Media Programing 1 - openFrameworks

- frame.02 - 20010.04.27 hrr

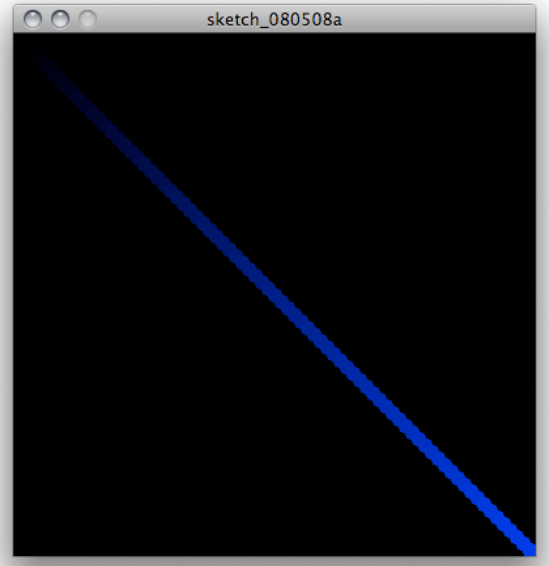
今回のテーマ

前回の残り（関数、アニメーション、マウス）
配列
オブジェクト指向入門

前回の復習

復習1：

右上から左下にかけて徐々に大きくなりながら、
色も段々と濃くなりなが少しずつ重なっている円を描く。



きちんと色の濃さが０から255までになるように計算して色を決定する。
ウインドウの大きさを変えても円の数は増えても、グラディエーションは
変わらないようにする。
ウインドウの幅は width という変数に入っています。
ちなみに、heightにはウインドウの高さが入っています。

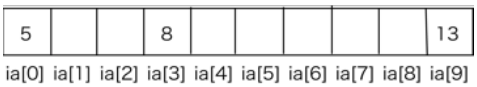
配列

複数の格納領域を持った変数を名前をつけてまとめたもの。
[] 内の番号（添字）によってアクセスする。
添字の数の分だけ変数が作られて、添字を使う事でプログラム内では
計算した結果に応じた変数アクセスが可能になる。

配列の生成

--

```
   int ia[10];
   // 10個分の整数を大きさをもつ、iaという名前の配列


void testApp::setup(){

   ofSetWindowShape(400, 400);
   ofBackground(0, 0, 0);

   ia[0] = 5;
   ia[3] = 8;
   ia[9] = ia[0] + ia[3];
}
```



配列を使わないで、

```
   i1 = 5;
   i2 = 8;

   sum = i1 + i2;
```

とやるのと同じようなものですが、配列にすると、次のように
ループでまとめて繰り返し処理ができるようになります。

```
   sum  = 0;
   for ( int i = 0; i < 10 ; i++){
      sum = sum + ia[i];
}
--
```

練習問題１：

整数の配列を引数として受け取り、配列の中のすべての整数値の平均を返す
関数 average() を作成する。
配列の大きさ（要素の数）は、変数の大きさ（バイト数）を返す、
sizeof() を使って、配列のバイト数を各要素の型の大きさで割ると得られる。
例えば、
　　int　ia[10];
で宣言された配列では、
　　len = sizeof(ia) / sizeof(int);
とすると、len に配列の大きさ 10 が入る。（ただしlenは整数型の変数）
呼び出し部分は以下のようにする（あくまで例として）

```
--
const int N = 100;
int  ary[N];

//ここにaverage関数を記述

void testApp::setup(){
```

```
    ofSetWindowShape(400, 400);
    ofBackground(0, 0, 0);

    for(int i =0; i < N; i++){
        ary[i] = int(ofRandom(0,1000));
}

int average(int *ar){
    int sum = 0;

    int len = sizeof(ar) / sizeof(int);
    for(int i = 0; i < len; i++){
        sum = sum + ar[i];
    }
    return sum/len;

}

void testApp::draw(){
    int result;

    result = average(ary); // この関数を作成する
    string str;
    str = ofToString(result, 0);
    ofDrawBitmapString(str, 10, 20);
}
```

二次元配列

配列は何次元にも拡張できます。

２次元空間を扱うなら、(x,y) の２つの変数からどの配列か決定できる方が便利に使えます。

int ia[100];

int ma[10][10];

この２つは配列の大きさとしては一緒です。

例えば、座標(5, 3)の場所にある配列の添字は、ia[15],　ma[5][3]

