet();
int IntRand(int, int);
void EnemySet(vector<ENEMY>&, int);
```

・SHOTクラス

プレイヤー、NPC、敵が全て使用できるSHOTクラスを作成する。

[basic.h]

```
/**クラスのプロトタイプ宣言**/
 ・・・
class SHOT;
```

[class_def.h]

```
class SHOT {
private:
        SDV point;              //座標
        SDV vellocity;          //速度
        SDV length;             //長さ
        int velNorlm;           //速度ベクトルの大きさ
        double angle;           //角度
        int strong;             //ダメ―ジ
```

・SHOTクラス②

```
public:
//コンストラクタ
SHOT() {
        point.Set(0, 0);
        vellocity.Set(0, 0);
        length.Set(0, 0);
        velNorIm = 0;
        angle = 0;
        strong = 1;
}
//コンストラクタ2
SHOT(SDV p, int S, double A = 0) {
        point = p;
        velNorIm = 8;
        angle = A;
        vellocity.Set(velNorIm * cos(angle), velNorIm * sin(angle));
        length.Set(8, 8);
        strong = S;
}
```

・補足

[デフォルト引数]

SHOT(SDV p, int S, double A = 0)

↑これ

引数を初期化することによって、関数呼び出し時に引数に値を設置しなかった場合、自動的に引数に代入されるシステム。

[例]

```
//定義
void function(int X,int Y,int Z = 2){
 ・・・
}

//呼び出し
function(2,6,3);    ←Zに3が代入される
function(8,2);    ←引数が指定されていないのでZに2が代入される
```

・SHOTクラス②

```
        bool Move() {
                bool out = false;
                point += vellocity;
                if (point.X - length.X > WINDOW_WIDTH)
                        out = true;
                if (point.X + length.X < 0)
                        out = true;
                if (point.Y - length.Y > WINDOW_HEIGHT)
                        out = true;
                if (point.Y + length.Y < 0)
                        out = true;
                return out;
        }

        void Draw() {
                DrawCircleAA(point.X, point.Y, length.X, 32, WHITE);
        }
};
```

vectorの要素の削除

画面外に行った弾はもう必要ないので、vectorの要素から削除する必要がある。要素を削除する関数として、clearがあったが、全要素が削除されてしまうので特定の要素を削除するerase関数を使う。

vector変数名.erase(削除したい要素のイテレーター);

[イテレーター]

vectorの要素を指すポインタ。

[宣言]

vector<型名>::iterator 変数名;

[begin()]

vectorの最初の要素のイテレータを返すメンバ関数

[end()]

vectorの最後の要素の一つ先のイテレータを返すメンバ関数

[erase()]

要素の削除を行うメンバ関数。削除した要素の次の要素のイテレーターを返す

SHOTの実装(player)

[class_def.h]

```
class PLAYER {
protected:
・・・
        int shot_rug;                   //弾を撃ったらある程度間隔を持たせる

        void MoveShot() {
                //イテレーターを最初の要素で初期化して宣言
                vector<SHOT>::iterator iter = shot.begin();
                //最後の要素になるまでループ
                while (iter != shot.end()) {
                        //弾を動かし画面外にあるかチェック
                        if (!iter->Move())
                                iter++;
                        else
                                iter = shot.erase(iter);
                                //画面外にいれば削除
                }
        }
```

・SHOTの実装(player)

```
public:
        vector<SHOT> shot;

        PLAYER(){
         ・ ・ ・
                shot_rug = 0;
        }

        void Move(){
         ・ ・ ・
                point += vellocity;
                MoveShot();
         ・ ・ ・
        }
```

・SHOTの実装(player)

```
void CreateShot(){
        //shot_rugが0以下かつZキーが押されていれば弾の生成
        if (--shot_rug <= 0 && CheckHitKey(KEY_INPUT_Z) == 1) {
                SHOT *data = new SHOT(point, 4);
                shot.push_back(*data);
                shot_rug = 30;
                delete data;
        }
}

