C言語講習まとめ

情報研究会CACTUS

特に身に着けてほしいこと

- 構造体
- メモリ管理(ポインタ)
- ファイルの読み書き
- デバッグ力
- 検索，リファレンス参照する力
- コードリーディング力
- 設計する力
- 思いやりのあるコーディング

ブロック崩しゲームを参考に上記を学ぼう

今日のスケジュール

1. ブロック崩しゲームの概要
2. コードリーディング
3. コードの説明
4. ゲームの改造

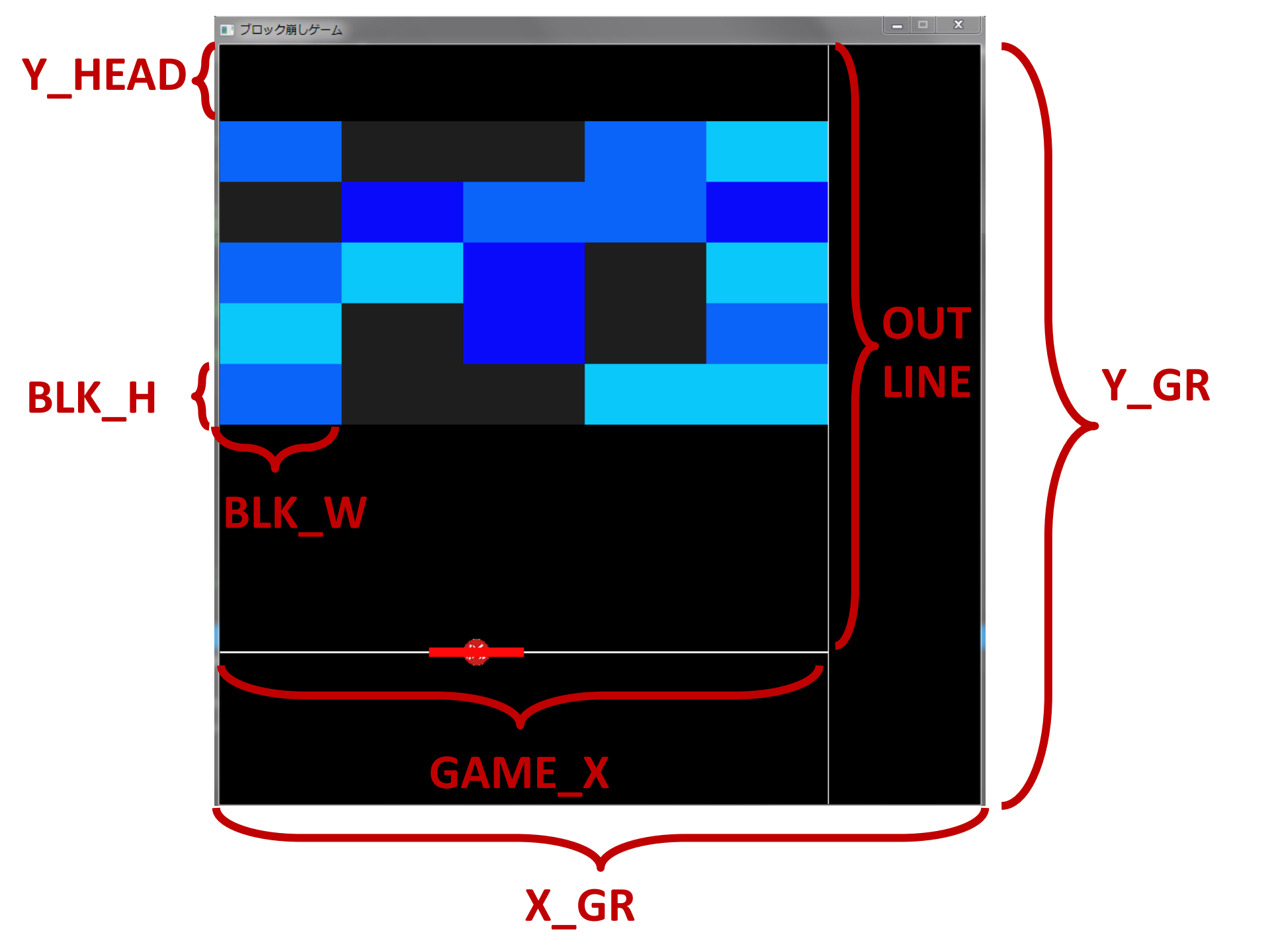
ゲーム概要

ただのブロック崩しゲームです.

