# DXライブラリを用いてブロック崩しを作る

## Part 4
## クラスを使ったブロック、ボールの制御

# 今回のスタートソース

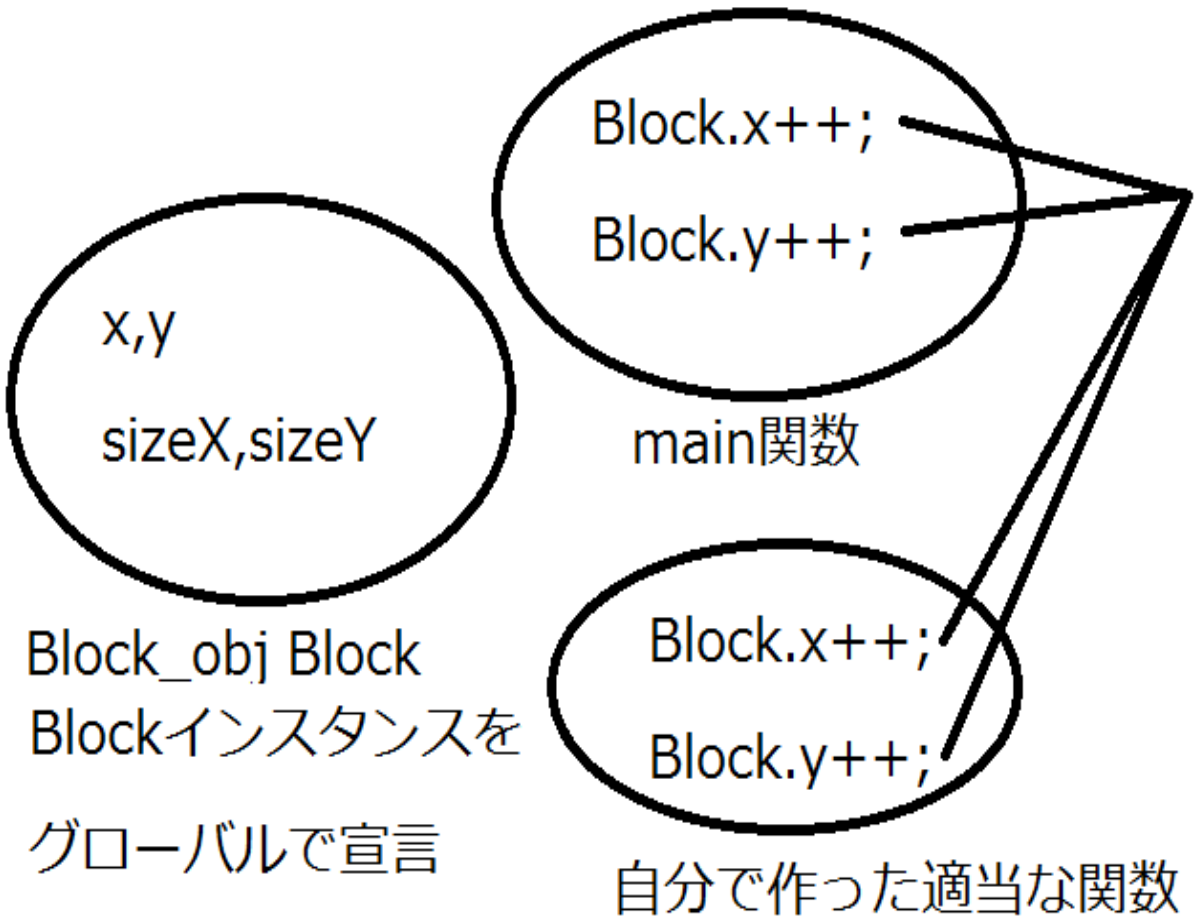
```
#include "DxLib.h"

//x方向のブロックの数
#define BLOCK_X_NUM 10
//y方向のブロックの数
#define BLOCK_Y_NUM 5

// プログラムは WinMain から始まります
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow )
{
    ChangeWindowMode(TRUE);//ウィンドウモードで起動
    DxLib_Init();   // ＤＸライブラリ初期化処理

    //裏画面に描画することを決定。
    SetDrawScreen(DX_SCREEN_BACK);

    //初期化設定はここに書く！

    while(ProcessMessage()==0){

        //描画されているものを消す。
        ClearDrawScreen();
        //メインループの動作はここに書く！！

        //裏画面の描画状態を表に反映
        ScreenFlip();
    }

    WaitKey() ;             // キー入力待ち

    DxLib_End() ;               // ＤＸライブラリ使用の終了処理

    return 0 ;              // ソフトの終了
}
```

