



# メディアプログラミング演習

第1回

# 本演習の目的

- プログラミングでメディアを取り扱う

- メディアとは

- media (メディア) は medium (メディウム) の複数形

- すなわち中間にあるもの・間に入って媒介するものたち

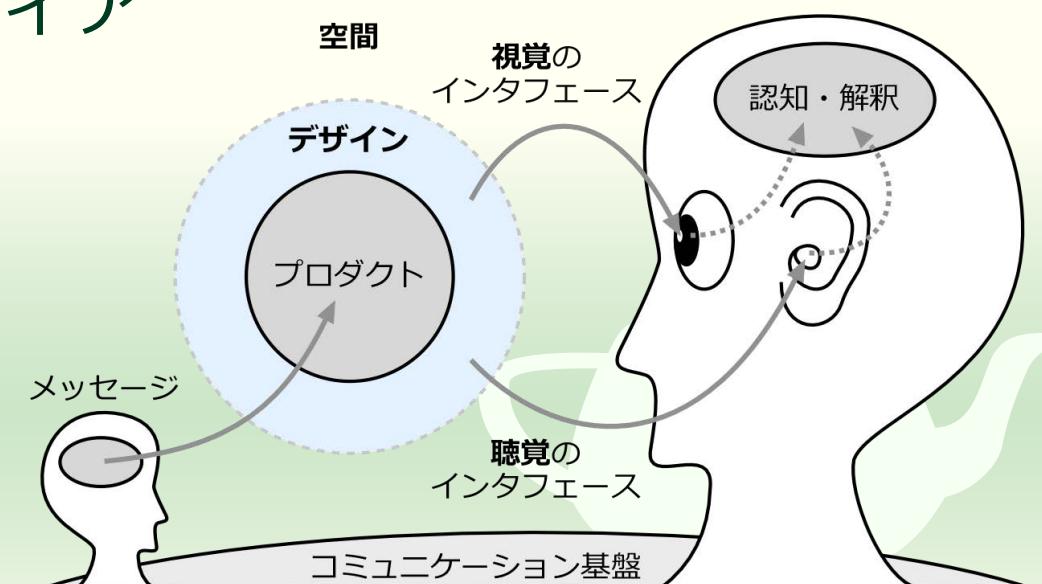
- コミュニケーションにおけるメディア

- 音, 音声, 音楽

- 画像, 映像

- デザイン

- 人と人の間にあるもの





# プログラミングとは

あなたがすべきことがどこかに書いてあるわけではない

# プログラムするということ・序

- 課題を解決する手順や方法を考える
  - つまりアルゴリズムを考える
    - 問題を解く手順を定式化したもの
    - 算法ともいう
- コンピュータが取り扱い可能な形式で記述する
  - つまりソースプログラムを作成する
    - アルゴリズムをもとに問題を解く手順を記述したもの
    - 原始プログラム, ソースコードともいう
    - かつては算譜という和製漢語が用いられていたが廃れた



# プログラムするということ・破

- 実行したプログラムが正しく動作するか確かめる
  - つまりテストする
    - プログラムの機能が期待通り動作するかどうか
    - プログラムの性能が期待通り得られるかどうか
    - プログラムが悪意のある操作によって予想外の動作をしないか
- プログラムが正しく動作するよう修正する
  - つまりデバッグする
    - プログラムが正しく動作しない原因や理由を見つける
    - プログラムを正しく動作させる修正方法を考える



# プログラムするということ・急

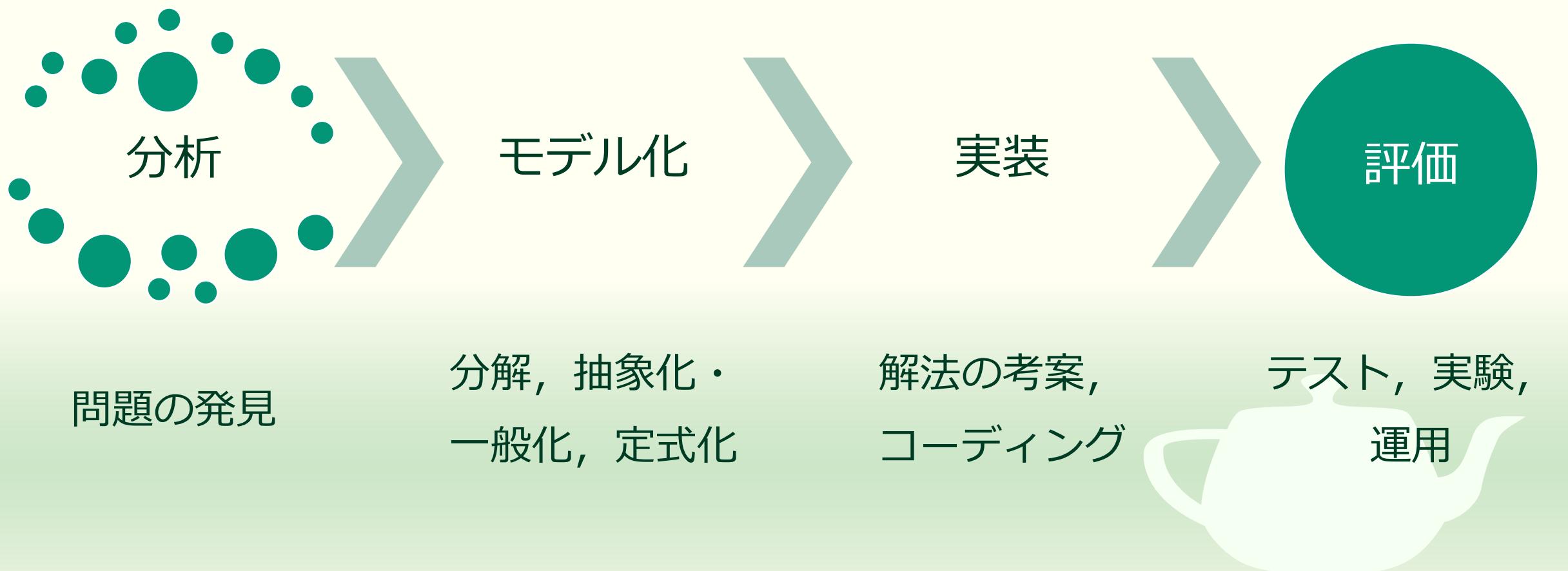
- まず課題を分析して正しい設計をする
- それでも書いたプログラムはたいてい正しく動かない
- 正しく動かない理由もたいてい分からない
- だから正しく動くようになるまで**試行錯誤**する

**試行錯誤できないとプログラミングできない**

# プログラミングの学習

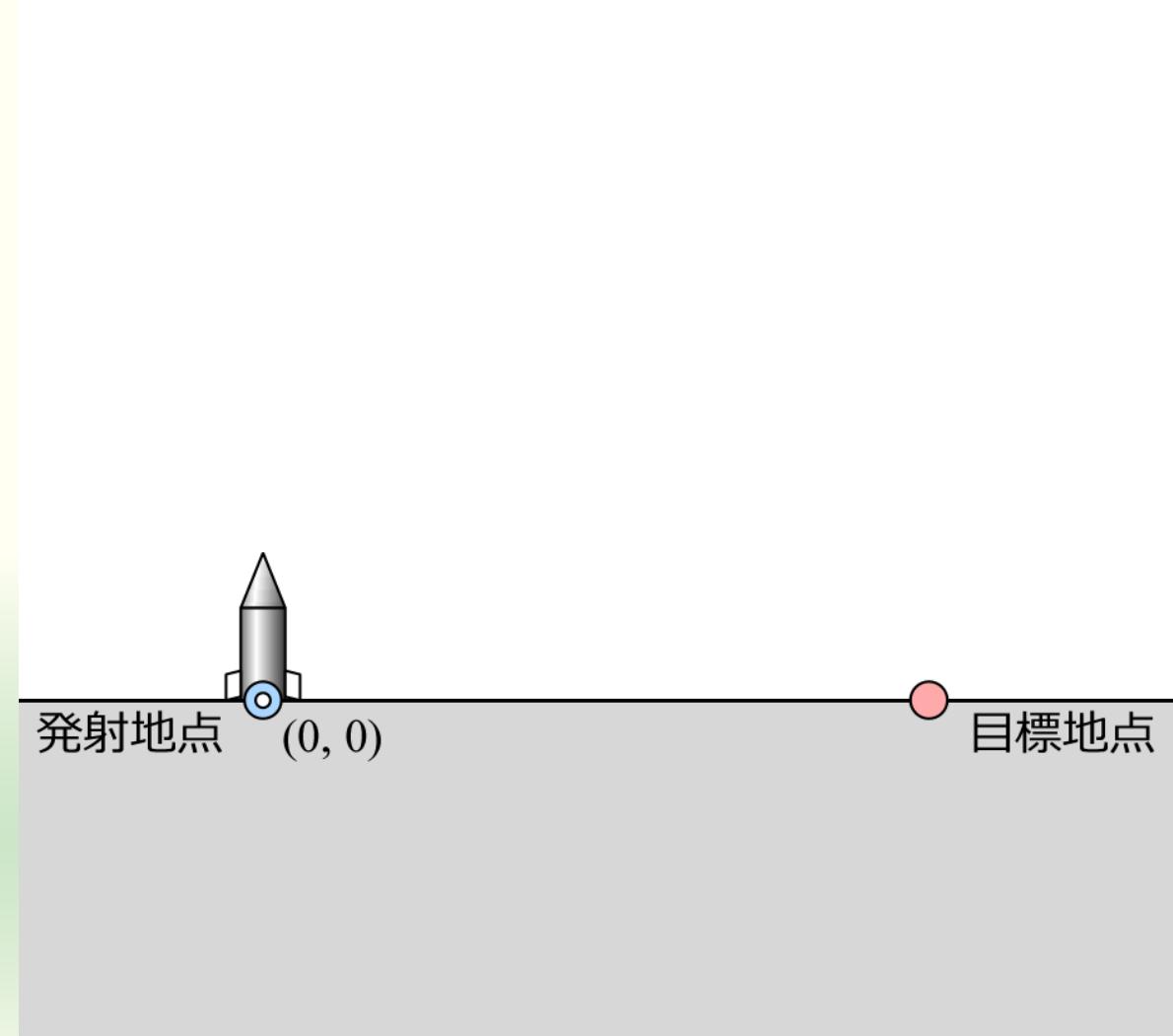
- 教科書を読んだだけでは書けるようにならない
  - 文法を学んでも目的を達成する方法は分からぬ
  - プログラムは書かないと書けるようにならない
- プログラミングが苦手という人のパラドックス
  - 書けないのに書くことはできないと思う
  - 最初に何をしたらいいのか分からぬ
- まず**何のため**（目的）に**何をする**（処理）の**か決める**
  - プログラミングは**処理の手順**（手続き）を考えること

# 「プログラミング的思考」



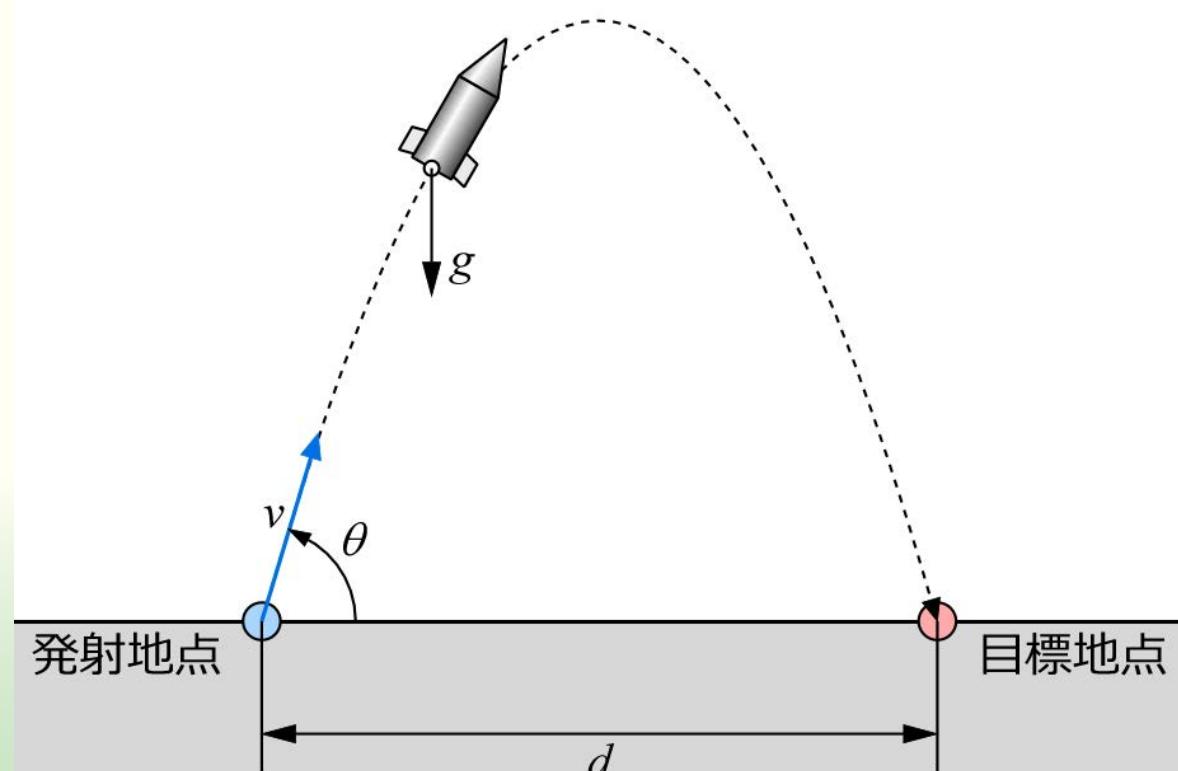
# 課題の例

- 弹道ミサイルを目標地点に到達させたい
- 考えること
  - 既知のこととは何か
  - 未知なことは何か
  - 求めることは何か



# 分析

- 弹道ミサイルの速度は  $v$
- 目標地点までの距離は  $d$
- 重力加速度は  $g$
- 条件
  - 空気抵抗は考慮しない
  - 自分で速度を変えない
  - 自分で向きを変えない
- 発射角  $\theta$  を求める

