

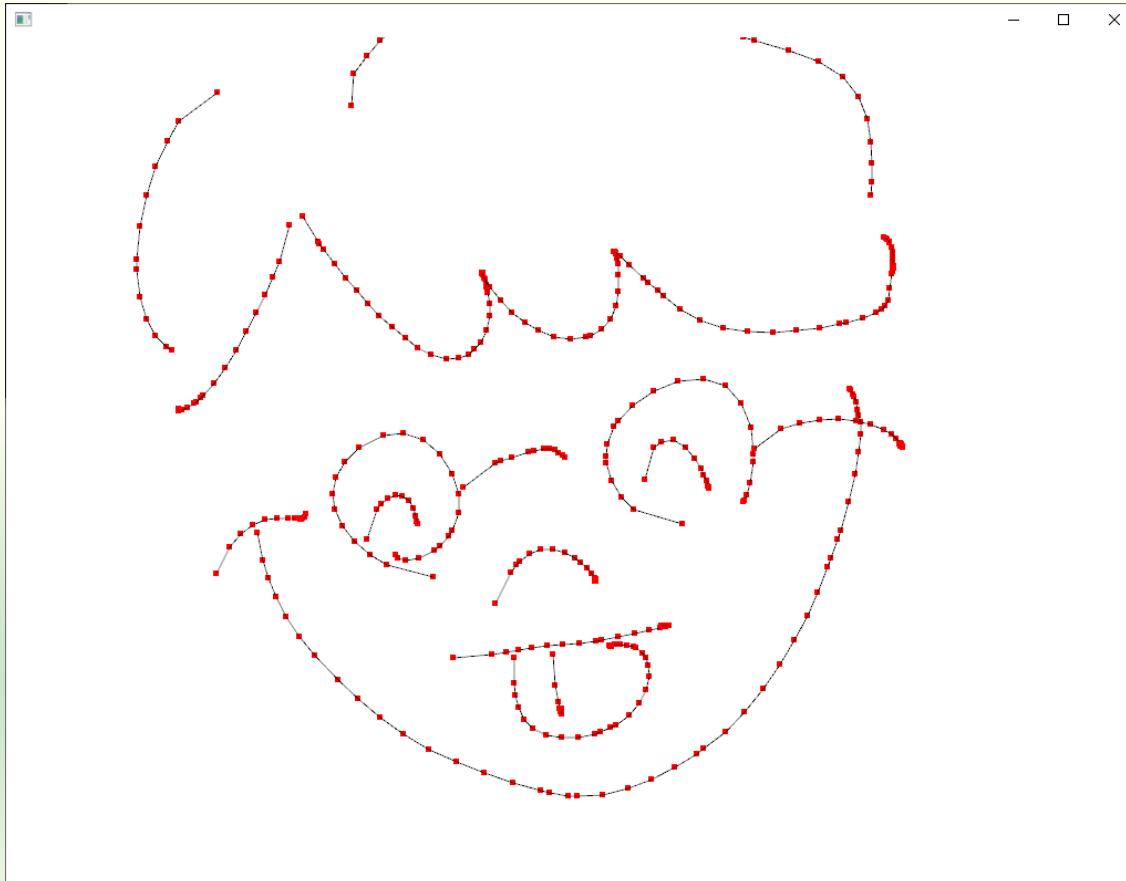


# メディアプログラミング演習

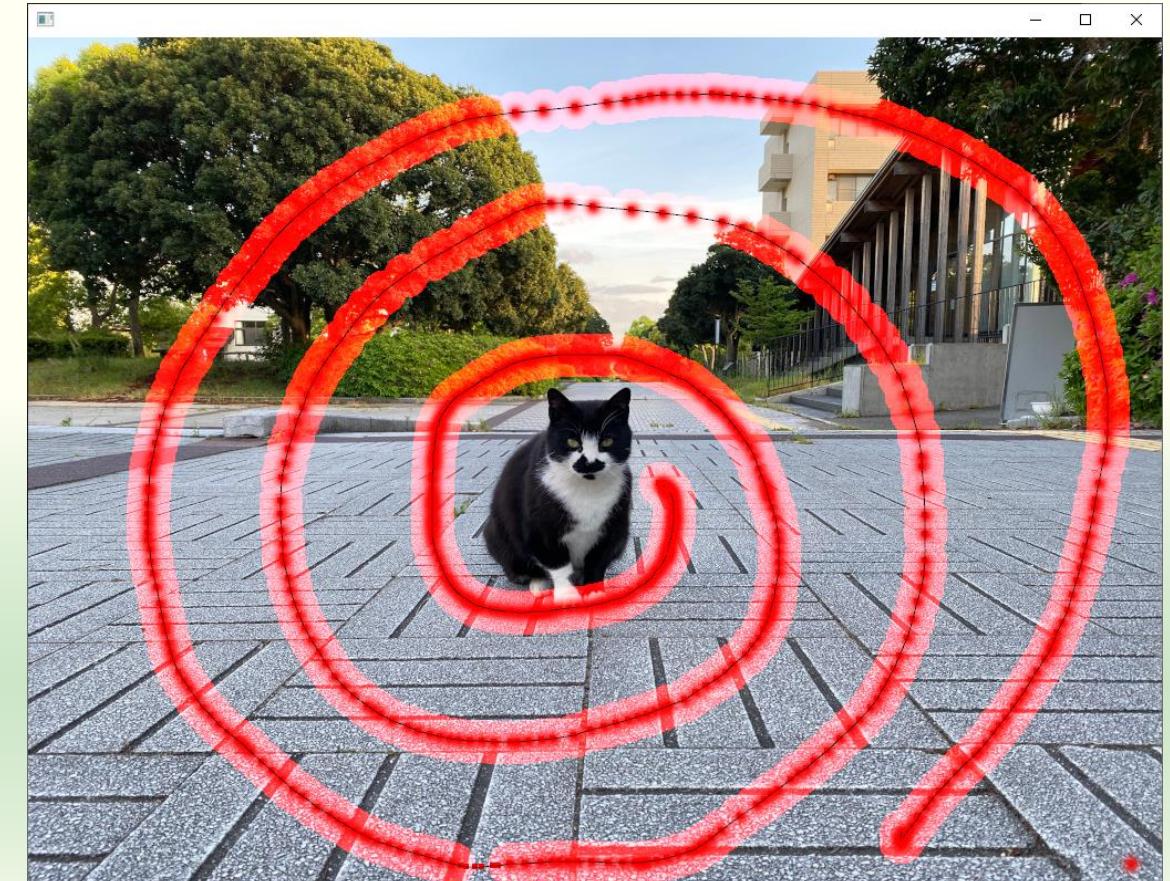
第4回

# 本日は作図・作画っぽいアプリの作成

## ドロー風



## ■ ペイント風



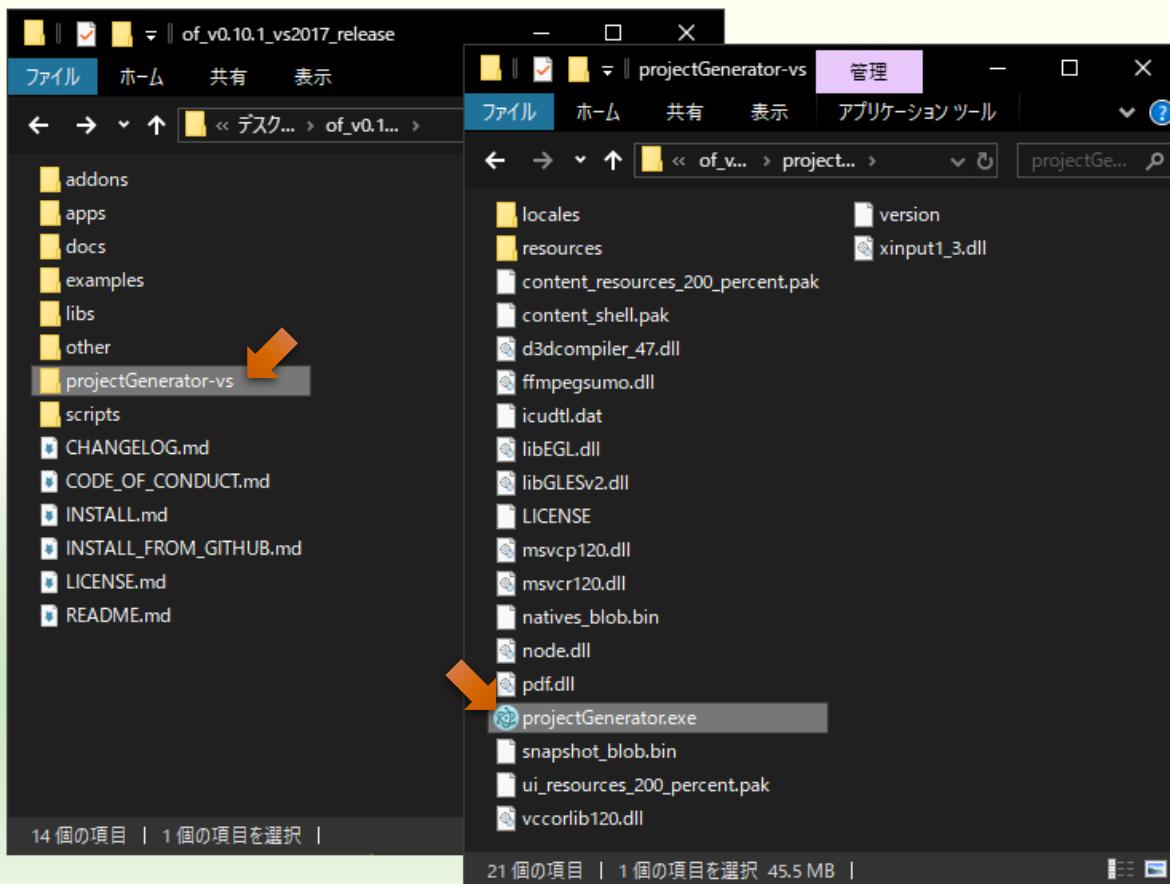


# 準備

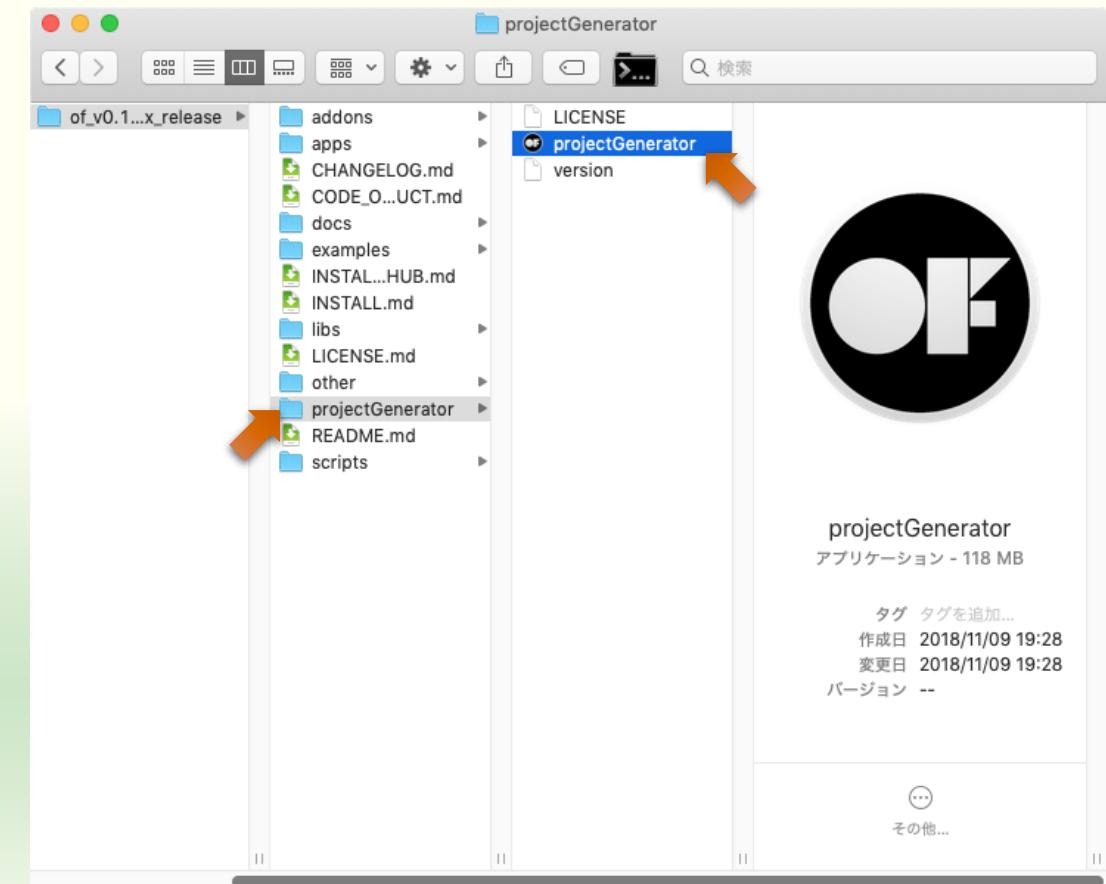
プロジェクトの作成

# projectGenerator を起動する

## windows 版のパッケージ



## macos 版のパッケージ



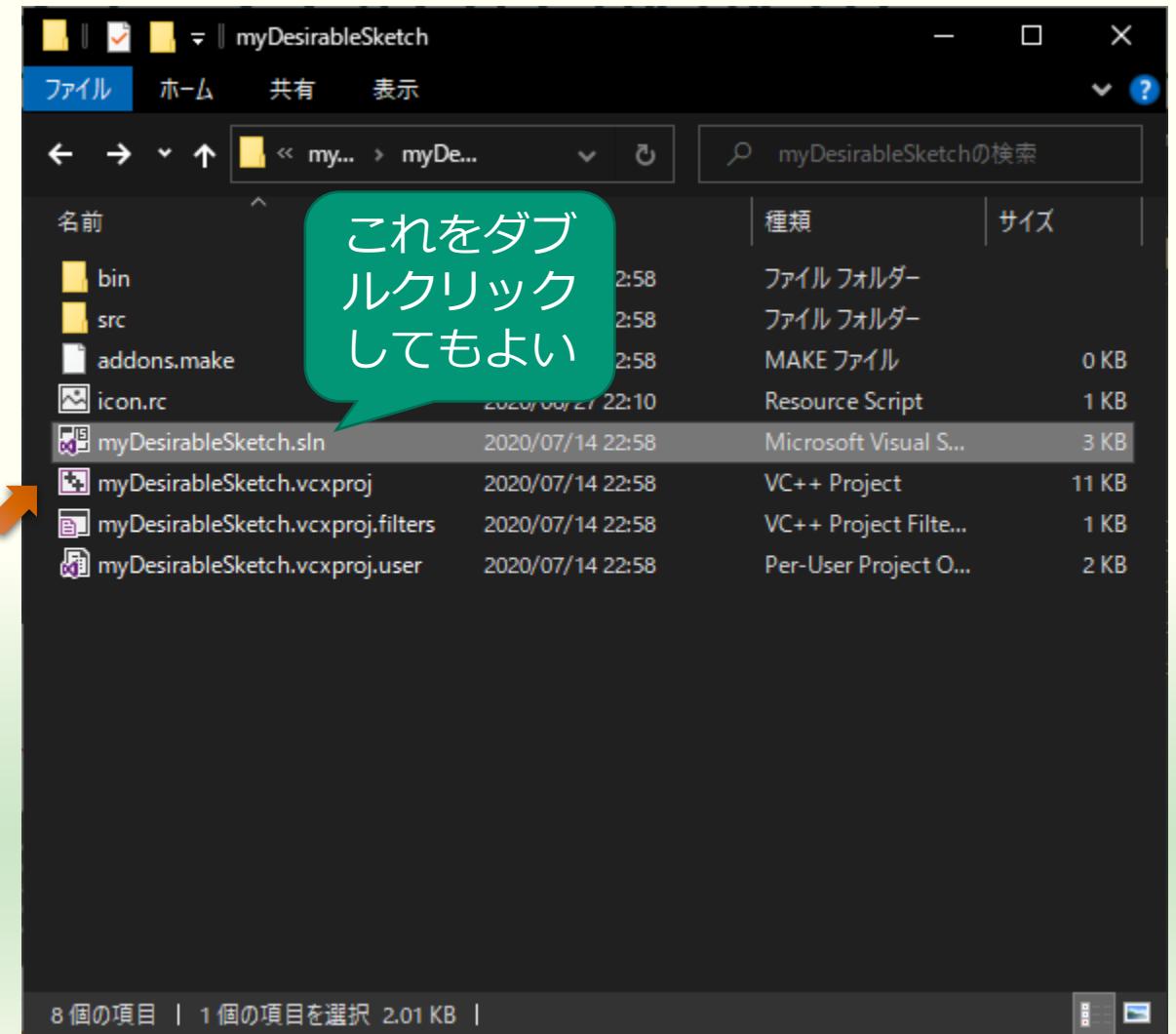
# 空のプロジェクトの作成



- Project name:
  - 作成するプロジェクト（プログラム）の名前
- Project path:
  - 作成するプロジェクトのファイルを置く場所
  - openFrameworks のパッケージを開いた場所の中の apps¥myApps