今回も前回やったことを
ある程度消してからスタート
前回の講座を行った理由は
今回は構造体をさらに
発展させたものを使うからである

# クラスとは？

- 構造体に機能を加えたものがクラス
- 構造体の変数をいろんなところでいじくって制御していたが、クラスではクラスの中の変数はクラスのもつ関数でしか制御させない。
- このことにより、どこでこの変数が計算されているのかが推定できるため、デバックがある程度簡単になる。

# 構造体の欠点




まだ小規模なプログラムなので、グローバル化はしていないので
上の図をみてもよくわからないかもしれないが、大規模なプログラムになってくると、
関数の処理が必要不可欠→グローバル化がどうしてもしたい→バグの特定不可能
となっていまう。これを防ぎたい。

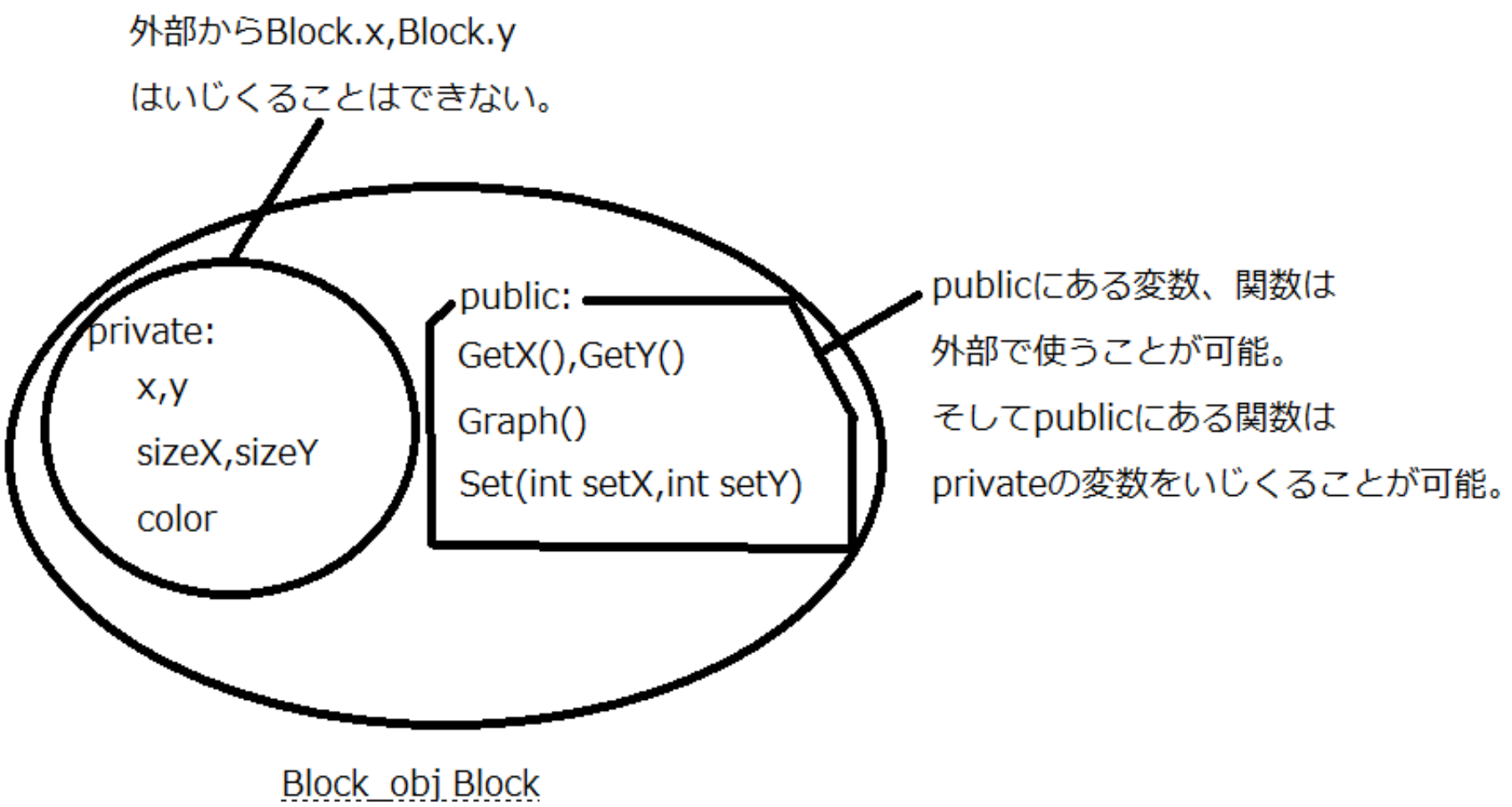
# クラスの例

```
class Block_obj{
private:
    int x,y;
    int sizeX,sizeY;
    int color;
public:
    int GetX(),GetY();
    void Set(int setX,int setY,int setColor);
    void Graph();
};
```