ゲームレイアウト

- レイアウトを決定する数はゲーム中変更しないので#defineで設定しておく

- プログラム中に生の数字は書かないようにする（変更がめんどうであるため）

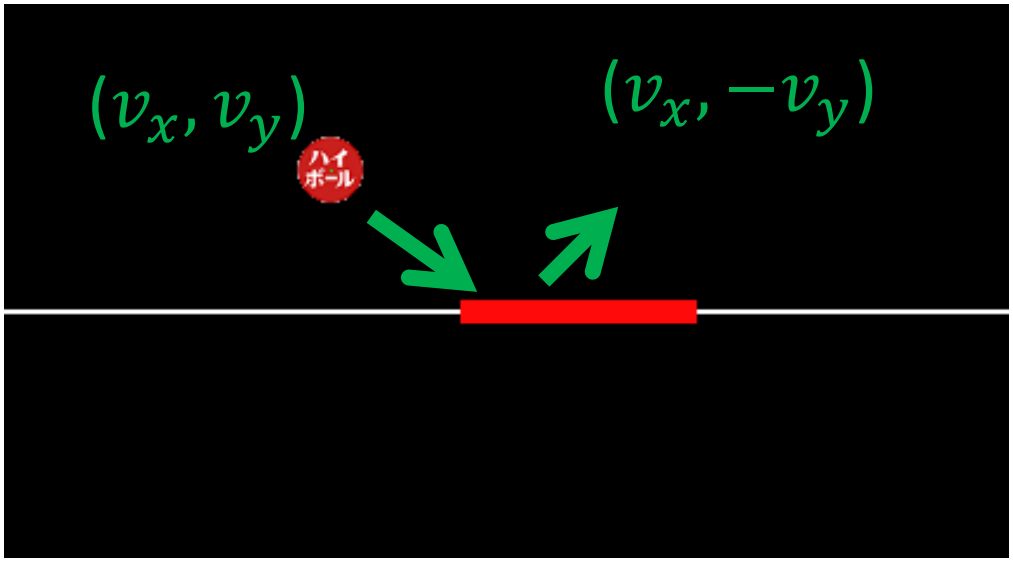

ブロック崩しゲーム
Y_HEAD
BLK_H
BLK_W
OUT
LINE
Y_GR
GAME_X
X_GR

ゲームの仕様

- ブロックの色はランダムで決める
- スコアは生き残ったフレーム×100である
- ブロックがすべて崩れたらクリア
- ボールがゲーム画面外に出たらDEAD

ゲームの仕様

- ボールは横から衝突したときの動作は下図のようにする



ゲームの仕様

- 各キーの割り当ては次のとおりである

キー入力	動作
SPACE	スタート
ESC	ゲーム強制終了
→キー	バーが右に移動
←キー	バーが左に移動

動作させてみよう

1. HachiKitフォルダのblock//block.exeを起動してみよう
2. HachiKitフォルダのblock//block.cppをプロジェクトに追加して動作させてみよう
3. HachiKitフォルダのblock!!//block!!.exeを起動してみよう（今日はこれを目指す）