void Draw(int graph){
        ・・・
        //弾の描画
        for (int i = 0; i < shot.size(); i++)
                shot[i].Draw();
}
```

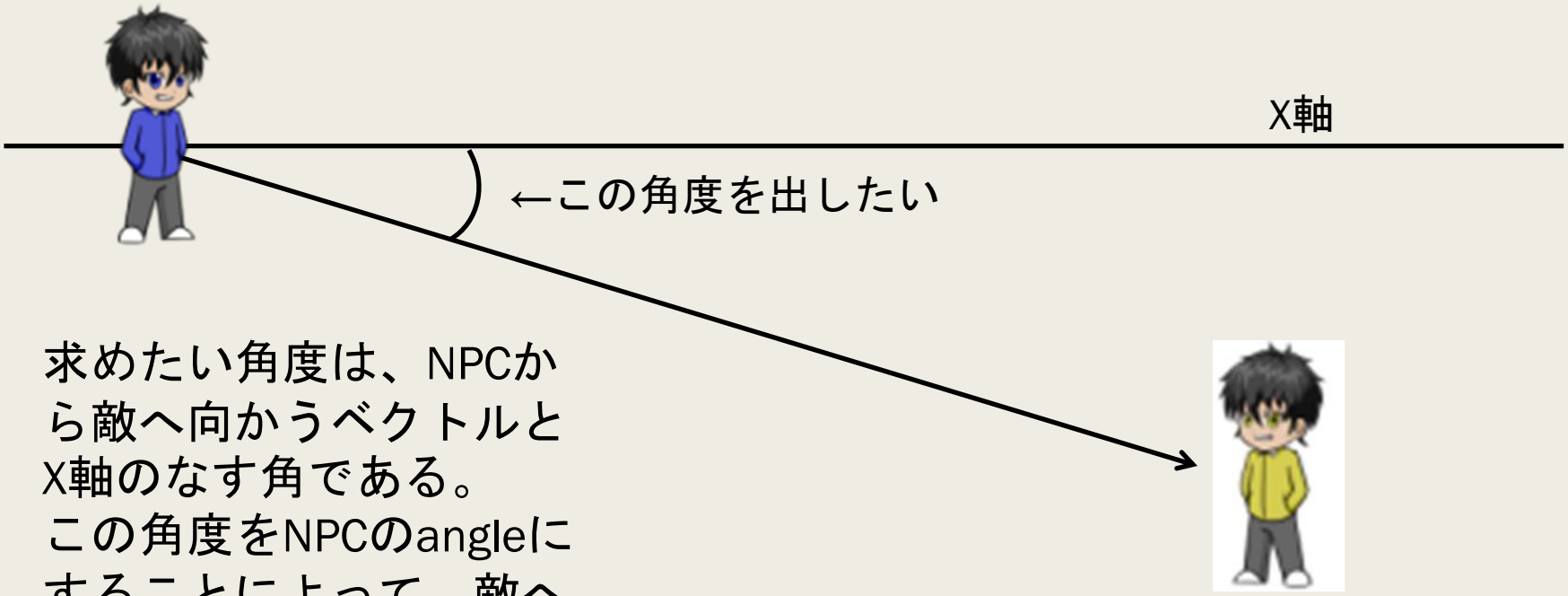
SHOTの実装(NPC)

NPCはY方向には動かない代わりに、敵を狙い、その方向に弾が放てるような仕様とする。

狙う敵は、NPCから最も距離が近い敵とする。

・敵を狙い撃つ

vellocity.X = velNorIm * cos(angle);
vellocity.Y = velNorIm * sin(angle);
としたため、弾がどの方向へ飛ぶかは角度に依存する。
つまりNPCから敵へ向かうような角度を求めれば、敵を狙う弾が実装できる。



求めたい角度は、NPCから敵へ向かうベクトルとX軸のなす角である。
この角度をNPCのangleにすることによって、敵へ向かう弾が実装できる。

・SHOTの実装(NPC)

[class_def.h]

```
Class NPC : public PLAYER{
        NPC(){
         ・ ・ ・
                shot_rug = 0;
        }

        NPC(int n){
         ・ ・ ・
                shot_rug = 0;
        }

        void Move(){
         ・ ・ ・
                point += vellocity;
                MoveShot();
         ・ ・ ・
        }
```

・SHOTの実装(NPC)

```
void CreateShot(vector<SDV> enemy_point){
        if (--shot_rug <= 0) {
                SHOT *data;
                double A, min;
                int min_index = 0;
                //最短距離を探索
                min = Norlm(enemy_point[0] - point);
                for (int i = 1; i < enemy_point.size(); i++) {
                        if (min > Norlm(enemy_point[i] - point)) {
                                min = Norlm(enemy_point[i] - point);
                                min_index = i;
                        }
                }
```

・SHOTの実装(NPC)