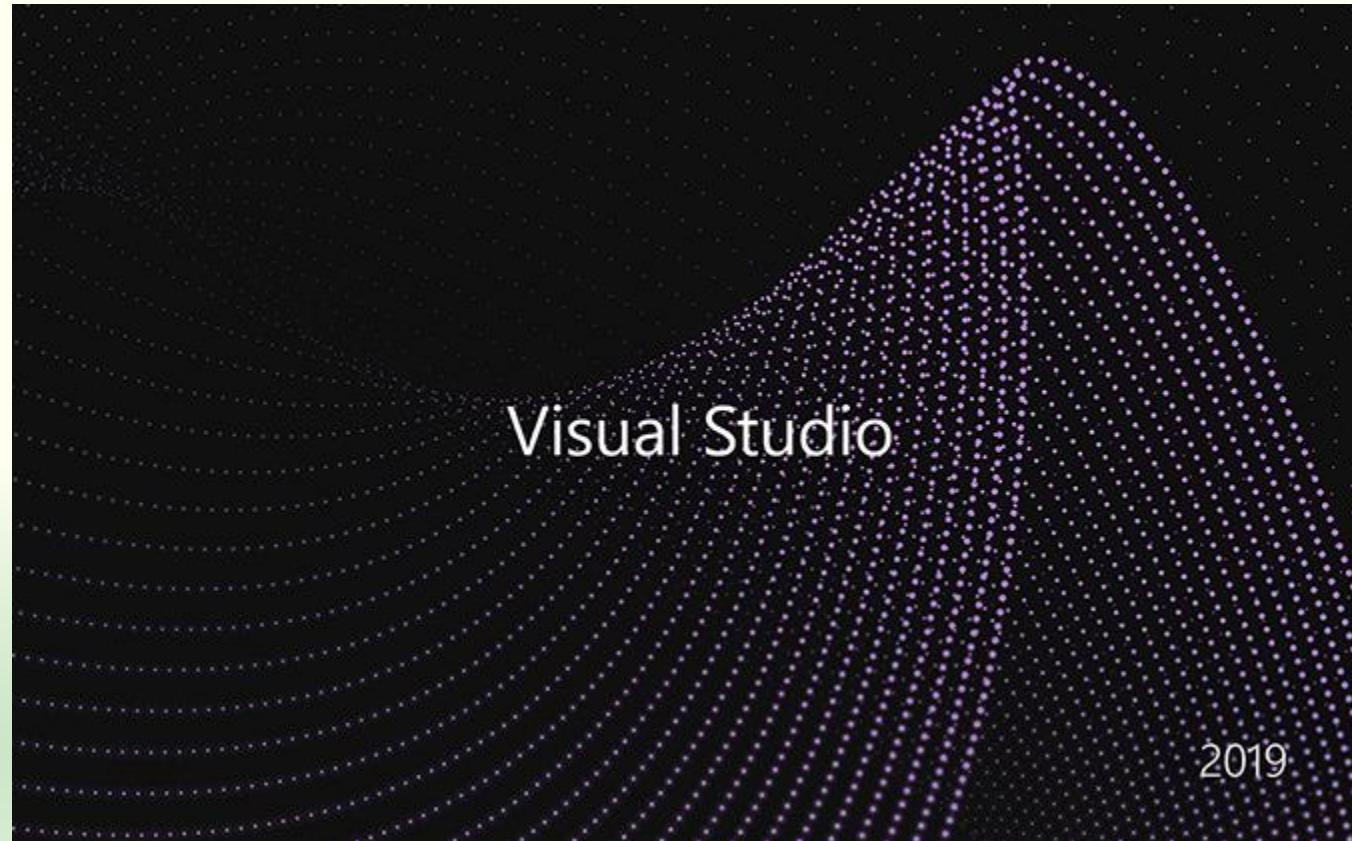


# プロジェクトの作成成功

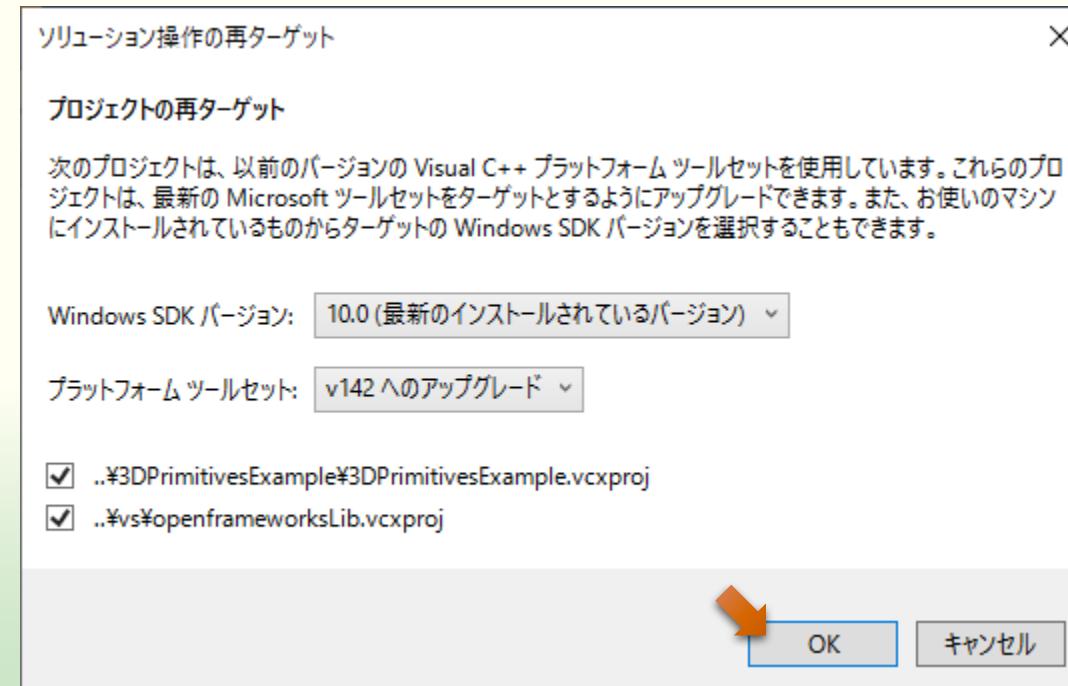


# Visual Studio が起動する

---

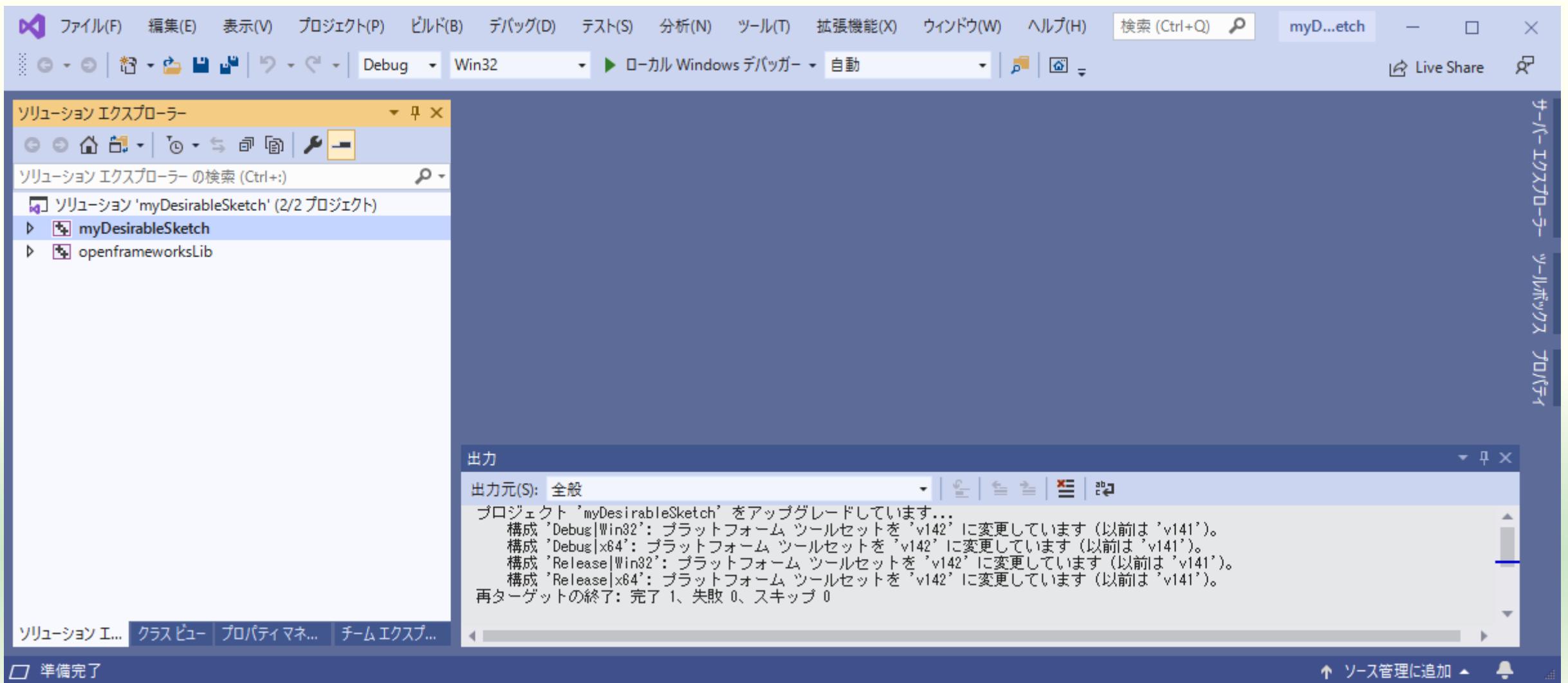


# ソリューションの再ターゲット



Visual Studio は頻繁に更新しているので皆さんのがお使いの Visual Studio SDK のバージョンと合わない場合がある

# Visual Studio 起動

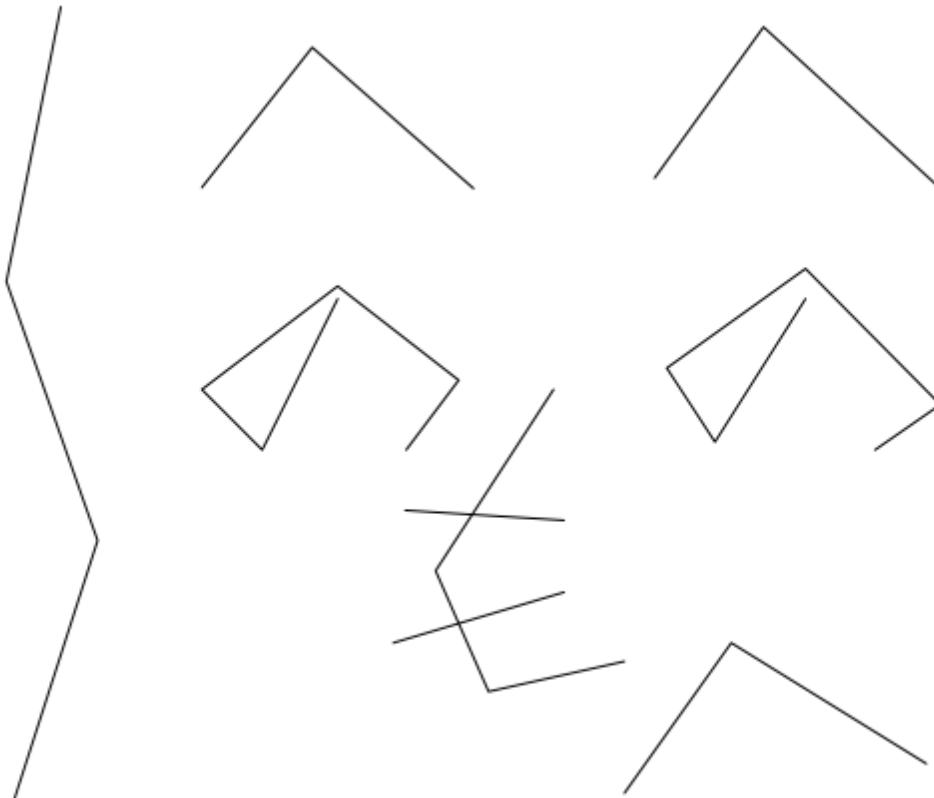




# マウスによる線の描画

ドローツールっぽいもの

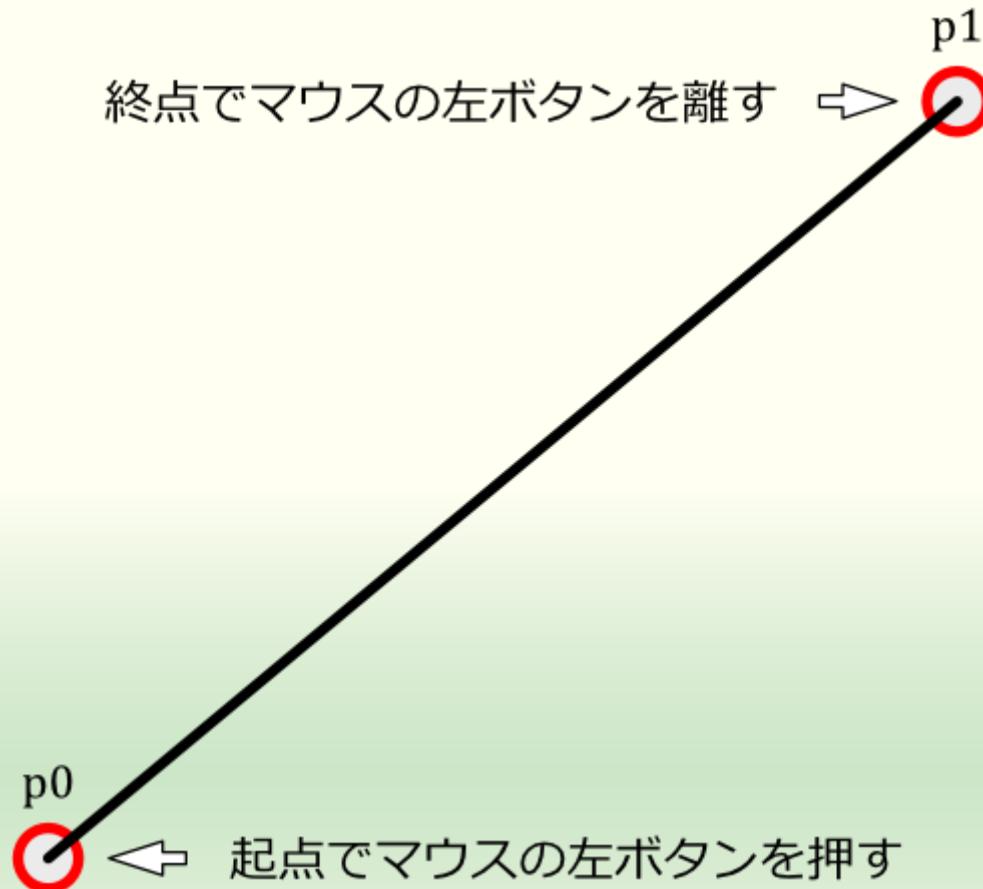
# ウィンドウ上にマウスで折れ線を描く



## ■ 仕様

- マウスの左ボタンをクリックしたところを折れ線で結ぶ
- マウスの右ボタンをクリックしたところで折れ線は終わる
- 次に左ボタンをクリックしたら新しい折れ線を描き始める
- 背景色は白
- 折れ線の色は黒
- 話の都合上 ofPath や ofPolyline などは知ってても使わない

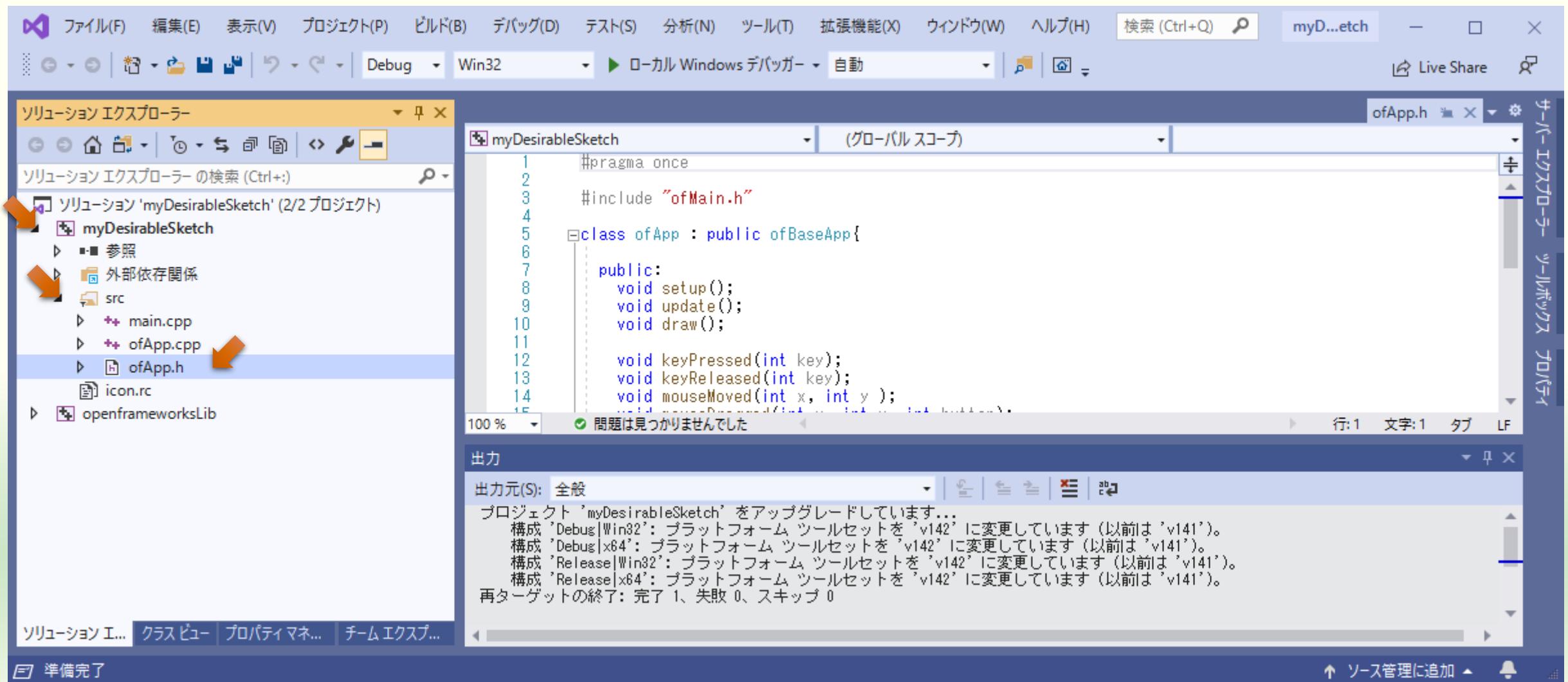
# 最初に 2 点間に線分を引くことを考える



- 起点でマウスの左ボタンを押す
  - 押した位置を  $p_0$  とする
- 終点でマウスの左ボタンを離す
  - 離した位置を  $p_1$  とする
- $p_0$  から  $p_1$  に線分を描く



## ofApp.h を開く



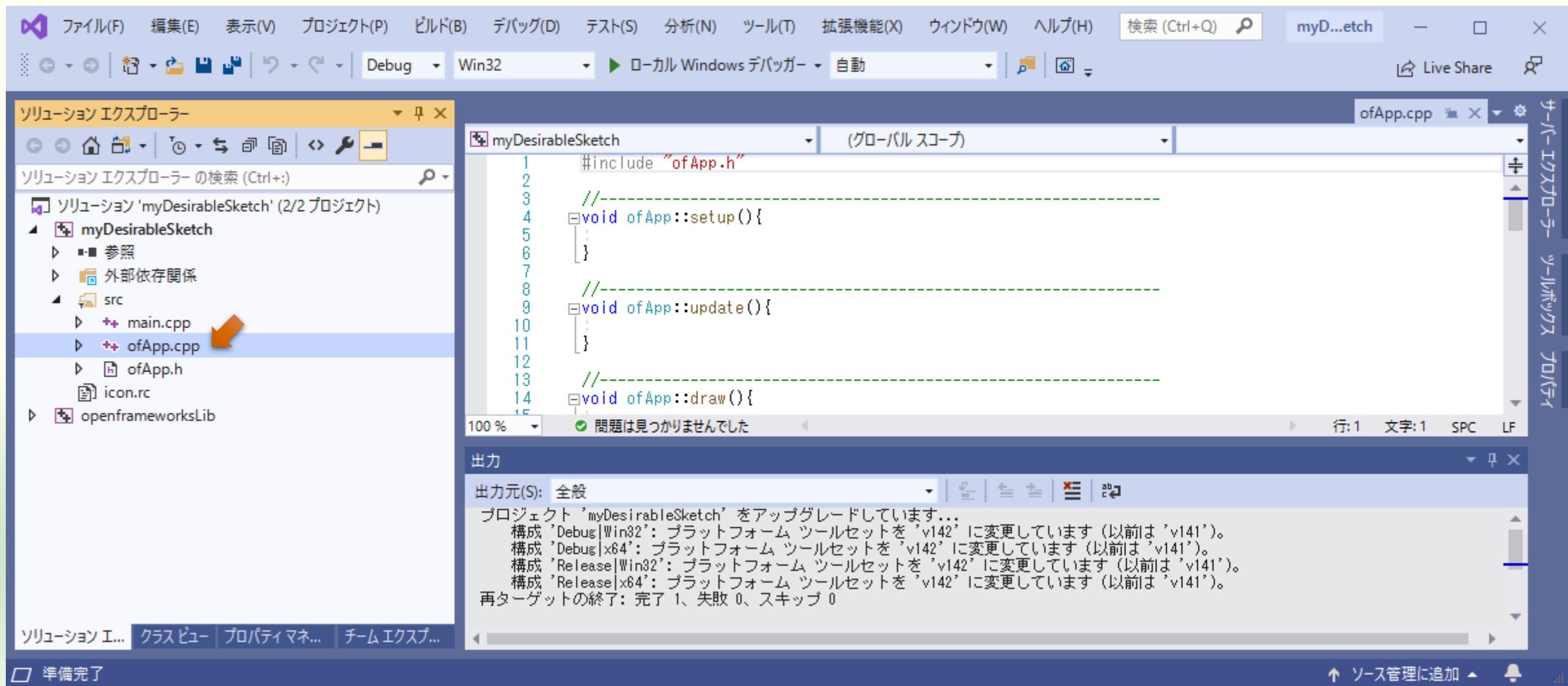
# ofApp クラスにメンバ変数 p0, p1 追加する

```
#pragma once  
  
#include "ofMain.h"  
  
using namespace glm;  
  
class ofApp : public ofBaseApp{  
    vec2 p0, p1;  
  
public:  
    void setup();  
    void update();  
    void draw();  
  
    void keyPressed(int key);  
    void keyReleased(int key);  
    void mouseMoved(int x, int y);  
    (以下略)
```

- p0, p1 は glm::vec2 クラスにする
  - “glm::” という名前空間の指定を省略するために using namespace glm; を入れておく



# ofApp.cpp を開く



# p0 と p1 にマウスの現在位置を代入する

```
//-----  
void ofApp::mouseDragged(int x, int y, int button){  
}  
  
//-----  
void ofApp::mousePressed(int x, int y, int button){  
    if (button == 0){  
        p0 = vec2{ x, y };  
    }  
}  
  
//-----  
void ofApp::mouseReleased(int x, int y, int button){  
    if (button == 0){  
        p1 = vec2{ x, y };  
    }  
}
```

- **mousePressed(x, y, button)**
  - マウスのボタンを押したときに実行される
  - もし押されたボタン button が 0 (左) なら起点の位置 p0 にマウスの現在位置 (x, y) を代入する
- **mouseReleased(x, y, button)**
  - マウスのボタンを離したときに実行される
  - もし離したボタン button が 0 (左) なら終点の位置 p1 にマウスの現在位置 (x, y) を代入する

# p0 から p1 に線分を描く

```
#include "ofApp.h"

//-----
void ofApp::setup(){
    ofBackground(255, 255, 255);
}

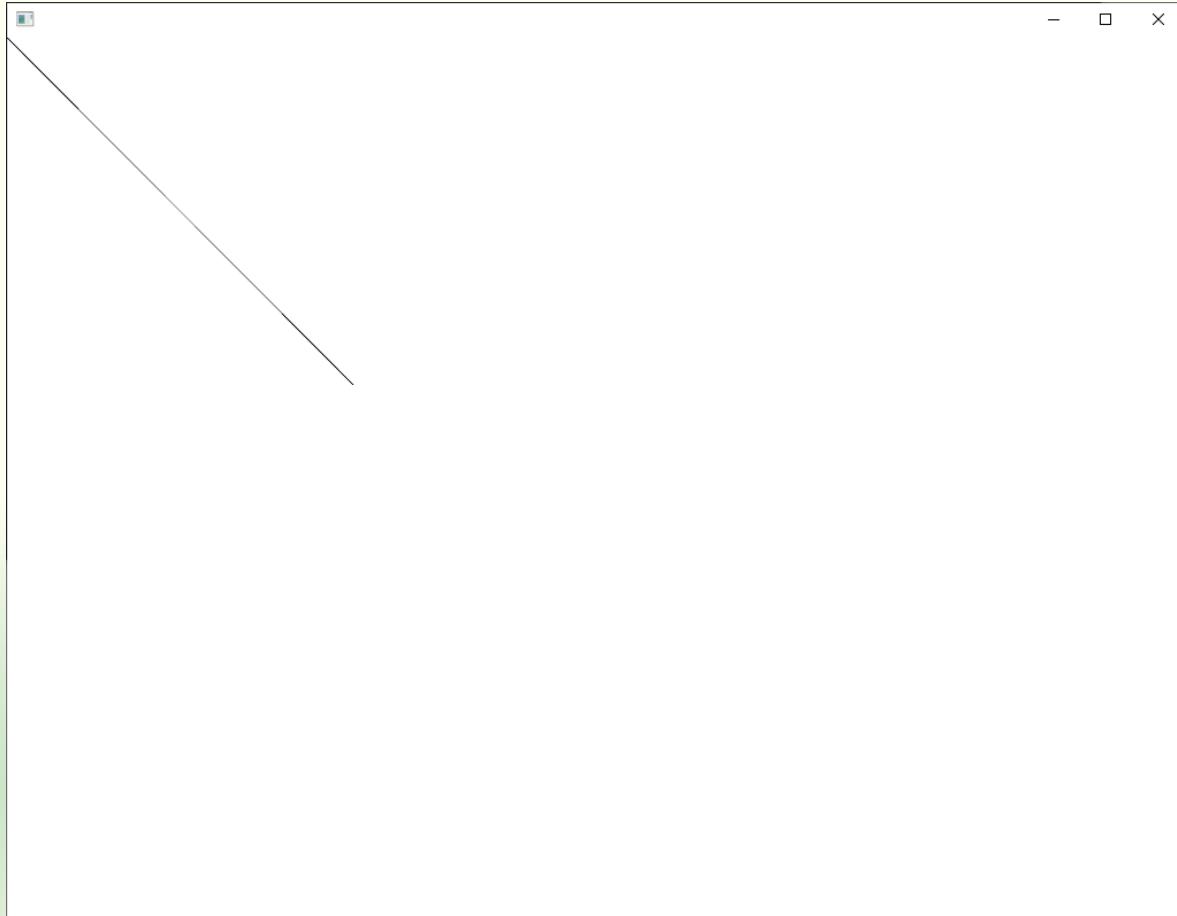
//-----
void ofApp::update(){

}

//-----
void ofApp::draw(){
    ofSetColor(0, 0, 0);
    ofDrawLine(p0, p1);
}
```