ゲームの設計

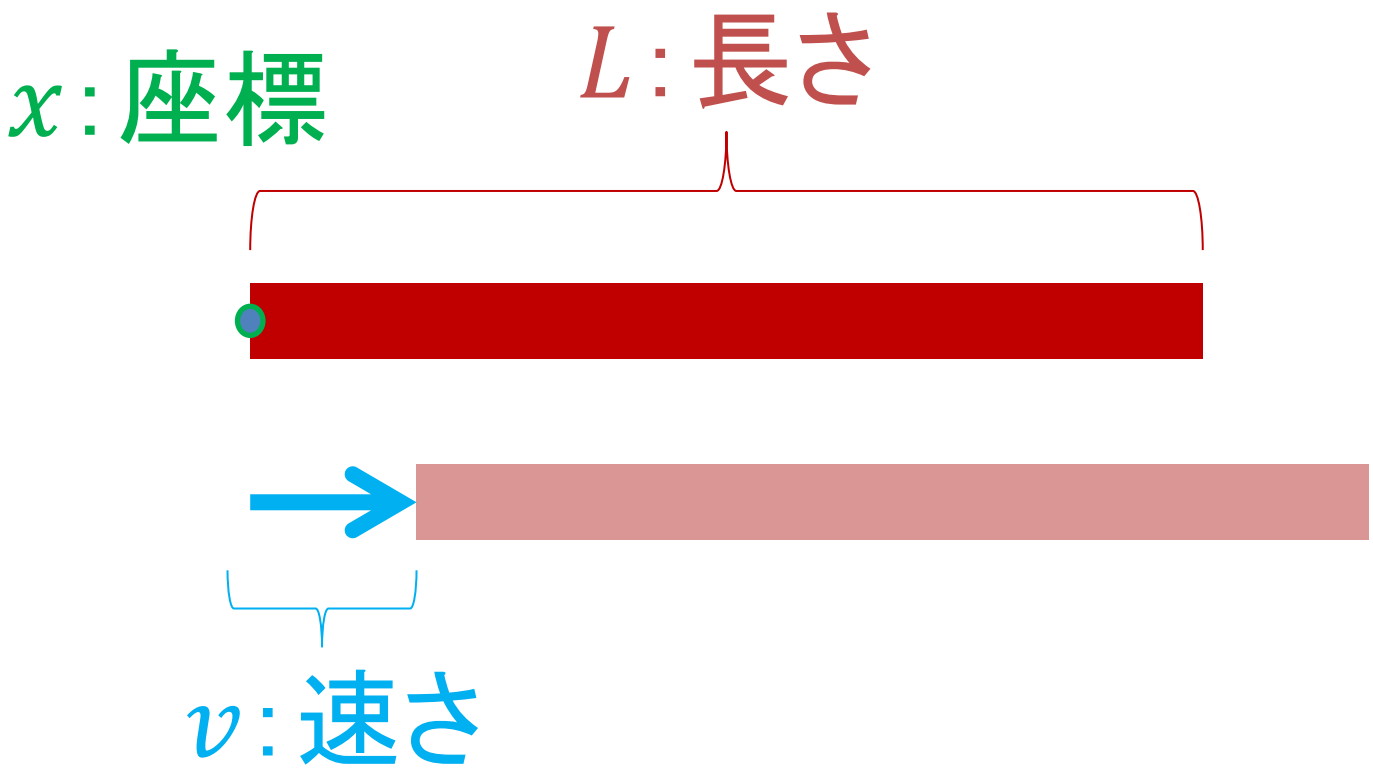
- ブロック，ボール，バーは構造体で定義する
- ブロックは二次元配列によって管理する

- モノをモノとして扱おう
- モノの抽象化=>構造体の定義

BAR構造体

```
typedef struct {
        int x;              //座標
        int v;              //速度
        int L;              //長さ
}BALL;
```

BAR構造体



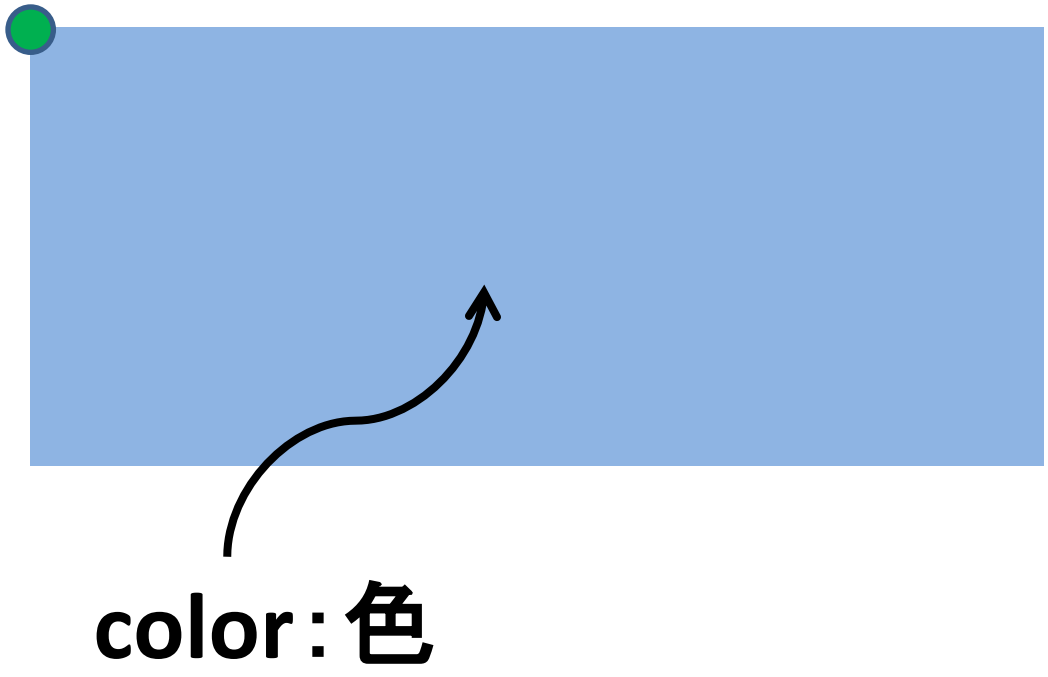
BLOCK構造体

```
typedef struct {
        int x,y;            //座標
        int color;          //色
}BLOCK;

BLOCK blk[5][5];
```