```
                //X軸と敵へ向かうベクトルのなす角を求める
                A = BetAngle(X_AXIS, enemy_point[min_index] - point);
                //敵がNPCよりも上にいるときの処理
                if (enemy_point[min_index].Y < point.Y)
                        A = -A;

                data = new SHOT(point, 4, A);
                shot.push_back(*data);
                shot_rug = 60;
                delete data;
        }
}
```

・補足

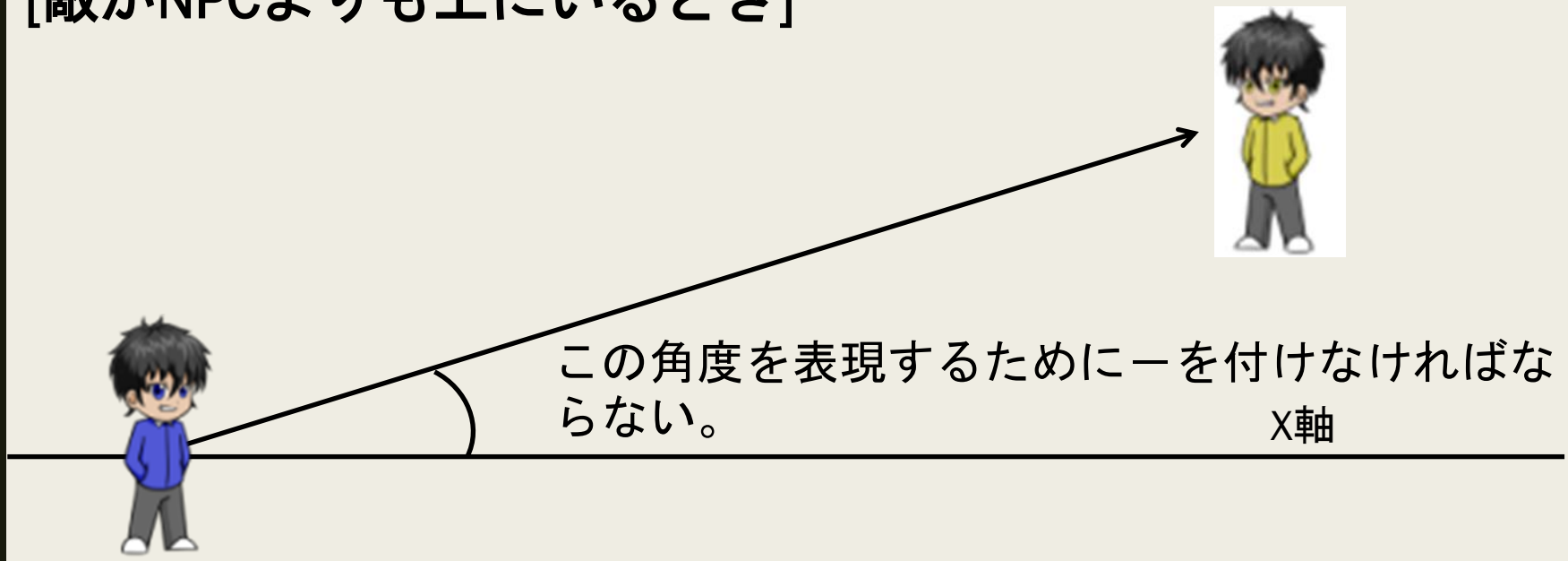
[X_AXIS]
SecondVector.hに入っている定数。値は(1,0)

[BetAngle(SDV,SDV)]
SecondVector.hに入っている関数。二つのベクトルのなす角を求める。

[NorIm(SDV)]
SecondVector.hに入っている関数。ベクトルの大きさを求める。

・補足

[敵がNPCよりも上にいるとき]



・SHOTの実装(NPC)

```
void Draw(int graph){
        ・・・
        //弾の描画
        for (int i = 0; i < shot.size(); i++)
                shot[i].Draw();
}
```

[basic.h]

```
/**関数のプロトタイプ宣言**/
 ・・・
//敵の座標を求める
vector<SDV> getEnemiesPoint(vector<ENEMY>);
```

SHOTの実装(NPC)

[function_def.h]

```
vector<SDV> getEnemiesPoint(vector<ENEMY> enemy){
        vector<SDV> points;
        for (int i = 0; i < enemy.size(); i++)
                points.push_back(enemy[i].getPoint());
        return points;
}
```

[main.cpp]

```
/*MoveとCreateShotを呼び出して、動作の処理*/
player->Move();
player->CreateShot();

for (int i = 0; i < NPC_NUM; i++) {
        npc[i]->Move(player->getPoint());
        npc[i]->CreateShot(getEnemiesPoint(enemy));
}

for (int i = 0; i < enemy.size(); i++)
        enemy[i].Move();
```

SHOTの実装(ENEMY)

敵もNPC同様に、playerもしくはNPCの中で最も距離が近いものを狙い撃つ仕様にする。

NPCを参考にして考えてみよう

・SHOTの実装(ENEMY)

[class_def.h]