- setup() で背景色を白にする
  - ofBackground(int r, int g, int b, int a = 255);
    - r = g = b = 255 は白, a (不透明度) は省略すると 255 (不透明)
- draw() で黒い線を描く
  - ofSetColor(int r, int g, int b),  
ofSetColor(int r, int g, int b, int a)
    - r = g = b = 0 は黒, 引数が 3 つのときは不透明度 a に 255 が設定される
  - drawLine(vec2 p0, vec2 p1)
    - p0 から p1 に線分を描く

# ボタンを離す前に線が引かれる



- 左ボタンを押した瞬間に原点から線が引かれてしまう
  - $p0, p1$  の初期値は  $\text{vec2}\{ 0, 0 \}$  すなわち原点なので
  - 左ボタンを押して  $p0$  に値を入れたことで  $p1$  の原点まで線が引かれてしまう
- 実は何もしない状態でも原点に点を描いている
  - $p0, p1$  が初期値の  $\text{vec2}\{ 0, 0 \}$  のままだから

# 原点までの線が出ないようにする

```
//-----
void ofApp::mouseDragged(int x, int y, int button){}

//-----
void ofApp::mousePressed(int x, int y, int button){
    if (button == 0){
        p0 = p1 = vec2{ x, y };
    }
}

//-----
void ofApp::mouseReleased(int x, int y, int button){
    if (button == 0){
        p1 = vec2{ x, y };
    }
}
```

- `mousePressed(x, y, button)`
  - マウスの左ボタンを押したときに終点の `p1` にもボタンを押した位置を入れてれば原点までの線は描かれない
  - 代入 = は**右から順番に実行される**

`p1 = vec2{ x, y }`

`p0 = p1`

# ドラッグ中には何も出ない



- ドラッグ中は  $p_0$  と  $p_1$  が等しい
  - マウスの左ボタンを押した位置が入っている
- ボタンを離すと  $p_1$  に終点の位置が入る
  - 線が描かれる



# ドラッグ中に線を引く

```
//-----
void ofApp::mouseDragged(int x, int y, int button){
    if (button == 0){
        p1 = vec2{ x, y };
    }
}

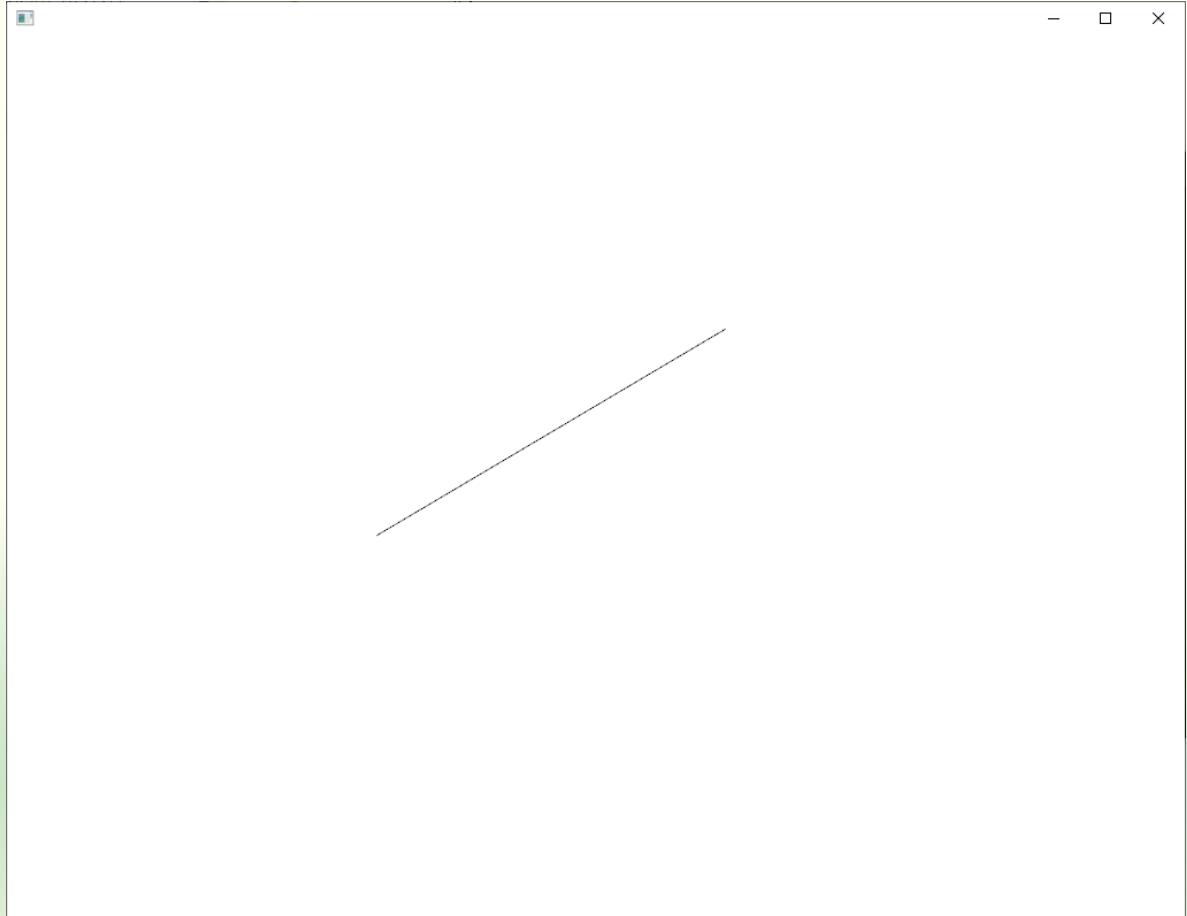
//-----
void ofApp::mousePressed(int x, int y, int button){
    if (button == 0){
        p0 = p1 = vec2{ x, y };
    }
}

//-----
void ofApp::mouseReleased(int x, int y, int button){
    if (button == 0){
        p1 = vec2{ x, y };
    }
}
```

- **mouseDragged(x, y, button)**
  - マウスのドラッグ中に実行される
  - もしドラッグ中に押されているマウスのボタン button が 0 (左) なら終点の位置 p1 にマウスの現在位置 (x, y) を代入する



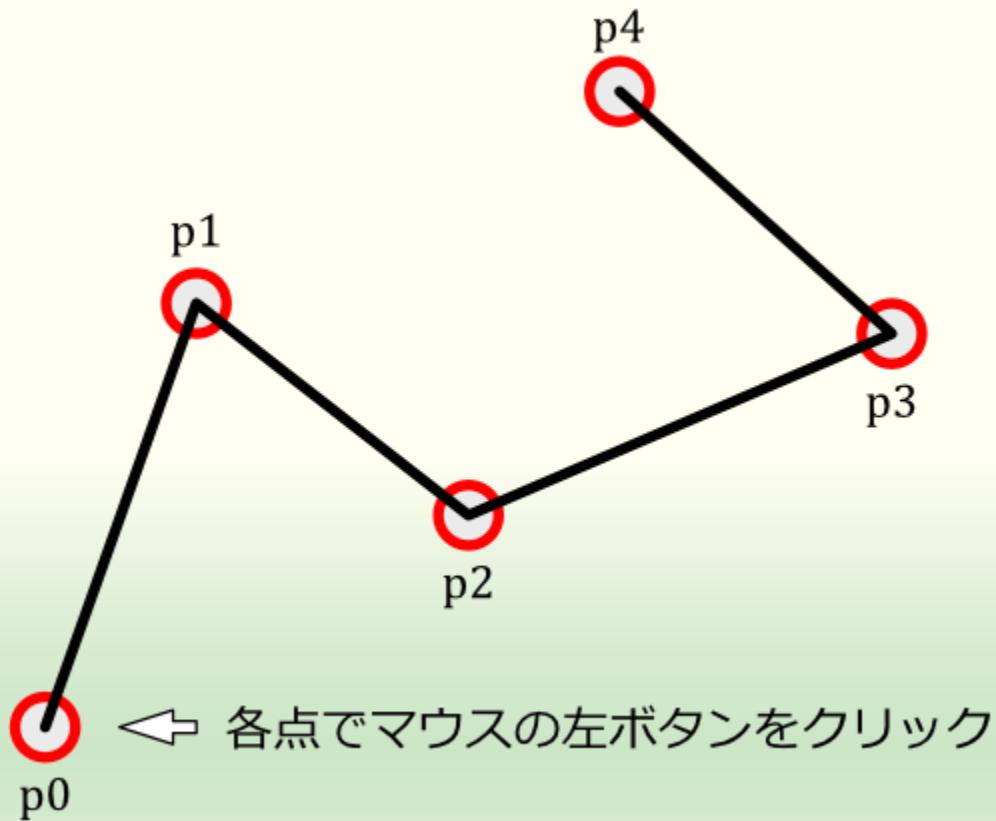
# ドラッグ中も線が引かれる



- マウスのボタンを離すと線が固定される
- しかしそれにマウスのボタンを押すと消える



# 折れ線を描くことを考える



- 左ボタンをクリックするたびにマウスの位置を記録する
  - ドラッグはしない
  - std::vector を使う



# ofApp.h でメンバ変数を vector にする

```
#pragma once  
  
#include "ofMain.h"  
  
using namespace glm;  
  
class ofApp : public ofBaseApp{  
    vector<vec2> points;  
  
public:  
    void setup();  
    void update();  
    void draw();  
  
    void keyPressed(int key);  
    void keyReleased(int key);  
    void mouseMoved(int x, int y);  
    (以下略)
```

- 変数名は points とかにする



# points は空の vector

```
points.size() == 0  
points.begin() == points.end()
```



# ofApp.cpp のマウス操作の処理を修正する

```
//-----
void ofApp::mouseDragged(int x, int y, int button){
    if (button == 0){
        p1 = vec2{ x, y };
    }
}

//-----
void ofApp::mousePressed(int x, int y, int button){
    if (button == 0){
        points.emplace_back(x, y);
    }
}

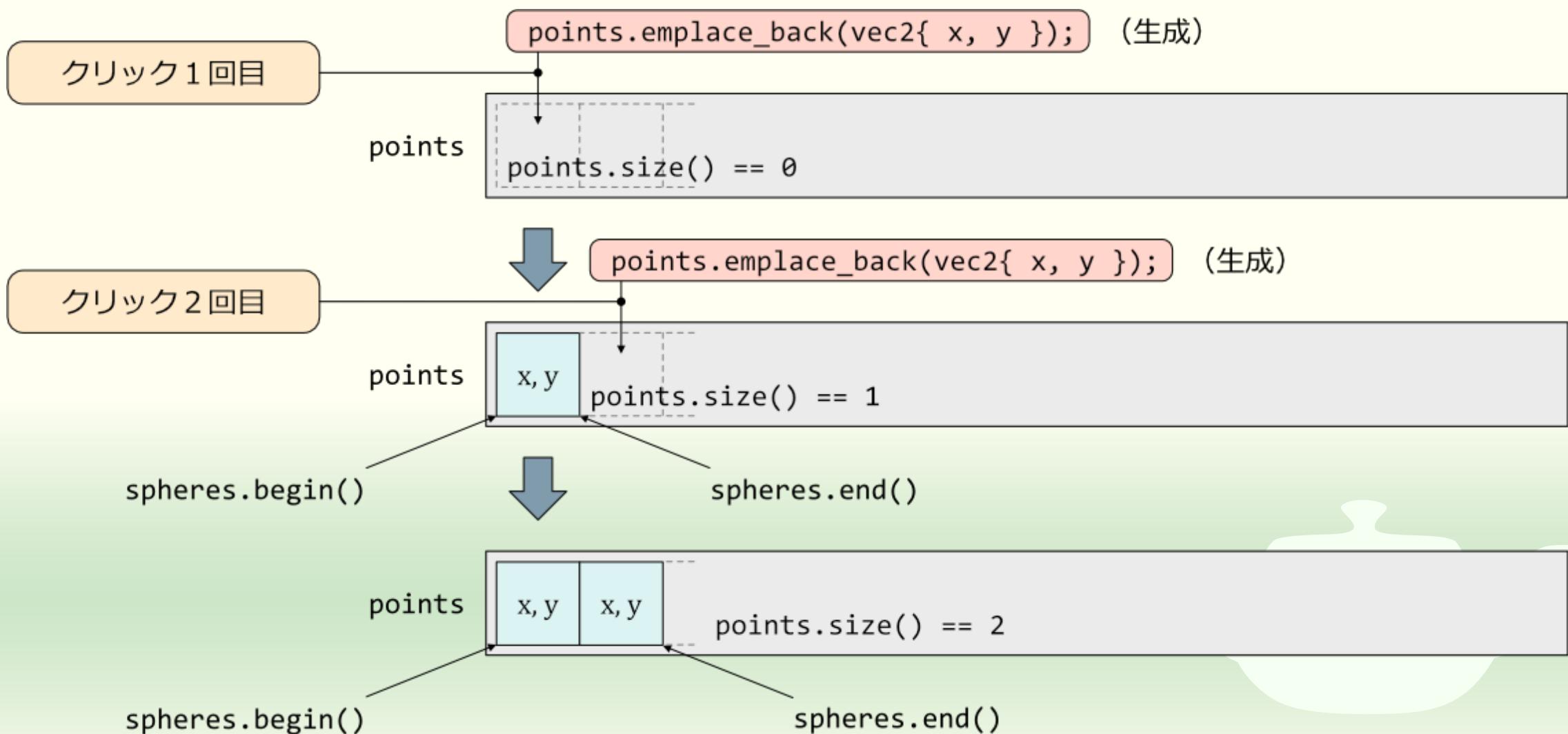
//-----
void ofApp::mouseReleased(int x, int y, int button){
    if (button == 0){
        p1 = vec2{ x, y };
    }
}
```

削除

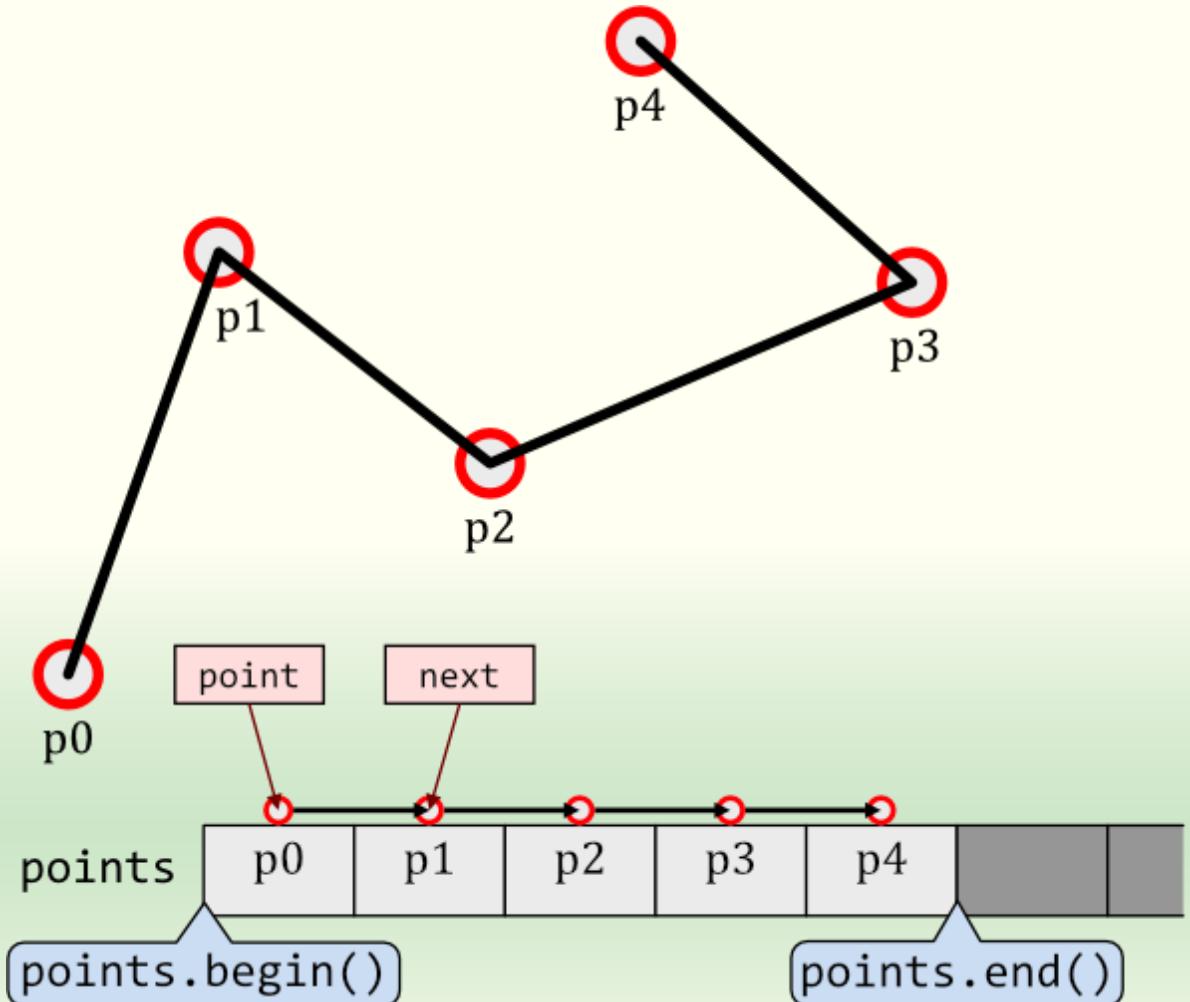
削除

- **mousePressed(x, y, button)**
  - マウスのボタンが押されたとき押されたボタン button が 0 (左) なら points にマウスの現在位置 (x, y) を追加する
- **mouseDragged(x, y, button)**
  - マウスのドラッグ中は何もしない
- **mouseReleased(x, y, button)**
  - マウスのボタンを離したときも何もしない

# emplace\_back() で vector の末尾に要素を生成

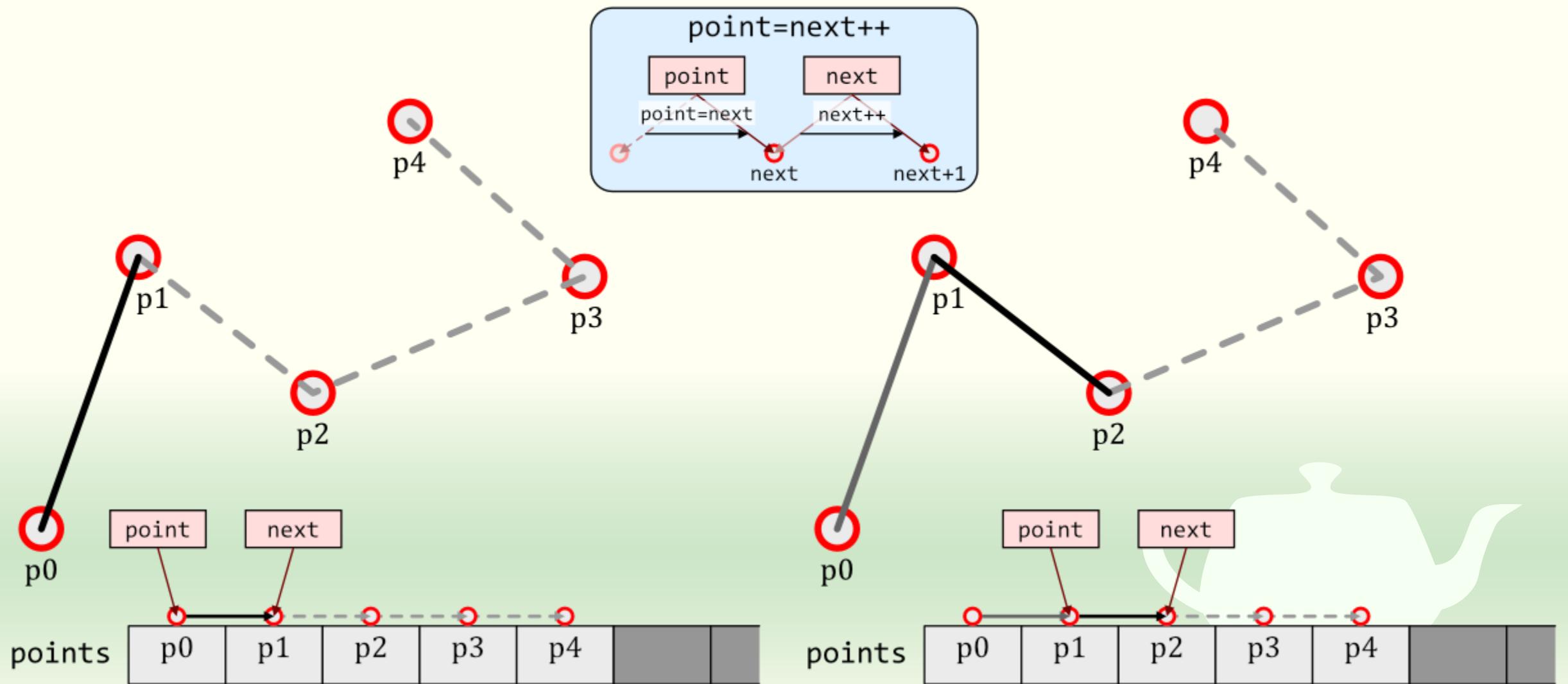


# クリックしたところを線分で結ぶ



- クリックした位置は `points` という `vector` に  $p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow \dots$  という順に入っている
- 最初 `point` に  $p_0$  の場所、`next` に  $p_1$  の場所を入れておく
- `point` が指すデータから `next` が指すデータに線分を引く
- `point` に `next` を代入し、`next` を次のデータに進める
- `next` がデータの終わりを超えていたら終わる