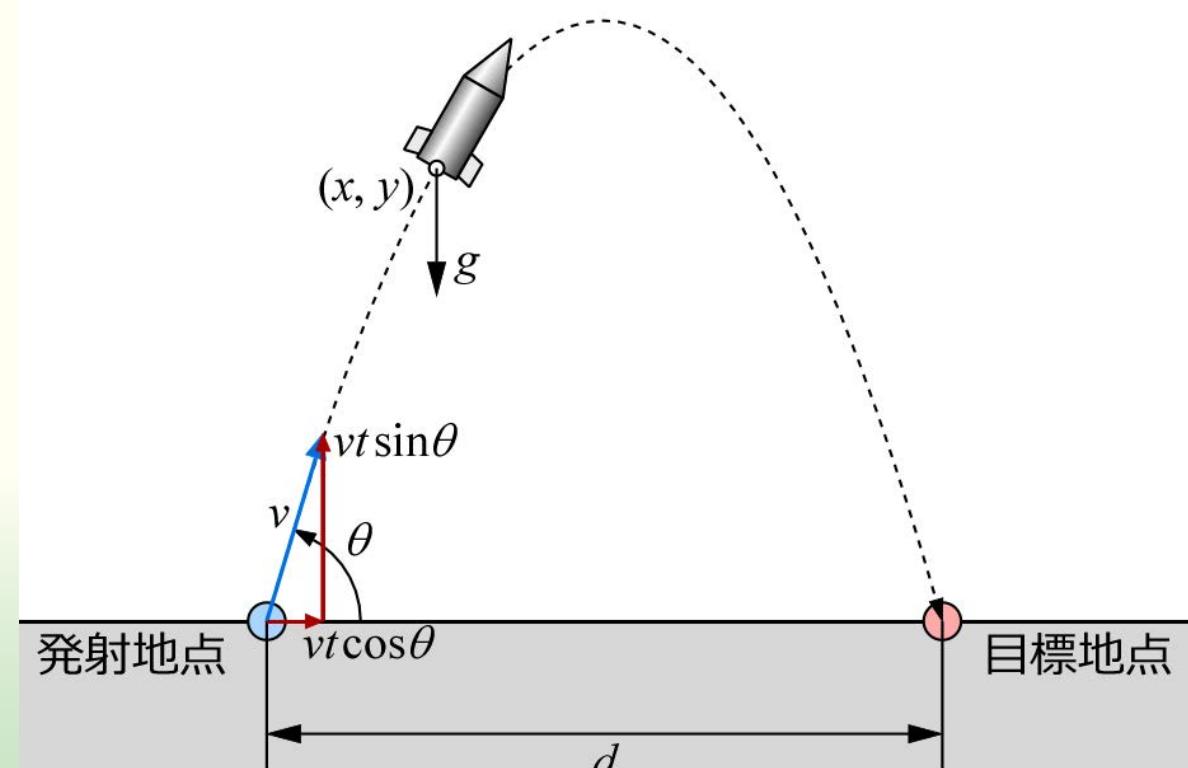


# モデル化

- 弹道ミサイルの軌跡
  - 軌跡は放物線を描く
  - 時刻を  $t$  とする

$$x = vt \cos \theta$$

$$y = vt \sin \theta - \frac{1}{2} gt^2$$

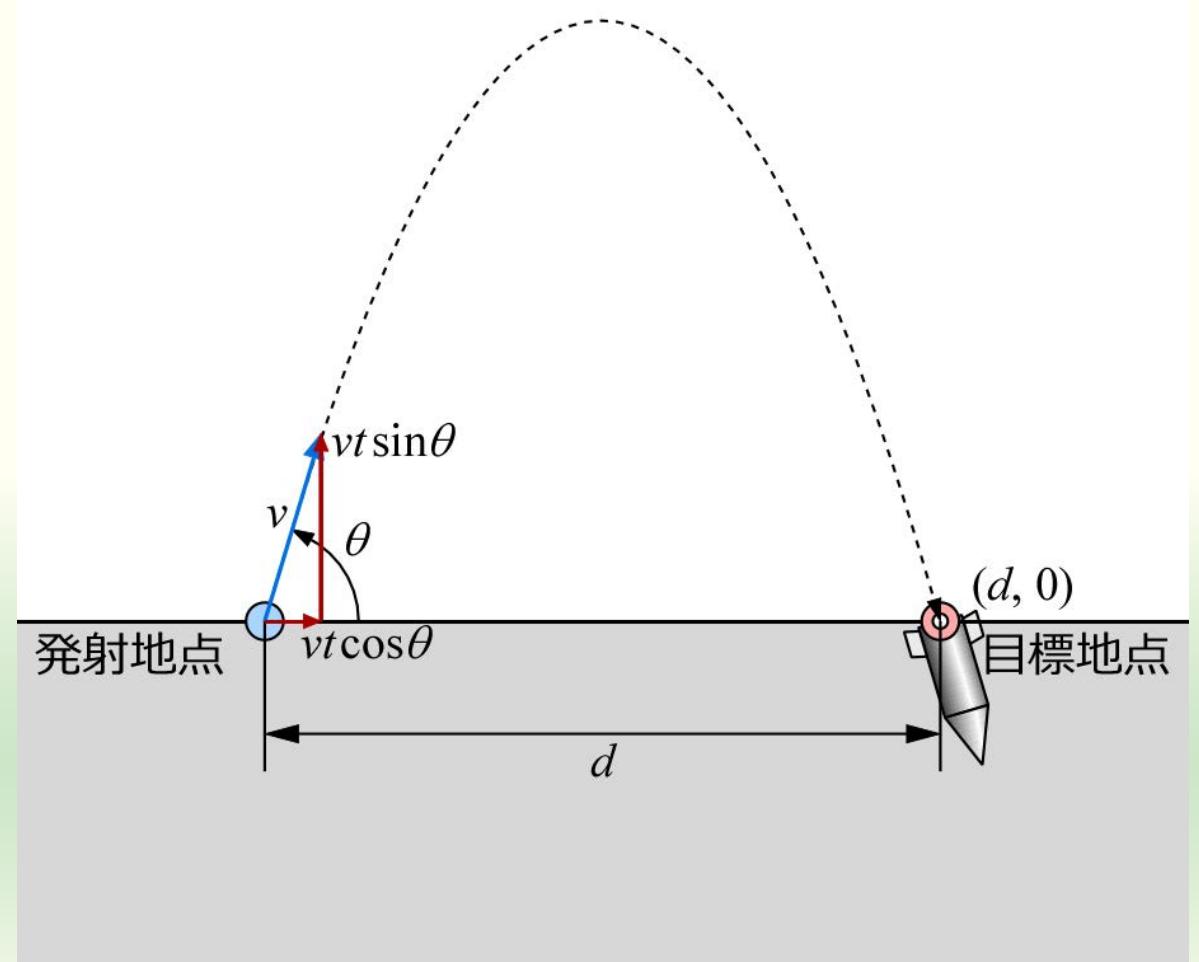


# 実装 (解法の考案)

## ■ 目標地点に到達したとき

$$x = vt \cos \theta = d$$

$$y = vt \sin \theta - \frac{1}{2}gt^2 = 0$$



# 実装 (解法の考案)

- $x$  の式より

$$t = \frac{d}{v \cos \theta}$$

- $y$  の式の  $t$  に代入して

$$\frac{d \sin \theta}{\cos \theta} - \frac{gd^2}{2v^2 \cos^2 \theta} = 0$$

- $(2\cos^2 \theta)/d$  倍して移項

$$2 \sin \theta \cos \theta = \frac{gd}{v^2}$$

- 二倍角の公式より

$$\sin 2\theta = \frac{gd}{v^2}$$

$$\theta = \frac{1}{2} \sin^{-1} \frac{gd}{v^2}$$

これをプログラムで計算するということを決めてようやくコーディング

# 実装（コーディング）

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    // 重力, 速度, 距離
    double gravity, velocity, distance;

    // パラメータの入力
    cout << "重力加速度:";
    cin >> gravity;
    cout << "発射速度:";
    cin >> velocity;
    cout << "目標までの距離:";
    cin >> distance;

    // パラメータのチェック
    if (gravity <= 0.0) {
        cout << "重力が小さすぎます\n";
        return 1;
    }
```

## ソースプログラム

```
if (velocity <= 0.0) {
    cout << "速度が小さすぎます\n";
    return 1;
}
if (distance <= 0.0) {
    cout << "距離が近すぎます\n";
    return 1;
}
```

解法のコードはこれだけ

```
double s = gravity * distance;
double t = velocity * velocity;

if (s > t) {
    cout << "目標に届きません\n";
    return 1;
}

// 結果の出力
cout << "発射角度:" << asin(s / t) * 0.5 << "\n";
```

# プログラミング言語

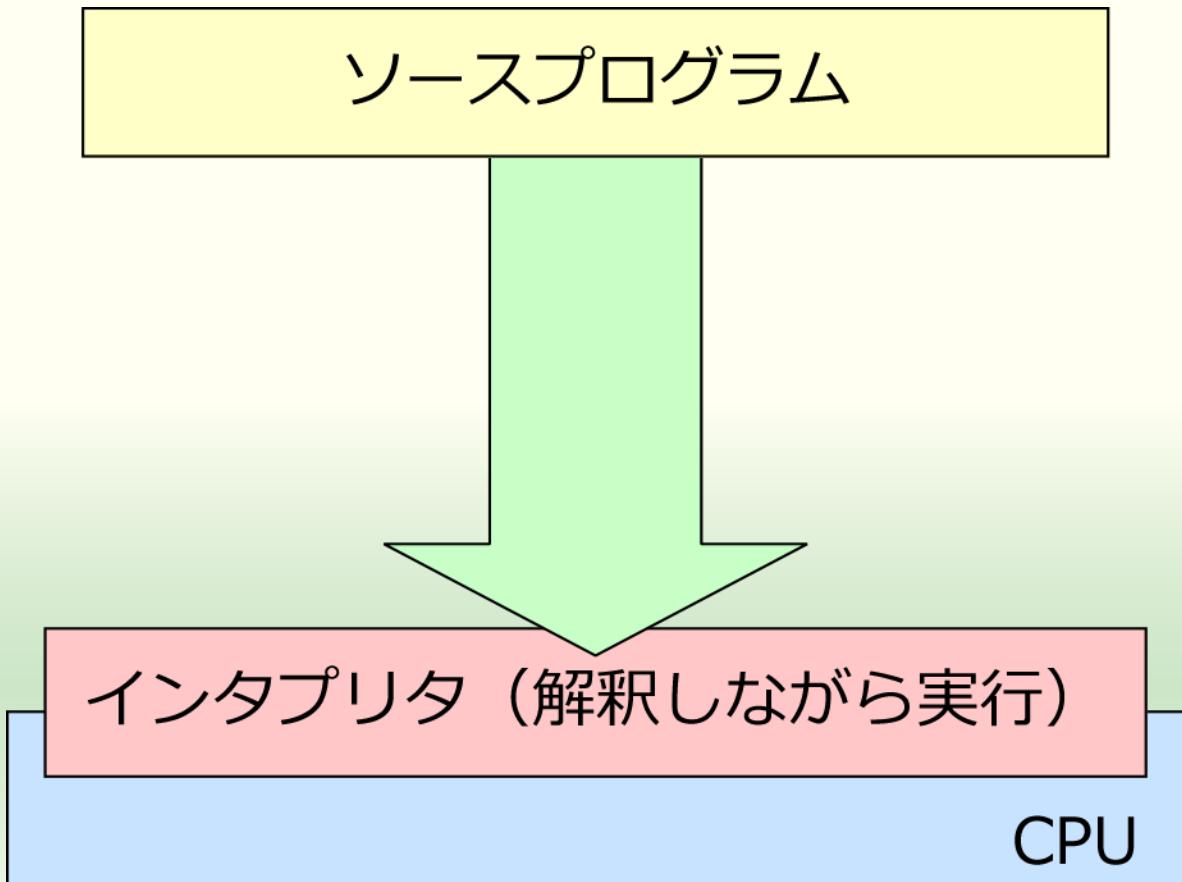
---

- ソースプログラムの記述に用いる言語
  - 言語なので**文法**が決められている
  - 文法に違反した記述は**エラー**になる
- ソースプログラムは人間が記述する
  - コンピュータが理解可能な**機械語**への翻訳が必要
    - 逐次的に解釈しながら実行
      - インタプリタ方式
    - 一括して翻訳した後に実行
      - コンパイラ方式

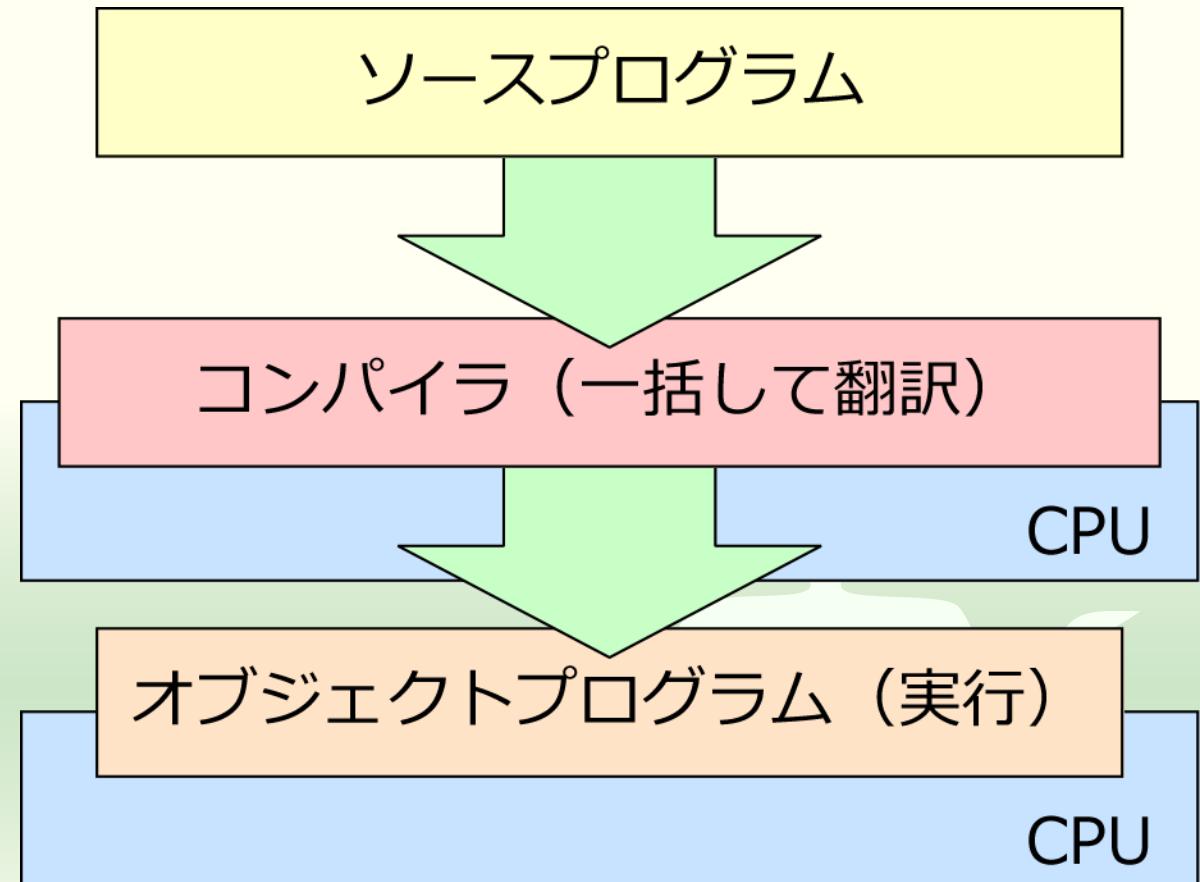


# インタプリタ方式とコンパイラ方式

## インタプリタ方式



## コンパイラ方式



# ソースプログラムのコンパイル

## C++ 言語のソースプログラム

```
// 関数 add の定義
//
int add(int x, int y)
{
    // 変数 z の宣言
    int z;

    // 引数 x と引数 y を足して z に代入する
    z = x + y;

    // z を関数の戻り値 (関数の値) とする
    return z;
}
```

コンパイル  
(一括翻訳)

## アセンブリ言語 (機械語に対応)