となるので、２次元配列の方が余計な計算をせずに直感的に利用できます。

練習問題２：

2次元配列を使って、色が連続的に変化していくアニメーションを作成する。

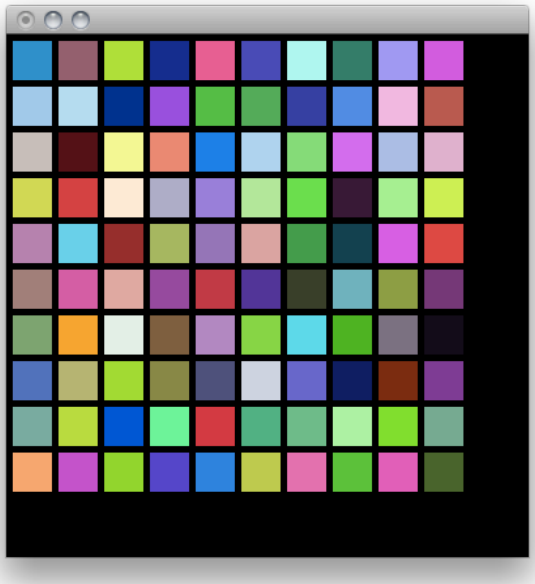
10x10の合計100個の四角を描き、各々の四角についてRGBの１パラメータを連続的に

変化させることで色を変えていく。

例として、プログラムの最初の部分を示すので、setup() および draw()を完成させる。

```
const int MX = 10;
const int MY = 10;

ofColor mtx[MX][MY];      //カラー型の２次元配列

void testApp::setup(){
```

```
    ofSetWindowShape(400, 400);
    ofBackground(0, 0, 0);
    ofSetVerticalSync(true);
    ofSetFrameRate(30);
    int i,j;

    for(i = 0; i < MX; i++){                // 二重ループで初期値となる色を設定
        for(j = 0; j < MY; j++){
            mtx[i][j].r = ofRandom(0, 255);
            mtx[i][j].g = ofRandom(0, 255);
            mtx[i][j].b = ofRandom(0, 255);
        }
    }
}

void testApp::draw(){
// 色を変化させるアニメーション
    r,g,bのどれか一緒を連続的に変化させてください。
    乱数を使って、条件を分けをする事で変化する色を変えてもよいかも。

}
----
```



オブジェクト指向

openFrameworksはC++をベースとしたライブラリであるため、オブジェクト指向型の
プログラムが可能である。
オブジェクト指向型言語では、プログラムをオブジェクト（もの）という

概念で扱うことで、より現実に近い形でプログラムを作ることができる。
オブジェクトには、**状態（データ）**とその**振る舞い（関数、メソッド）**が定義されていて、
オブジェクトに対して、「何々する」という**操作を行う**こと（メッセージを送る）ことで、
処理が行われる。

オブジェクト指向言語の特徴としては、
・カプセル化、隠蔽
・継承
・ポリモルフィズム
・メッセージパッシング
などが挙げられる。

例えば、現実の例として、自転車を考える。
自転車には、ハンドルの向き、ギアの段数、ブレーキの状態、ペダルの状態などの
変数がある。
また、ペダルをこぐ、ハンドルを切る、ブレーキをかけるなどの操作がある。
つまり、自転車というオブジェクトをこれらの変数と関数を定義してあげれば
よいことになる。
これらのオブジェクトの定義は、クラスと呼ばれ、クラスを定義することが
オブジェクトの設計図にあたり、オブジェクト自体は設計図から生成された
実体といえる。
オブジェクトを操作するには、そのクラスに定義されている関数を呼ぶことで、
オブジェクトに対しメッセージを送り、対応する処理を行ってもらう。

カプセル化とは、クラスの定義おいて、クラス内だけ必要な変数や関数を
クラスの外から見えないようにすることで、必要のない情報を隠蔽し、
間違いを起こしにくくすることである。
継承とは、元となるクラスから、その子供となるクラス（サブクラス）を定義
することで、似たような機能をもつクラスを作ることを意味する。
例えば、自転車でいえば、２つの車輪とハンドル、ペダルをもつ自転車クラスから
かごの付いたママチャリクラスや、ギアがたくさんあるMTBクラスとかを定義する
ことにあたる。
継承により、既に用意されたクラスを目的に応じて再利用しながら、
新しいクラスを作るなど、再利用性を高めることができると同時に
概念的にもわかりやすいプログラムを作ることができる。
ポリモルフィズムは、概念的には同じでも処理が異なる機能を同じ名前で
実現することである。
たとえば、「足す」という機能は、数値どうしと文字列同士では処理がことなる。
これを同じ足すという名前で行えるような仕組みのことを示す。
実際の例では、既に用意されているライブラリ、他の人が作ってくれたライブラリは
クラスとして用意されています。ドキュメントでも"class ofSoundPlayer"など
クラス単位でいろいろな機能が用意されているのがわかります。
それらのクラスを利用する事、つまり、クラスからオブジェクトを生成して
オブジェクトの機能を呼び出す事で、様々なことが可能となります。
さらに、用意されたクラスからサブクラスを作る事で、自分用に機能拡張や