クラスの定義の方法は
クラス (クラス名){
private:
  (privateなパラメータ)
public:
  (publicなパラメータ)
};
とする。

ここで、privateとpublicについて、
privateにある変数、関数は外部で使われることはない。保護されている変数、関数である。
これにより、全ての場所で使われるのを防ぐ。
publicにある変数、関数は外部で使われることが可能。
なので、ここには外部で使えたら便利な変数、関数を置く場合が多い。
ここでは x,y,sizeX,sizeY,colorをprivateにし、保護している。

# クラスのイメージ




上の図にも置いたとおり、publicにある関数は、privateな変数もいじくることが可能。
つまり、これからはpublicの関数を経由して、privateな変数をいじくることになる。

# クラスの関数の使い方

```
    void Graph();
};

int Block_obj::GetX(){
    return x;
}
int Block_obj::GetY(){
    return y;
}
int Block_obj::Set(int setX,int setY,int setColor){
    x = setX;
    y = setY;
    color = setColor;
    sizeX = 40;
    sizeY = 10;
}
void Block_obj::Graph(){
    DrawBox(x,y,x+sizeX,y+sizeY,color,TRUE);
}

// プログラムは WinMain から始まります
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPr
```

クラスの関数の定義の仕方としては

（型名）（クラス名）::（関数名）

という感じである。

ここでいう型名は
int や voidのことである。

クラス名は今回は
Block_obj

関数名はそれぞれ
GetX()等である。

もちろん普通に関数の定義する時と同じように、
引数もここにしっかり明示してやる。

# 作った関数の解説、GetX(),GetY()

```
int  Block_obj::GetX(){
     return x;
}
int  Block_obj::GetY(){
     return y;
}
```

関数にて,x,yがそれぞれ返されているが、このx,yは何者かというBlock_objのx,yである。
「え？普通にx,yを知りたいならmain関数でBlock.xみたいにしてやればいいんじゃない？」
って思う人もいるだろうが、今回x,yはprivateな変数なので、外部から参照することができない。
なので、このような関数をpublicで設定してやらないと、Blockのx,yが知ることができないのである。
もちろん,BlockのsizeX,sizeYが知りたい時が来たら、同じようにGetSizeX関数のようなものも
作らないといけない。

# 作った関数の解説
# Set(int setX,int setY,int setColor)

```
void Block_obj::Set(int setX,int setY,int setColor){
    x = setX;
    y = setY;
    color = setColor;
    sizeX = 40;
    sizeY = 10;
}
```