```
add:
    pushq %rbp
    movq %rsp, %rbp
    subq $16, %rsp
    movl %ecx, 16(%rbp)
    movl %edx, 24(%rbp)
    movl 16(%rbp), %edx
    movl 24(%rbp), %eax
    addl %edx, %eax
    movl %eax, -4(%rbp)
    movl -4(%rbp), %eax
    addq $16, %rsp
    popq %rbp
    ret
```

アセンブリ言語は数値の羅列である  
機械語の命令を文字による簡略表記  
(ニーモニック) で表したもの

# インタプリタ方式のプログラミング言語

- Python, Ruby, Perl, JavaScript, PHP, Prolog, ...
  - 軽量言語 (Light Weight Language, LL), スクリプト言語
  - Web や機械学習などサービス開発で利用されるものが多い
- 特徴
  - CPU によって実行されるのはインタプリタ
  - インタプリタがソースプログラムを解釈しながら実行する
  - コンパイル (翻訳) の必要がない



# コンパイラ方式のプログラミング言語

- C, C++, Objective-C, Swift, Go, Fortran, COBOL, ...
  - OS やアプリケーションの開発に使用されるものが多い
  - 他のプログラミング言語の処理系の開発にも使用される
    - C 言語は C 言語で開発されているが Python (のリファレンス実装の一つ) CPython は C 言語で開発されている
- 特徴
  - CPU は機械語のオブジェクトプログラムを直接実行する
  - 実行前にあらかじめコンパイル (翻訳) する必要がある
  - 実行時にソースプログラムの解釈を行わないので高速

# 中間言語方式のプログラミング言語

- C#, Java, Kotlin, Scala, Pascal, Smalltalk, ...
  - 近年のアプリケーション開発に用いられることが多い
- 特徴
  - コンパイラを使って中間言語に翻訳
  - 中間言語のプログラムをインタプリタによって実行する
  - インタプリタ方式とコンパイラ方式の中間的な特徴をもつ
  - コンパイルが速く実行速度もコンパイル方式に迫る
  - 中間言語のインタプリタが用意された多様な環境で使える

JIT (Just In Time) コンパイラ  
により解釈しながら機械語  
への変換を行うものが主流

# プログラミングパラダイム

- 命令型プログラミング言語
  - 手続き型プログラミング言語
    - C, C++, C#, Java, JavaScript など大半のプログラミング言語
- 宣言型プログラミング言語
  - 関数型プログラミング言語
    - LISP, Ocaml, Erlang, Scala, Haskell, F#
  - 論理型プログラミング言語
    - Prolog, GHC

パラダイム  
ある時代や分野において  
支配的規範となる  
「物の見方や捉え方」

# オブジェクト指向

---

- プログラミングパラダイムの一つ
  - 命令型・宣言型の分類などとは直交する概念
  - C++, C#, Java, Python など近年のプログラミング言語で採用
- オブジェクト
  - 何らかの役割を与えられた実体（メモリ）
- 長くなるので細かい話は割愛
  - 別に「オブジェクト指向」という講義がある（くらい）
  - この演習では便利な機能として利用するだけ





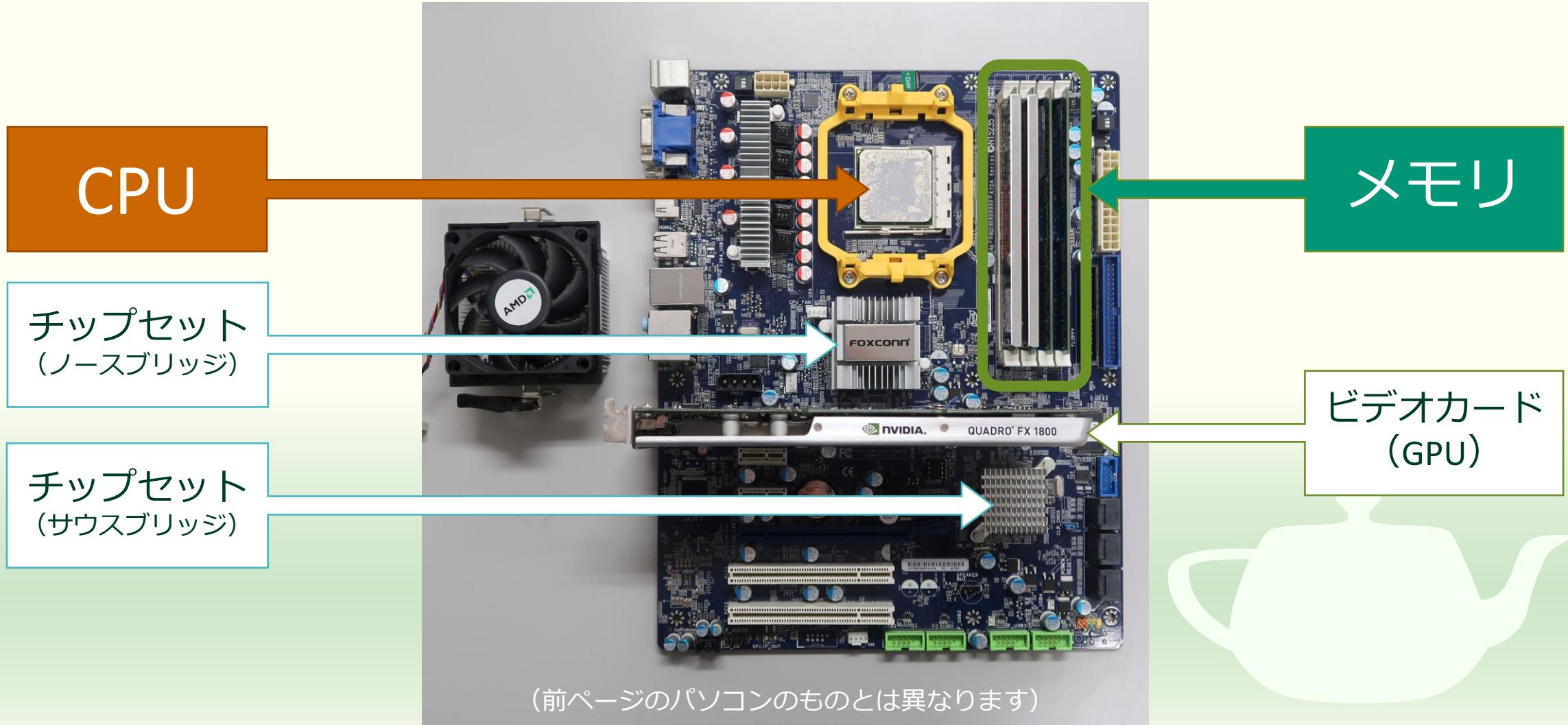
# C++ 言語によるプログラミング

C++ 言語は初心者に向いていないらしい

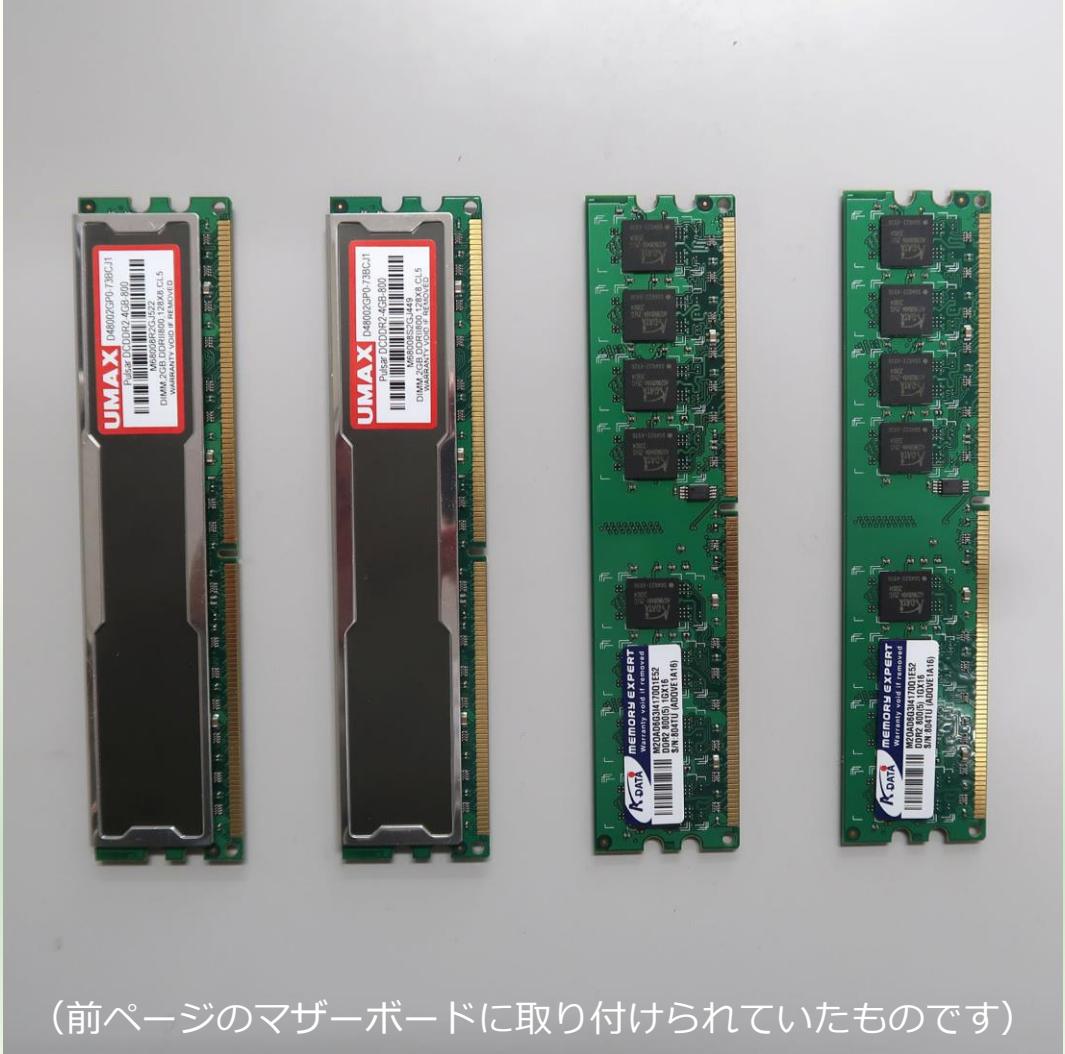
# パソコンの外観と内部



# マザーボード

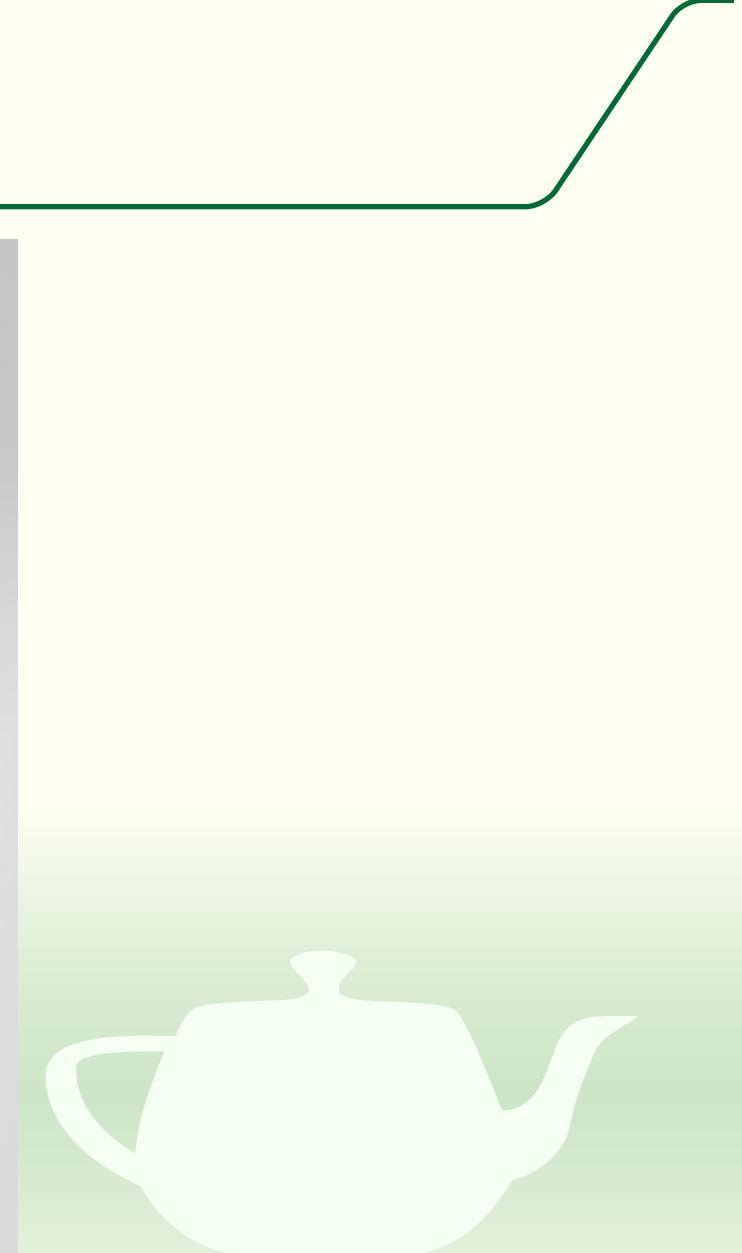


# CPU とメモリ



(前ページのマザーボードに取り付けられていたものです)

# ビデオカード (GPU)



# 周辺機器との接続



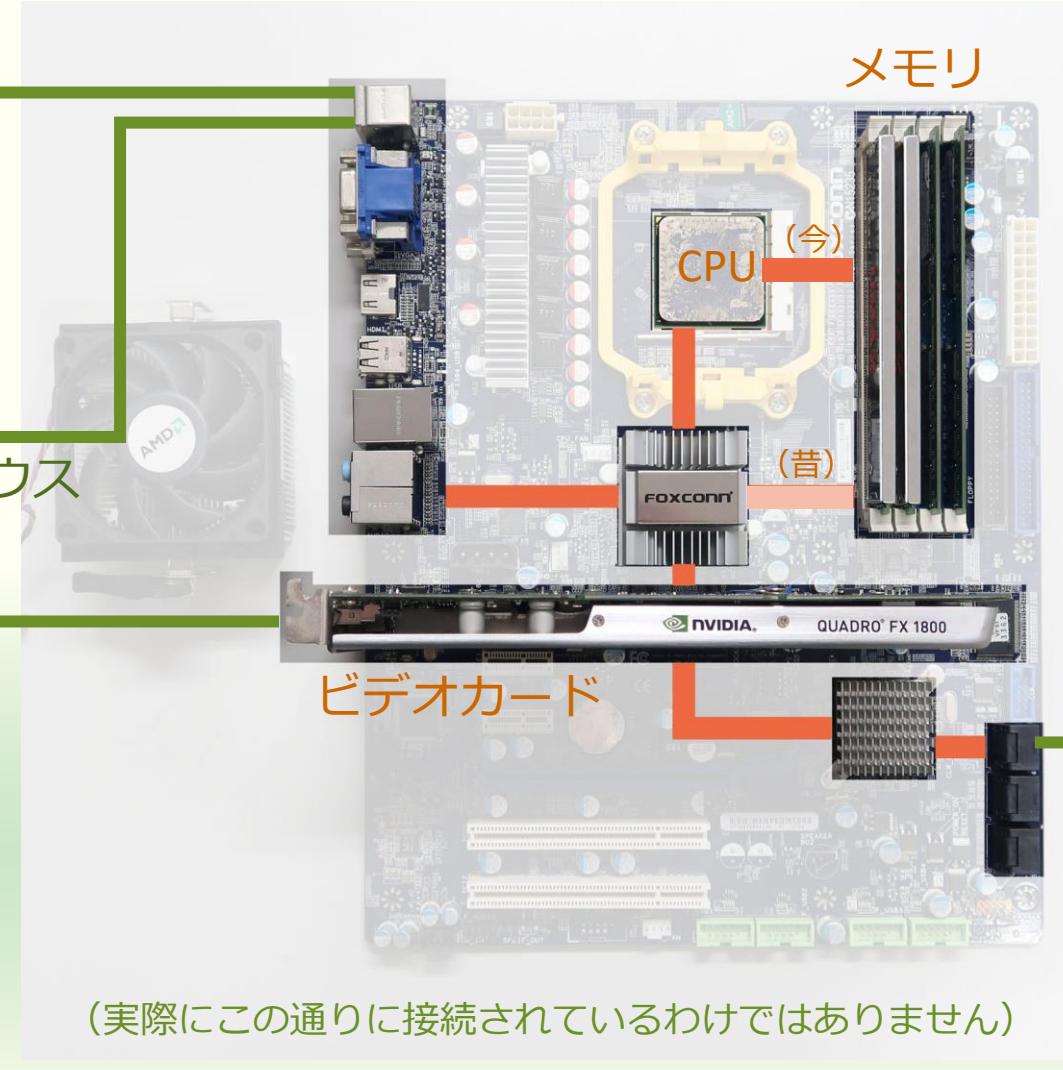
キーボード



ディスプレイ

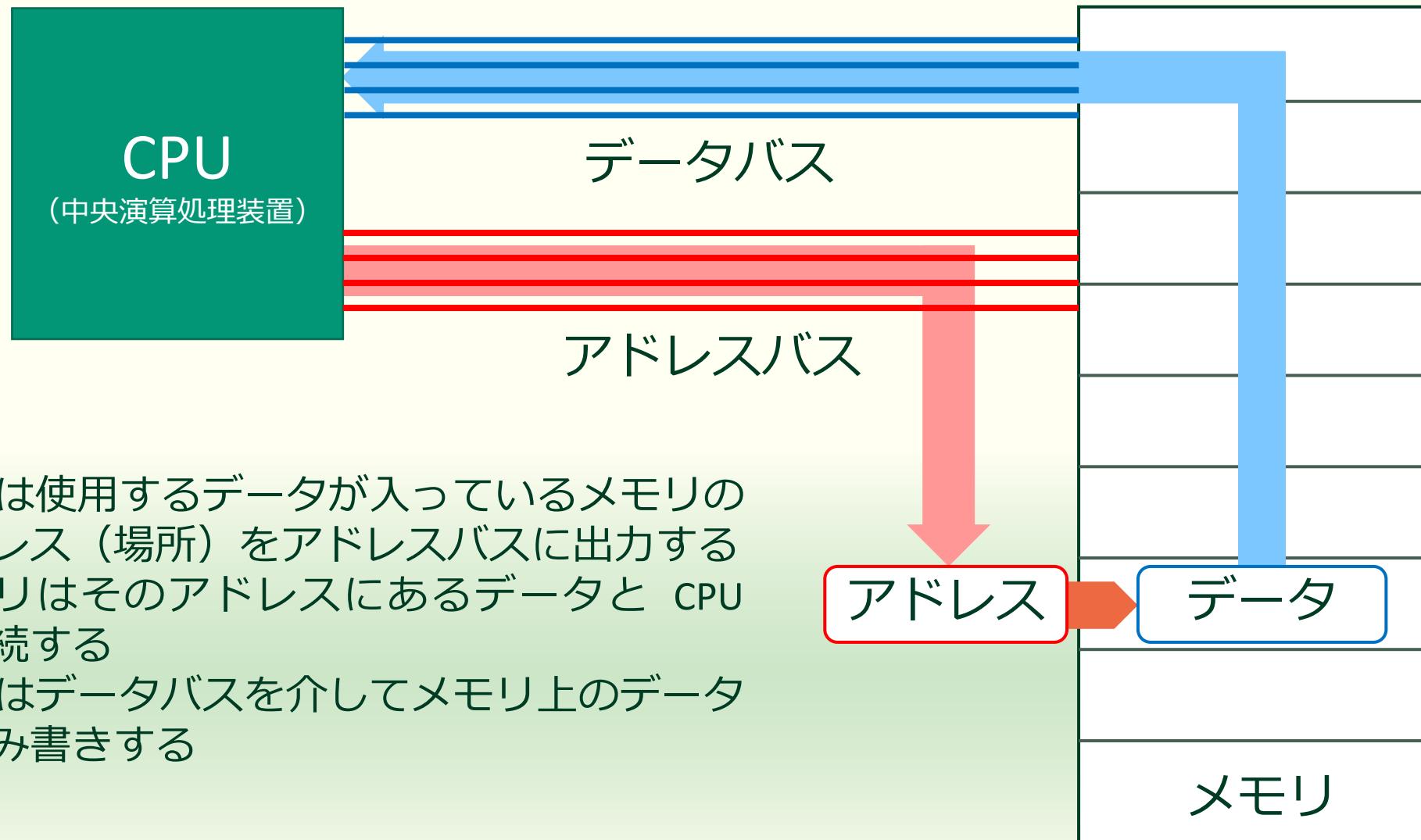


マウス

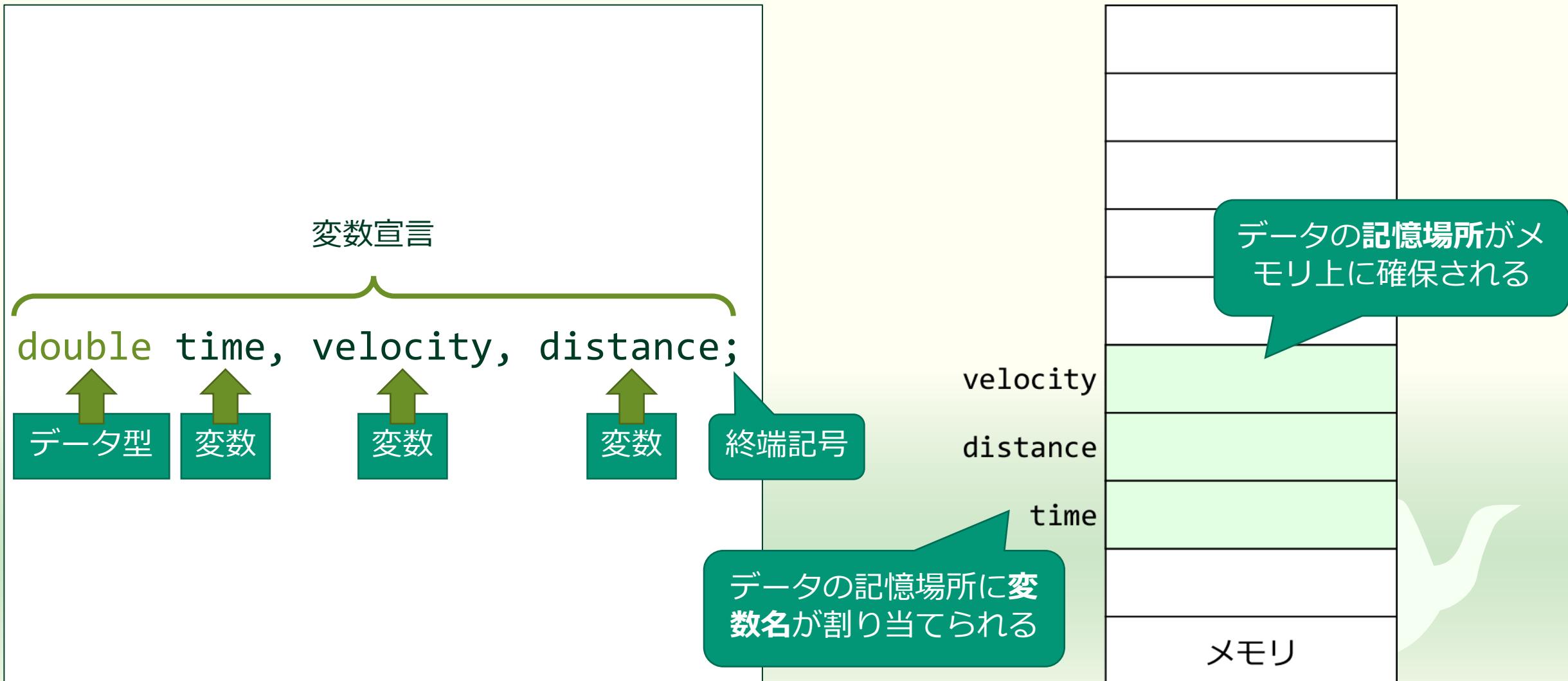


HDD / SSD

# CPU とメモリの関係



# メモリを使うには変数を宣言する



# C++ 言語ではデータ型が重要な意味を持つ

- データ型とは
  - プログラミング言語などでデータを性質ごとに分類した際の種類
    - 整数型
    - 実数型
    - 文字型
    - ...
- 「数値」と「数字」は性質の違うデータ
- データ型をあまり気にしないように見える言語もある
  - データ型という考え方初学者には難しいと思われていたりする
    - Python, JavaScript, Perl, PHP, ...
  - でもデータ型を明示しないだけで気を付けてないとハマる