独自のカスタマイズをすることが簡単にできるようになっています。

クラスの定義

もう少し、openFrameworksらしい例を考えてクラスを作成していきます。
配列を使って、１０個の点を作ろうとすると、x[10], y[10] という２つの
配列を作って、x座標とy座標を別々に扱う必要がある。
点を扱うのに、座標値と色をまとめて定義すると点をたくさん扱うときには
便利になりそうである。

```
--
   class Zukei{
      float x;
      float y;
      ofColor c;
   }
--
```

これがZukeiというクラスの最初の**定義**である。
クラスで定義されているオブジェクトを利用するには、変数と同じように宣言します。
オブジェクトが生成されるときに自動的に実行される関数は、
クラス名と同じ関数で定義される。これを**コンストラクタ**と呼ぶ。

```
--
   class Zukei{
      private:

      float x;
      float y;
      color c;

      public:                  // 誰でもアクセスできるpublicなメンバーやメソッド
                               // であることを宣言する
                               // クラスの内部だけで使う物には、private: を使います。

      Zukei(){                 // Zukeiクラスを生成するときに自動実行
         x = ofRandom(200);
         y = ofRandom(200);
         c.r = ofRandom(0,255);
         c.g = ofRandom(0,255);
         c.b = ofRandom(0,255);
         c.a = ofRandom(0,255);
      }
   };



   Zukei z;                 // Zukeiオブジェクトの宣言

   void testApp::setup(){

   }
```

--

これで、Tenクラスから、ランダムな座標値と色を持つZukeiオブジェクトが
生成された。

このままでは、何も描画できないので、Zukeiクラスに描画部分を加える。
ついでに、座標値もウインドの大きさに合わせるように変更する。

--

```
#include "testApp.h"

class Zukei {
   float x;
   float y;
   ofColor c;

   public:

   Zukei(){
      x = ofRandom(0, 400);
      y = ofRandom(0, 400);
      c.r = ofRandom(0, 255);
      c.g = ofRandom(0, 255);
      c.b = ofRandom(0, 255);
      c.a = ofRandom(0, 255);
   }

   void ten(){                          // 点を描画するためのメソッド（関数）
      ofSetColor(c.r, c.g, c.b , c.a);
      ofRect(x, y, 1, 1);               // 点なので、サイズは1x1
   }
};


Zukei z;

void testApp::setup(){
   ofSetWindowShape(400, 400);
   ofBackground(0, 0, 0);
   ofSetVerticalSync(true);
   ofSetFrameRate(30);
}

void testApp::draw(){
     z.ten();              // zオブジェクトのtenメソッドの実行
}
```

--

これで一つの点が描画できた。
ここまで、座標値、と色という状態を持ち、描画という機能をもった
点を「もの」を定義するZukeiというクラスがとりあえず完成した。

点だけだと小さすぎて良くわからないので、適当にサイズを変えて確認が必要。

練習問題3：

Zukei クラスと同じように円というものを定義するEnクラスを作成する。
座標値、色、直径という状態（属性）と描画する機能を記述する。
直径もrandomで良い。

オーバーロード

クラスの定義では、同じ名前であるが引数や返り値が異なる関数を
重複して定義できる。
引数の数や型が一致した関数が自動的に呼ばれる。
コンストラクタでも、この仕組みを利用して引数によって初期値を設定して
オブジェクトの生成（インスタンス化）が行える。
座標を引数として受け取るコンストラクタを定義すると、座標値を指定して
Zukeiオブジェクトを生成出来る。

```
--

class Zukei {
   float x;
   float y;
   ofColor c;

public:

   Zukei(){
      x = ofRandom(0, 400);
      y = ofRandom(0, 400);
      c.r = ofRandom(0, 255);
      c.g = ofRandom(0, 255);
      c.b = ofRandom(0, 255);
      c.a = ofRandom(0, 255);
   }

   void ten(){
      ofSetColor(c.r, c.g, c.b , c.a);
      ofRect(x, y, 1, 1);
   }

   void ten(float x, float y){       // 座標を渡して、点を描画する。
      this->x = x;                   // このクラスのx,yの値を受けとった値に変更
      this->y = y;
      ofSetColor(c.r, c.g, c.b, c.a);
      ofRect(x, y, 1, 1);            //
   }
};


--
```