BLOCK構造体

(x, y): 座標



color：色


ブロック崩しゲーム
[0][0] [0][1] [0][2] [0][3] [0][4]
[1][0] [1][1] [1][2] [1][3] [1][4]
[2][0] [2][1] [2][2] [2][3] [2][4]
[3][0] [3][1] [3][2] [3][3] [3][4]
[4][0] [4][1] [4][2] [4][3] [4][4]

unsigned int MyGetColor(int n);

概略　　：nに対応する色を返す関数
引数　　：0~4の整数n（n=0の時はEMPTY）
戻り値　：nに対応する色

色	n
空色	2
水色	1
青	3
METAL	4
水色	1

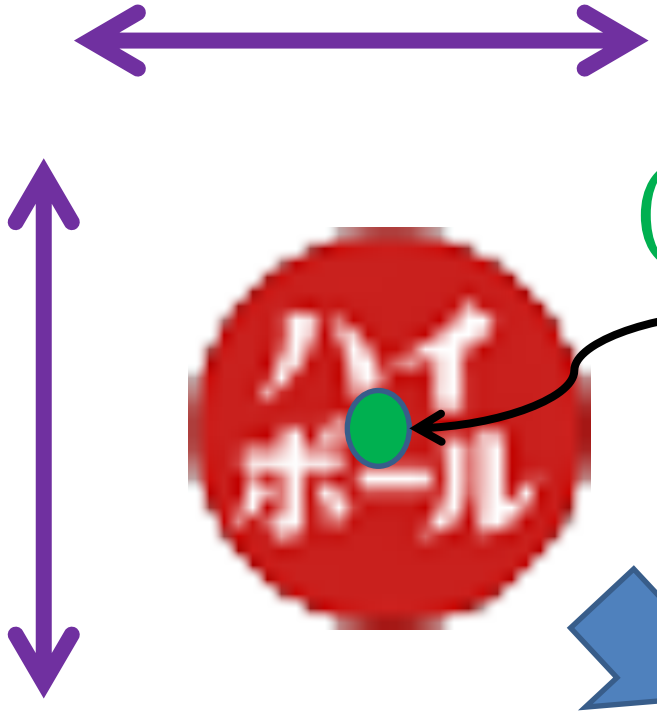
BALL構造体

```
typedef struct {
        int x,y;            //座標
        int imag;           //色
        int v_x,v_y;        //速度
        int ImgSizeX; //画像幅
        int ImgSizeX; //画像高
}BALL;
```

BALL構造体

img：画像ハンドル

imgSizeX：画像幅



imgSizeY：
画像高さ

(x, y)：座標

(v_x, v_y)：速度

構造体のポインタ

- 構造体のメンバにアクセスする

```
kouzoutai.member = 1;
int a = kouzoutai.member;
```

- 構造体のポインタからメンバにアクセスする

```
kouzoutai *Kpointer;
Kpointer->member = 1;
int a = Kpointer->member;
```

グローバル変数

- プログラム（〇〇〇.c）内の中でどこでも使える変数（mainの中でも関数の中でも）

- アクセスできる範囲が広いため競合が起きやすいできるだけ使わないようにしよう

リファレンスを読むこと

- ライブラリ中の関数は使えるようになる必要がありますが暗記する必要はありません
- ○○できる関数があったな…くらいにしておきましょう
- 関数のリファレンスを読む力をつけよう

宣言　　**int DrawFormatString(int x , int y , unsigned int Color ,**
char *FormatString , ...) ;

概略　　書式付き文字列を描画する

引数　　int x , y : 文字列を描画する起点座標
unsigned int Color : 描画する文字列の色
char *FormatString : 描画したい書式付き文字列のアドレス
... : 書式付き文字列に付随する引数

戻り値　　０：成功
－１：エラー発生

解説　　printf という関数はご存知でしょうか？
書式を指定することにより容易に数値変数や文字列配列の内容を画面に出力することの出来る便利なＣ言語の標準関数です。
そんな便利な printf 関数ですが、ＤＸライブラリを含む DirectX を使う環境では printf 関数は使うことが出来ません。
そこで登場するのがこの DrawFormatString 関数です。
この関数は printf と違い描画する座標や色を指定するという違いはあるものの、ほぼ printf と同じ機能を提供します。