# データ型 (基本)

## 整数型

char	■ 8ビット精度符号付き整数 ■ -128～127
int	■ 32ビット精度符号付き整数
long	■ -2,147,483,648～2,147,483,647

## 実数型

float	■ 32ビット単精度浮動小数点 ■ 精度は10進で約6.92桁
double	■ 64ビット倍精度浮動小数点 ■ 精度は10進で約15.65桁

## 整数型のバリエーション

signed char	char と同じ
unsigned char	8ビット符号無し整数, 0～255
short, short int, signed short int	16ビット符号付き整数, -32,768～32,767
unsigned short, unsigned short int	16ビット符号無し整数, 0～65,535
signed, signed int, long, long int, signed long int	int と同じ
unsigned, unsigned int, unsigned long, unsigned long int	32ビット符号無し整数, 0～4,294,967,2965

# そのほかのデータ型（基本）

bool	論理型, true (真) と false (偽) の値だけを持つ
enum	列挙型, 名前を付けた定数の組のうちのどれか一つの値を持つ
long long, signed long long	64ビット精度符号付き整数, -9,223,372,036,854,775,808～9,223,372,036,854,775,807
unsigned long long	64ビット精度符号無し整数, 0～18,446,744,073,709,551,615
long double	double と同じ

## int と long が同じ理由

初期のパソコンの CPU は 8 ビットで、それが 16 ビット、32 ビット、64 ビットと発展してきました。C++ 言語のもとになった C 言語がパソコンで使われ始めたのは 16 ビットの頃だったため、その int は 16 ビット、long は 32 ビットになっていました。パソコンの CPU が 32 ビット化したことで C 言語の int にも 32 ビット割り当てられるようになりましたが、long は互換性のために 32 ビットのままに据え置かれました。CPU が 64 ビット化したときも互換性のために int、long とともに 32 ビットに据え置かれました。なお、64 ビットの整数を扱う場合は long long、unsigned long long が使用されます。

## long double と double が同じ理由

これは Visual Studio の C++ 言語処理系である Visual C++ の仕様で (int と long の関係も Visual C と x86 系の CPU に依存した話)、他の言語処理系では同じ x86 系でも long double が 80 ビットになっているものもあります。これは初期の x86 系 CPU にオプションで追加する浮動小数点演算ハードウェアが内部的に 80 ビットで計算していたのをそのまま使えるようにするためにですが、データサイズは 128 ビットとなり 48 ビット無駄に使います。128 ビットをフルに使い切る精度の実数データを扱えるように考えられてはいるものの、現時点では実装がまちまちの状態になっているようです。

# くどいようだが整数型と実数型がある

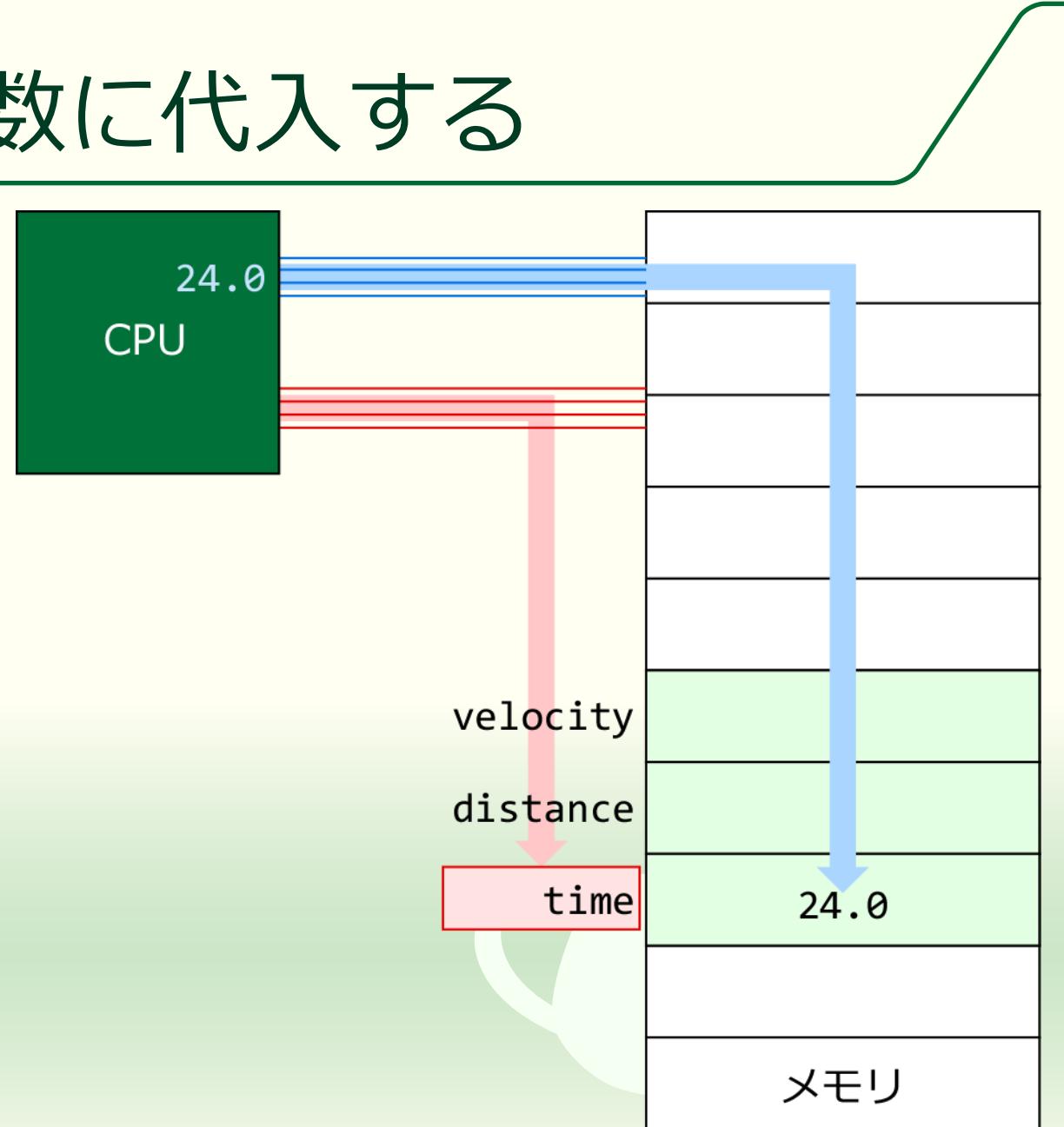
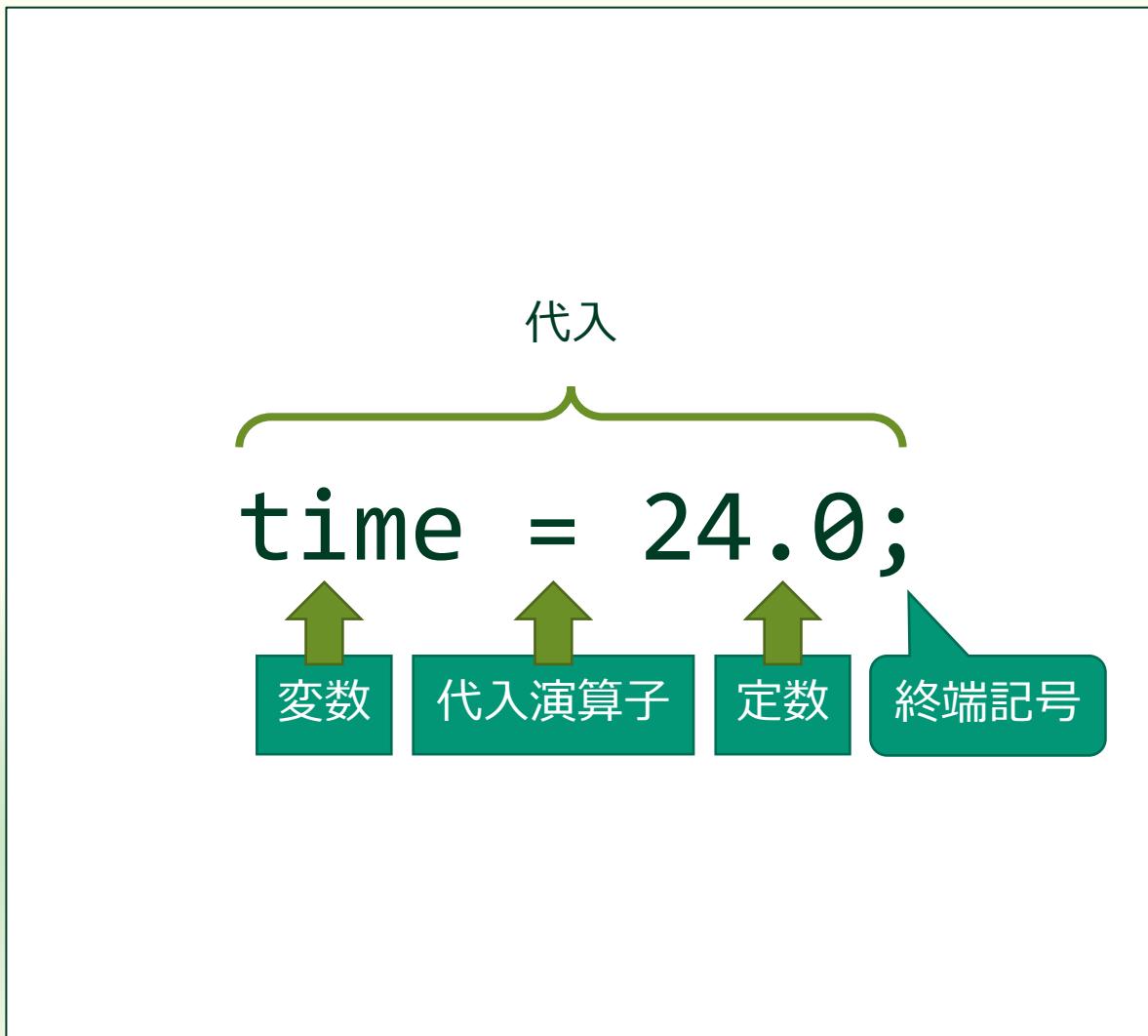
## 整数型 (char, int, long 等)

- (主に) 数を数えるのに使う
- 小数点以下を持たない
- 隣り合う数値の間隔は 1
- 除算は小数部が**切り捨て**される
  - $5 / 3 == 1$
  - $-5 / 3 == -1$
  - $1 / 2 == 0$
- 実数と混在するときは実数になる
  - $5 / 3.0f \doteq 1.66667$
  - $1.0f / 2 == 0.5f$

## 実数型 (float, double 等)

- 数値計算に使う
- 小数点以下を持つ
- 隣り合う数値の間隔が一定でない
  - $0.1f$  は実は  $0.1$  の**近似値**
- 非常に大きな数や非常に小さな数が表現できる
- 指数表記が可能
  - $1.23 \times 10^5$  は  $1.23e5f$
  - 末尾に  $f$  が付いているものは float 型の定数、無ければ double 型の定数

# データを記憶するには変数に代入する



# 手続き（処理）は関数として記述する

## ■ 数学

### ■ 関数を定義する

$$f(x) = x^2$$

### ■ 関数を評価する

$$y = f(2)$$

↑  
これは  $y$  が  $f(2)$  の値と  
等しいという意味

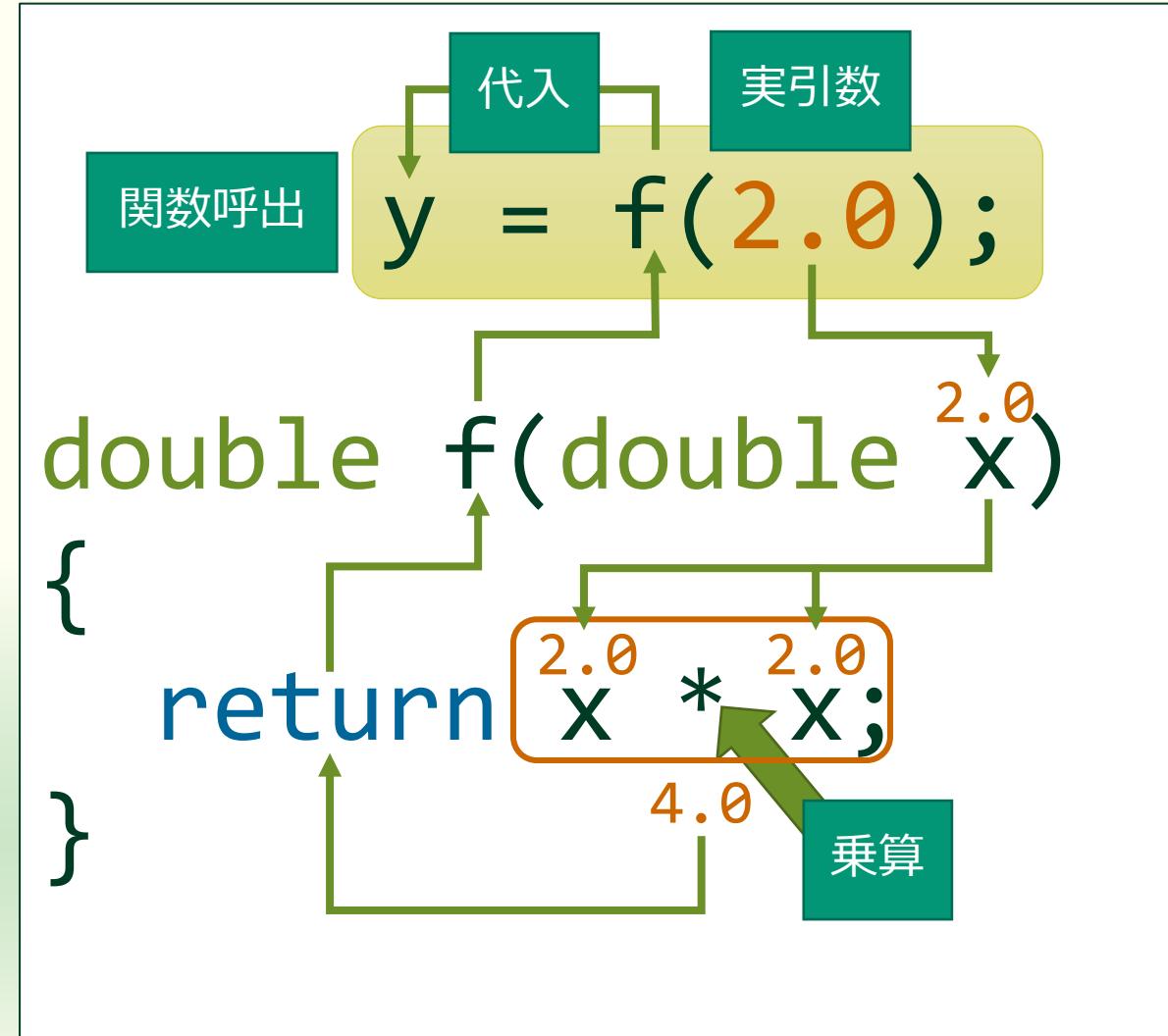
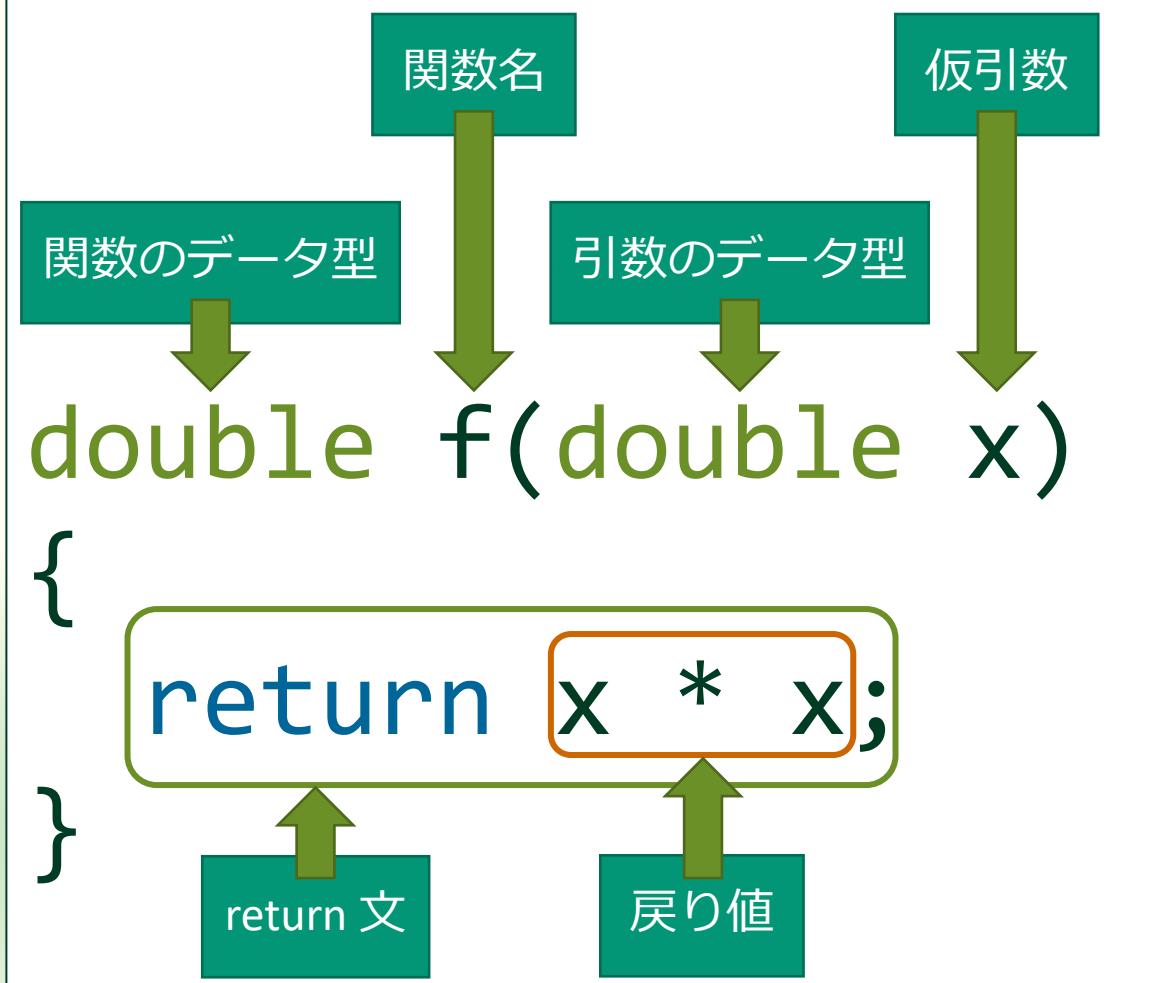
```
// 関数名が f の関数の定義
//
double f(double x)
{
    // 関数の戻り値として引数 x の二乗を返す
    return x * x;
}