# 折れ線の描画



# vector が空でなければ最初のデータを取り出す

```
#include "ofApp.h"

//-----
void ofApp::setup(){
    ofBackground(255, 255, 255);
}

//-----
void ofApp::update(){

}

//-----
void ofApp::draw(){
    if (points.empty()) return;
    auto point{ points.begin() };
    (次のページに続く)
}
```

- if (points.empty()) return;
  - points にデータが入っていないければ何もせずに戻る
    - empty() メソッドは vector の中に何も入っていないければ true になる
    - if の {} 内に ; (セミコロン) がひとつしかないときは {} を省略できる
- auto point{ points.begin() };
  - points の最初のデータ (p0) の場所を point に代入する
    - begin() メソッドは vector の最初のデータの場所 (イテレータ) を取り出す
    - begin() メソッドは vector<vec2>::iterator というデータ型だが長いので point は auto を使って型推論して宣言している

# 2つずつ点を取り出して線分で結ぶ

(前のページの続き)

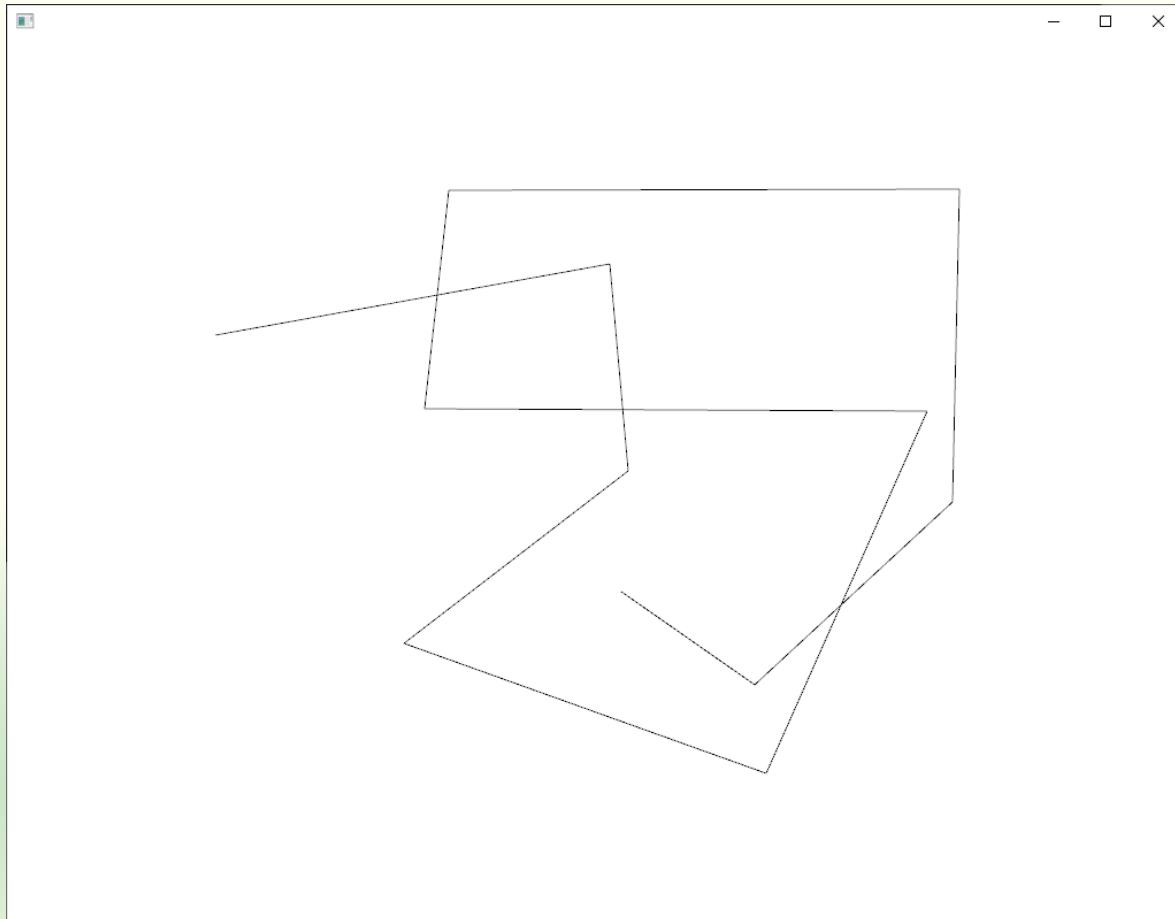
```
for (auto next = point + 1;  
    next != points.end(); point = next++){  
    ofSetColor(0, 0, 0);  
    ofDrawLine(*point, *next);  
}
```

間接参照演算子 (\*) を使って  
イテレータが指している  
データの実体を取り出す



- auto next = point + 1
  - point の次のデータ（最初は p1）の場所を next に代入する
    - イテレータに 1 を足すと次のデータの場所が得られる
- next != points.end()
  - next が points の最後のデータの次でなければ {} 内を実行する
    - end() メソッドは vector の最後のデータの次の場所を取り出す
- point = next++
  - point に next を代入した後に next が指す場所を次に進める

# 折れ線が描ける



- ひと続きの折れ線しか描けない

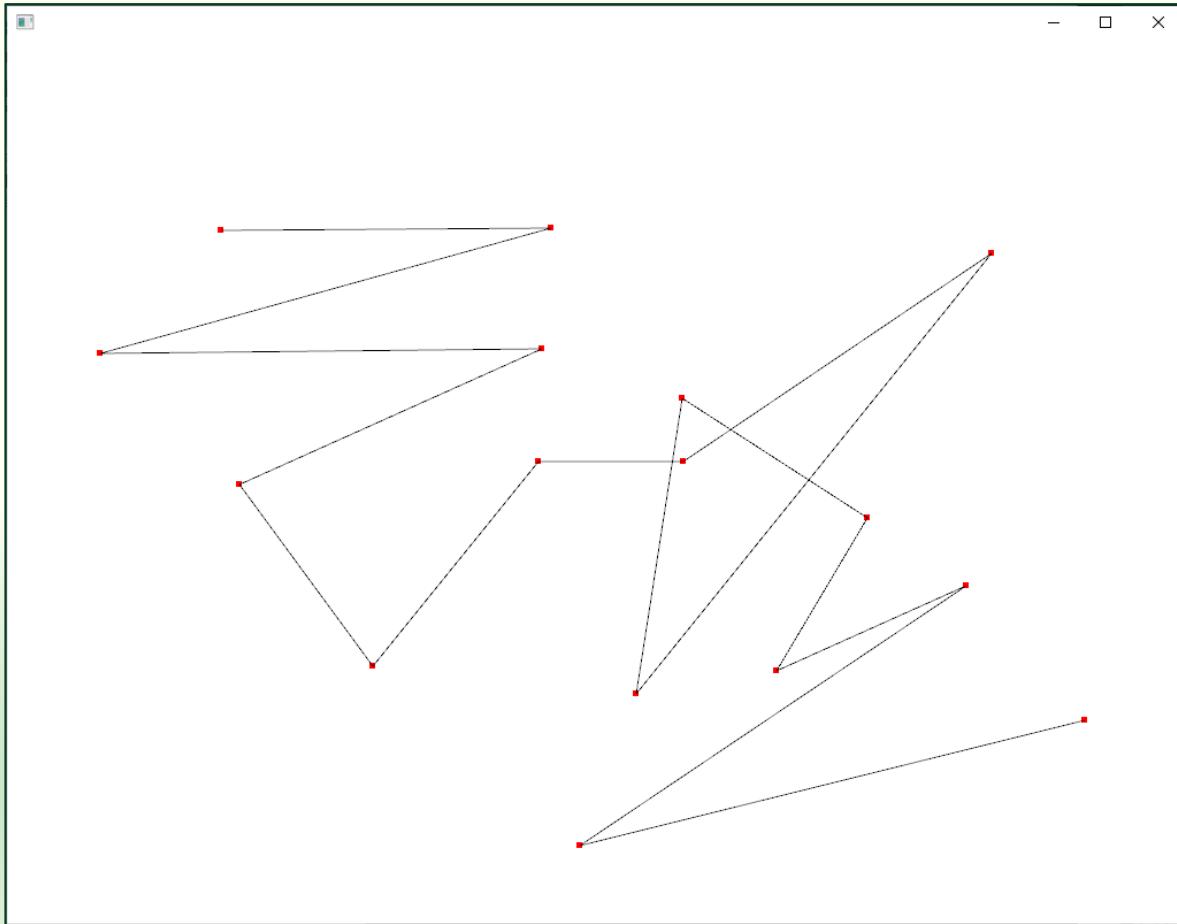




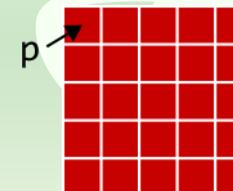
# 課題 4 – 1

クリックしたところに点を打つ

# クリックしたところに点を打つ



- クリックしたところに点を打つ  
ようにしてください
- `p` に縦横5ドットの赤い点を打つとしたら次のようにになります  
`ofSetColor(255, 0, 0);  
ofDrawRectangle(p, 5, 5);`
- `ofDrawRectangle()`は矩形を描く
- `p` は左上の位置で `vec2` 型の値



# 課題のアップロード

---

- 作成したプログラムの実行結果のスクリーンショットを撮つて **4-1.png** というファイル名で保存し、Moodle の第 4 回課題にアップロードしてください
  - 線が赤いドットの中央を通るよう工夫してみてください
  - 最後の点にも赤いドットを打つように工夫してみてください

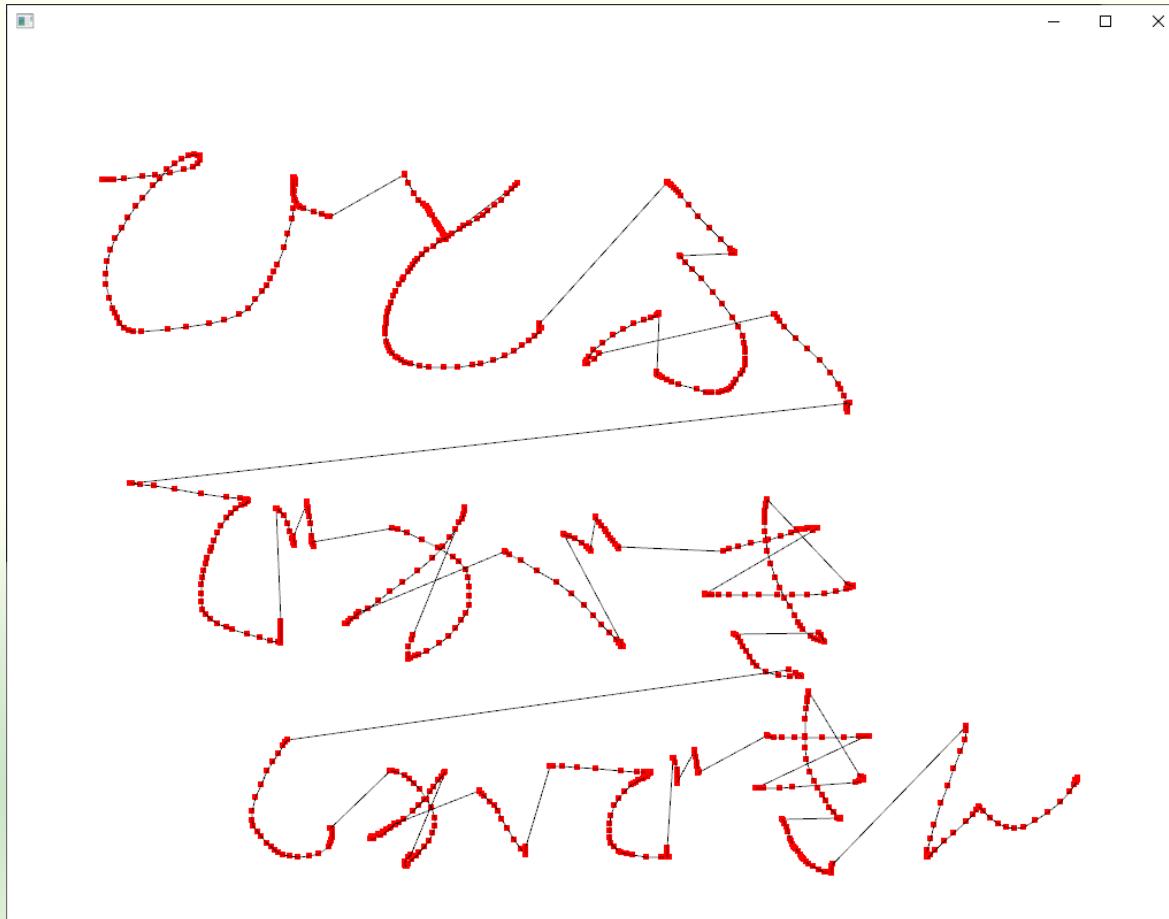




# 課題 4 – 2

ドラッグ中も線を引く

# ドラッグ中も線を引く



- マウスの左ボタンを押しながら  
ドラッグしているときも点を記  
録するようにしてください



# 課題のアップロード

---

- 作成したプログラムの実行結果のスクリーンショットを撮つて **4-2.png** というファイル名で保存し、Moodle の第 4 回課題にアップロードしてください

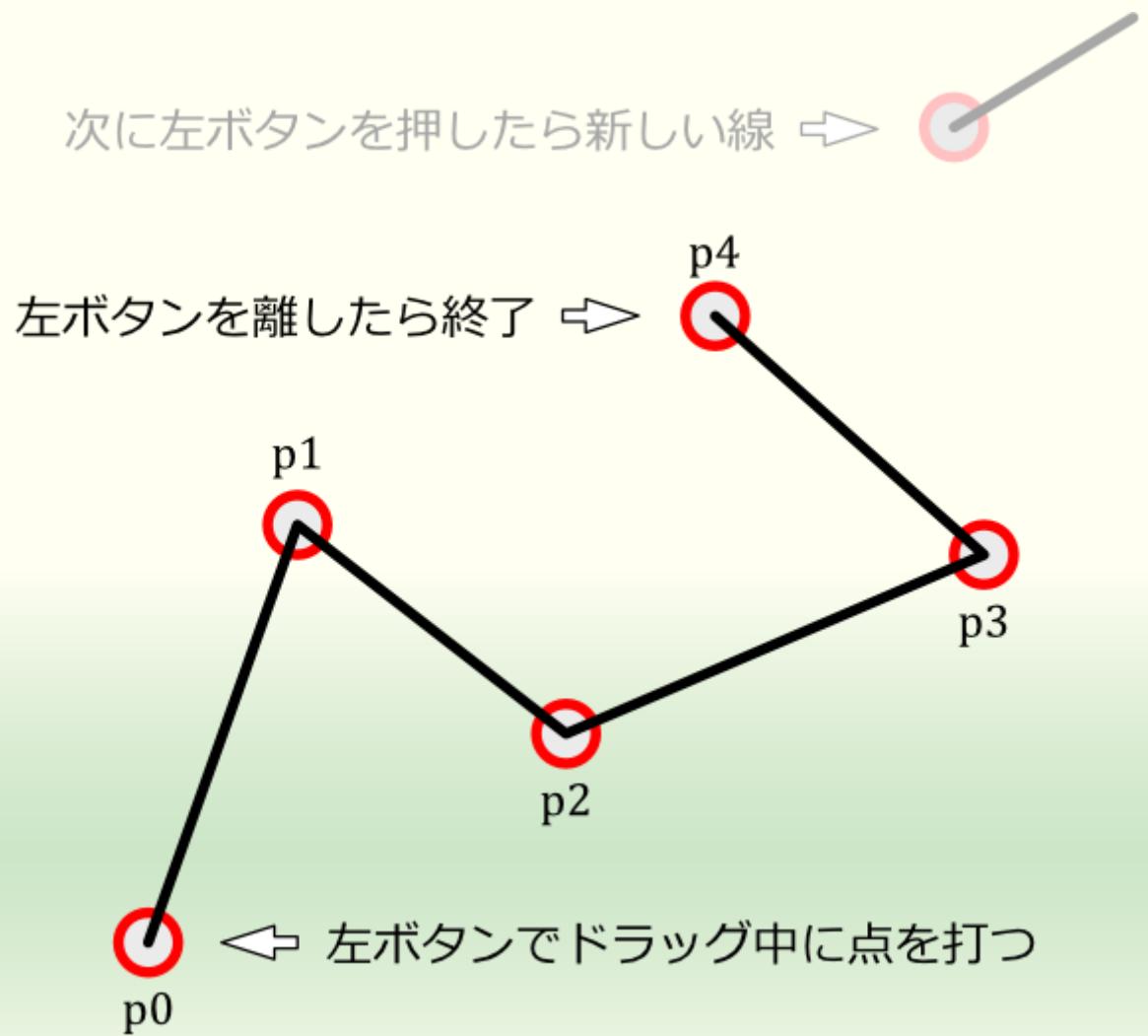




# 課題 4 – 3

複数の折れ線を描く

# 複数の折れ線を描く



- 左ボタンを押しながらマウスをドラッグしている間に点を打つ
- 左ボタンを離したら折れ線の作成を終わる
  - 次に左ボタンを押したら新しい折れ線を書き始める



# ofApp.h で折れ線の vector を作る

```
#pragma once

#include "ofMain.h"

using namespace glm;

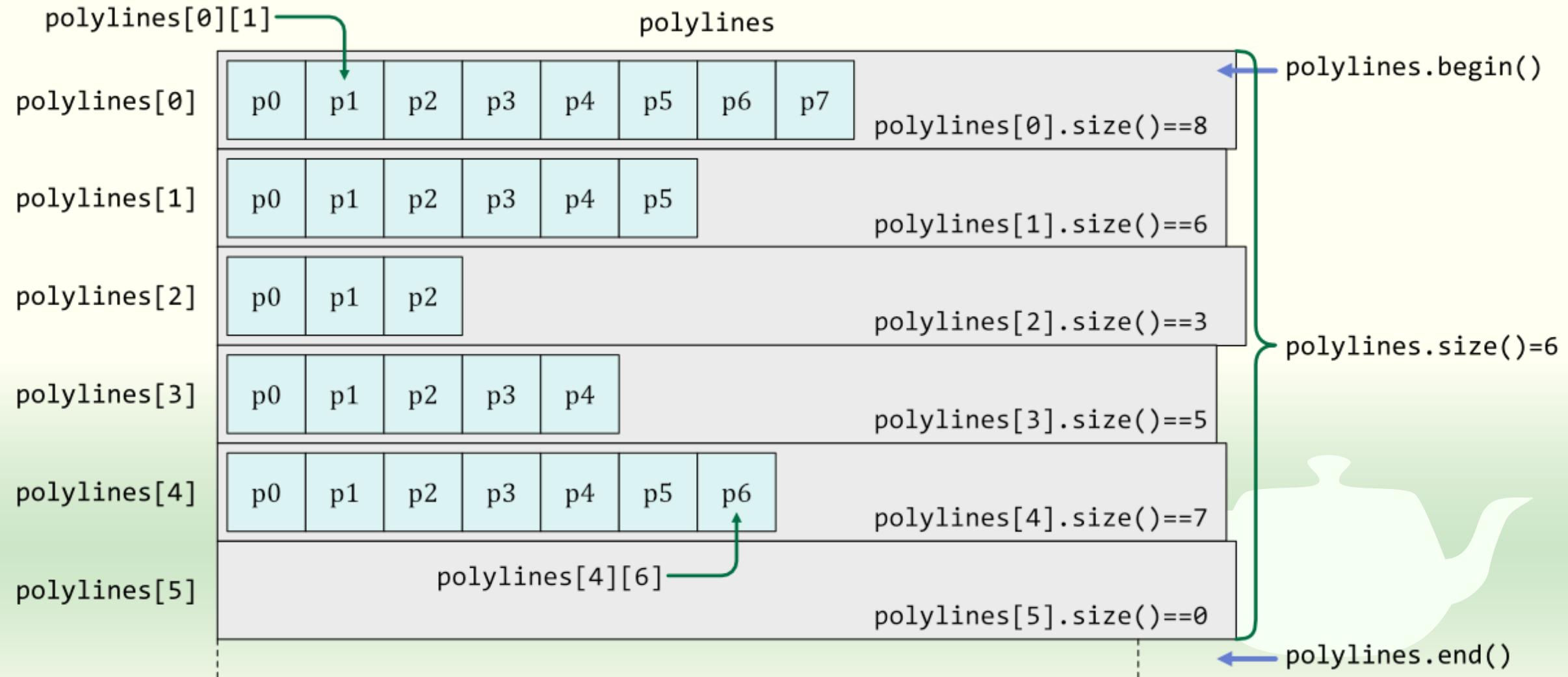
class ofApp : public ofBaseApp{
    vector<vector<vec2>> polylines;

public:
    void setup();
    void update();
    void draw();

    void keyPressed(int key);
    void keyReleased(int key);
    void mouseMoved(int x, int y);
    (以下略)
```

- **vector<vec2>**
  - 点のコンテナ (vec2) の vector
  - 折れ線のコンテナ
- **vector<vector<vec2>> polylines;**
  - 折れ線のコンテナ (vector<vec2>) の vector
- **vector の vector**
  - 一本の折れ線は点の vector
  - 複数の折れ線は折れ線の vector