宣言　　**int DrawLine(int x1 , int y1 , int x2 , int y2 , unsigned int Color) ;**

概略　　線を描画

引数　　x1 , y1　：　描画する線の起点座標
x2 , y2　：　描画する線の終点座標
Color　　：　描く線の色

戻り値　　０：成功
－１：エラー発生

解説　　画面上に点(x1 , y1)と点(x2 , y2)を結ぶ線をColorで指定した色で描きます。 (終端座標は描かれないので、実際には描き切りたい座標＋１の値を指定する必要があります)
パソコン画面は一般に画面左上が座標(０ , ０)で、 画面左から右に向かう方向が x のプラス方向の 画面上から下に向かう方向が y のプラス方向となります。 そして標準では画面の右端の x 座標値は 639 画面最下の y 座標は 479 となります。 この最大値は関数『SetGraphMode』によって変更する事が出来ます。
Colorの値は画面の色の表現できる色の数によってかわってきます。
この色の値はライブラリの関数『GetColor』を使って取得する事をお勧めします。

宣言　　**int DrawBox(int x1 , int y1 , int x2 , int y2 ,**
unsigned int Color , int FillFlag) ;

概略　　四角形を描画

引数　　x1 , y1　：　描画する四角形の左上の頂点座標
x2 , y2　：　描画する四角形の右下＋１の頂点座標
Color　　：　描画する四角形の色
FillFlag：　四角の中身を塗りつぶすか、のフラグ。TRUEで塗りつぶし、FALSEで塗りつぶさない

戻り値　　０：成功
－１：エラー発生

解説　　(x1, y1)を四角形左上の頂点、(x2 - 1, y2 - 1)を右下の頂点とする四角形をColorで指定された色で描画します。
（パソコン画面の座標の持ち方、色の指定のし方についての説明はすぐ上のDrawLine関数の解説を参照してください。）
FillFlagをTRUEにすると四角形の中身もColorで指定した色で塗りつぶし、FALSEを指定すると輪郭のみを描画します。（TRUEと言うのは#defineマクロで定義された定数で1と同義です。FALSEについては0と同義です）

宣言　　**int DrawPixel(int x , int y , unsigned int Color) ;**

概略　　点を描画する

引数　　x , y :　点を描画する座標
Color :　描画する点の色

戻り値　　０：成功
－１：エラー発生

解説　　座標(x , y)にColorで指定した色で点を描画します。
（パソコン画面上での座標のとり方、色の指定方法は『DrawLine』 の解説を参照してください）

宣言　　**int LoadGraph(char *FileName) ;**

概略　　画像ファイルのメモリへの読みこみ、及び動画ファイルのロード

引数　　FileName :　ロードする画像、及び動画ファイルの
　　ファイルパス文字列へのポインタ

戻り値　　－１　　：　エラー発生
－１以外　：　グラフィックのハンドル

解説　　画像ファイルをメモリにロードします。
これは表示する必要が出るたびにディスクにアクセスすると画像処理の負荷が非常に高くなってしまうので、ディスクよりも高速に処理を行うことが出来るメモリ上に画像を保存してしまおう、と言う考えから来ているものです。
この関数が成功するとグラフィックハンドルと言うものが返って来ます。 これはメモリに保存した画像の識別番号で int 型の数値です、 読みこんだ画像を描画する際にこの識別番号を指定するとメモリに読みこんだ画像を描画することが出来ます。

例　　test1.bmpをロードし、戻り値であるグラフィックハンドルを
int 型変数 GrHandle に保存します

```
int GrHandle ;

GrHandle = LoadGraph( "test1.bmp" ) ;
```

尚、読み込むことの出来る画像形式は BMP,JPEG,PNG,DDS,ARGB,TGA の６種類です。

宣言　　**int DrawGraph(int x, int y, int GrHandle, int TransFlag) ;**

概略　　メモリに読みこんだグラフィックの描画

引数　　x , y　　：グラフィックを描画する領域の左上頂点の座標
GrHandle　：描画するグラフィックのハンドル
TransFlag　：　画像の透明度を有効にするかどうか(TRUE：有効にする　FALSE：無効にする)

戻り値　　０：成功
－１：エラー発生

解説　　LoadDivGraph、LoadGraph、MakeGraph等で読みこんだ（作成した） グラフィックを(x , y)を描画する画像の左上頂点として描画します。
TransFlagをTRUEにすると画像の透明度が有効になります。
(画像の透明度の詳細については SetTransColor の解説を参照してください)

宣言　　**int GetGraphSize(int GrHandle ,**
int *SizeXBuf , int *SizeYBuf) ;