int main()
{
    double y;

    // 関数 f を評価した値を y に代入する
    y = f(2.0);
}
```

↑  
これは  $y$  に  $f(2.0)$  の値を  
代入（格納）するという意味

# 関数の定義と関数の呼出し



## “//”より右はコメント

```
//  
// 関数名が f の関数の定義  
//  
double f(double x)  
{  
    // 関数の戻り値として引数 x の二乗を返す  
    return x * x;  
}
```

コメント  
プログラムの動作に  
影響を与えない  
メモや注釈



# 最初に評価される関数 main()

```
// // 関数名が f の関数の定義
//
double f(double x)
{
    // 関数の戻り値として引数 x の二乗を返す
    return x * x;
}

// // メインプログラム
//
int main()
{
    // 変数宣言
    double y;

    // 関数 f を評価した値を y に代入する
    y = f(2.0);
}
```

- プログラムは main() 関数から実行を開始する
- main() から関数 f() が呼び出される
- 関数 f() が実行された後 main() に戻る
- main() の return 文を省略すると 0 を返す





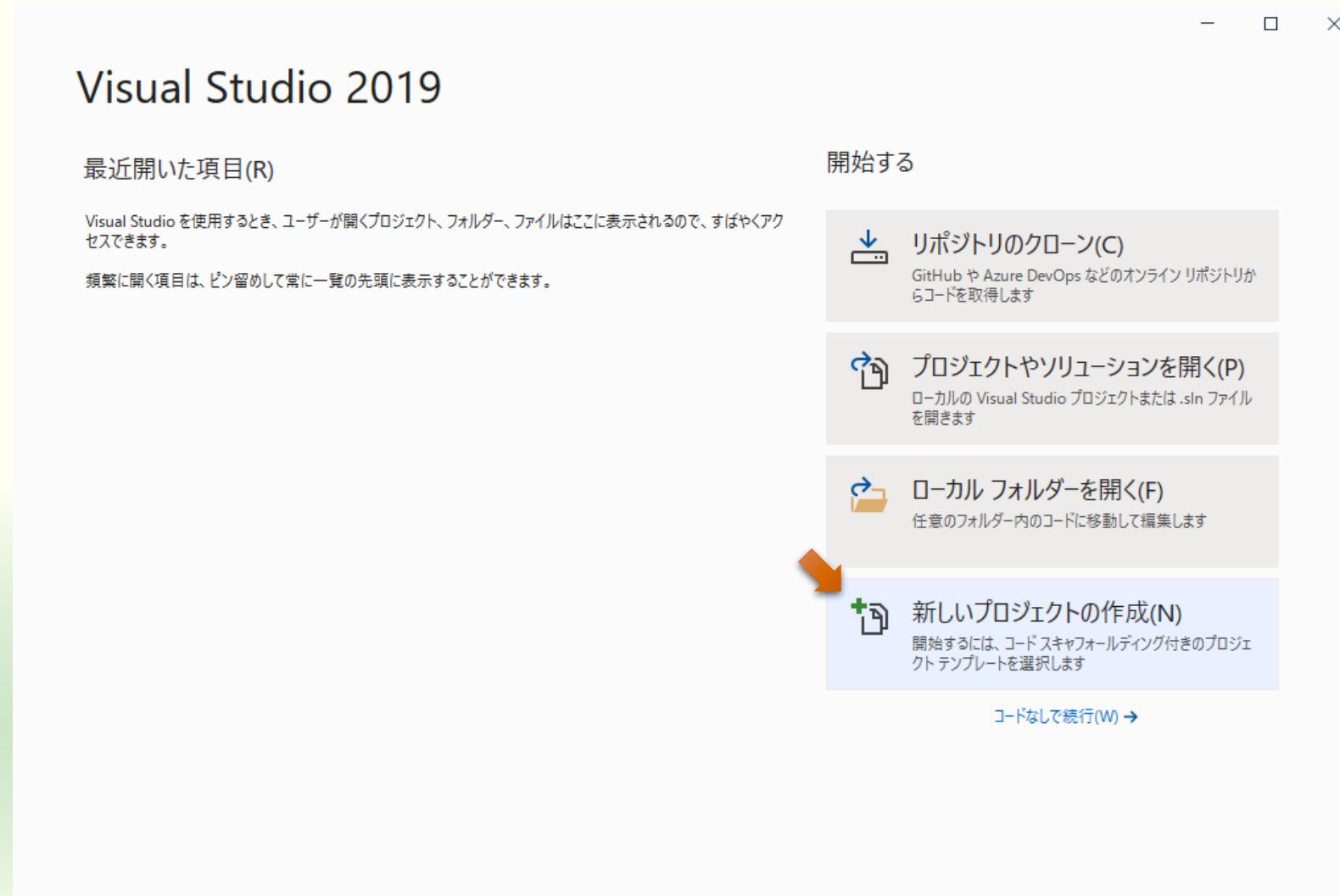
# 例題

Visuall Studio による C++ プログラム作成

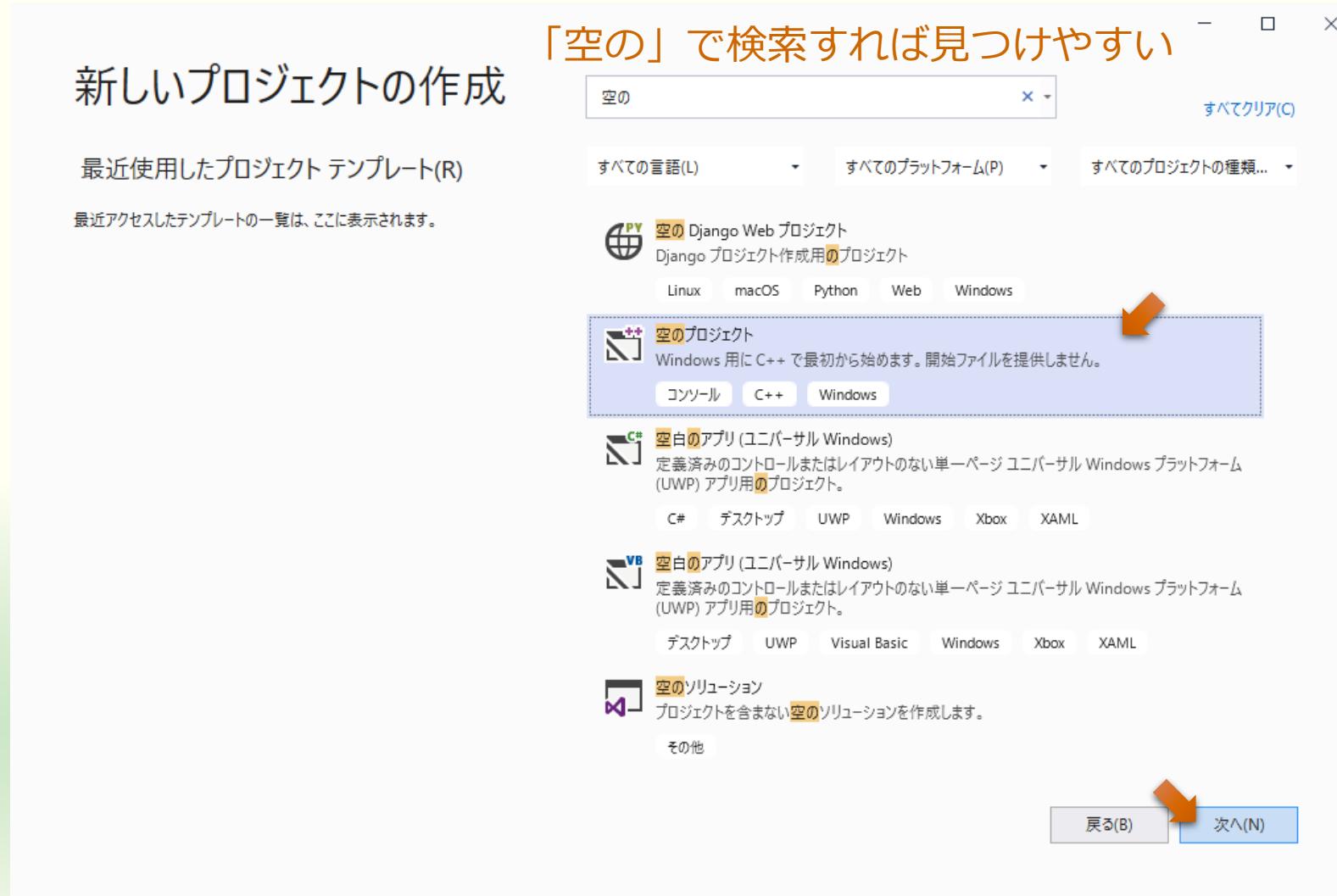
# Visual Studio を起動する



# 「新しいプロジェクトの作成 (N)」を選ぶ



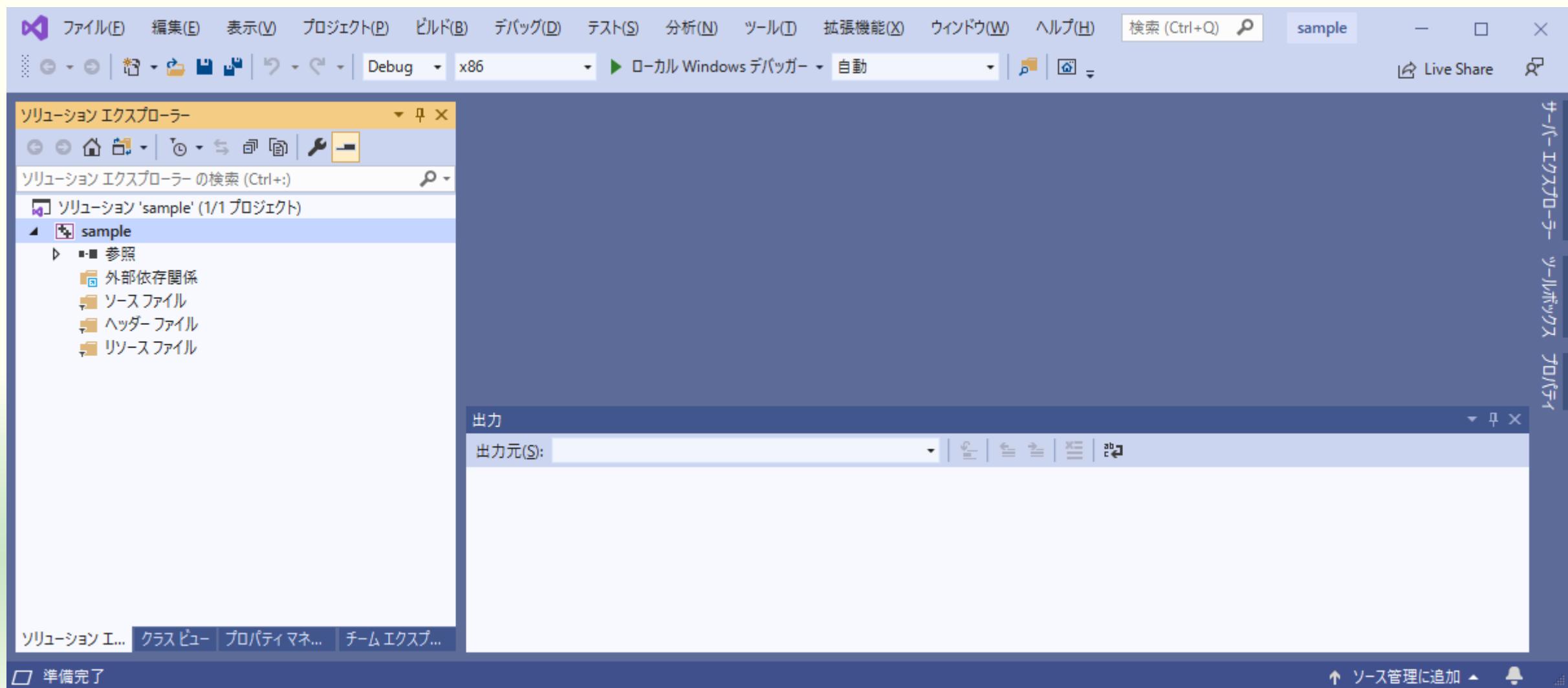
# 「空のプロジェクト」を作成する



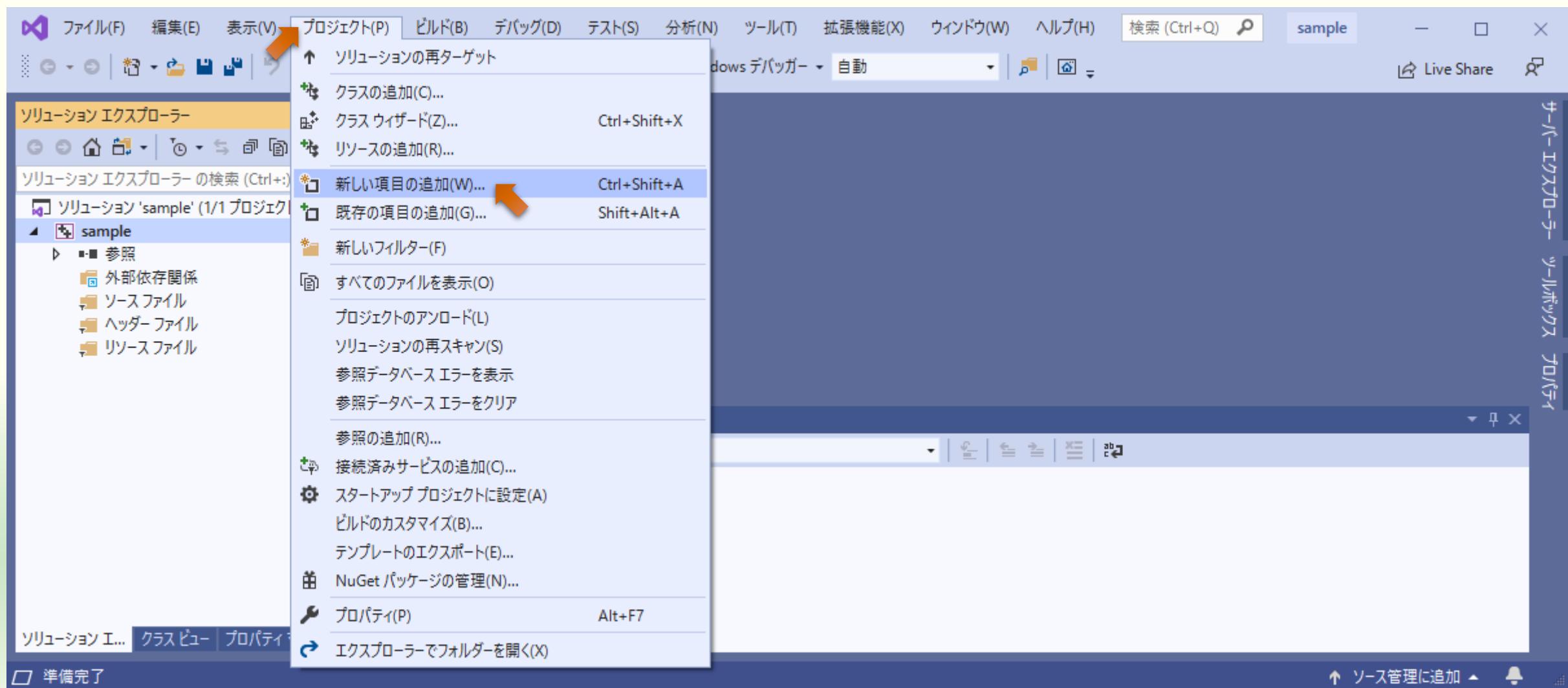
# 「プロジェクト名 (N)」を入力する



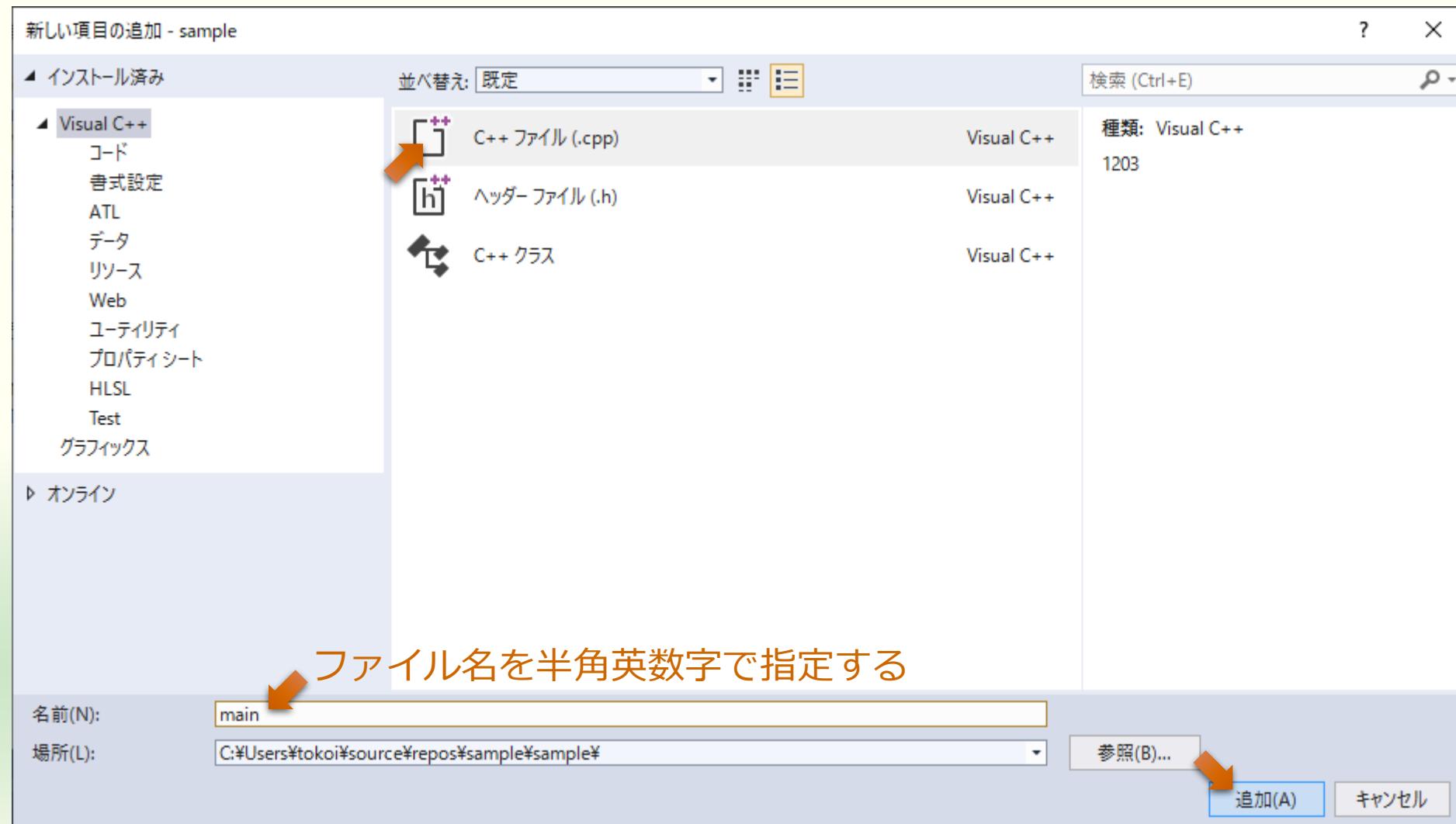
# 新しいプロジェクトが作成される



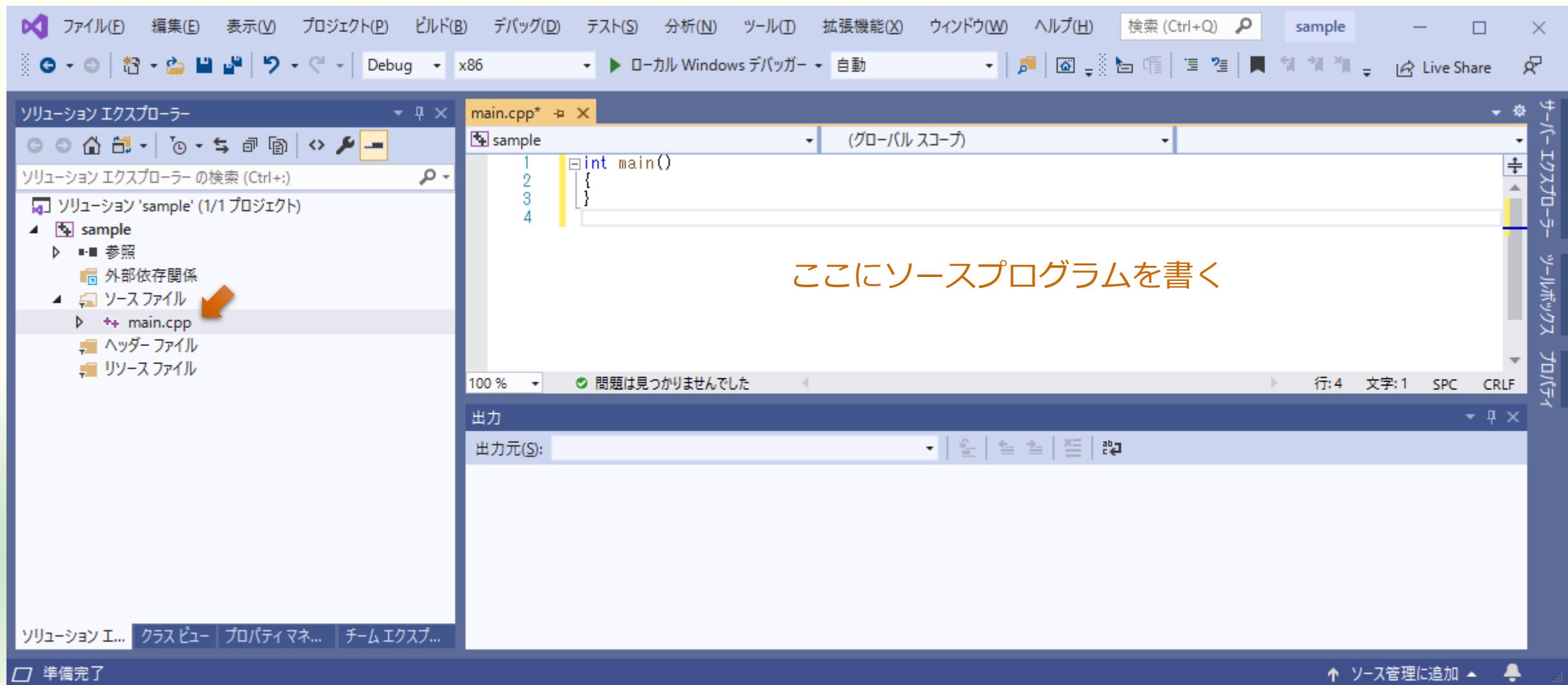
# 「新しい項目の追加 (W)」を選択



# 「C++ ファイル (.cpp)」を追加する



# ソースプログラムの編集



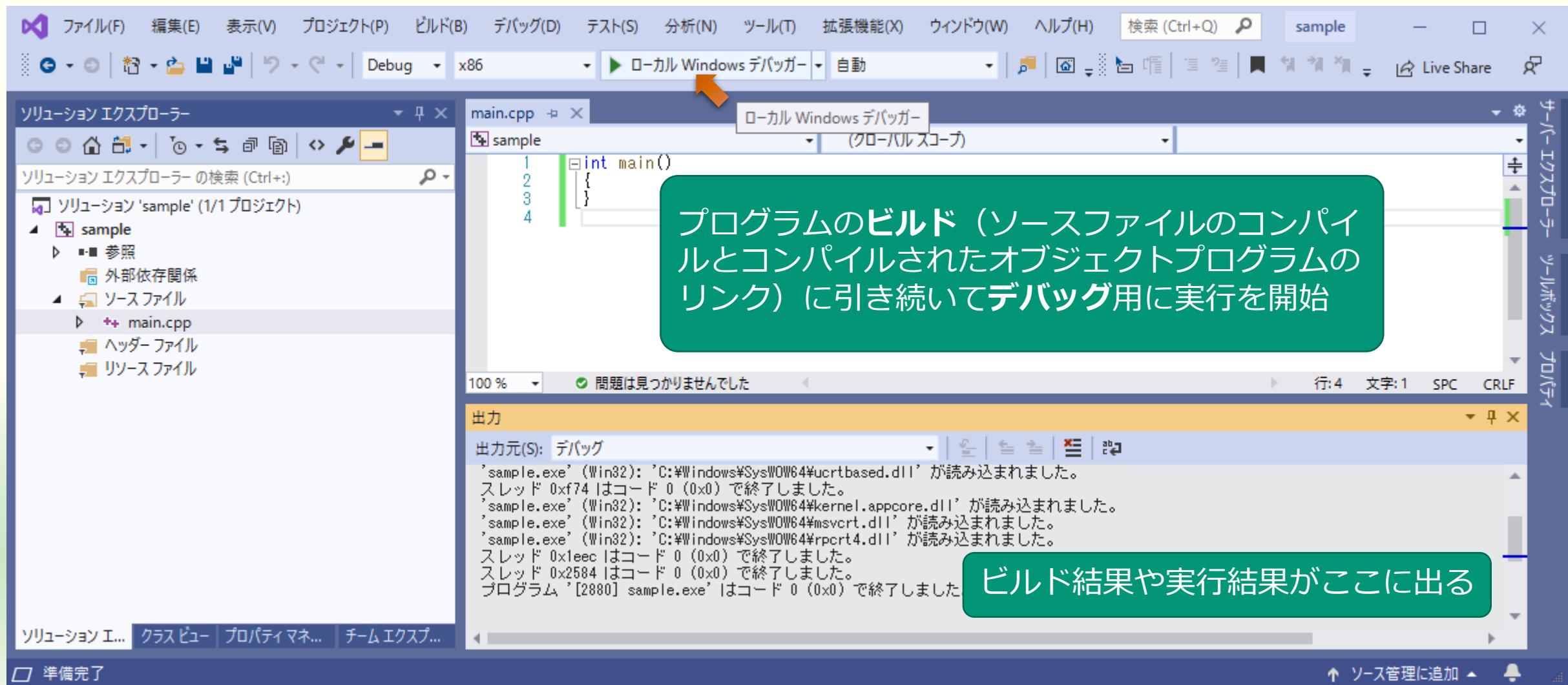
# 作成するプログラム