このように、同じ名前で引数の違うコンストラクタを定義すると、
引数に応じて、対応したコンストラクタを呼んでくれる。

同じように、同じ名前で引数の違う関数（メソッド）を定義すると
引数に応じて適切なメソッドを呼んでくれる。

--

```
class Zukei{
   float x;
   float y;
   ofColor c;

   Zukei(){
      x = ofRandom(0, 400);
      y = ofRandom(0, 400);
      c.r = ofRandom(0, 255);
      c.g = ofRandom(0, 255);
      c.b = ofRandom(0, 255);
      c.a = ofRandom(0, 255);
   }

   Zukei(float x, float y){    // 座標を渡してTenを生成
      this->x = x;
      this->y = y;
      c.r = ofRandom(0, 255);
      c.g = ofRandom(0, 255);
      c.b = ofRandom(0, 255);
      c.a = ofRandom(0, 255);
   }

   void ten(){
      ofSetColor(c.r, c.g, c.b , c.a);
      ofRect(x, y, 1, 1);
   }

   void ten(float x, float y){
      this->x = x;
      this->y = y;
      ofSetColor(c.r, c.g, c.b, c.a);
      ofRect(x, y, 1, 1);
   }
};


Zukei z;
Zukei z2 = Zukei(50,50);

void testApp::setup(){
   ofSetWindowShape(400, 400);
   ofBackground(0, 0, 0);
   ofSetVerticalSync(true);
   ofSetFrameRate(30);

}
```

```
   void testApp::draw(){
      z2.ten();
      z.ten(10,10);
   }
```

--

練習問題4：

Zukeiクラスを使って、マウスをクリックした場所に点を描画するプログラムを作成する。
testApp:mousePressed() 内にマウスをクリックした座標を記録する部分、
testApp:draw() 内にその座標に点を書く部分を作成する。

クラスの配列

乱数を使っていても、毎回同じ座標や色が書かれてしまうことがあります。
このような時は乱数のSeedを変更する事で、乱数の系列が変わる。
ofSeedRandom();　を使うと乱数のSeedを変更できます。
ただし、使うにはSeedを変更してからオブジェクトの生成をする必要があるので、
動的なオブジェクト生成を行う必要があります。

ここまで、基本的な点描画のクラスを作ることができた。
次に、配列を使って複数の点を描画できるようにする。

--

```
   Zukei za[10];                // Zukeiオブジェクトを入れる配列を生成
// Zukei *za;                   // 動的生成の場合同じ内容をこのように書く事もできる

   void testApp::setup(){
      ofSetWindowShape(400, 400);
      ofBackground(0, 0, 0);
      ofSetVerticalSync(true);
      ofSetFrameRate(30);
      ofSeedRandom();
//    za = new Zukei[10];          // 動的生成の場合
   }

   void testApp::draw(){
      for(int i = 0; i < 10; i++){
         za[i].ten();
//       (za+i)->ten();            // 動的生成の場合
   }
```

--

授業内課題１：

一つの整数を引数として与えると、その整数の範囲内のランダムな値で
座標を設定して、オブジェクトを生成するコンストラクタを定義しなさい。

```
   Zukei(int dsize){
      ....
```

```
}
```

サブクラスの作成

オブジェクト指向の特徴であるクラスの継承機能を使ってサブクラスを作成する。
サブクラスは、既存のクラスをその属性や機能（変数やメソッド）を引き継いだ上で
必要に応じた拡張を行ったクラスである。