概略　　グラフィックのサイズを得る

引数　　GrHandle　：　サイズを調べるグラフィックのハンドル
SizeXBuf　：　グラフィックの幅を保存するint型変数のポインタ
SizeYBuf　：　グラフィックの高さを保存するint型変数のポインタ

戻り値　　０：成功
－１：エラー発生

解説　　GrHandleで指定したハンドルが持つグラフィックのサイズを 取得する関数です。サイズはそれぞれSizeXBufとSizeYBufが示す int型変数に保存されます。

例　　test1.bmpを読みこんでそのビットマップのサイズをint型変数SizeX,SizeYに格納します

```
int SizeX , SizeY , GrHandle ;

GrHandle = LoadGraph( "test1.bmp" ) ;
GetGraphSize( GrHandle , &SizeX , &SizeY ) ;
```

宣言　　**int GetHitKeyStateAll(char *KeyStateBuf) ;**

概略　　キーボードのすべてのキーの押下状態を取得する

引数　　char *KeyStateBuf : すべてのキーの押下状態を格納するバッファのポインタ

戻り値　　０：成功
－１：エラー発生

解説　　CheckHitKey 関数はキーボードのキーの押下状態を取得するための関数です。
沢山のキーの状態を CheckHitKey 関数で調べる場合は当たり前ですが沢山 CheckHitKey 関数を使わなければなりません。
ですが、実は CheckHitKey 関数はとても無駄の多い関数です。
理由は聞かないで下さい。とにかく無駄が多い関数なのです。
ですから複数のキーの状態を一度に知りたい場合は、 CheckHitKey 関数よりこの GetHitKeyStateAll 関数を使ってください。 この関数はすべてのキーについて CheckHitKey 関数を使った場合と同じものを一度に配列に格納してくれます。

コードリーディングしていこう

HachiKitフォルダのblock//block.pdfをみんなで読みましょう

練習問題A

1. コードを呼んで隠しコマンドを探せ．どのキーを押せばよいか

2. 画像をball.pngに変更してみよう

練習問題B

現在初期ブロックの色はランダムに決定している

ブロックの色をテキストファイルから読み込んで使ってみよう（HachiKit//map//block_color.txt）

```
4 4 4 4 4
3 3 4 3 3
2 3 4 3 2
1 2 2 2 1
1 1 1 1 1
```

練習問題C

- ボールを移動する関数

  void MoveBALL(BALL *ball);を完成させよ

概略：ボールの座標を(x, y)，速度をそれぞれv_x, v_yとすると次の座標$(x + v_x, y + v_y)$に移動させる

引数：ボール構造体のポインタ

戻り値：なし

キー入力

```
//すべてのキーの監視をするための関数
//グローバル変数を使ってるので注意
int GetHitKeyStateAll_2(char stateKey[]) {
    char GetHitKeyStateAll_Key[KEY_MAX];        //とりあえず保存のための配列

    //キー配列の取得
    GetHitKeyStateAll(GetHitKeyStateAll_Key);

    //何回押されたかを監視する
    for (int i = 0; i<256; i++) {
        if (GetHitKeyStateAll_Key[i] == 1)  stateKey[i]++;
        else                                stateKey[i] = 0;
    }
    return 0;
}
```

キー入力

(stateKey[KEY_INPUT_LEFT] >= 1)

(stateKey[KEY_INPUT_LEFT] == 1)

ふたつの違いはなんだろうか？変更してみて違いを確かめてみよ

乱数

乱数とは人間にとってランダムに見える数列のことである.

例 : 3.14159265359…

しかし，毎回3141…だとまったくランダムに見えない

なので，数列のどこからスタートするかをランダムに決めなくてはならない（乱数の初期化）

乱数の初期化

乱数の初期値のことをシードという

シードは時間で決定するとよい

C言語ではsrand()を使ってシードを決定する

```
//乱数初期化
srand((unsigned int)time(NULL));
```

このように書くことが望ましいとされる

時間に依存させることで，あたかもランダムであるように見える

乱数を使ってみる

C言語の乱数は非常に高速であるが実は規則性がある．あまり使わないようにしよう

int rand(void);