```
int main()  
{  
}
```

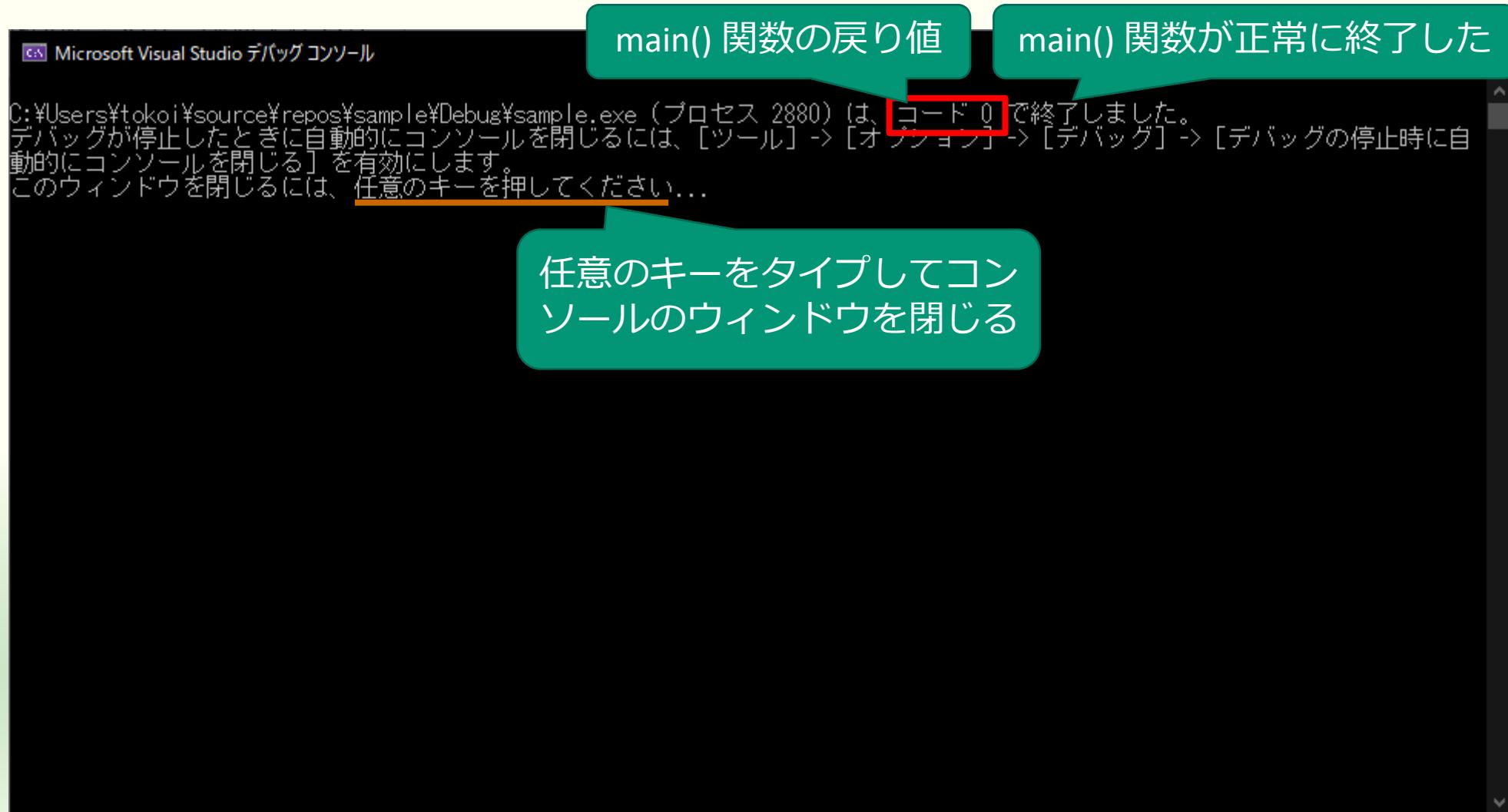
- **main()** 関数を定義する
  - このプログラムには中身がないので何もしない
  - main() 関数の戻り値のデータ型は int 型
- **main()** 関数に限り return を省略しても戻り値として 0 を返す



# プログラムのビルドと実行



# プログラムのコンソール出力



Microsoft Visual Studio デバッグコンソール

```
C:\$Users\$tokoi\$source\$repos\$sample\$Debug\$sample.exe (プロセス 2880) は、コード 0 で終了しました。  
デバッグが停止したときに自動的にコンソールを閉じるには、[ツール] → [オプション] → [デバッグ] → [デバッグの停止時に自動的にコンソールを閉じる] を有効にします。  
このウィンドウを閉じるには、任意のキーを押してください...
```

main() 関数の戻り値

main() 関数が正常に終了した

任意のキーをタイプしてコンソールのウィンドウを閉じる

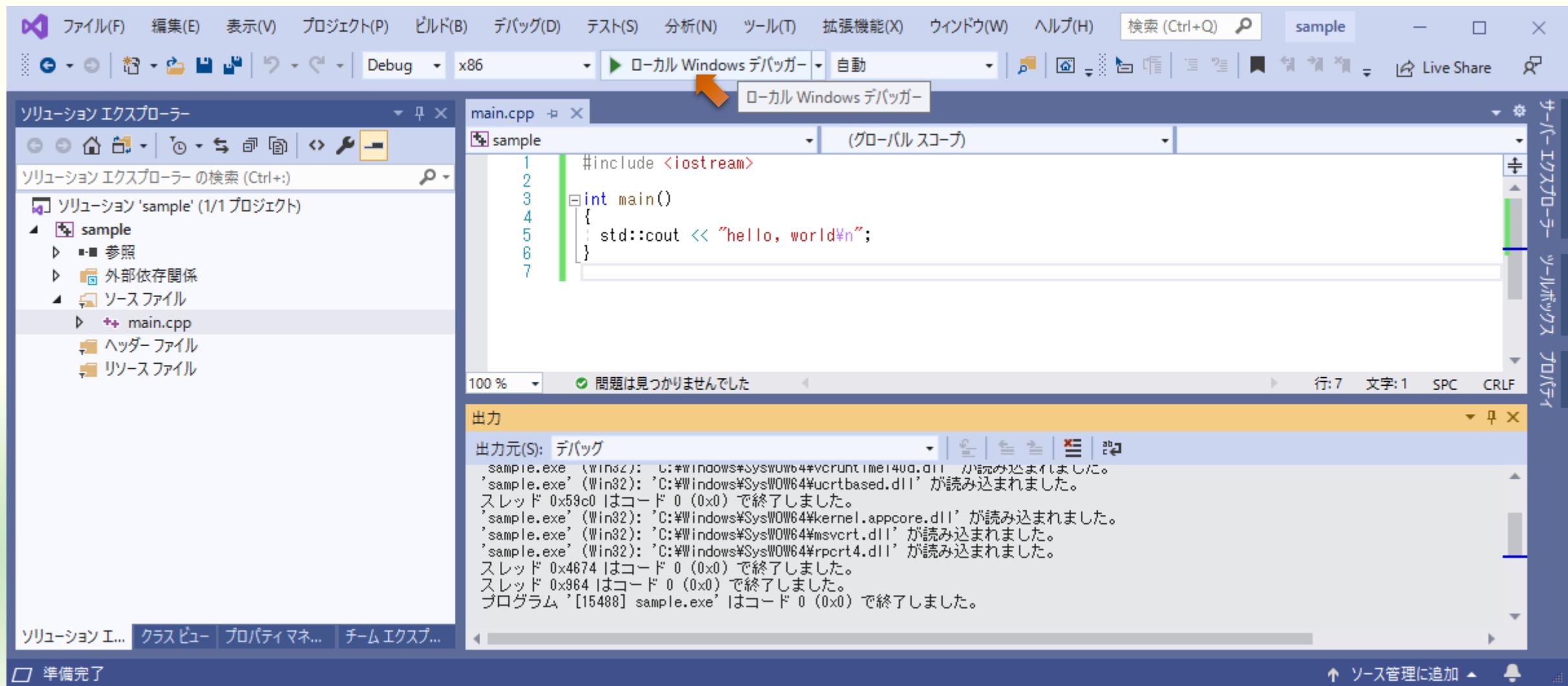
# コンソールに文字を出力する

```
#include <iostream>

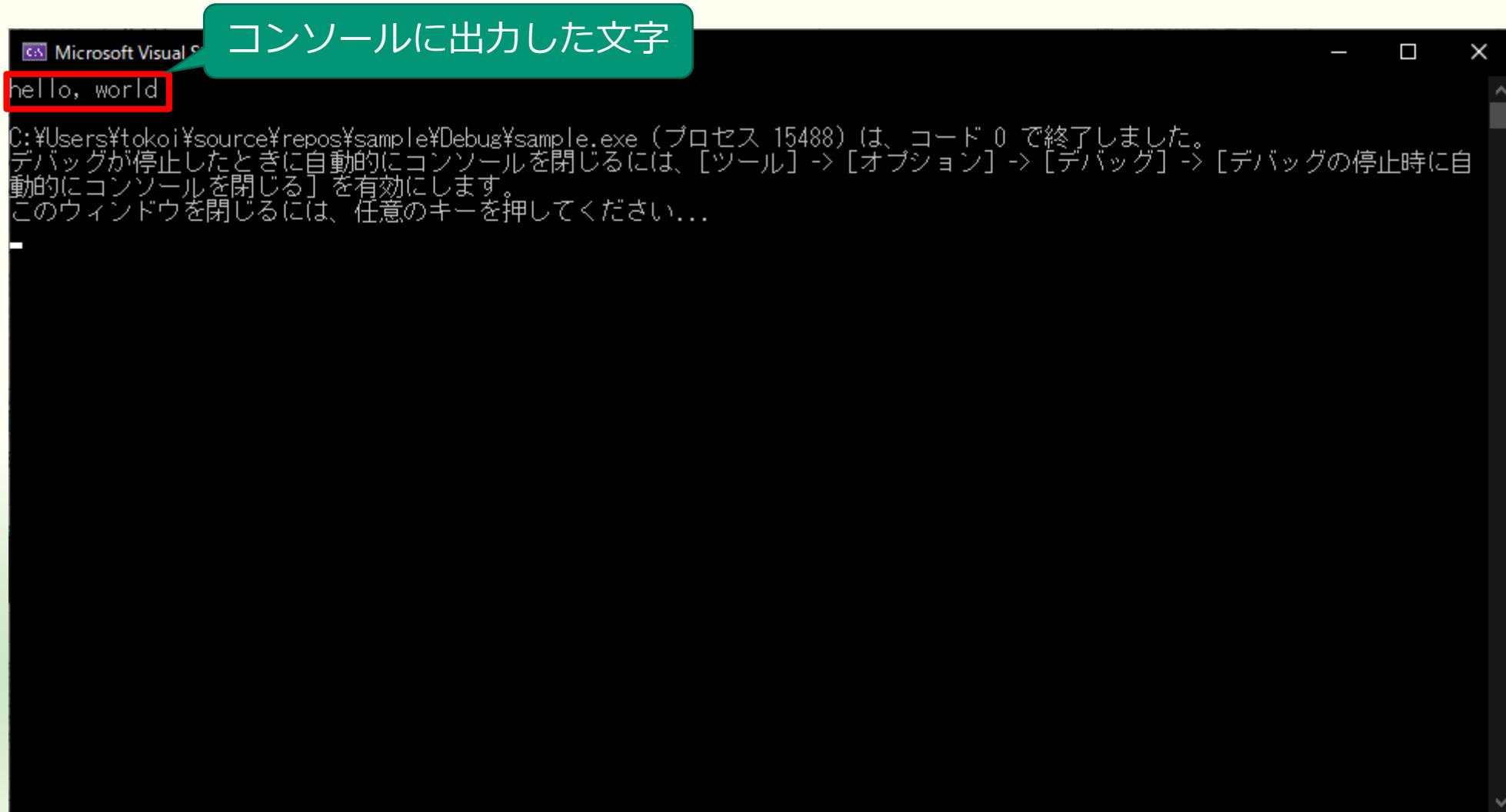
int main()
{
    std::cout << "hello, world\n";
}
```

- マーカー部分を追加する
  - #include <iostream>
    - iostream という**標準ライブラリ**の定義をソースプログラムのこの部分に埋め込む
    - 標準ライブラリは C++ 言語に最初から用意されている機能
  - std::cout << "hello, world\n";
    - コンソールに hello, world を表示
    - std::cout は標準ライブラリに含まれるコンソール出力の機能

# 修正したプログラムのビルドと実行



# 修正したプログラムのコンソール出力



Microsoft Visual Studio

コンソールに出力した文字