# `vector<vector<vec2>> polylines;` のイメージ



# polylines に空のコンテナを追加する

```
#include "ofApp.h"

//-----
void ofApp::setup(){
    ofBackground(255, 255, 255);
    polylines.emplace_back();
}

//-----
void ofApp::update(){

}
```

(次のページに続く)

- polylines は空
  - 折れ線を格納する `vector<vec2>` のコンテナが一つもない
    - 点が追加できない
- 最初に polylines に `vector<vec2>` のコンテナを追加する
  - 内容は空
    - マウスのドラッグしたときに点のデータをこのコンテナに追加する
    - 点のデータは常に polylines の最後のコンテナに追加するようにする

# 全部の折れ線を描画する

```
//-----
void ofApp::draw(){
    for (auto &points : polylines){
        if (points.empty()) return;
        auto point{ points.begin() };
        for (auto next = point + 1;
            next != points.end(); point = next++){
            ofSetColor(0, 0, 0);
            ofDrawLine(*point, *next);
        }
    }
}
```

- `polylines` の一つ一つの要素（折れ線のコンテナ）を取り出す
- 取り出したコンテナに格納されている折れ線を描画する
- この処理を繰り返す

このプログラムには点を描く  
処理を含めていませんが点も  
描くようにしてください

# 左ボタンでドラッグした時に点を追加する

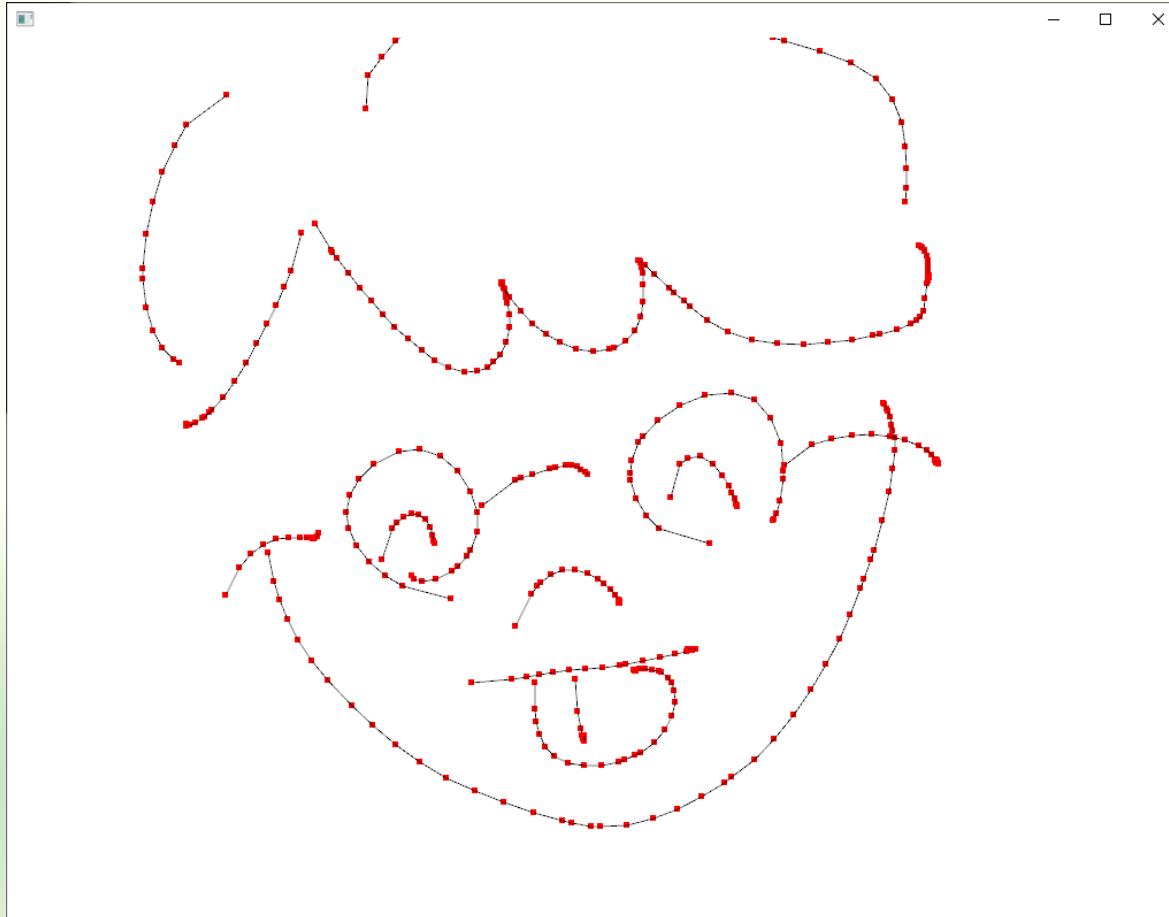
```
//-----
void ofApp::mouseDragged(int x, int y, int button){
    if (button == 0){
        auto &points{ polylines.back() };
        points.emplace_back(x, y);
    }
}

//-----
void ofApp::mousePressed(int x, int y, int button){
    if (button == 0){
        auto &points{ polylines.back() };
        points.emplace_back(x, y);
    }
}

//-----
void ofApp::mouseReleased(int x, int y, int button){
}
```

- `polylines` の最後の要素のコンテナを取り出す
  - 最初に一つだけ入れていた
  - この要素は空
- 取り出したコンテナに点を追加する
- `auto &points{ polylines.back() };`
  - 参照演算子 & が付いているので `points` は `polylines` の最後の要素そのものになる
  - コピーではない

# 複数の折れ線を描く



- 複数の折れ線を描くようにしなさい
  - マウスのドラッグを終了して左ボタンを離したら 1 本の線分の作成を終える
  - 次にマウスの左ボタンを押してドラッグを開始したら新しい折れ線を描く
- `polylines` の最後に空のコンテナを 1 つ追加するだけ

# 課題のアップロード

---

- 作成したプログラムの実行結果のスクリーンショットを撮つて **4-3.png** というファイル名で保存し、Moodle の第 4 回課題にアップロードしてください





# 画像データの変更

ペイントツールっぽいもの

# ofApp クラスに画像のメンバ変数を追加する

```
#pragma once

#include "ofMain.h"

using namespace glm;

class ofApp : public ofBaseApp {
    vector<vector<vec2>> polylines;
    ofImage image;

public:
    void setup();
    void update();
    void draw();

    void keyPressed(int key);
    void keyReleased(int key);
    void mouseMoved(int x, int y);
    (以下略)
```

## ■ ofImage

- 画像の読み込み、保存、描画を行うクラス
  - 画面に画像を描画する
  - 画像のピクセルデータを操作する
  - 画像ファイルを読み込む
  - OpenGL テクスチャを作成する



# ofApp.cpp の setup() で image に画像を読み込む

```
#include "ofApp.h"

//-----
void ofApp::setup(){
    ofBackground(255, 255, 255);
    polylines.emplace_back();
    image.load("image.jpg");
}
```

- `image.load("image.jpg");`
  - プロジェクトのフォルダの bin の data の中にある image.jpg という画像ファイルを image に読み込む
- "image.jpg" は画像ファイル名
  - JPEG, PNG, GIF 画像が読み込める

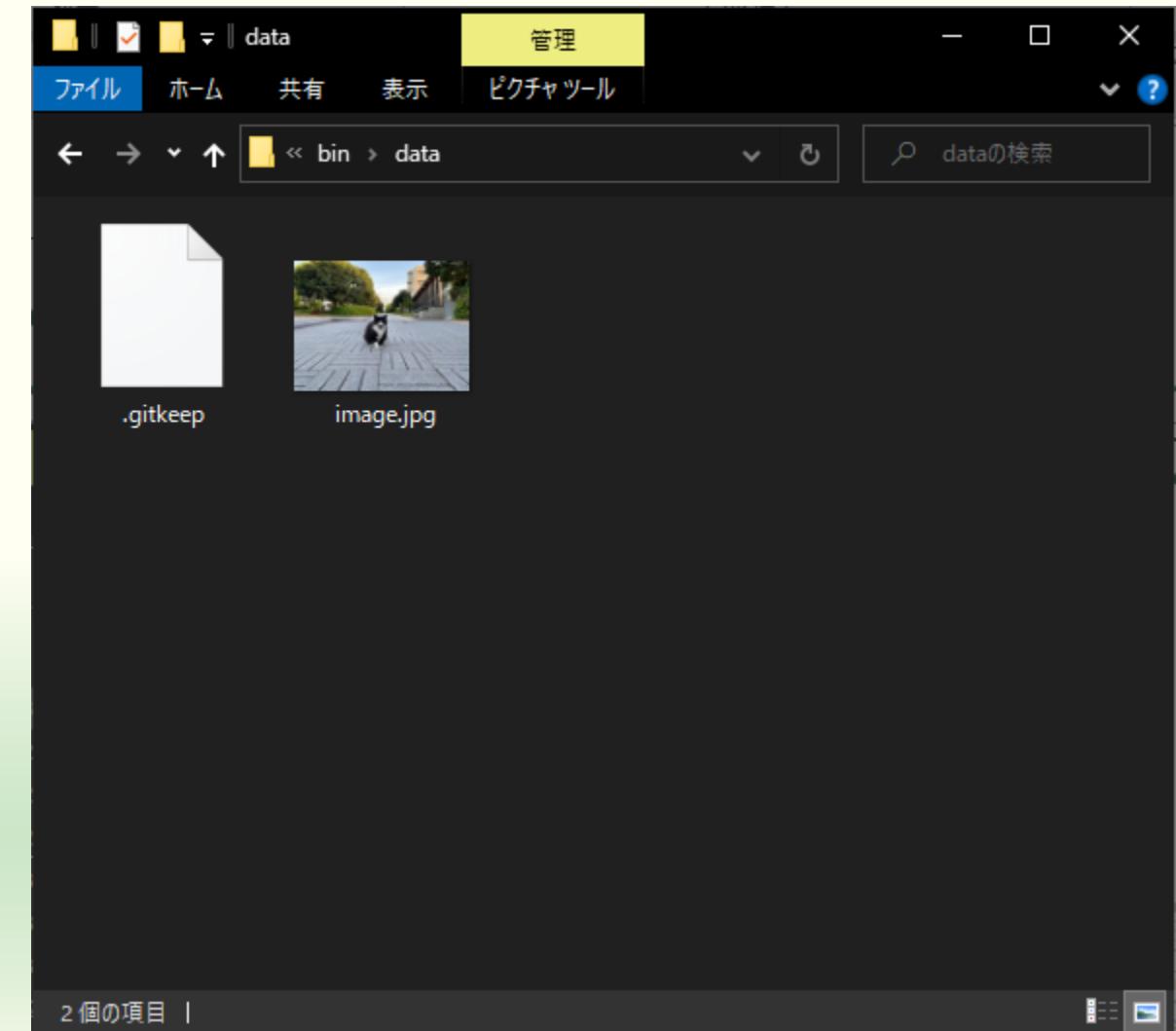
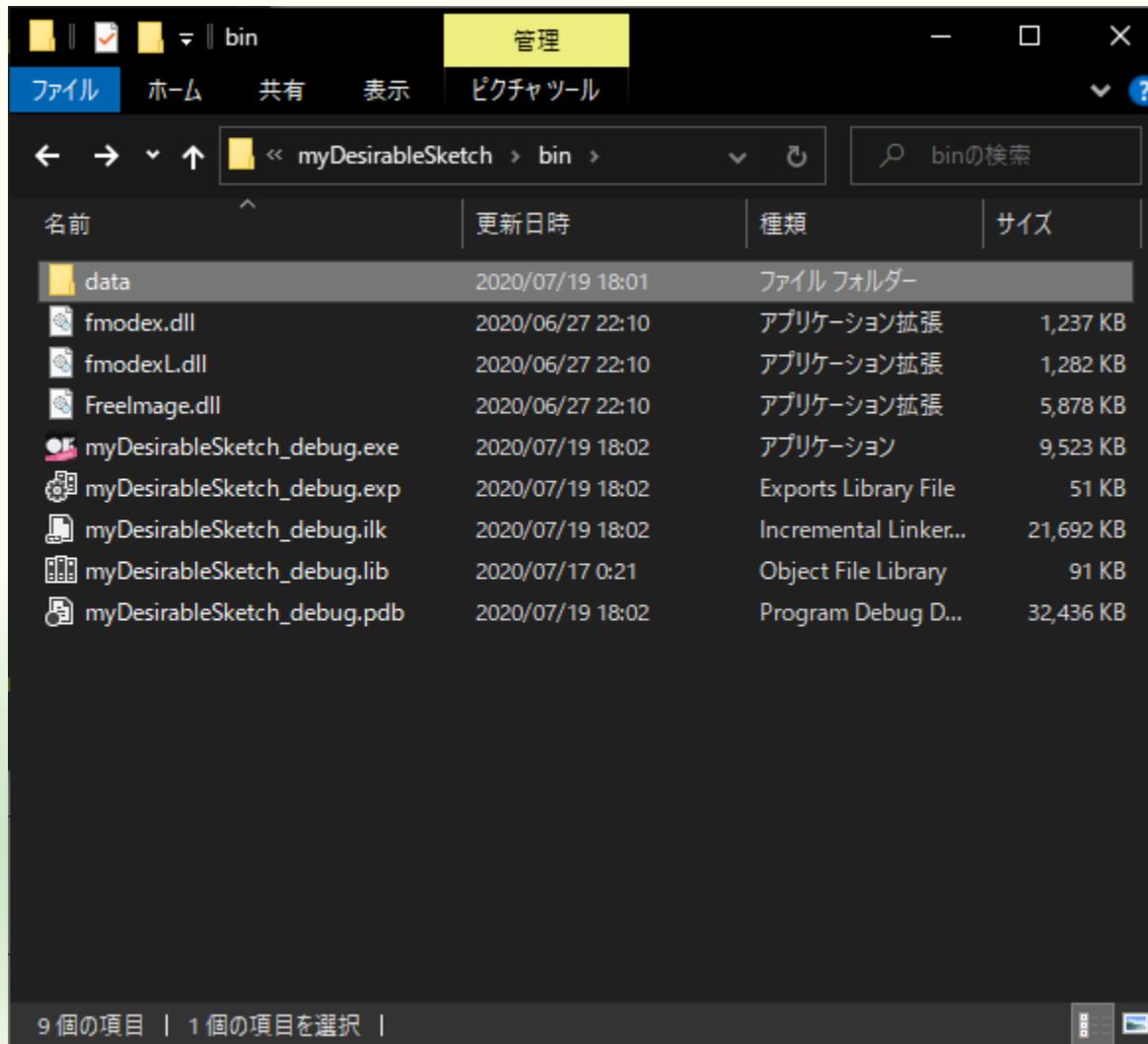


# load() メソッド

---

- `bool ofImage_::load(const filesystem::path &fileName, const ofImageLoadSettings &settings)`
  - `filename`: 画像ファイルのパス
  - `settings`: 画像ファイルの読み込み設定
    - メンバ: `bool accurate, bool exifRotate, bool grayscale, bool separateCMYK`
    - 省略可能
- `fileName` に指定した画像ファイルから画像を読み込む
  - JPEG, PNG, GIF 形式のファイルが読み込み可能
  - パスを省略したときはプロジェクトのフォルダの **bin/data** の中
  - 戻り値は読み込みに成功したとき `true`、失敗したとき `false` を返す

# 画像は bin フォルダの中の data に配置する



# draw() で image を描画する

```
//-----
void ofApp::draw(){
    ofSetColor(255, 255, 255);
    image.draw(0, 0);
    for (auto &points : polylines){
        if (points.empty()) return;
        auto point{ points.begin() };
        ofSetColor(255, 0, 0);
        ofDrawRectangle(points[0] - 3.0f, 5, 5);
        for (auto next = point + 1;
            next != points.end(); point = next++){
            ofSetColor(255, 0, 0);
            ofDrawRectangle(*next - 3.0f, 5, 5);
            ofSetColor(0, 0, 0);
            ofDrawLine(*point, *next);
        }
    }
}
```

- image.draw(0, 0);
  - image をウィンドウの (0, 0) の位置に描画する
- 表示は image の色と ofSetColor() で設定された色との積になる
  - ofSetColor(255, 0, 0); とすると画像の赤成分だけが表示される
  - ofSetColor(127, 127, 127); とすると明度が半分になる
  - ofSetColor(0, 0, 0); とすると画像の内容にかかわらず黒になる

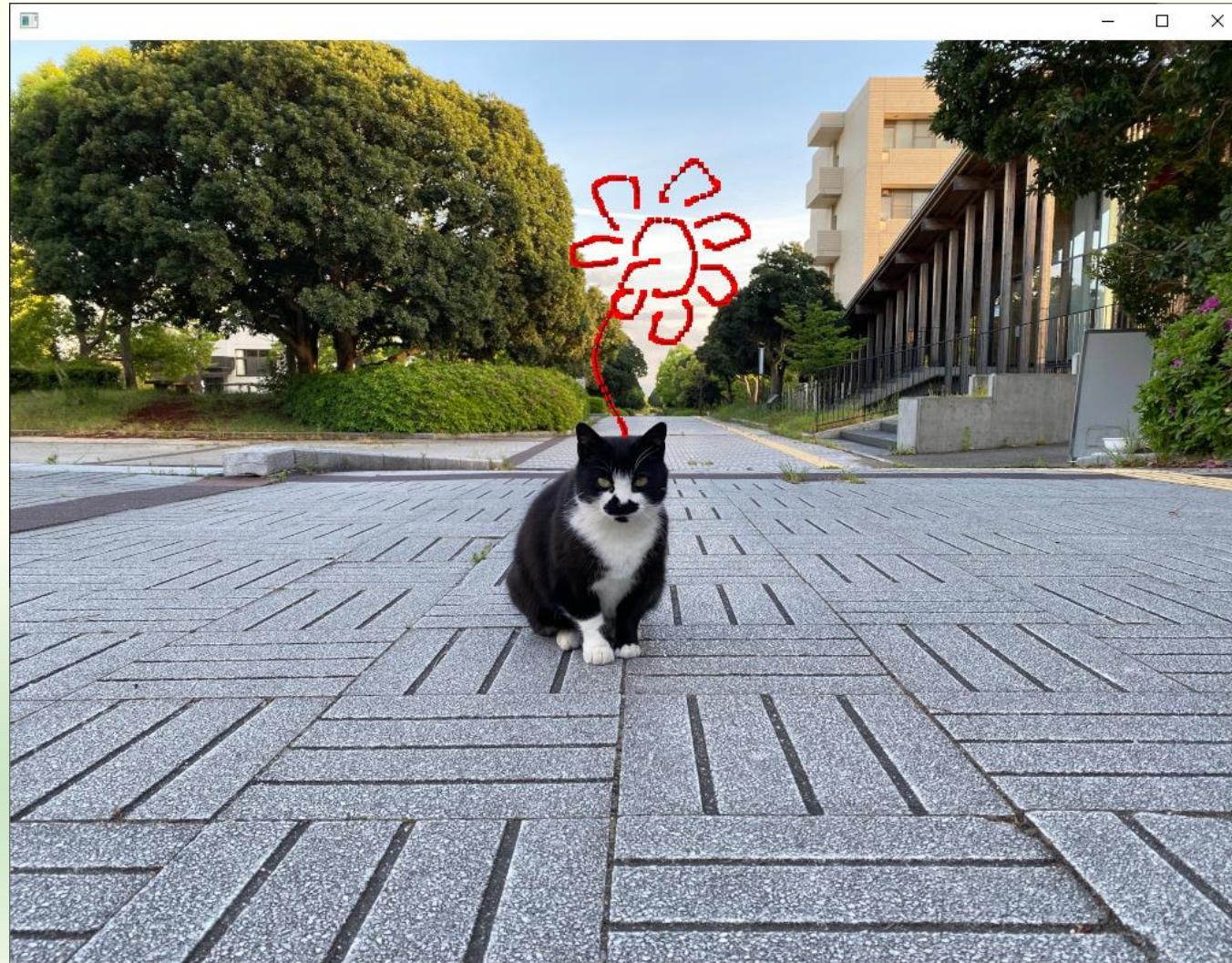
# draw() メソッド

---

- `void ofImage_::draw(float x, float y)`
  - ウィンドウの (x, y) の位置を左上として画像を描画する
  - ウィンドウからはみ出た部分は表示されない
  
- `void ofImage_::draw(float x, float y, float w, float h)`
  - ウィンドウの (x, y) の位置を左上として幅 w 高さ h の領域に画像を拡大縮小して描画する
    - `image.draw(0, 0, ofGetWidth(), ofGetHeight());` とすると画像全体が常にウィンドウいっぱいに表示される
  - ウィンドウからはみ出た部分は表示されない



# 実行結果



折れ線も  
描ける



# ファイル選択ダイアログを使う

```
//-----  
void ofApp::setup(){  
    ofBackground(255, 255, 255);  
    polylines.emplace_back();  
    image.load("image.jpg");  
    auto result{ ofSystemLoadDialog() };  
    if (result.bSuccess){  
        image.load(result.filePath);  
    }  
}
```

## 注意

ofSystemLoadDialog() は日本語を含む  
ファイルパスに対応していない  
(日本語が? になってしまう)