0~RND_MAXの値をランダムに返す

引数：なし　　戻り値：乱数

たとえばサイコロ（現実世界の疑似乱数）のように1~6の乱数を生成するにはどうすればよいか？

乱数を使ってみる

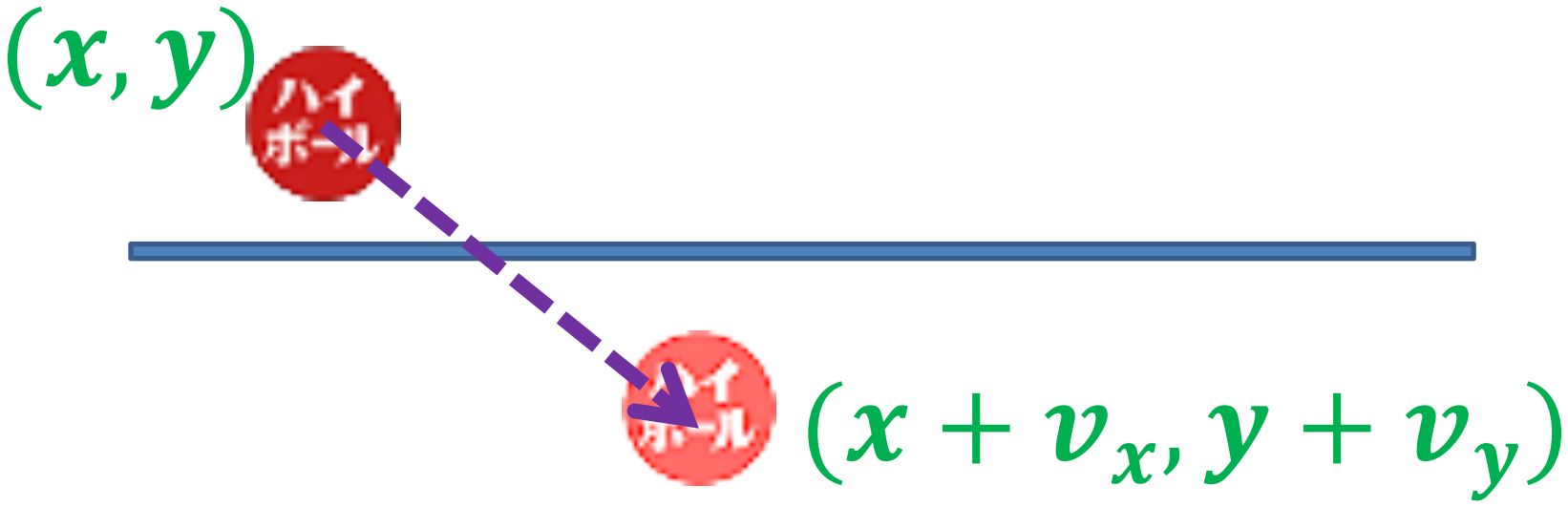
たとえば次のようにすればよい

```
rand() % 6 + 1
```

あくまで疑似的な乱数である.