```
hello, world
```

C:\\$Users\\$tokoi\\$source\\$repos\\$sample\\$Debug\\$sample.exe (プロセス 15488) は、コード 0 で終了しました。  
デバッグが停止したときに自動的にコンソールを閉じるには、[ツール] -> [オプション] -> [デバッグ] -> [デバッグの停止時に自動的にコンソールを閉じる] を有効にします。  
このウィンドウを閉じるには、任意のキーを押してください...

# 関数を定義する

```
#include <iostream>

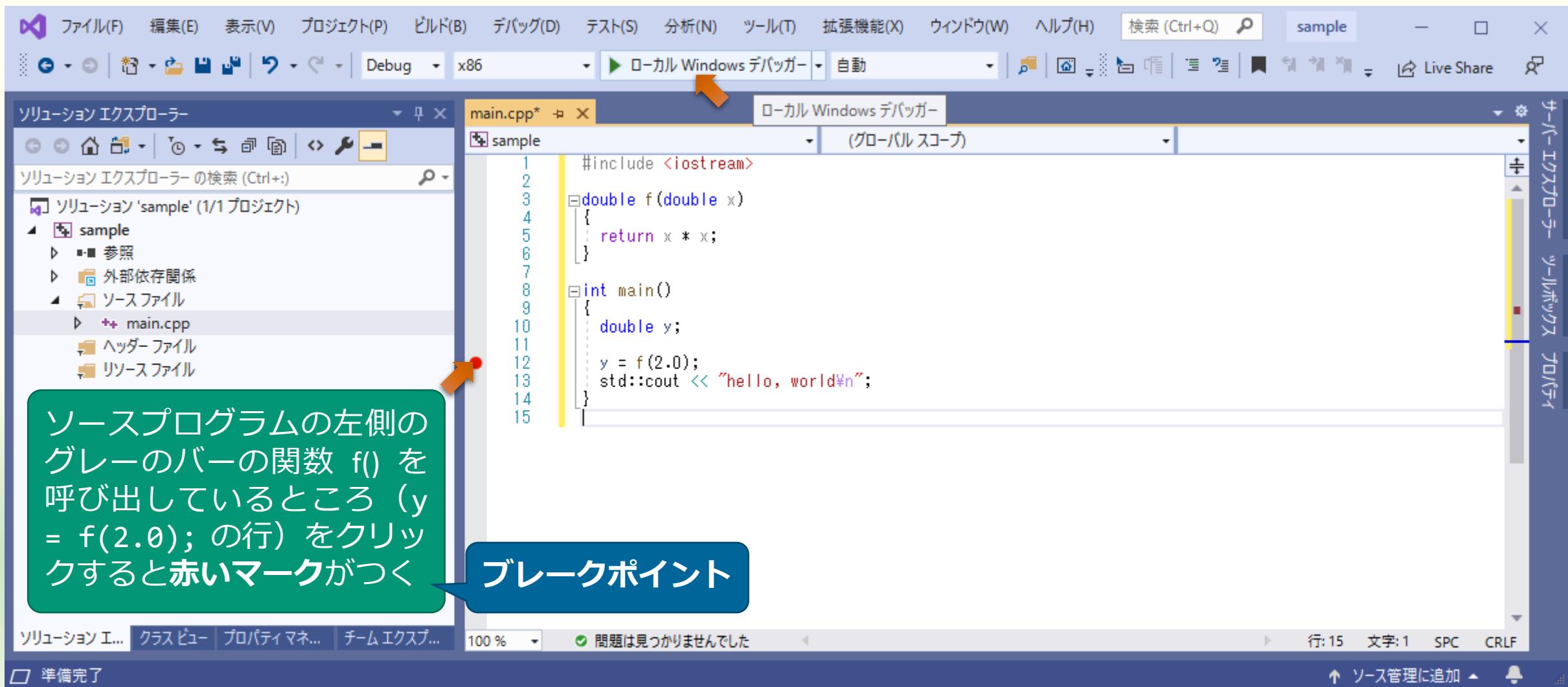
double f(double x)
{
    return x * x;
}

int main()
{
    double y;

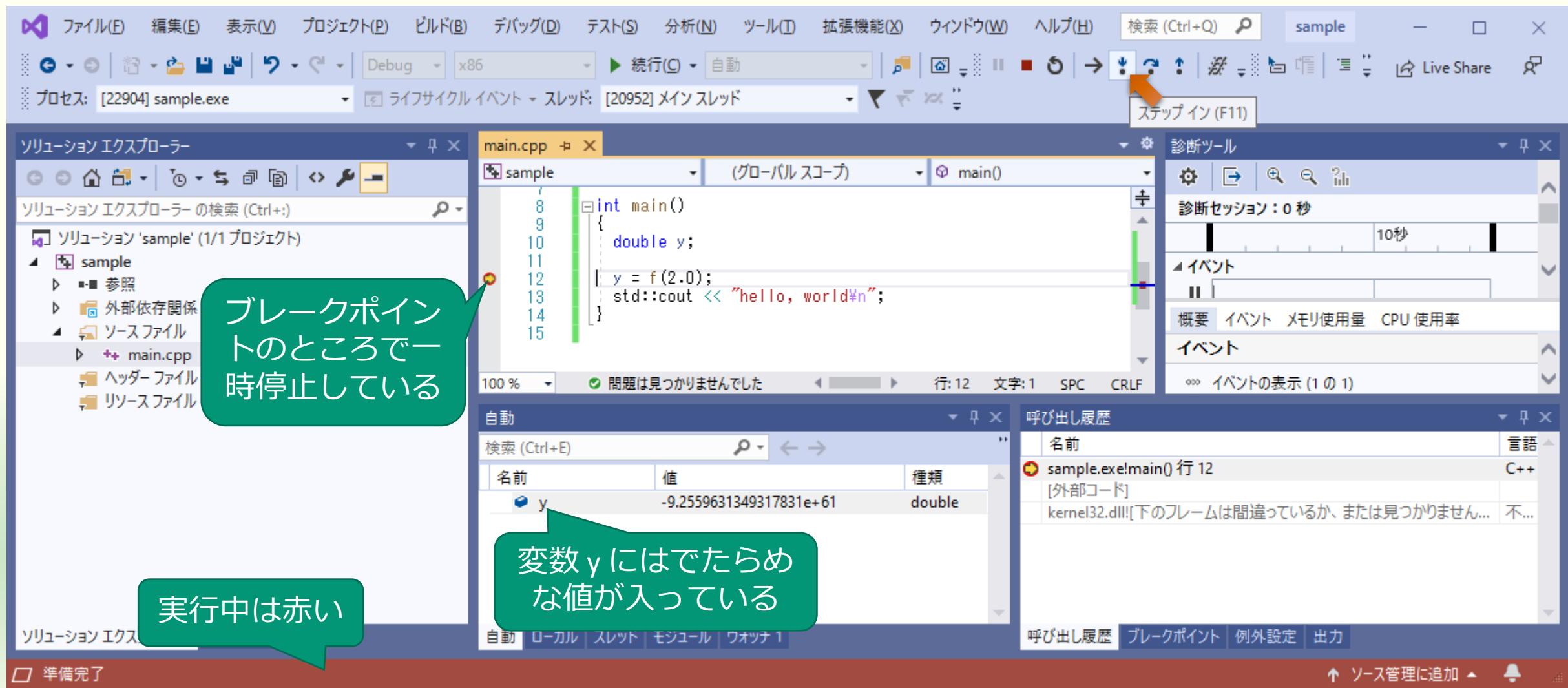
    y = f(2.0);
    std::cout << "hello, world\n";
}
```

- マーカー部分を追加する
  - `double f(double x)`
    - 戻り値のデータ型 `double` の関数 `f()` を定義する
    - 仮引数 `x` のデータ型は `double`
    - 戻り値として `x * x` を返す
  - `y = f(2.0);`
    - 実引数に `2.0` を指定して関数 `f()` を呼び出す
    - 関数 `f()` の戻り値は変数 `y` に代入する

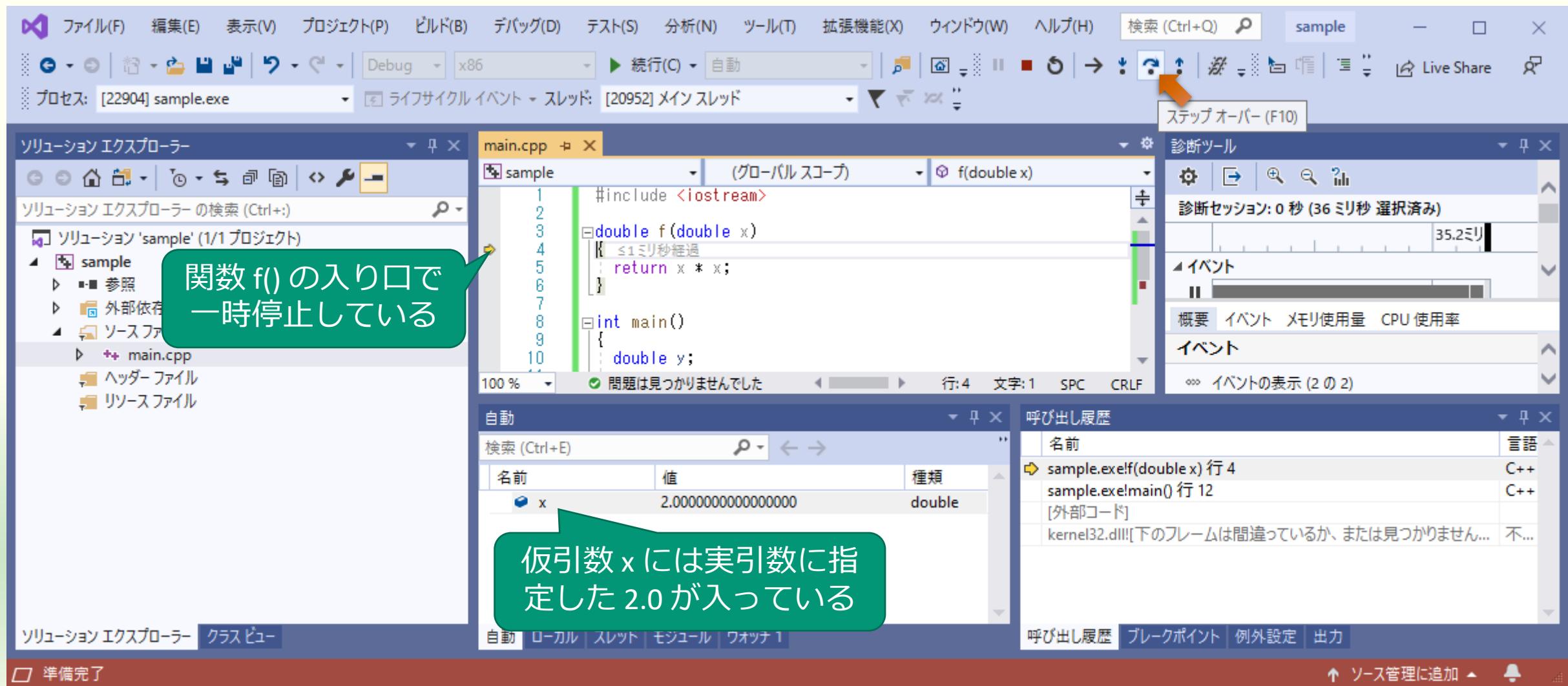
# ブレークポイントを設定してビルドと実行



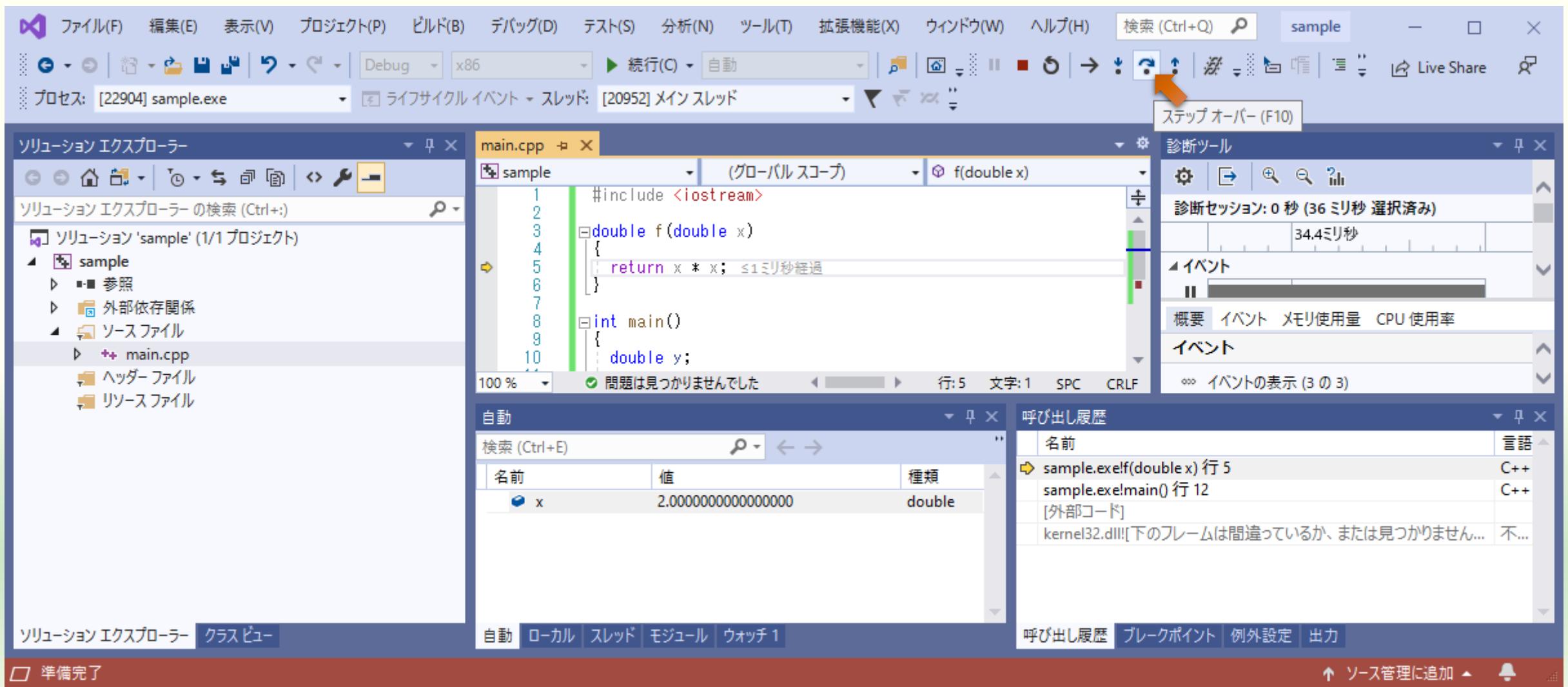
# 一時停止したらステップインする



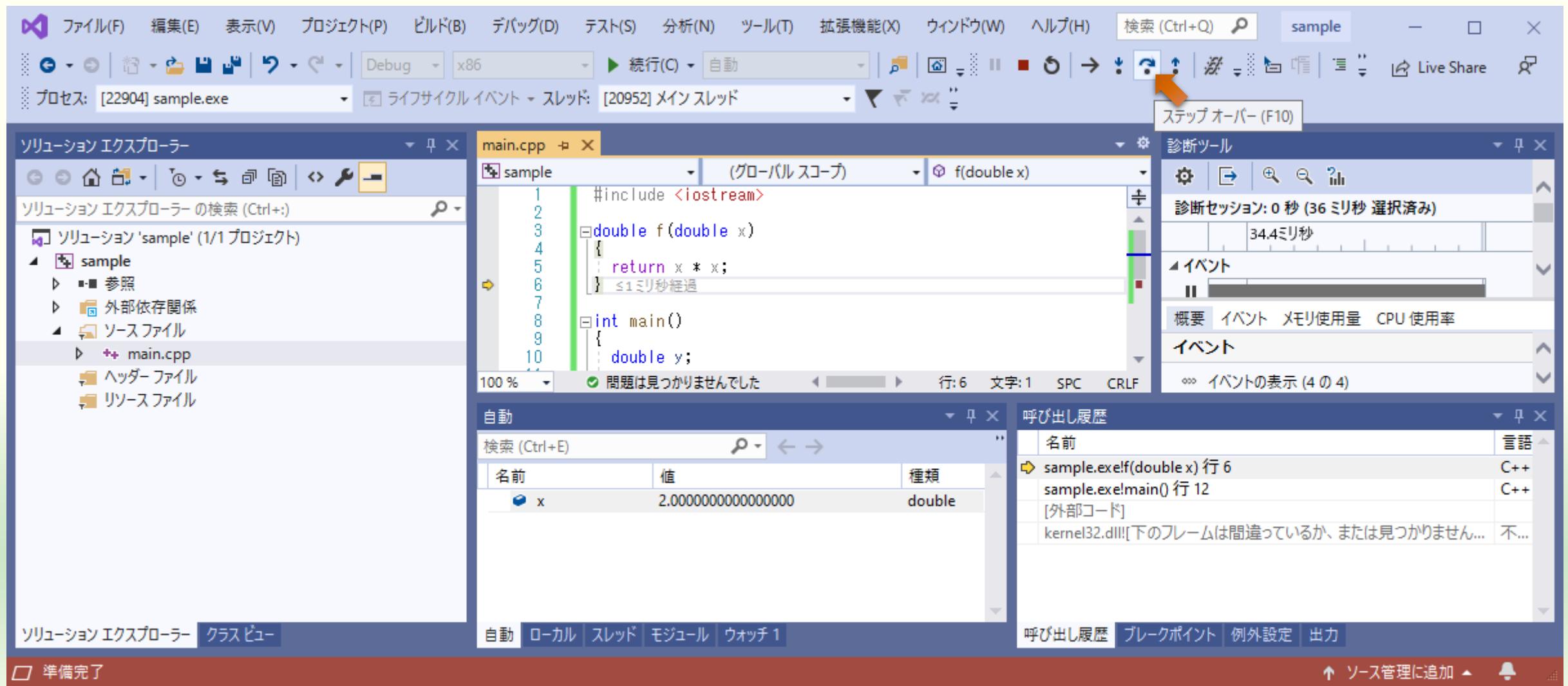
# 関数 f() に移動するのでステップオーバー



# 次の行に進んだらステップオーバー



# もう一度ステップオーバー



# main() 関数の f() の呼び出し位置に戻る

Screenshot of Microsoft Visual Studio showing the debugger interface during a step-over operation. The main window displays the code in `main.cpp`:

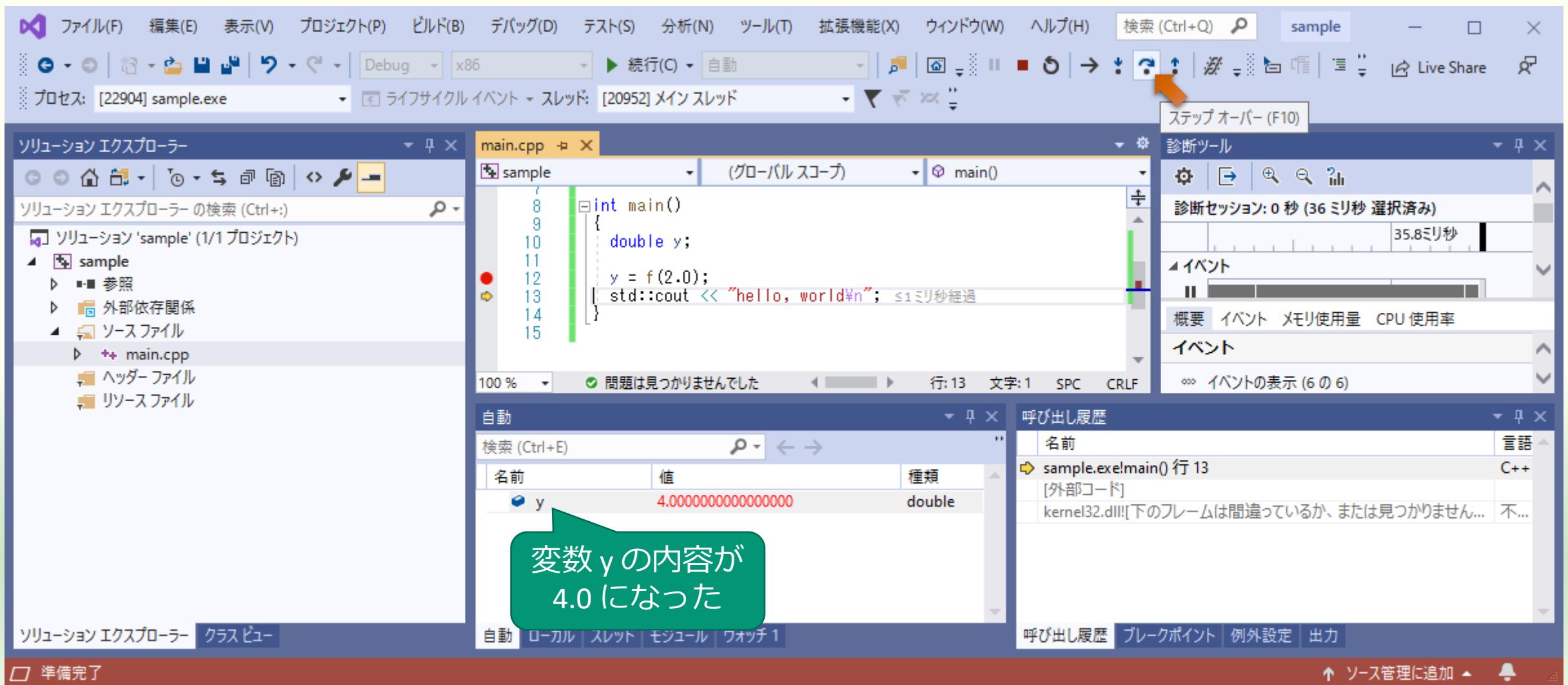
```
int main()
{
    double y;
    y = f(2.0); // 1ミリ秒経過
    std::cout << "hello, world\n";
}
```

The debugger toolbar at the top right shows the **Step Over (F10)** button highlighted with an orange arrow. The status bar at the bottom indicates the current step: **100 %** and **自動** (Automatic).

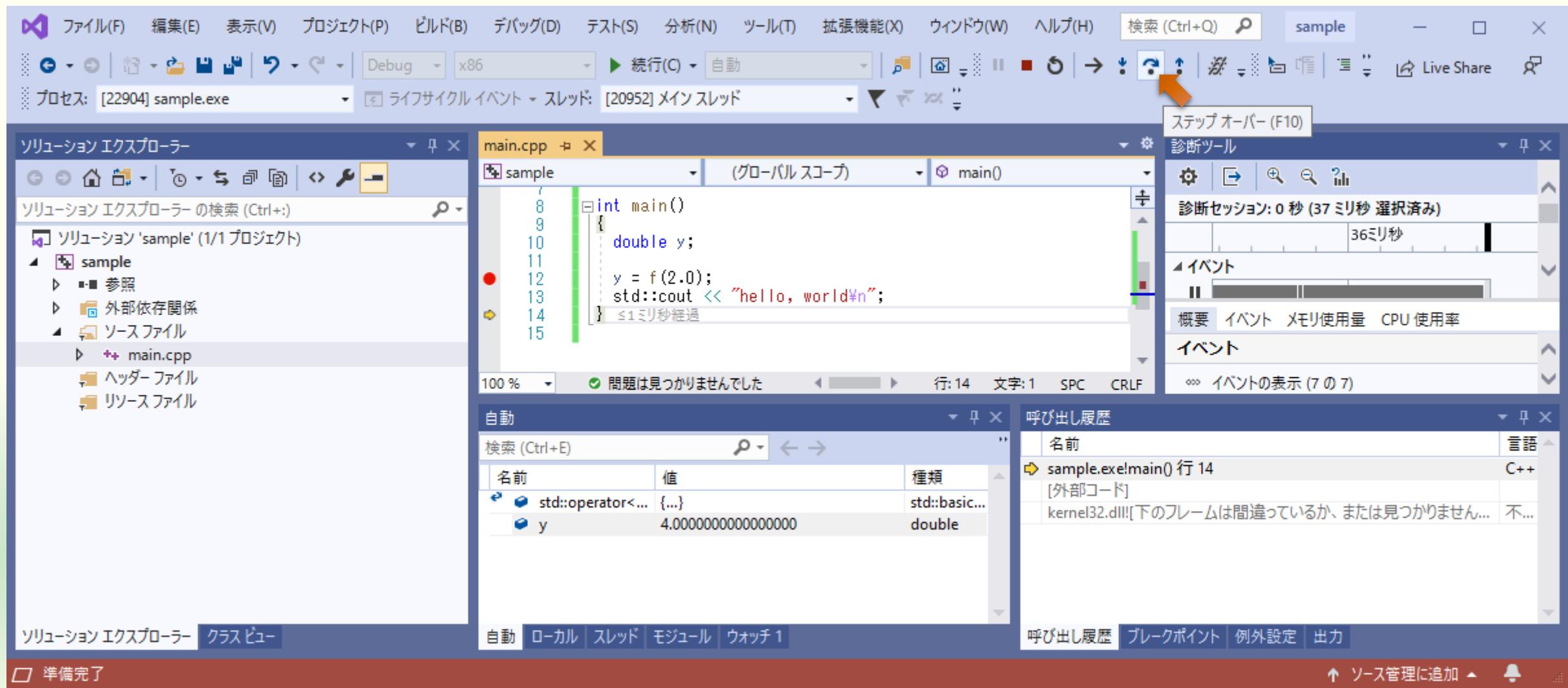
Annotations in the image:

- 関数 f() を呼び出した位置に戻っている** (The position where the function f() was called is being returned to) points to the line `y = f(2.0);`.
- 関数 f() が 4.0 を返している** (The function f() is returning 4.0) points to the **呼び出し履歴** (Call History) window, which shows the return value of `f` as `4.0000000000000000`.
- 変数 y の内容はでたらめなまま** (The content of variable y is still random) points to the **自動** (Automatic) window, which shows the value of `y` as `-9.2559631349317831e+61`.

# ステップオーバーすると y に代入される



# std::cout をステップオーバーする

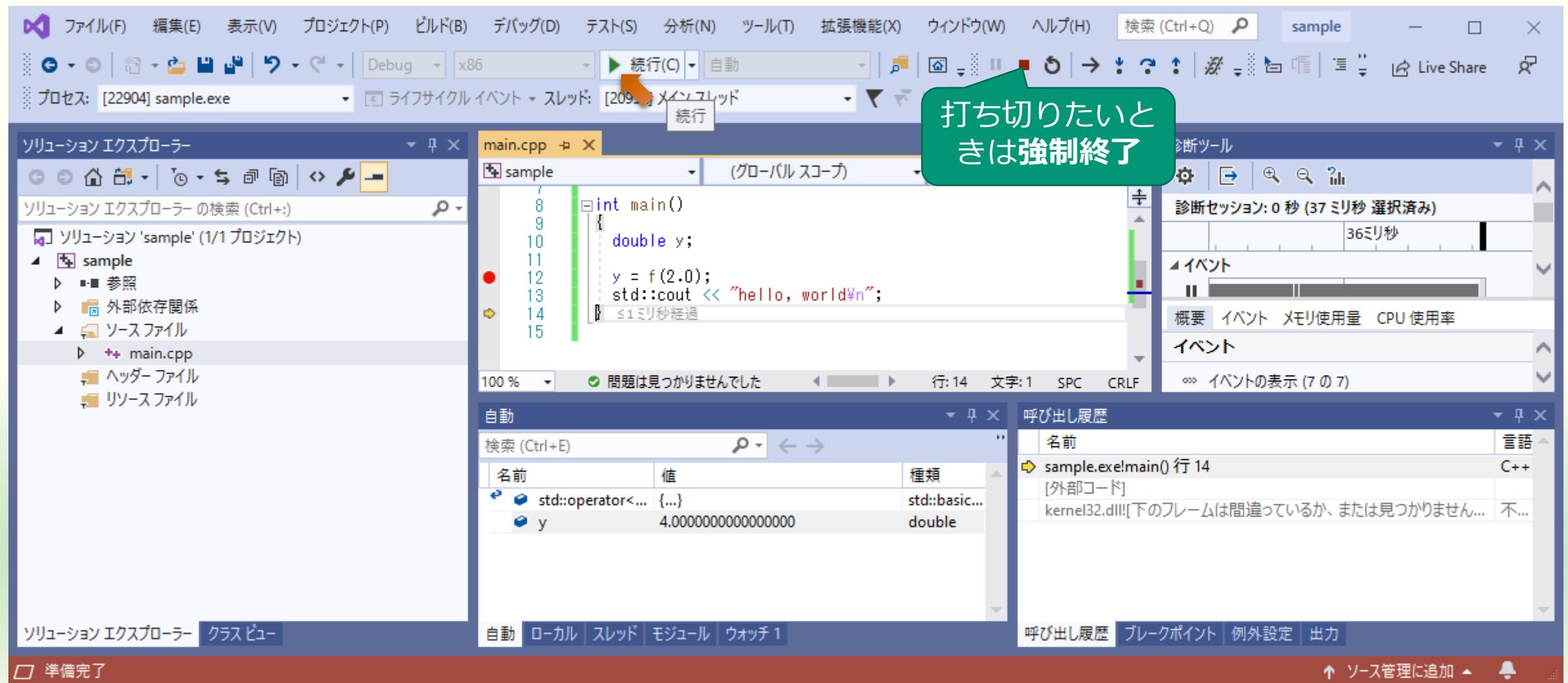


# コンソールに出力される



A screenshot of a Windows Command Prompt window. The window title is 'C:\Users\tokoi\source\repos\sample\Debug\sample.exe'. The command entered is 'sample.exe'. The output 'hello, world' is displayed in the window, with the first few characters 'hello, world' highlighted with a red rectangular box. The window has standard minimize, maximize, and close buttons in the top right corner.

# 「続行」すると残りを一気に実行する



# プログラムの実行が終了する