引数として,setX,setY,setColorが指定されていて
それをそのままBlock_objのx,y,colorに代入している。
これにより、privateなx,y,colorをこの関数で設定することが可能となっている。
また今回sizeX,sizeYは全て同じのブロックをつくろうとしてるので
この関数内で同じ値を設定している。
もちろんブロックごとに違う数値にしたければ、引数にさらに
setSizeXなどを追加してやればいいだろう。

# 作った関数の解説
# Graph()

```
void Block_obj::Graph(){
    DrawBox(x,y,x+sizeX,y+sizeY,color,TRUE);
}
```

DrawBoxの引数にBlock_objのパラメータを指定してやり、
箱を描画している。
前回の構造体を使った時のDrawBoxの引数に
比べるとかなり見やすいのが分かる。
個人的にはここもかなりのポイントだと思う。

# これらの関数を実際に使ってみる。

```
// プログラムは WinMain から始まります
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE hPrevI
                    LPSTR lpCmdLine, int nCmdShow )
{
    ChangeWindowMode(TRUE);//ウィンドウモードで起動
    DxLib_Init();   // ＤＸライブラリ初期化処理

    //裏画面に描画することを決定。
    SetDrawScreen(DX_SCREEN_BACK);

    //初期化設定はここに書く！
    Block obj Block;
    //ブロックの設定
    Block.Set(50,20,GetColor(255,255,255));

    while(ProcessMessage()==0){

        //描画されているものを消す。
        ClearDrawScreen();
        //メインループの動作はここに書く！！
        Block.Graph();
        //裏画面の描画状態を表に反映
        ScreenFlip();
    }
```