```
Class ENEMY : public PLAYER{
        ENEMY(){
         ・ ・ ・
                shot_rug = 0;
        }

        ENEMY(int n,int total){
         ・ ・ ・
                shot_rug = 0;
        }

        void Move(){
         ・ ・ ・
                point += vellocity;
                MoveShot();
         ・ ・ ・
        }
```

・SHOTの実装(ENEMY)

```
void CreateShot(SDV player_point, vector<SDV> npc_point) {
        if (--shot_rug <= 0) {
                SHOT *data;
                double A, min;
                int min_index = -1;

                min = NorIm(player_point - point);
                for (int i = 0; i < npc_point.size(); i++) {
                        if (min > NorIm(npc_point[i] - point)) {
                                min = NorIm(npc_point[i] - point);
                                min_index = i;
                        }
                }
```

・SHOTの実装(ENEMY)

```
                if (min_index == -1) {
                        A = BetAngle(X_AXIS, player_point - point);
                                if (player_point.Y < point.Y)
                                        A = -A;
                }
                else {
                        A = BetAngle(X_AXIS, npc_point[min_index] - point);
                                if (npc_point[min_index].Y < point.Y)
                                        A = -A;
                }

                data = new SHOT(point, 1, A);
                shot.push_back(*data);
                shot_rug = 60;
                delete data;
        }
}
```

・SHOTの実装(ENEMY)

```
void Draw(int graph){
        ・・・
        for (int i = 0; i < shot.size(); i++)
                shot[i].Draw();
}
```

[basic.h]

```
/**関数のプロトタイプ宣言**/
 ・・・
vector<SDV> getNpcPoint(NPC*[]);
```

[function_def.h]

```
vector<SDV> getNpcsPoint(NPC *npc[]) {
        vector<SDV> points;
        for (int i = 0; i < NPC_NUM; i++)
                points.push_back(npc[i]->getPoint());
        return points;
}
```

・SHOTの実装(ENEMY)

[main.cpp]

```
/*MoveとCreateShotを呼び出して、動作の処理*/
player->Move();
player->CreateShot();

for (int i = 0; i < NPC_NUM; i++) {
		npc[i]->Move(player->getPoint());
		npc[i]->CreateShot(getEnemiesPoint(enemy));
}

for (int i = 0; i < enemy.size(); i++){
		enemy[i].Move();
		enemy[i].CreateShot(player->getPoint(),getNpcPoint(npc));
}
```

・当たり判定

```
class PLAYER{
protected:
 ・・・
        int life;          //ライフ
public:
        PLAYER(){
         ・・・
                life = 100;
        }
```

・当たり判定

```
int HitShot(vector<SHOT> &shot) {
	vector<SHOT>::iterator iter = shot.begin();
	while (iter != shot.end()) {
		//当たっていたらダメージを与え、弾を消す。
		if (iter->point.Y - iter->length.Y <= point.Y + length.Y &&
			iter->point.Y + iter->length.Y >= point.Y - length.Y &&
			iter->point.X - iter->length.X <= point.X + length.X &&
			iter->point.X + iter->length.X >= point.X - length.X) {
			life -= iter->strong;
			iter = shot.erase(iter);
		}
		else
			iter++;
		}
	return life;
}
```

・メンバ変数にアクセス

SHOTクラスのメンバ変数はprivateに置かれているためアクセスできず、当たり判定の処理が行えない。そのため、friend classを使い、private領域に他クラスからアクセスできるようにする。

friend class クラス名;・・・指定したクラスはprivate領域にアクセスできるようになる。

```
class SHOT {
//PLAYERクラスがSHOTクラスのprivateにアクセスできるようになる
        friend class PLAYER;
 ・・・
```

・当たり判定

[main.cpp]

・・・

```
/*当たり判定*/
//敵とプレイヤー
for (int i = 0; i < enemy.size(); i++) {
        player->HitShot(enemy[i].shot);
        enemy[i].HitShot(player->shot);
}

//敵とNPC
for (int i = 0; i < NPC_NUM; i++) {
        for (int j = 0; j < enemy.size(); j++) {
                npc[i]->HitShot(enemy[j].shot);
                enemy[j].HitShot(npc[i]->shot);
        }
}
```

- 念のため

PLAYER、NPC、ENEMY　各クラスのDraw関数の最後に以下を記述

```
DrawFormatString(point.X, point.Y, WHITE, "%d", life);
```