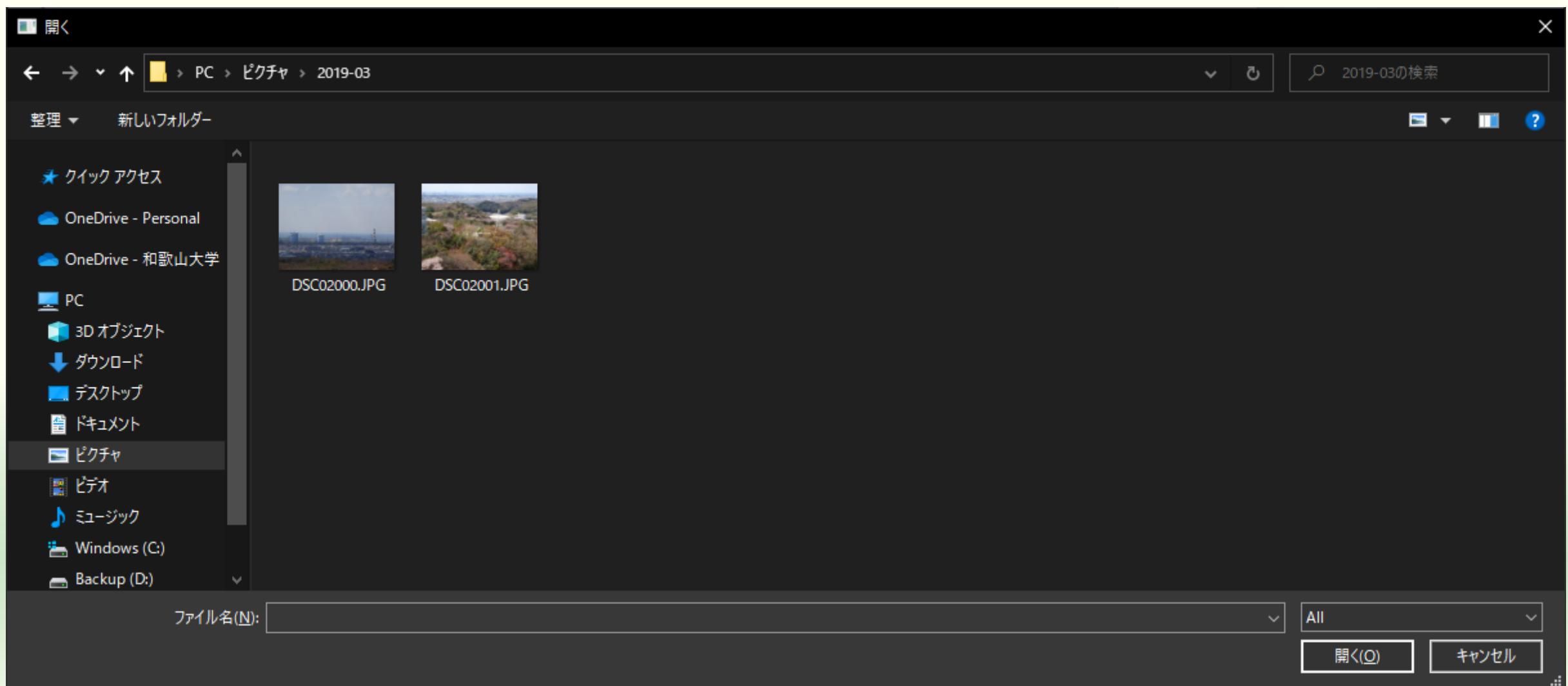
- 好きなところからファイルを読み込みみたい場合
- auto result{ ofSystemLoadDialog() };
  - ファイル選択ダイアログを開く
- if (result.bSuccess){
  - もしファイルの選択に成功していれば
  - image.load(result.filePath);
    - 選択したファイル (result.filePath) を読み込む

# ofSystemLoadDialog()

- ofFileDialogResult ofSystemLoadDialog(string windowTitle, bool bFolderSelection=false, string defaultPath)
  - windowTitle: ダイアログウィンドウのタイトルバーに表示する文字列
  - bFolderSelection: true ならフォルダの選択, false ならファイルの選択
  - defaultPath: ファイルパスを省略したときのデフォルトのパス名
- ofFileDialogResult
  - bSuccess: ファイルの選択がキャンセルされずに成功したら true
  - getName(), fileName: ファイル名
  - getPath(), filePath: ファイル名のフルパス



# ファイル選択ダイアログウィンドウ



# 読み込みに失敗したらエラーを表示する

```
//-----
void ofApp::setup(){
    ofBackground(255, 255, 255);
    polylines.emplace_back();
    image.load("image.jpg");
    auto result{ ofSystemLoadDialog() };
    if (result.bSuccess){
        if (!image.load(result.filePath)){
            ofSystemAlertDialog("Can't load file: "
                + result.filePath);
        }
    }
}
```

## 注意

ofSystemAlertDialog() は日本語を含む  
メッセージに対応していない  
(日本語が文字化けする)

- `if (!image.load(result.filePath))`
  - もし `result.filePath` の読み込みに成功しなかつたら
    - `!` は論理反転演算子、`true`→`false`,  
`false`→`true`
  - `ofSystemAlertDialog("message")`
    - `message` を表示する
- `"Can't load file: " + result.filePath`
  - `"Can't load file: "` に `result.filePath` の内容を連結する
  - このときの `+` は文字列 (string) を連結する演算子

# 'o' か 'O' キーのタイプでファイルを読み込む

```
//-----
void ofApp::setup(){
    ofBackground(255, 255, 255);
    polylines.emplace_back();
    image.load("image.jpg");
}

//-----
(void ofApp::keyPressed(int key){
    if (key == 'o' || key == 'O'){
        auto result{ ofSystemLoadDialog() };
        if (result.bSuccess){
            if (!image.load(result.filePath)){
                ofSystemAlertDialog("Can't load file: "
                    + result.filePath);
            }
        }
    }
}
```

移動

- プログラム実行時に毎回ファイル選択ダイアログを表示したくない
- **keyPressed()**
  - キーボードのキーを押したときに実行される
- **if (key == 'o' || key == 'O') {**
  - もし押されたキー **key** が 'o' または 'O' なら
    - ファイル選択ダイアログを表示する
    - 選択された画像ファイルを読み込む

# Windows で日本語のファイル名を選択する

```
//-----
#include <commndlg.h>

void ofApp::keyPressed(int key){
    if (key == 'o' || key == 'O'){
        TCHAR filePath[MAX_PATH] = TEXT("");
        OPENFILENAME ofn;

        memset(&ofn, 0, sizeof(OPENFILENAME));
        ofn.lStructSize = sizeof(OPENFILENAME);
        ofn.lpstrFilter = TEXT("JPEG (*.jpg)*.jpg*PNG (*.png)*.png*GIF (*.gif)*All (*.*)*.*");
        ofn.lpstrFile = filePath;
        ofn.nMaxFile = sizeof(filePath);
        ofn.Flags = OFN_FILEMUSTEXIST | OFN_HIDEREADONLY;
        ofn.lpstrDefExt = TEXT("jpg");

        if (GetOpenFileName(&ofn) && !image.load(filePath)) {
            MessageBox(NULL, filePath, TEXT("ファイルが開けません"), MB_ICONERROR);
        }
    }
}
```

WIN32 (Windows の機能) を直接使って  
ファイル選択ダイアログを表示する

# ofApp クラスにブラシのメンバ変数を追加する

```
#pragma once  
  
#include "ofMain.h"  
  
using namespace glm;  
  
class ofApp : public ofBaseApp {  
    vector<vector<vec2>> polylines;  
    ofImage image, brush;  
  
public:  
    void setup();  
    void update();  
    void draw();  
  
    void keyPressed(int key);  
    void keyReleased(int key);  
    void mouseMoved(int x, int y);  
(以下略)
```

- polylines は image に読み込んだ画像とは別に表示している
  - image の表示に重ねて表示しているが image の内容を**変更しない**
- 追加した画像の ofImage のインスタンス brush は image に格納されている画像そのものを修正するために使う
  - そのため、ここでは brush (ブラシ) という変数名にした

# brush にメモリを割り当てる

```
//-----  
void ofApp::setup(){  
    ofBackground(255, 255, 255);  
    polylines.emplace_back();  
    brush.allocate(50, 50, OF_IMAGE_COLOR);  
}  
-----
```

- brush には画像を読み込むのではなくメモリだけを割り当てる
- brush.allocate(50, 50, OF\_IMAGE\_COLOR);
  - brush のサイズは縦 50 画素（ピクセル）、横 50 画素
  - OF\_IMAGE\_COLOR は R, G, B の 3 つのチャネルをもつフルカラー画像



# allocate() メソッド

---

- void ofPixels\_::allocate(size\_t w, size\_t h, ofImageType imageType)
- void ofPixels\_::allocate(size\_t w, size\_t h, ofPixelFormat pixelFormat)
- void ofPixels\_::allocate(size\_t w, size\_t h, size\_t channels)
  - w, h: メモリを割り当てる画像のサイズ
  - imageType: OF\_IMAGE\_GRAYSCALE, OF\_IMAGE\_COLOR, OF\_IMAGE\_COLOR\_ALPHA
  - pixelFormat: OF\_PIXELS\_RGB, OF\_PIXELS\_RGBA, OF\_PIXELS\_BGRA, OF\_PIXELS\_MONO
  - channels: 1 画素当たりのチャネル数



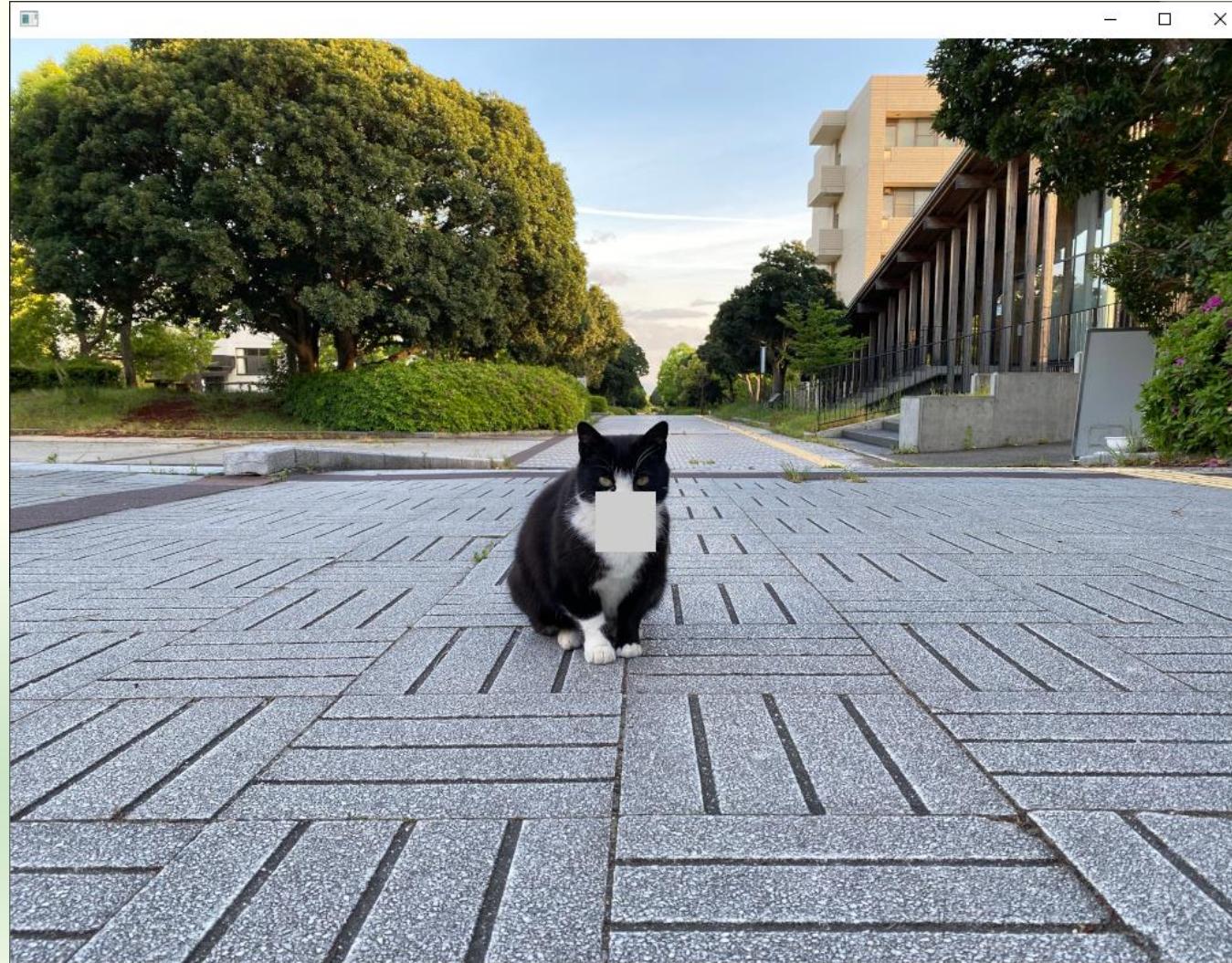
# brush を image に重ねて描画する

```
//-----
void ofApp::draw(){
    ofSetColor(255, 255, 255);
    image.draw(0, 0);
    brush.draw(mouseX, mouseY);
    for (auto &points : polylines){
        if (points.empty()) return;
        auto point{ points.begin() };
        ofSetColor(255, 0, 0);
        ofDrawRectangle(points[0] - 3.0f, 5, 5);
        for (auto next = point + 1;
            next != points.end(); point = next++){
            ofSetColor(255, 0, 0);
            ofDrawRectangle(*next - 3.0f, 5, 5);
            ofSetColor(0, 0, 0);
            ofDrawLine(*point, *next);
        }
    }
}
```

- brush.draw() を image.draw() の後に置く
  - mouseX, mouseY は現在のマウスカーソルの位置
  - brush の画像がマウスカーソルと一緒に動く
  - brush の色は初期値のグレー



# グレーの正方形がマウスカーソルと一緒に動く



# 画素データを取り出す

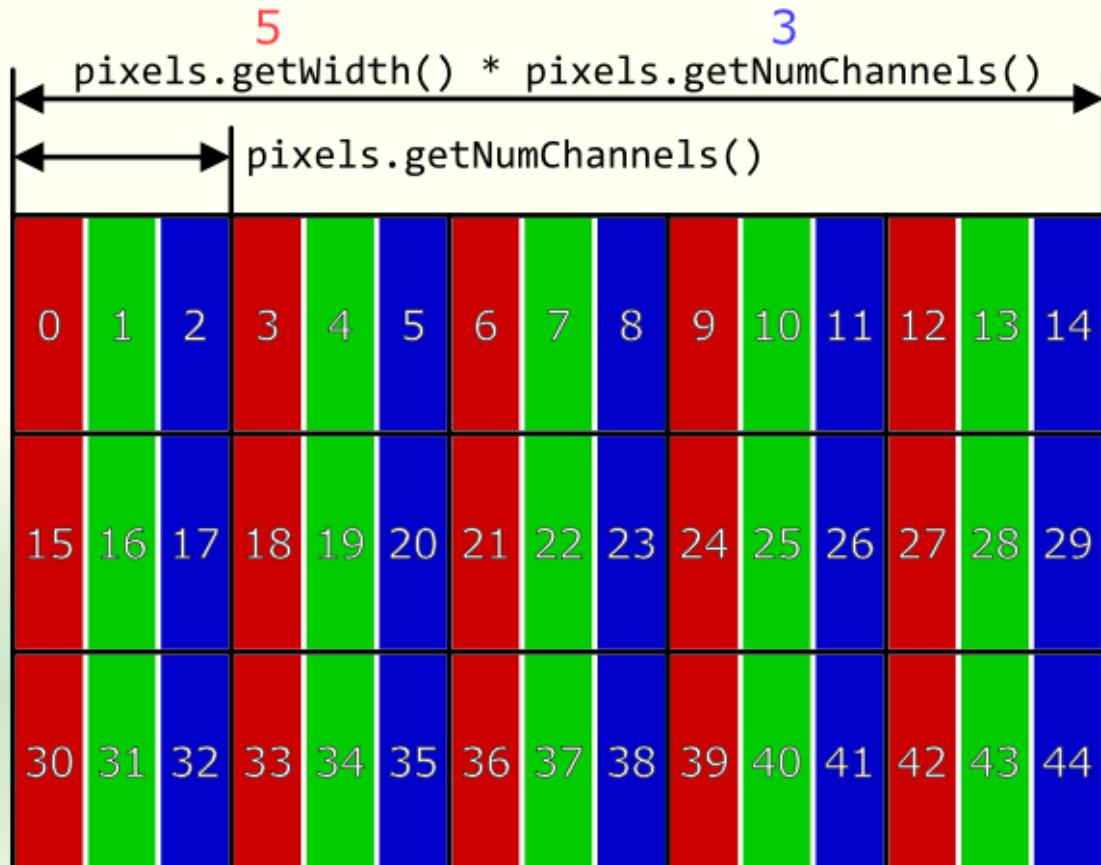
```
//-----
void ofApp::setup(){
    ofBackground(255, 255, 255);
    polylines.emplace_back();
    image.load("image.jpg");
    brush.allocate(50, 50, OF_IMAGE_COLOR);
    ofPixels &pixels{ brush.getPixels() };
}
```

- `getPixels()` メソッド
  - `ofImage` クラスのオブジェクト（画像）の画素データを取り出す
- `ofPixels` クラス
  - 画像の画素データのクラス



# ofPixels クラスの画素データ

```
brush.allocate(5, 3, OF_IMAGE_COLOR);
ofPixels &pixels = brush.getPixels();
```



- 例えば brush.allocate(5, 3, OF\_IMAGE\_COLOR); で割り当てた画素データは左のようになる
  - pixels.getWidth() == 5
  - pixels.getHeight() == 3
  - pixels.getNumChannels() == 3
  - pixels.size() == 45
- 1 画素分のデータは pixels の 3 つの要素に入っている
  - pixels[0]: 赤, pixels[1]: 緑, pixels[2]: 青, pixels[3]: 赤, pixels[4]: 緑, ...

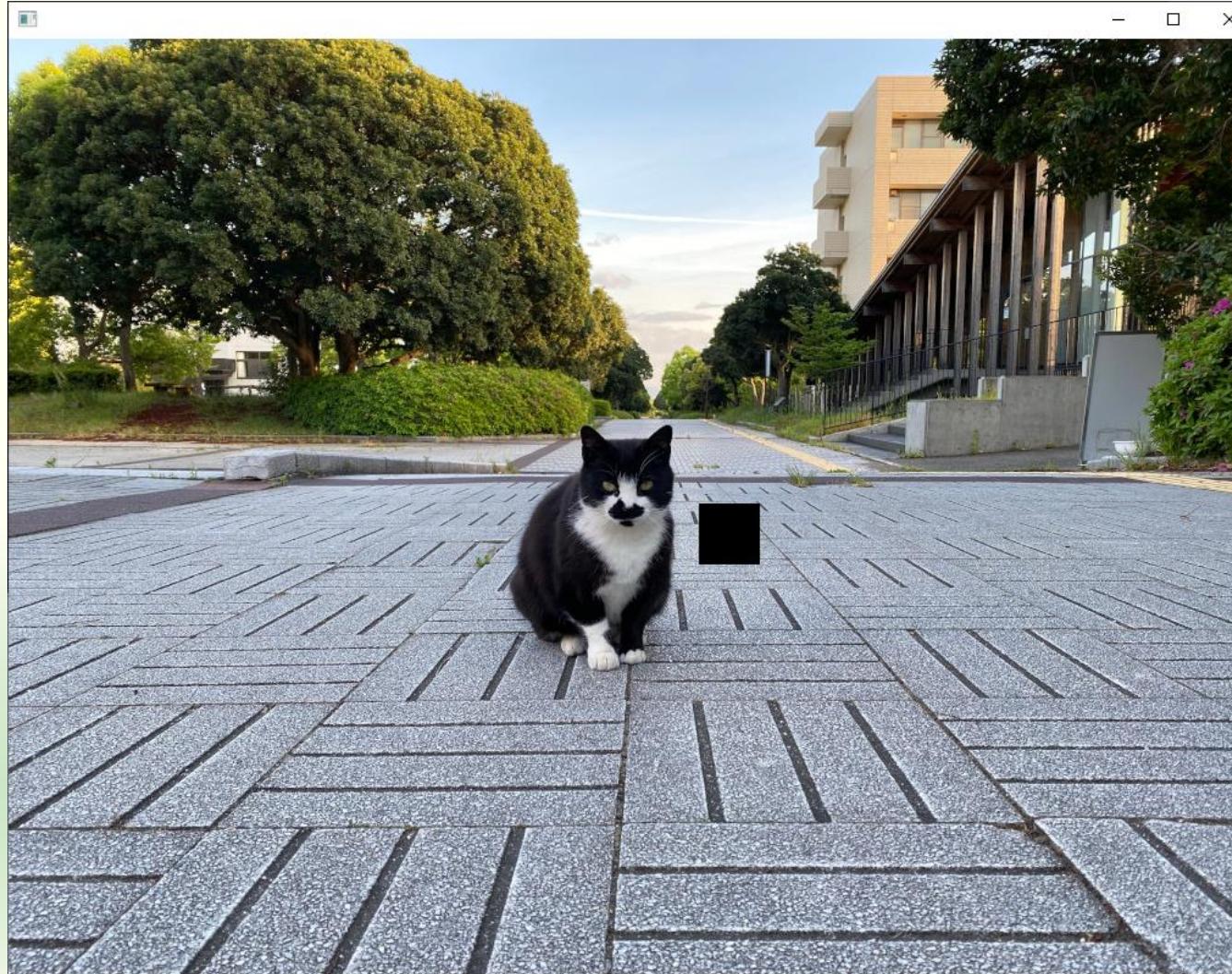
# brush を黒く塗りつぶす

```
//-----
void ofApp::setup(){
    ofBackground(255, 255, 255);
    polylines.emplace_back();
    image.load("image.jpg");
    brush.allocate(50, 50, OF_IMAGE_COLOR);
    ofPixels &pixels{ brush.getPixels() };
    for (size_t i = 0; i < pixels.size(); i += 3){
        pixels[i] = 0;
        pixels[i + 1] = 0;
        pixels[i + 2] = 0;
    }
    brush.update();
}
```

- size() メソッド
  - 画素データの数を返す
  - データ型は size\_t
- OF\_IMAGE\_COLOR の場合は3つごとに1画素のカラーデータ
  - pixels[i] 赤, pixels[i+1] 緑, pixels[i+2] 青
- pixels の各要素に値を代入すれば画素の色が変えられる
- 最後に update() メソッドを実行すれば brush に反映される