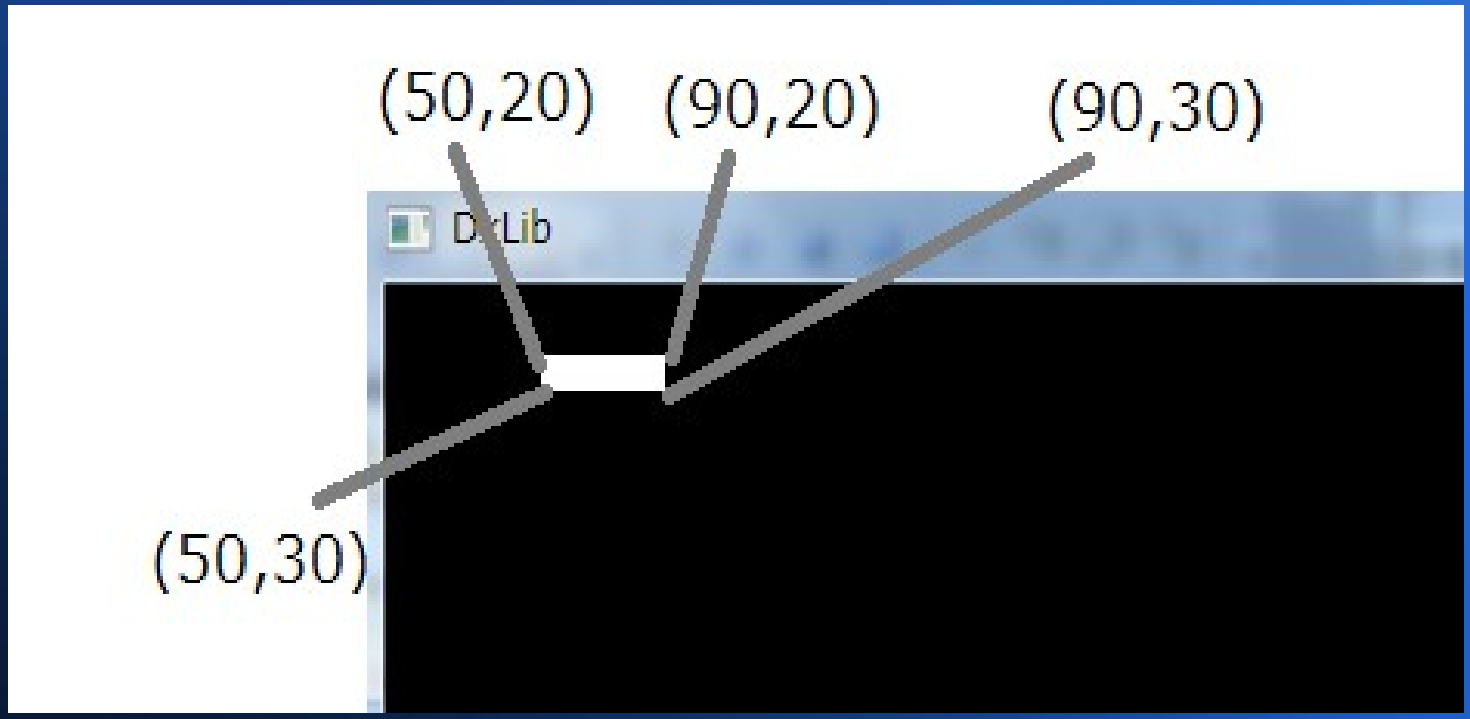
メイン関数に追加した処理は
左の図の赤線部のみ。

Block_obj Blockは
構造体の時と全く同じ
Block_objのパラメータを持った
Blockというインスタンスを生成。

ここでクラスの関数の使い方も
構造体の変数と同じように
(クラス名).(関数名)
のように用いる。
Block.Setは
1つめの引数にx,
2つめの引数にy,
3つめの引数にcolorを設定している。

そして、メインループ内で
Block.Graph()
これで、設定されたパラメータにより、
Blockが描画されている。

# 実行結果




このように、左上に白のブロックが描画されるはずである。
もちろんSet関数にてx=50,y=20,sizeX=40,sizeY=10
を指定しているので、このような座標配置になるのもわかるだろう。

# 大量にブロックを作る。

```
//初期化設定はここに書く！
Block obj Block[BLOCK_X_NUM][BLOCK_Y_NUM];
//ブロックの設定
for(int i=0;i<BLOCK_X_NUM;i++){
    for(int j=0;j<BLOCK_Y_NUM;j++){
        Block[i][j].Set(50*i,20*j,GetColor(255,255,255));
    }
}

while(ProcessMessage()==0){

    //描画されているものを消す。
    ClearDrawScreen();
    //メインループの動作はここに書く！！
    for(int i=0;i<BLOCK_X_NUM;i++){
        for(int j=0;j<BLOCK_Y_NUM;j++){
            Block[i][j].Graph();
        }
    }
```

例によって
二重配列によって
インスタンスを大量に宣言

そしてforの二重ループで
各ブロックのそれぞれの
x,yを設定。
例えばi=0,j=0の時
Block[0][0]のx,yは
x=0,y=0で設定。
I=2,j=8の時
Block[2][8]のx,yは
x=100,y=160で設定される。

さらに、メインループ内で
forの二重ループにより、
各ブロックに対して
Graph関数を使う。

# 実行結果


このように大量に描画されていることがわかる。
ここで説明するのが
Block[0][0].Graph()ではBlock[0][0]のx,y,colorを用いて描画
Block[2][8].Graph()ではBlock[2][8]のx,y,colorを用いて描画。
やはり意識して欲しいのが、全ての物体がそれぞれのパラメータを持っている。
ということなのである。

# ボールもクラスで作る。

```
void Block_obj::Graph(){
	DrawBox(x,y,x+sizeX,y+sizeY,color,TRUE);
}

class Ball_obj{
private:
	int x,y,r;
	int vx,vy;
	int color;
public:
	int GetX(),GetY(),GetR();
	int GetVX(),GetVY();
	void Set(int setX,int setY,int setR,
		int setVX,int setVY,int setColor);
	void Graph();
	void Calc();
};

// プログラムは WinMain から始まります
int WINAPI WinMain( HINSTANCE hInstance, HINSTANCE
```