Zukeiクラスを拡張して, Sikakuクラスを作成してみる。

```
--
   class Zukei {
   protected:                    // サブクラスでも使えるようにprotectedに変更する
      float x;
      float y;
      ofColor c;

   public:

      Zukei(){
         x = ofRandom(0, 400);
         y = ofRandom(0, 400);
         c.r = ofRandom(0, 255);
         c.g = ofRandom(0, 255);
         c.b = ofRandom(0, 255);
         c.a = ofRandom(0, 255);
      }

      Zukei(float x, float y){
         this->x = x;
         this->y = y;
         c.r = ofRandom(0, 255);
         c.g = ofRandom(0, 255);
         c.b = ofRandom(0, 255);
         c.a = ofRandom(0, 255);
      }

      void ten(){
         ofSetColor(c.r, c.g, c.b , c.a);
         ofRect(x, y, 10, 10);
      }

      void ten(float x, float y){
         this->x = x;
         this->y = y;
         ofSetColor(c.r, c.g, c.b, c.a);
         ofRect(x, y, 1, 1);
      }
   };

   // ここからサブクラス

   class Sikaku: public Zukei{
```

```
    float d;                    // 四角の大きさを入る変数

    public:

    Sikaku() : Zukei(){         //  Zukeiクラスのコンストラクタを呼んで初期化
      d = ofRandom(10, 100);    // 四角のサイズを乱数で決定
    }

    void paint(){               // 描画用のメソッド
      ofSetColor(c.r, c.g, c.b, c.a);
      ofRect(x, y, d, d);
    }


};

Sikaku sk;

void testApp::setup(){
  ofSetWindowShape(400, 400);
  ofBackground(0, 0, 0);
  ofSetVerticalSync(true);
  ofSetFrameRate(30);
}

void testApp::draw(){
  sk.paint();
}
```

--

練習問題５：

Sikakuクラスに塗りつぶすかどうかを指定する変数として論理型のisFill を
導入し、setFill(boolean) というメソッドを作成し、
塗りつぶすかどうかを決められるようクラスを変更する。

```
class Sikaku: public Zukei{

    float d;
    boolean isFill;

    public:

    Sikaku() : Zukei(){         //  Zukeiクラスのコンストラクタを呼んで初期化
      d = ofRandom(10, 100);    // 四角のサイズを乱数で決定
    }

    void setFill(boolean f){
      //  ここにコードが必要
    }
```

```
    void paint(){                  // 描画用のメソッド
      //   isFillの値に応じて塗りつぶしするかを決定する部分を追加
      ofSetColor(c.r, c.g, c.b, c.a);
      ofRect(x, y, d, d);
    }
};
```

動きを制御するメソッドの追加

描画するだけでなく動き方自体もクラスの中に入れる。
例えば、左右に往復する移動を示すメソッドとして moveH()という関数を作成すると
testApp::draw()内で、sk.moveH() を繰り返す事で、skオブジェクトの座標などを意識する
ことなく、移動のアニメーションを行う事ができる。
Zukeiクラスに追加見てみると、

```
--
  class Zukei {
    protected:
      float x;
      float y;
      ofColor c;
      float speed;

    public:

    Zukei(){
      speed = 1.0;
      x = ofRandom(0, 400);
      y = ofRandom(0, 400);
      c.r = ofRandom(0, 255);
      c.g = ofRandom(0, 255);
      c.b = ofRandom(0, 255);
      c.a = ofRandom(0, 255);
    }

    Zukei(float x, float y){
      speed = 1.0;
      this->x = x;
      this->y = y;
      c.r = ofRandom(0, 255);
      c.g = ofRandom(0, 255);
      c.b = ofRandom(0, 255);
      c.a = ofRandom(0, 255);
    }

    void ten(){
      ofSetColor(c.r, c.g, c.b , c.a);
      ofRect(x, y, 10, 10);
    }

    void ten(float x, float y){
      this->x = x;
      this->y = y;
      ofSetColor(c.r, c.g, c.b, c.a);
```

```
      ofRect(x, y, 1, 1);
   }

   void moveH(){
      x = x + speed;
      if( x > ofGetWidth()){
         speed = speed * -1.0;
      }
   }
};


Sikaku sk;

void testApp::update(){            // 描画を伴わない変更はupdate() でも良い
   sk.moveH();                     // skオブジェクトの座標を変更
}

void testApp:draw(){
   sk.paint();                     // skオブジェクトの描画
}
```

--

授業内課題2：
同じようにして垂直に移動するメソッド moveV() を作成する。

課題：

Zukeiクラス、Sikakuクラスを変更して以下の機能を追加する。

（1）このままでは、反射するときに壁にめり込んでしまう。
　　四角の大きさを考慮することで、めり込まずに反射できるように、
　　SikakuクラスのmoveH(), moveV()を書き直す。

（2）移動のスピードを変更するためのメソッド（関数） setSpeed(int) を追加する。

（3）以上の機能を追加した上で、配列を使って１０個の点と２０個の四角が動き回る
　　プログラムを作成する。
　　オブジェクトを作成したあとに、setup()内で、speedを乱数で決めたり、
　　draw()内でフレーム数に応じて、縦や横の動きに変化が出るようにすると良い。