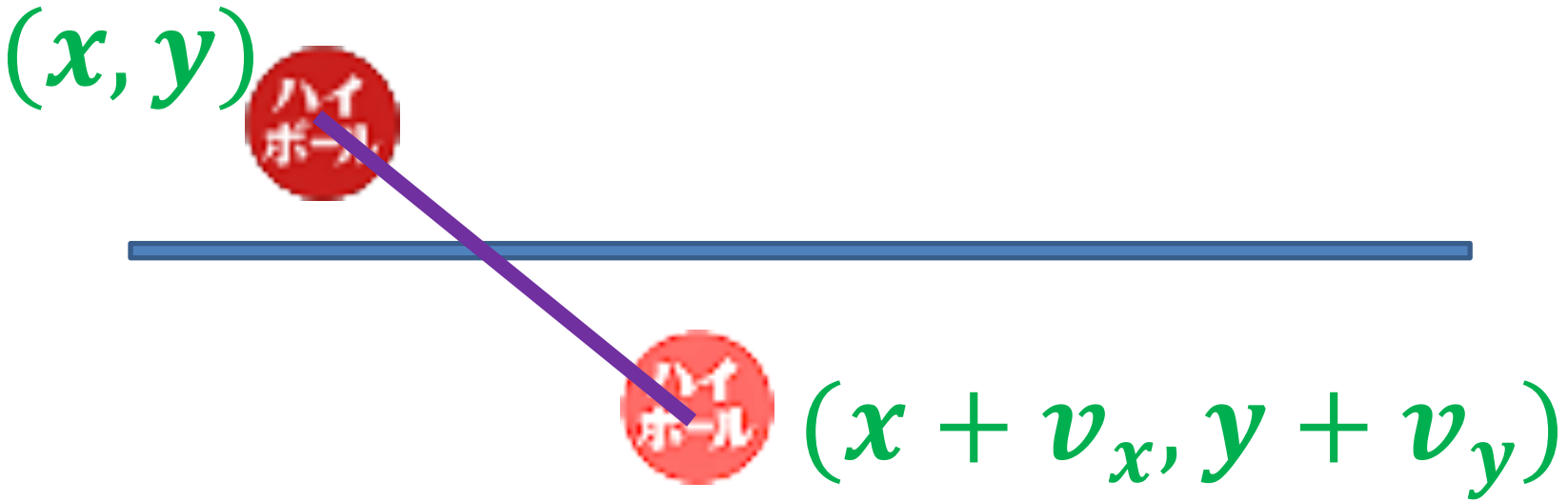
衝突判定

どのような時に衝突されたと判定するか？



衝突判定

青の線分と紫の線分が交差してる時衝突する



線分交差判定

- 線分交差判定アルゴリズムを使えばよい
- 高校数学程度で十分理解できる
- bool LineLineCollision(…)という関数でこの判定を行っている.

- （インターネットを使って線分交差判定アルゴリズムを理解せよ）

演習問題1

- 自分で好きなギミックを追加してみよう

（例：
アイテムが落ちてくる，
ボールが二つに増える，
ボーナスステージ（スコアが二倍），
壊れないブロックの追加，
衝突した時の音の追加
）

演習問題2

- ブロックの画像を用意してきた.

Dxlibの関数

int DrawExtendGraph(…);

を使ってブロックの画像を変更せよ

unsigned int MyGetColor(int n);
に代わる新しい関数
int MyGetImage(int n);
を定義すること

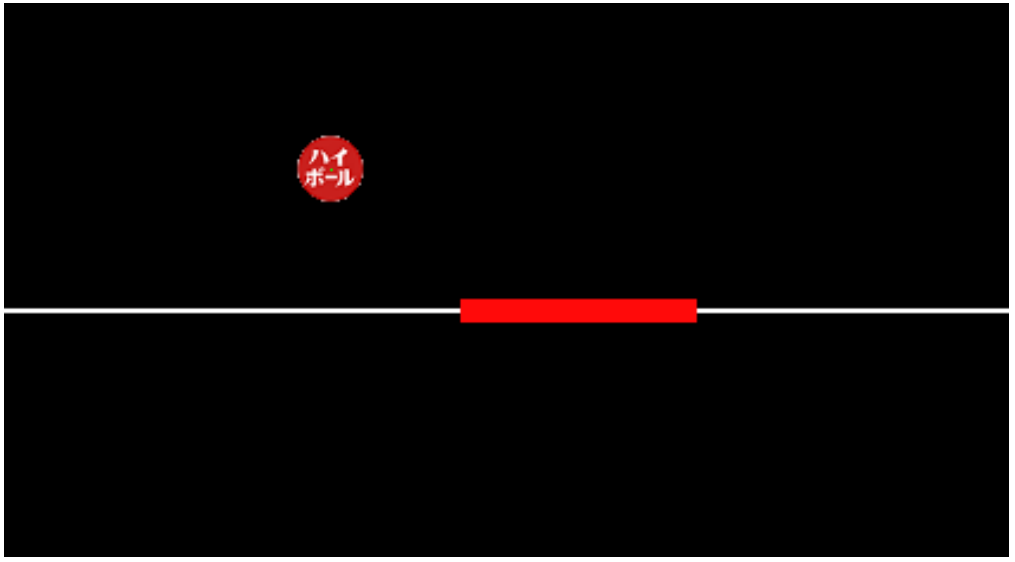
演習問題3

現在ボールは決まった動きしかしない.

バーの当たった位置に依存して跳ね返るよう変更してみよ

跳ね返りの仕様は自由とする

ヒントは次ページ



演習問題3ヒント

$$y = a_1 x + b_1$$
$$y = a_2 x + b_2$$

で表せる二つの直線の交点の座標は

$$x = -\frac{b_2 - b_1}{a_2 - a_1}$$
$$y = -a_1 x + b_1$$

である. ただし, $a_1 \neq a_2$とする.

フーリエ級数展開

- HachiKitフォルダのフーリエ級数展開フォルダFoueier.cppを読んでみよう
- kukeiha.exeは矩形波をフーリエ級数展開したものである
- Sanakuha.exeは三角波をフーリエ級数展開したものである
- ↑キーを押せば精度が上がる
- ↓キーを押せば精度が下がる

バリスティック凝集現象

- HachiKitフォルダのフーリエ級数展開フォルダBA.cppを読んでみよう.
- BA.PNGが出力画像例である
- BA_2.exeはバリスティック凝集現象モデル（BAモデル）をシミューレートしたものである
- BAモデルは沈殿などの物理現象をモデル化したものである