さて、ボールについてもクラスで作っていく。
簡単にいじれるようにしたくないパラメータはprivateに宣言。
それをいじくる権利がある関数はpublicに宣言。
Privateのパラメータは構造体の時に使っていたのと同じもの。
PublicにあるGet関数はそれぞれのパラメータを外部で参照したいときに使用。
Set関数によって、ボールのパラメータ全てセットする予定。
Graphでボールを描画,Calc関数でボールを動かす予定である。

# ボールもクラスで作る。

```
class Ball_obj{
private:
    int x,y,r;
    int vx,vy;
    int color;
public:
    int GetX(),GetY(),GetR();
    int GetVX(),GetVY();
    void Set(int setX,int setY,int setR,
        int setVX,int setVY,int setColor);
    void Graph();
    void Calc();
};

int Ball_obj::GetX(){
    return x;
}
int Ball_obj::GetY(){
    return y;
}
int Ball_obj::GetR(){
    return r;
}
int Ball_obj::GetVX(){
    return vx;
}
int Ball_obj::GetVY(){
    return vy;
}
```

クラスの関数は、前述したとおり
(型名)(クラス名)::(関数名)
で定義に入る。

Get系の関数はただ数値を返すだけなので
特に記述することはない。

ただどういう時に使うのかというと
次回やる当たり判定の部分で
使っていくことになる。

# ボールもクラスで作る。

```
int Ball_obj::GetVY(){
    return vy;
}
void Ball_obj::Set(int setX,int setY,int setR,
    int setVX,int setVY,int setColor){

        x = setX;
        y = setY;
        r = setR;
        vx = setVX;
        vy = setVY;
        color = setColor;

}
void Ball_obj::Graph(){
    DrawCircle(x,y,r,color,TRUE);
}
void Ball_obj::Calc(){
    x+=vx;
    y+=vy;
}
```

Set関数では
x,y,r,vx,vy,color
を決定する。

そしてGraph関数では
DrawCircleによって
ボールを描画

Calc関数では
ボールを移動させている。

# ボールもクラスで作る。

```
//初期化設定はここに書く！
Block_obj Block[BLOCK_X_NUM][BLOCK_Y_NUM];
Ball_obj Ball;

//ブロックの設定
for(int i=0;i<BLOCK_X_NUM;i++){
    for(int j=0;j<BLOCK_Y_NUM;j++){
        Block[i][j].Set(50*i,20*j,GetColor(255,255,255));
    }
}

//ボールの設定
Ball.Set(200,400,5,5,-3,GetColor(0,0,255));

while(ProcessMessage()==0){
```

main関数内でBall_obj BallによりBall_objの変数・関数を持ったBallインスタンスを生成。
さらにBall.Setにて Ballのx,y,r,vx,vy,colorを指定している。