# brush が黒くなる

---

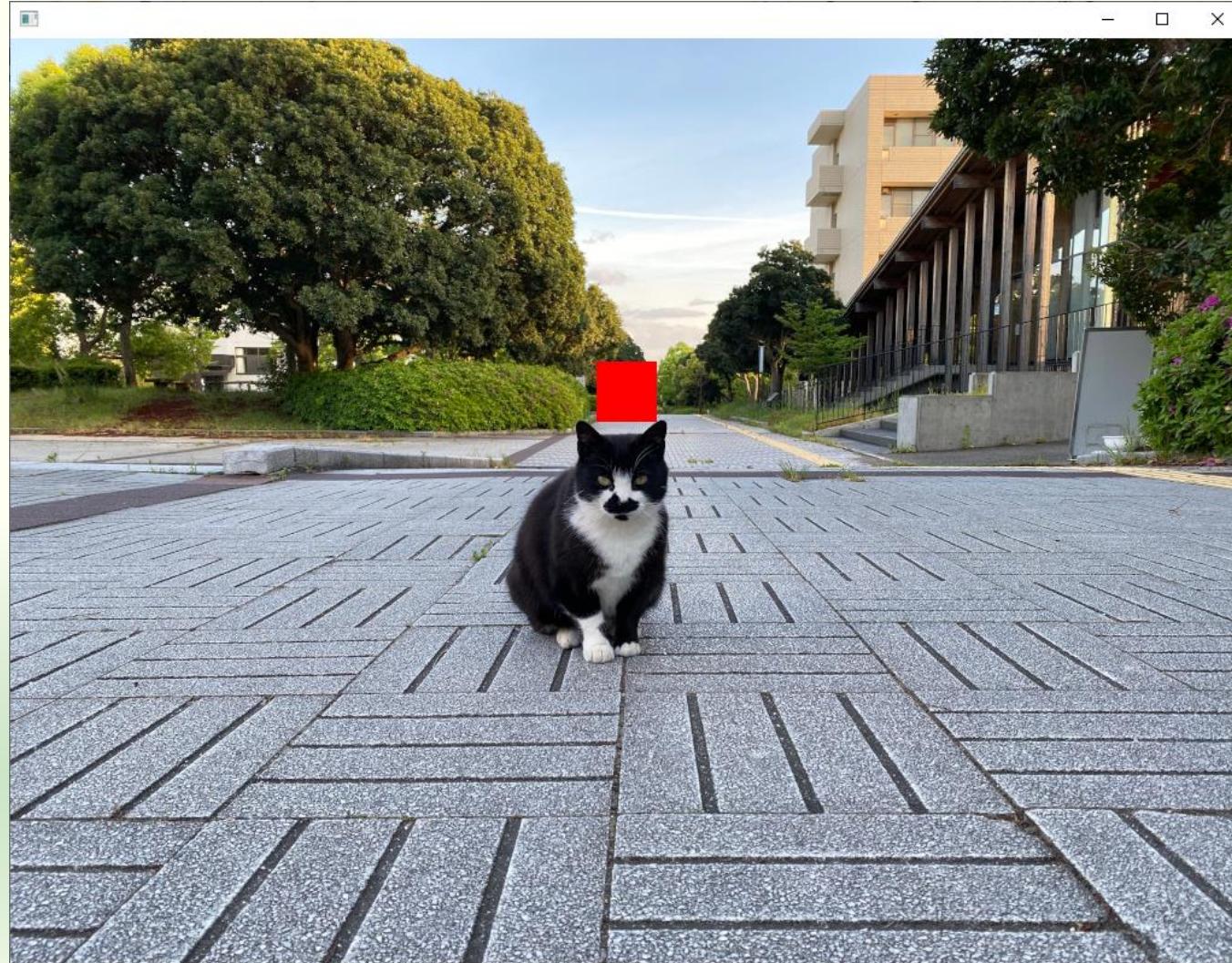




# 課題 4 – 4

赤いブラシを作る

# brush を赤くしてください



# 課題のアップロード

---

- 作成したプログラムの実行結果のスクリーンショットを撮つて **4-4.png** というファイル名で保存し、Moodle の第 4 回課題にアップロードしてください

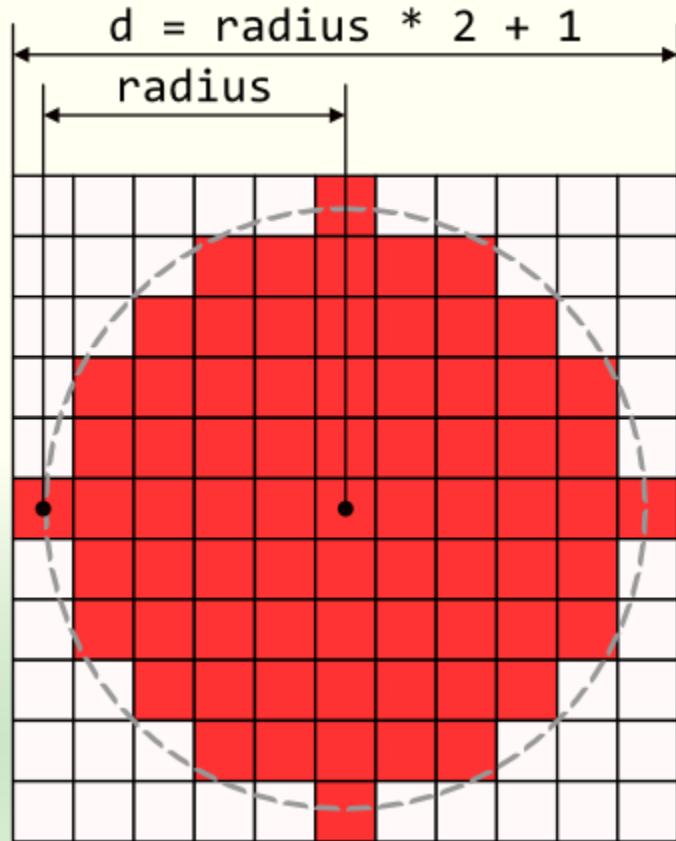




# 課題 4 – 5

ブラシを円形にする

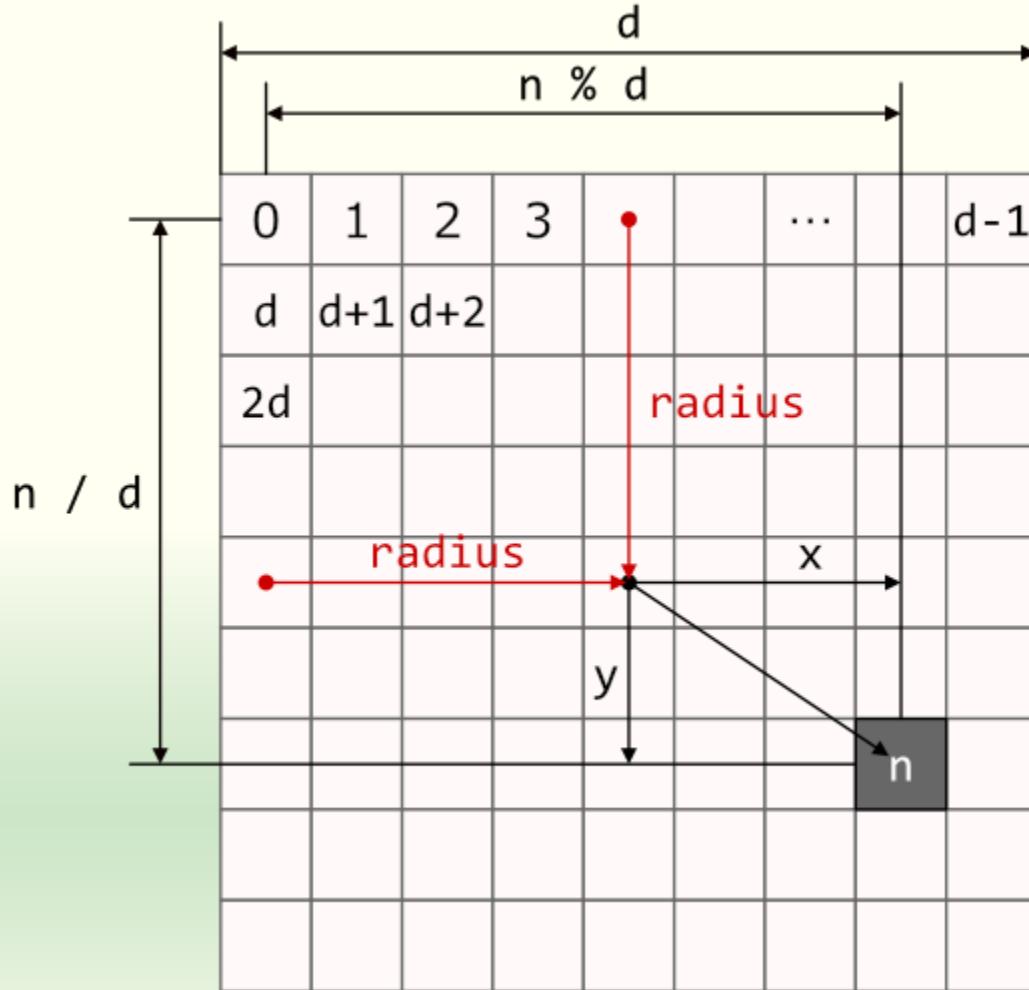
# brush を丸くする



- 半径が  $\text{radius}$  の円を塗りつぶす
- この円がぴったり収まる領域は  
辺の長さ  $d$  が  $d = \text{radius} * 2 + 1$   
の正方形である



# 画素の番号 $n$ と座標値 $x, y$



- 画像の幅（横方向の画素数）を  $d$  とする
- 画像の左上の画素を起点として、その画素の番号を 0 とする
- 番号  $n$  の画素の画像の左端からの画素数は  $n \% d$
- 番号  $n$  の画素の画像の上端からの画素数は  $n / d$
- それらから半径  $radius$  を引いて中心からの画素位置  $x, y$ を得る

# brush のサイズを求める

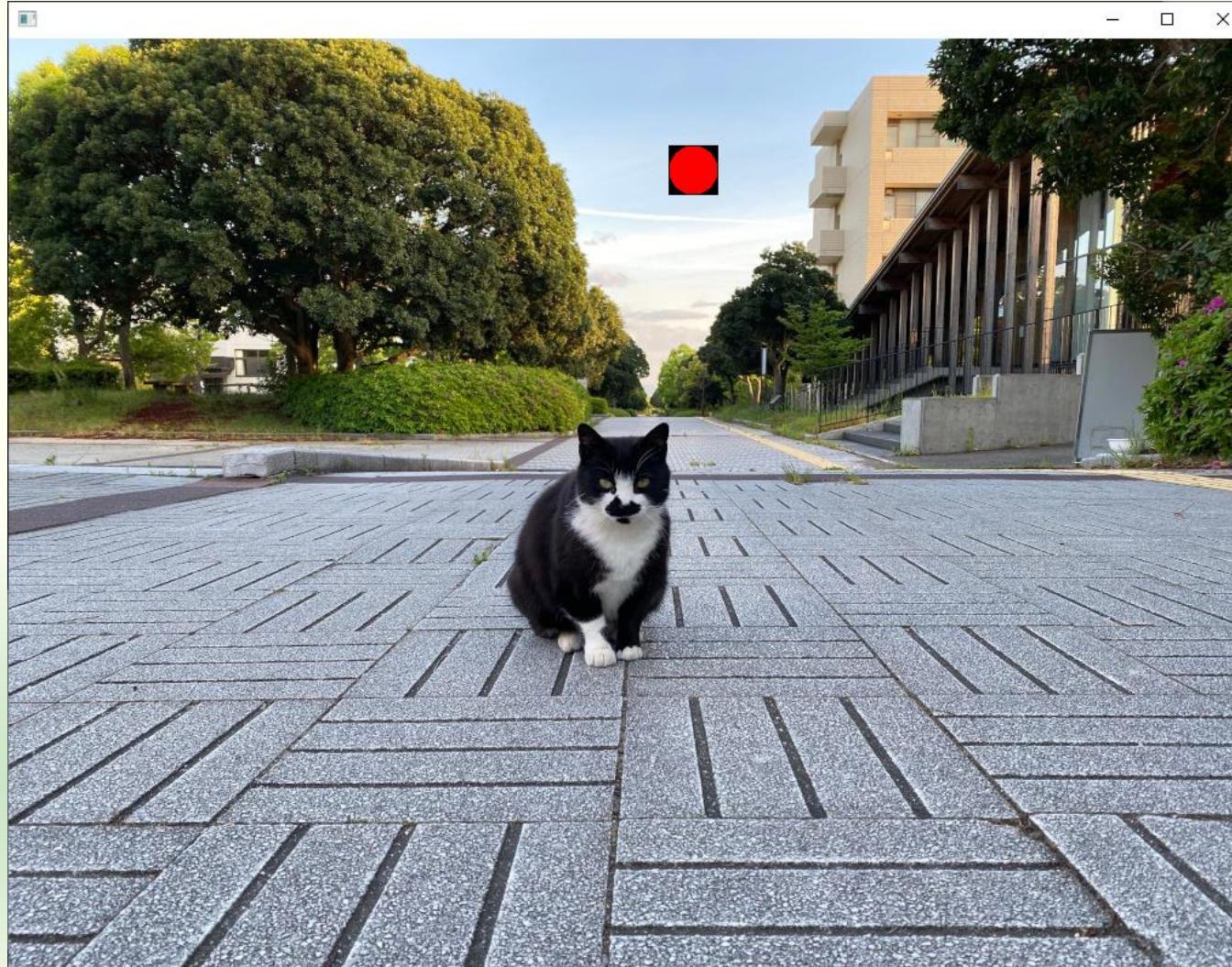
```
#include "ofApp.h"

const auto radius{ 20 };

//-----
void ofApp::setup(){
    ofBackground(255, 255, 255);
    polylines.emplace_back();
    image.load("image.jpg");
    const auto d{ radius * 2 + 1 };
    brush.allocate(d, d, OF_IMAGE_COLOR);
    ofPixels &pixels{ brush.getPixels() };
    for (size_t i = 0; i < pixels.size(); i += 3){
        const auto n{ int(i / 3) };
        const auto x{ n % d - radius };
        const auto y{ n / d - radius };
        (途中略)
    }
}
```

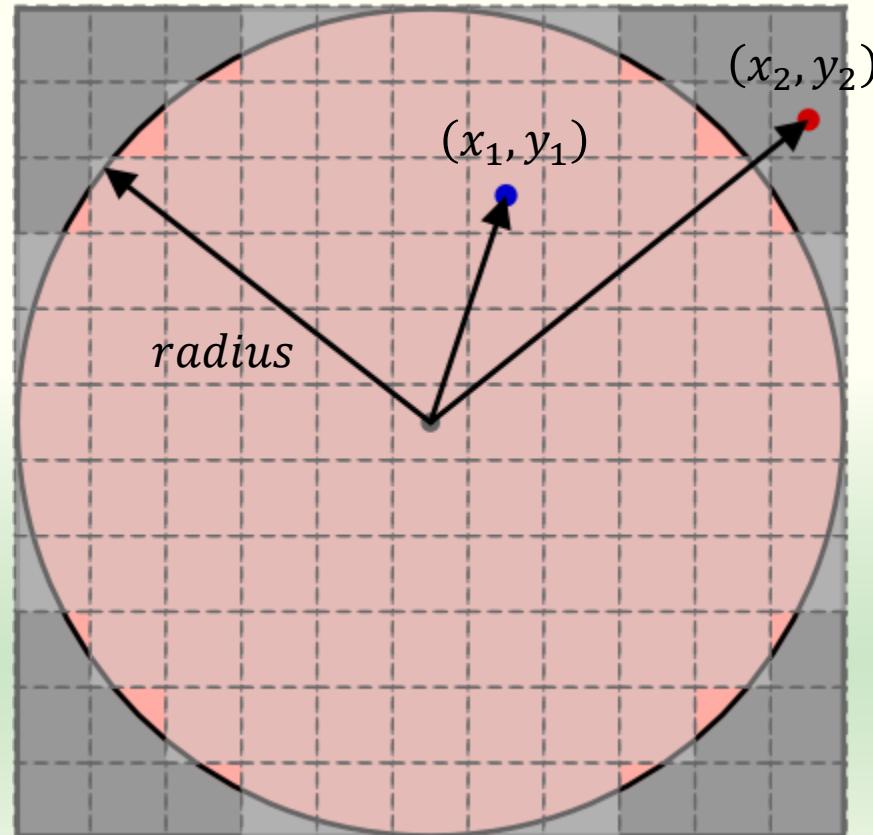
- `const auto n{ i / 3 };`
  - $i$  は 1 画素ごとに 3 進むから 3 で割れば画素の番号  $n$  が得られる
- `const auto x{ n % d - radius };`
  - $n$  を領域の幅  $d$  で割った剰余は画素の領域の左端からの位置となる
  - これから  $radius$  を引けば領域の中心からの  $x$  座標値が得られる
- `const auto y{ n / d - radius };`
  - $n$  を領域の幅  $d$  で割った商の整数部は画素の領域の上端からの位置となる
  - これから  $radius$  を引けば領域の中心からの  $y$  座標値が得られる

# brush を円形にしてください



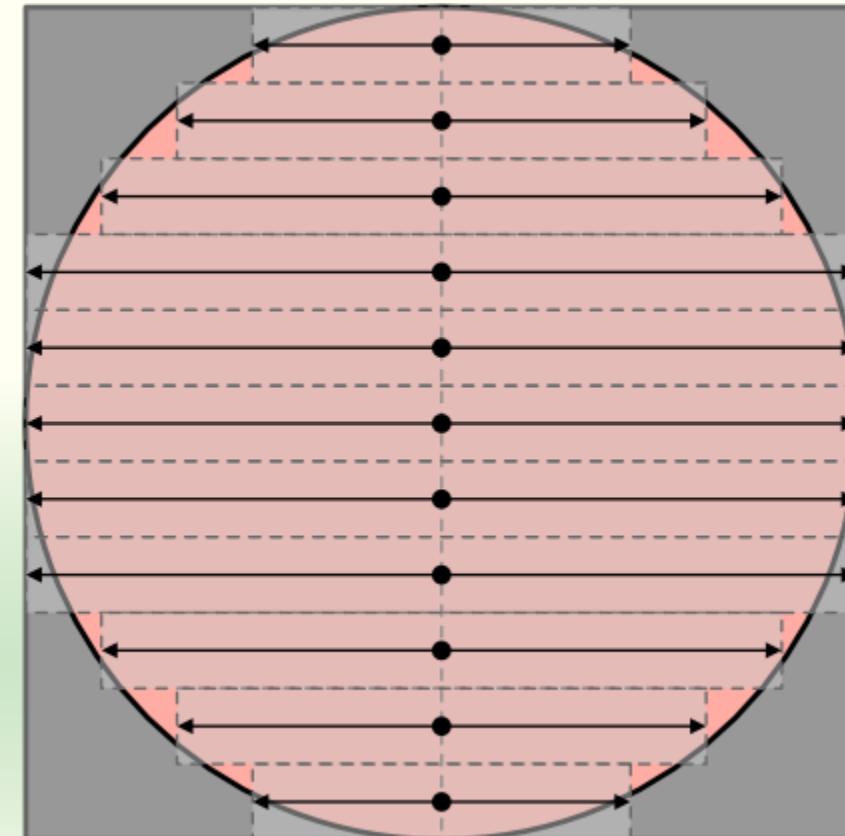
# 円の内部にある画素に色を付ける

画素が円の内部にあるか判定する



やり方は色々

画素の高さごとに幅を求める



# マウスカーソルの先に円の中心を置く

```
//-----
void ofApp::draw(){
    ofSetColor(255, 255, 255);
    image.draw(0, 0);
    brush.draw(mouseX - radius, mouseY - radius);
    for (auto &points : polylines){
        if (points.empty()) return;
        auto point{ points.begin() };
        ofSetColor(255, 0, 0);
        ofDrawRectangle(points[0] - 3.0f, 5, 5);
        for (auto next = point + 1;
            next != points.end(); point = next++){
            ofSetColor(255, 0, 0);
            ofDrawRectangle(*next - 3.0f, 5, 5);
            ofSetColor(0, 0, 0);
            ofDrawLine(*point, *next);
        }
    }
}
```

- マウスカーソルの位置から半径 radius を引く



# 課題のアップロード

---

- 作成したプログラムの実行結果のスクリーンショットを撮つて **4-5.png** というファイル名で保存し、Moodle の第 4 回課題にアップロードしてください