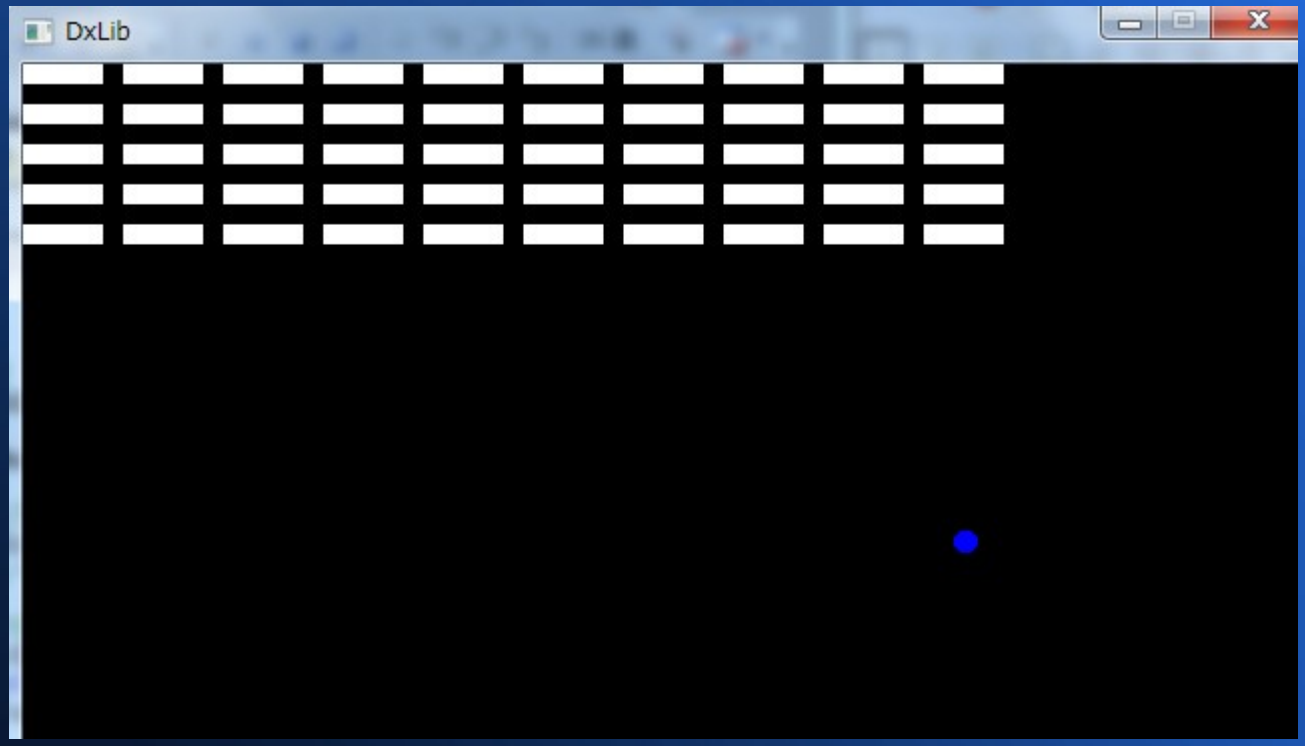
# ボールもクラスで作る。

```
//ボールの設定
Ball.Set(200,400,5,5,-3,GetColor(0,0,255));

while(ProcessMessage()==0){

    //描画されているものを消す。
    ClearDrawScreen();
    //メインループの動作はここに書く！！

    //ブロックの描画
    for(int i=0;i<BLOCK_X_NUM;i++){
        for(int j=0;j<BLOCK_Y_NUM;j++){
            Block[i][j].Graph();
        }
    }
    //ボールの描画＋移動
    Ball.Graph();
    Ball.Calc();
    //裏画面の描画状態を表に反映
    ScreenFlip();
}
```

メインループ内では
描画とボールの移動関数を追加した。

これにより、Ball.Setでセットされた
Ballの各パラメータを用いて、
ボールの描画や、
ボールの移動を行なっている。

# 実行結果




このように、青いボールが移動しつつ、描画されているのがわかる。
ただ壁で反射していないのがわかるだろうか。
そこでボールを移動させている関数、Ball_objのCalc関数を書き換えよう。

# 壁で反射させる。

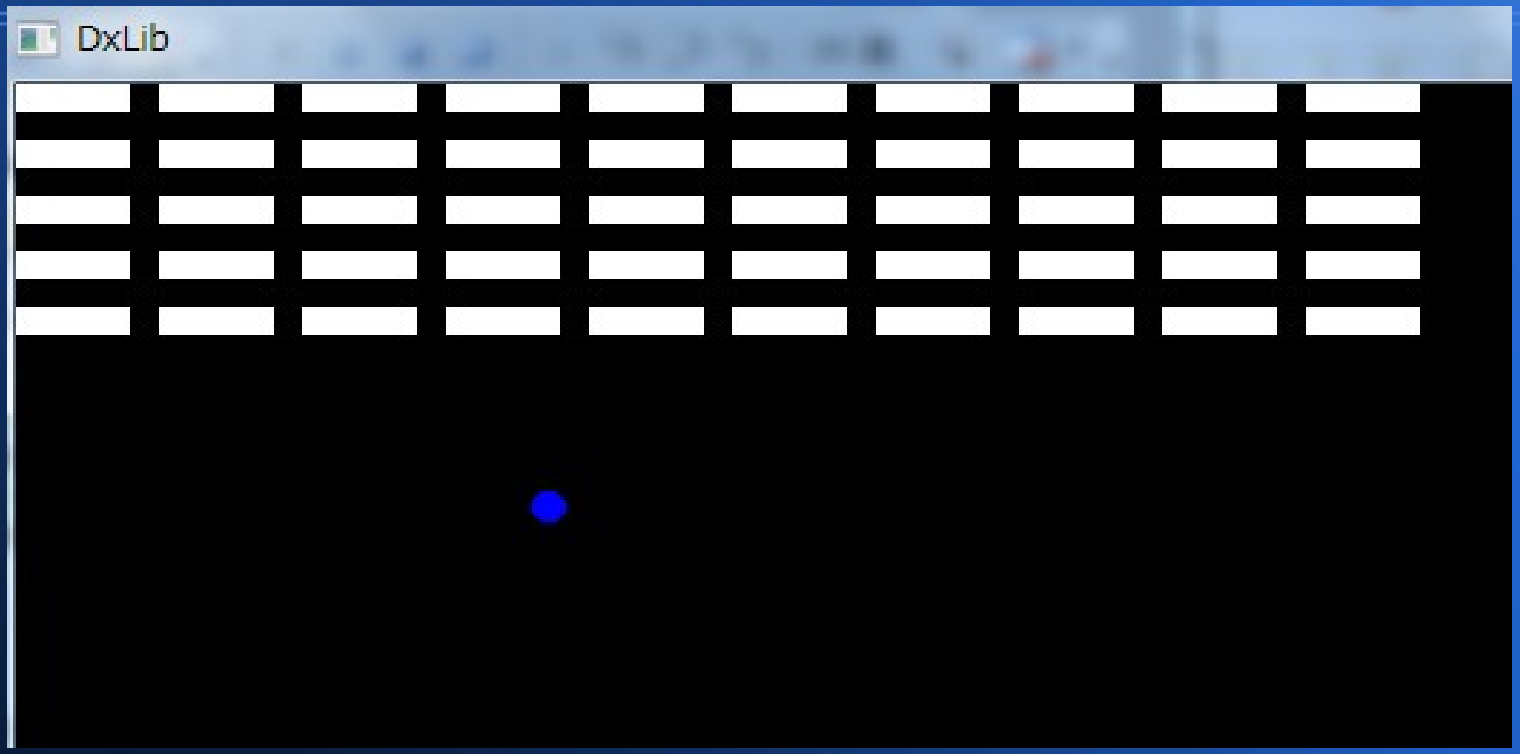
```
void Ball_obj::Calc(){
    if( x + vx >= 640 || x + vx <= 0 ){
        vx *= -1;
    }
    if( y + vy >= 480 || y + vy <= 0 ){
        vy *= -1;
    }
    x+=vx;
    y+=vy;
}
```

壁の反射の方法は前回のパートと同じ方法で、
ある進めてみたとき一定領域より進んでしまっていたら速度を変更させる。
という方法をとる。

このように、今回は移動を制御しているCalc関数内に壁の反射を追加した。

クラスに関わらず、関数を作るときのコツとしては
ある程度関数に独立性をもたせることである。
例えば描画処理と移動処理を同じ関数に書くのは非常にオススメしない。
ということである。

# 実行結果




さて、壁でボールが反射するようになっただろう。
以後、このクラスで作ったブロックとボールに処理を加えることで、
ブロック崩しを作ることにする。

# まとめ

- 構造体ではどこでも変数がいじくれてしまう！
  →どこがバグの原因か特定できない。
- クラスだと変数をいじれる関数が限定される。
  →どこがバグの原因かある程度特定可能。
- 関数を作る時は独立性をできるだけ高い状態で維持する。