# 課題 4 – 6

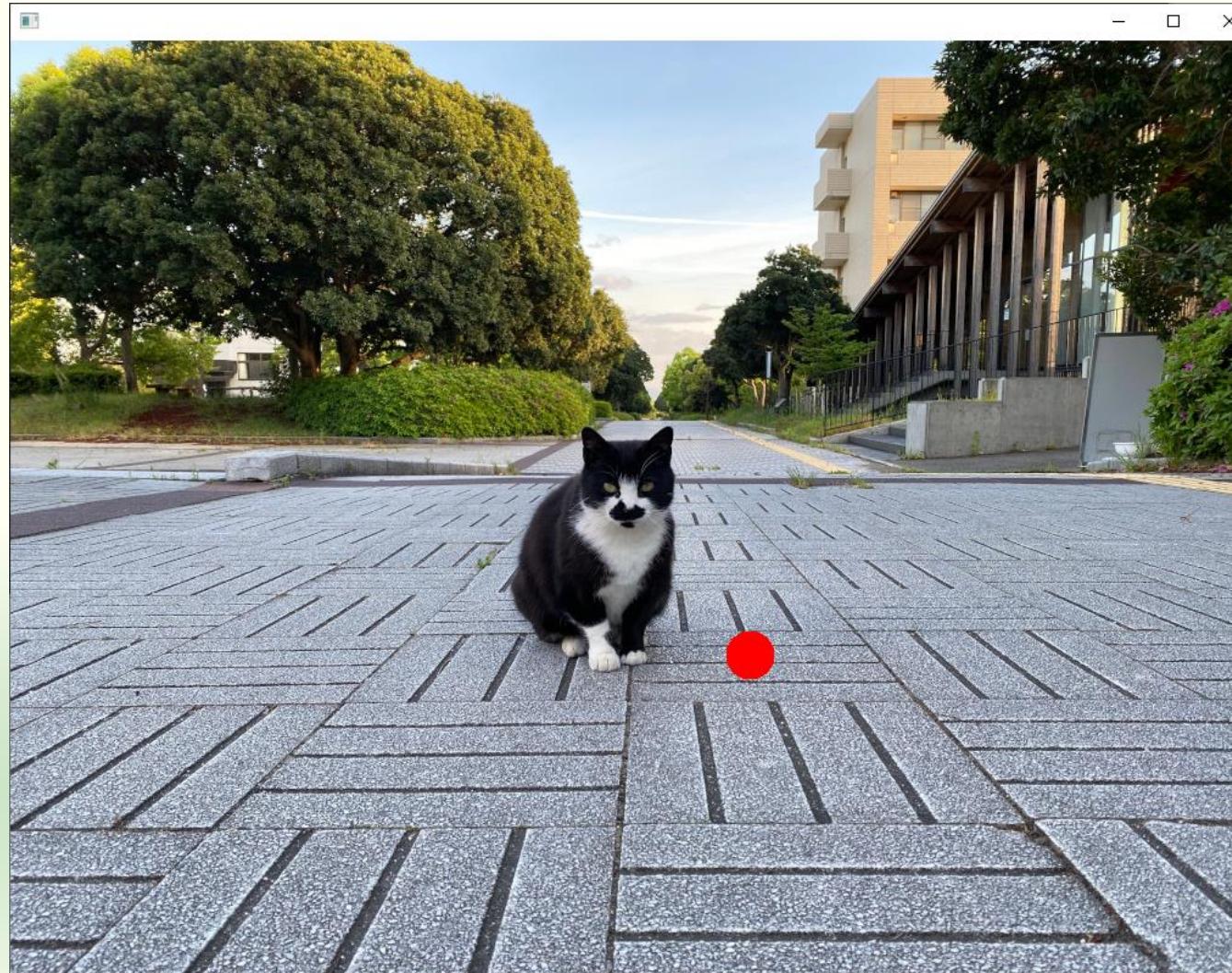
ブラシの範囲外を透明にする

# 円の範囲外を透明にする

```
//-----
void ofApp::setup(){
    ofBackground(255, 255, 255);
    polylines.emplace_back();
    image.load("image.jpg");
    const auto d{ radius * 2 + 1 };
    brush.allocate(d, d, OF_IMAGE_COLOR_ALPHA);
    ofPixels &pixels{ brush.getPixels() };
    for (size_t i = 0; i < pixels.size(); i += 4){
        const auto t{ int(i / 4) };
        const auto x{ t % d - radius };
        const auto y{ t / d - radius };
        (途中略)
    }
}
```

- `allocate()` メソッドの `imageType` として `OF_IMAGE_COLOR` の代わりに `OF_IMAGE_COLOR_ALPHA` を用いる
  - アルファチャンネルを設ける
- 1 画素当たり 4 チャネル用いる
  - 円の範囲外ではアルファチャネル（第 4 チャネル）を 0 にする
  - 円の範囲内ではアルファチャネルを 255 にする

# ブラシの範囲外を透明にする



# 課題のアップロード

---

- 作成したプログラムの実行結果のスクリーンショットを撮つて **4-6.png** というファイル名で保存し、Moodle の第 4 回課題にアップロードしてください





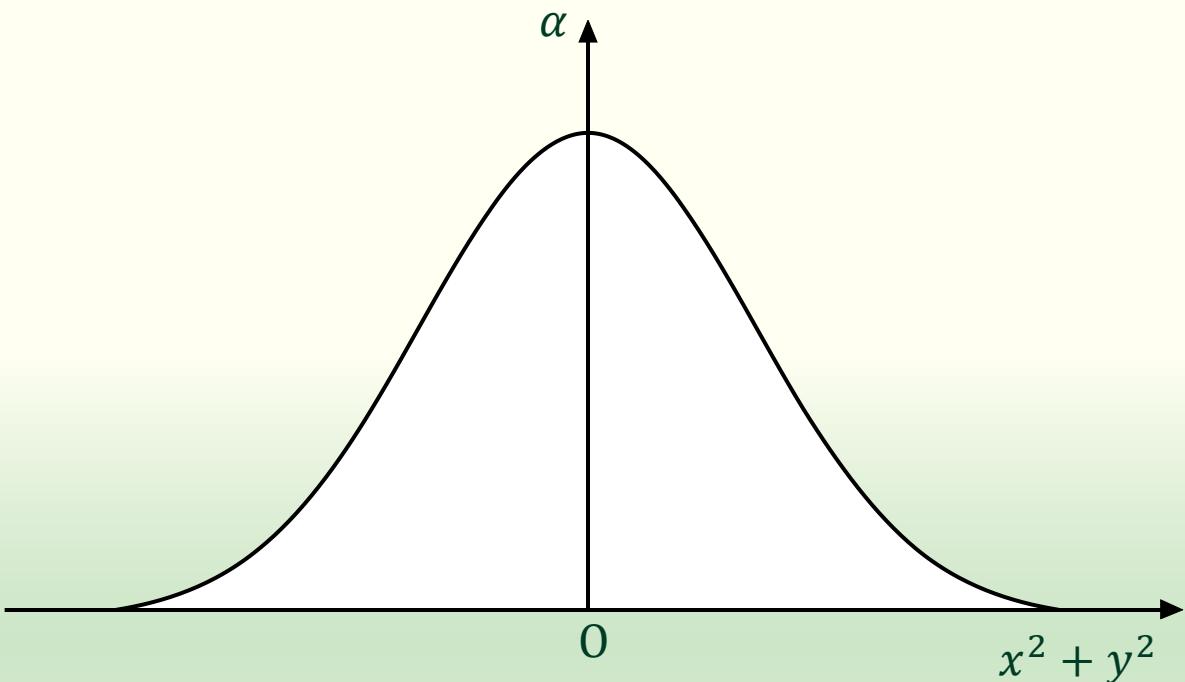
# 課題 4 – 7

ブラシの中心から円周に向かって透明にする

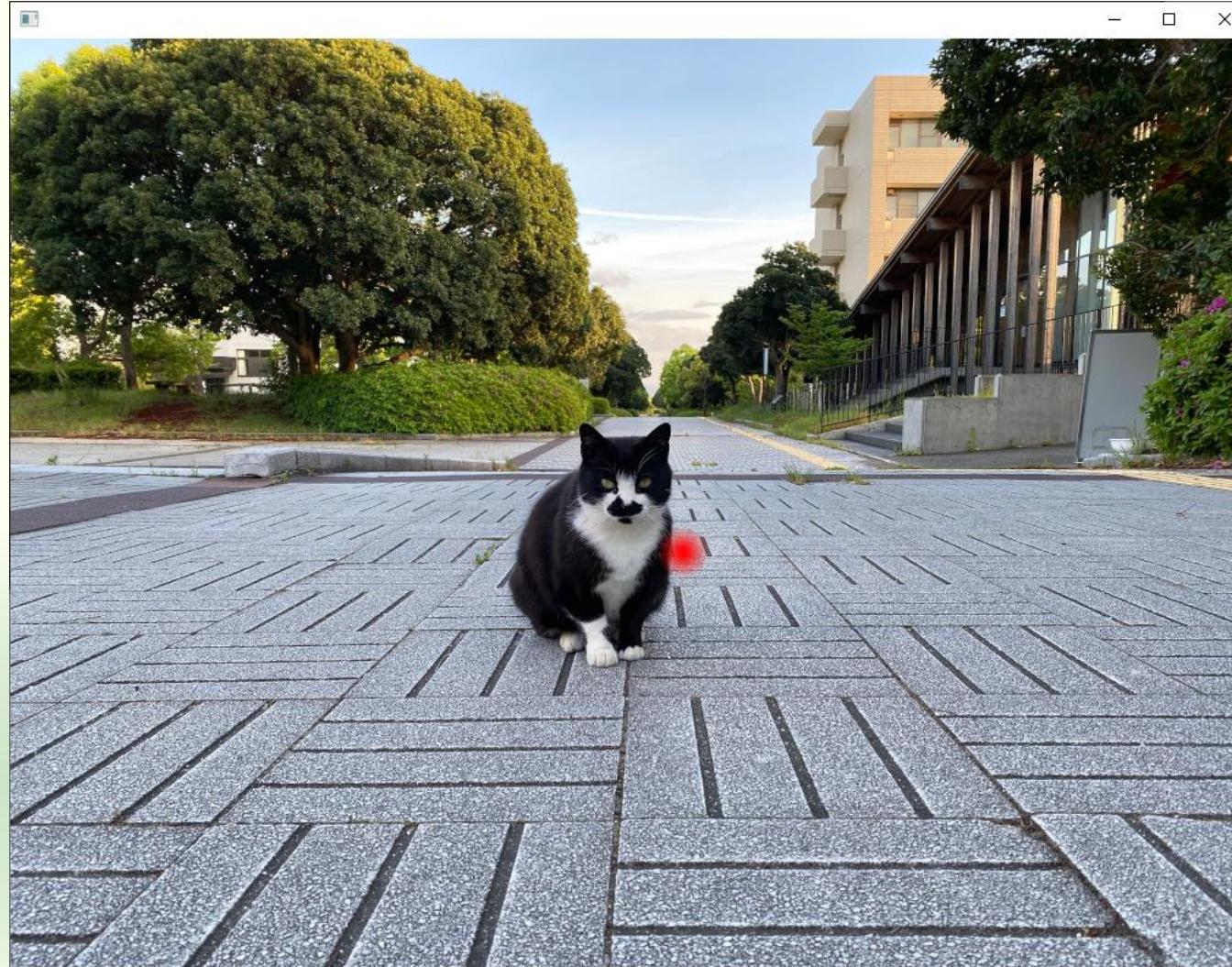
# ブラシの中心から円周に向かって透明にする

- 中心から離れるにしたがって値が減少する関数を塗りつぶす値とアルファ値に用いる
  - 例えば正規分布など

$$\alpha = 255 \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$



# ブラシを中心から円周に向かって透明にする



# 課題のアップロード

---

- 作成したプログラムの実行結果のスクリーンショットを撮つて **4-7.png** というファイル名で保存し、Moodle の第 4 回課題にアップロードしてください





# 課題 4 – 8

ブラシを読み込んだ画像に書き込む

# 読み込んだ画像にアルファチャネルを追加する

```
#include "ofApp.h"

const auto radius{ 20 };

//-----
void ofApp::setup(){
    ofBackground(255, 255, 255);
    polylines.emplace_back();
    image.load("image.jpg");
    image.setImageType(OF_IMAGE_COLOR_ALPHA);
    const auto d{ radius * 2 + 1 };
    brush.allocate(d, d, OF_IMAGE_COLOR_ALPHA);
    ofPixels &pixels{ brush.getPixels() };
    for (size_t i = 0; i < pixels.size(); i += 4){
        const auto n{ int(i / 4) };
        const auto x{ n % d - radius };
        const auto y{ n / d - radius };
        (途中略)
    }
}
```

```
//-----
#include <commddlg.h>

void ofApp::keyPressed(int key){
    if (key == 'o' || key == 'O'){
        (途中略)

        if (GetOpenFileName(&ofn)){
            if (!image.load(filePath)) {
                MessageBox(... 途中略 ...);
            }
            else{
                image.setImageType(OF_IMAGE_COLOR_ALPHA);
            }
        }
    }
}
```

# brush を image にブレンド

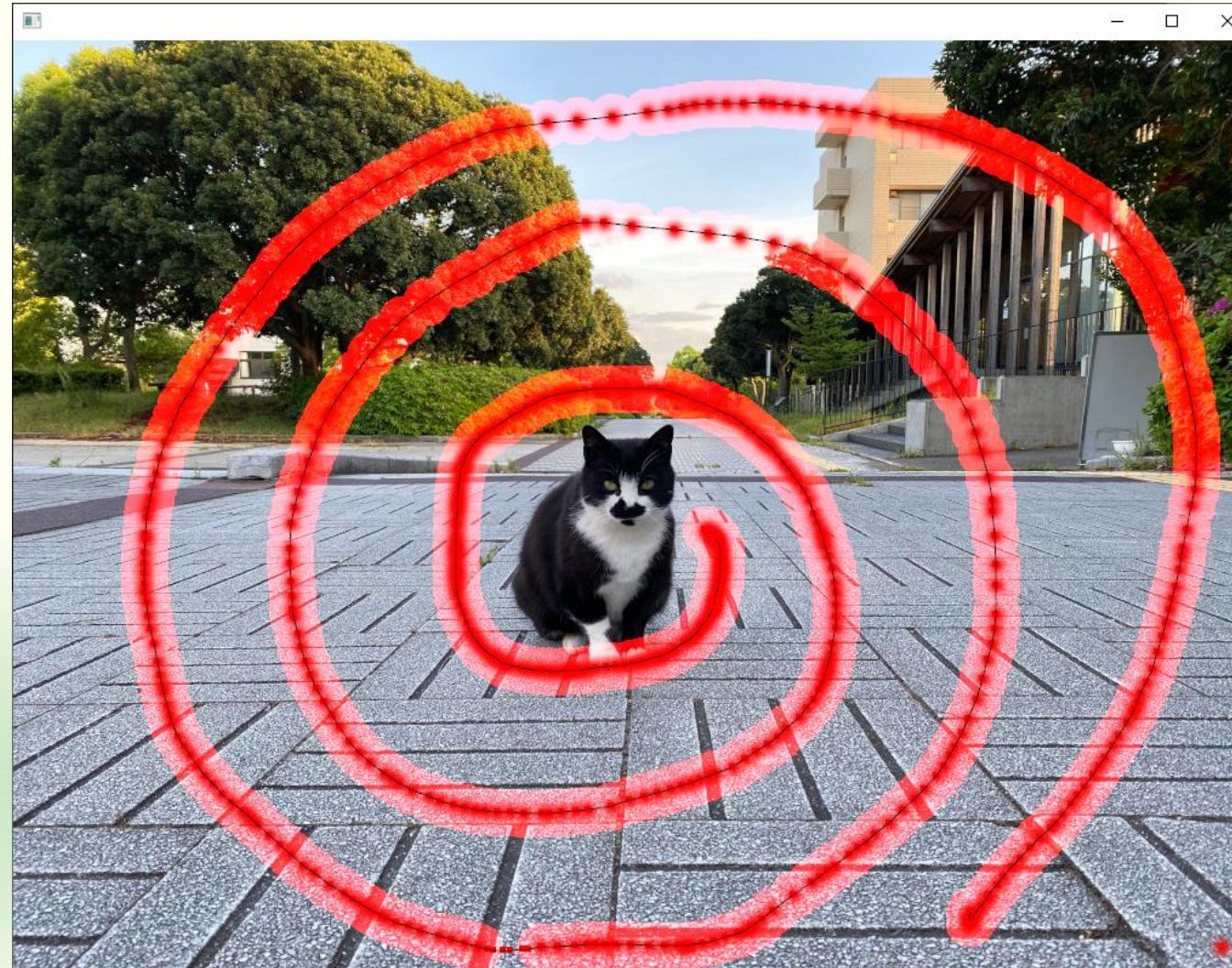
```
//-----
void ofApp::mouseDragged(int x, int y, int button){
    if (button == 0){
        auto &points{ polylines.back() };
        points.emplace_back(vec2{ x, y });
        brush.getPixels().blendInto(
            image.getPixels(), x - radius, y - radius);
        image.update();
    }
}

//-----
void ofApp::mousePressed(int x, int y, int button){
    if (button == 0){
        auto &points{ polylines.back() };
        points.emplace_back(vec2{ x, y });
        brush.getPixels().blendInto(
            image.getPixels(), x - radius, y - radius);
        image.update();
    }
}
```

- アルファブレンディングする
  - 書き込む画像のアルファ値（不透明度） $\alpha$ を使って合成する
  - 下地の色 $\times(1-\alpha)$ +書き込む色 $\times\alpha$
- blendInto() メソッド
  - bool ofPixels\_::blendInto(ofPixels\_ &dst, size\_t x, size\_t y)
  - pixels の画像を dst に指定した画像の x, y の位置にブレンドする



ブラシをつなげるとここまで行かなかった...



# 課題のアップロード

- 作成したプログラムの実行結果のスクリーンショットを撮つて **4-8.png** というファイル名で保存し、Moodle の第 4 回課題にアップロードしてください
- ソースプログラム **ofApp.h** と **ofApp.cpp** を Moodle の第 4 回課題にアップロードしてください

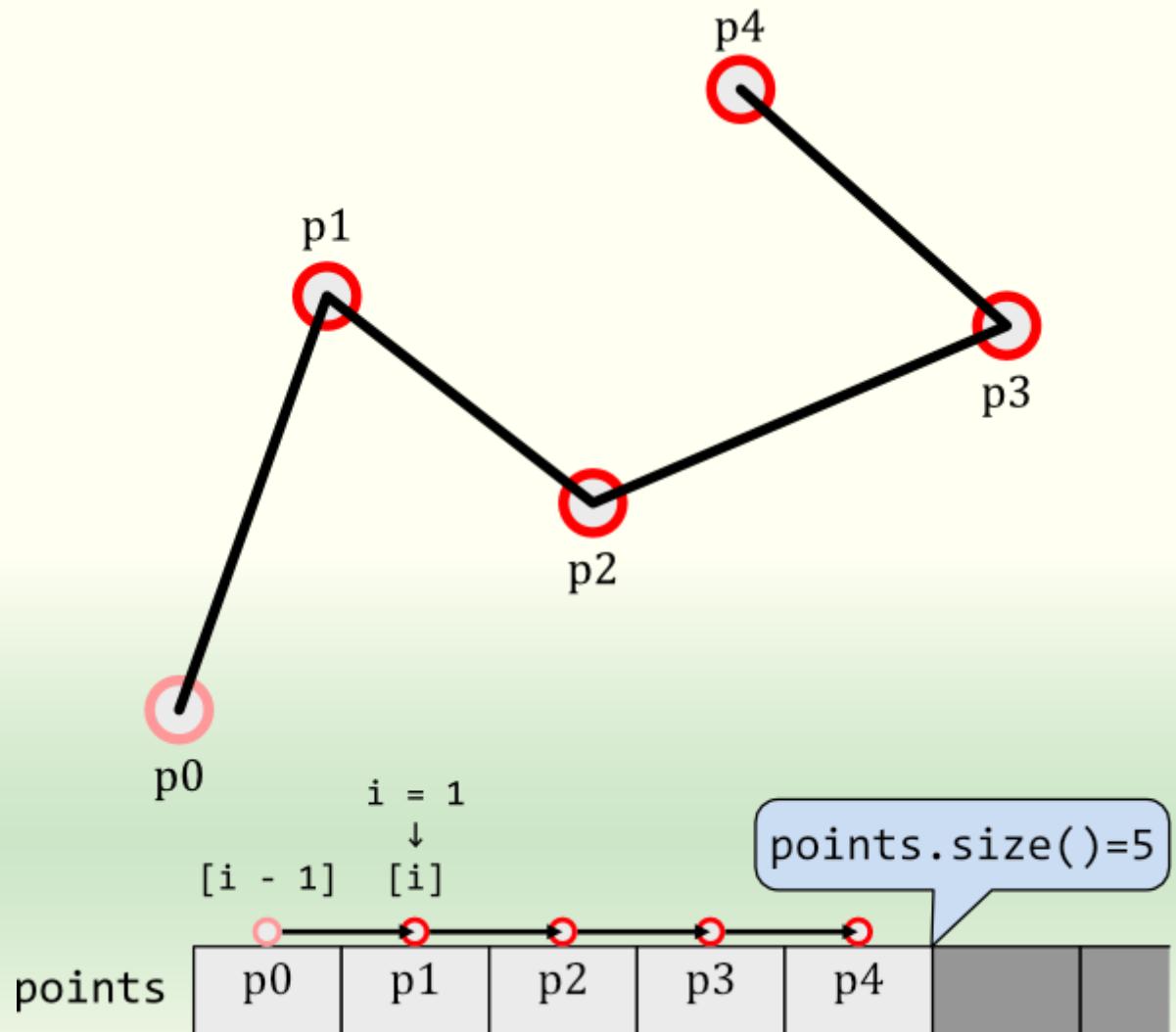




# 補足

添え字を使ったデータの取り出し

# クリックしたところを線分で結ぶ



- クリックした位置は `points` という `vector` に  $p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow \dots$  という順に入っている
- $p_1$  から 1 つ前の点 ( $p_1$  に対しては  $p_0$ ) との間に線分を引く
- $p_2, p_3, \dots$  と進めて最後の点で終わる
- 点の数は `points.size()` で調べられる

# 2つずつ点を結ぶ

```
#include "ofApp.h"

//-----
void ofApp::setup(){
    ofBackground(255, 255, 255);
}

//-----
void ofApp::update(){

}

//-----
void ofApp::draw(){
    for (size_t i = 1; i < points.size(); ++i){
        ofSetColor(0, 0, 0);
        ofDrawLine(points[i - 1], points[i]);
    }
}
```

- $i = 1$  ( $p_1$ ) から始める
- $i$  が点の数  $\text{points.size()}$  に達しないなければ ( $i < \text{points.size}()$ ) 以後の処理を行う
- 1つ前の点  $i - 1 = 0$  ( $p_0$ ) から現在の点  $i = 1$  ( $p_1$ ) との間に線分を引く
- $++i$  により  $i$  を  $2, 3, \dots$  と増やしながら繰り返す ( $p_2, p_3, \dots$  と進める)