```
Microsoft Visual Studio デバッグ コンソール
hello, world
C:\$Users\$tokoi\$source\$repos\$sample\$Debug\$sample.exe (プロセス 15488) は、コード 0 で終了しました。
デバッグが停止したときに自動的にコンソールを閉じるには、[ツール] -> [オプション] -> [デバッグ] -> [デバッグの停止時に自動的にコンソールを閉じる] を有効にします。
このウィンドウを閉じるには、任意のキーを押してください...
-
```

# yをコンソールに出力する

```
#include <iostream>

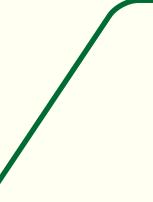
double f(double x)
{
    return x * x;
}

int main()
{
    double y;

    y = f(2.0);
    std::cout << "hello, world\n" << y;
}
```

- マーカー部分を追加する
  - 「std::cout << "..."」全体も std::cout と同じ機能を持つ
  - したがって std::cout << "..." << "..." のように続けて書ける
  - std::cout << y とすると変数 y の値を文字に直してコンソールに出力する
  - したがって std::cout << "..." << y とすれば "..." の後ろに y の値を文字直して出力される

# hello, world の次に 4 が output される



```
Microsoft Visual Studio デバッグ コンソール
hello, world
4
C:\$Users\$tokoi\$source\$repos\$sample\$Debug\$sample.exe (プロセス 19980) は、コード 0 で終了しました。
デバッグが停止したときに自動的にコンソールを閉じるには、[ツール] -> [オプション] -> [デバッグ] -> [デバッグの停止時に自動的にコンソールを閉じる] を有効にします。
このウィンドウを閉じるには、任意のキーを押してください...
```

# 改行文字を移動する

```
#include <iostream>

double f(double x)
{
    return x * x;
}

int main()
{
    double y;

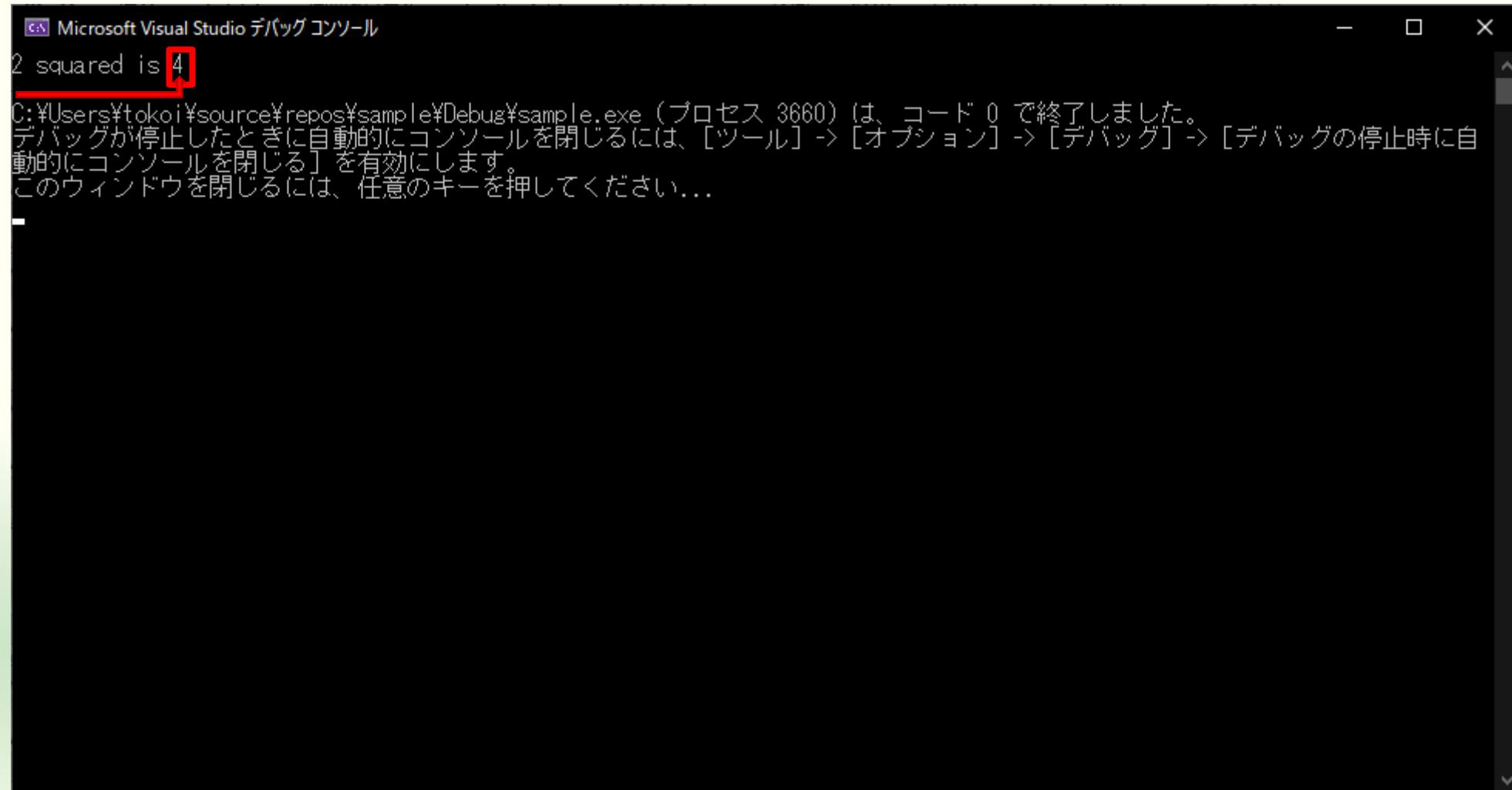
    y = f(2.0);
    std::cout << "2 squared is " << y << "\n";
}
```

改行文字を移動

- マーカー部分を変更する
  - `\n` は改行文字
    - `\` はフォントや文字コードの環境によっては`\`と表示される
    - 英語圏で用いられていた文字集合 (ASCIIコード) の字形は`\`だったが日本の文字集合の規格 JIS X 0201 では字形を`¥`にした



# 改行位置が移動する



Microsoft Visual Studio デバッグ コンソール

2 squared is 4

C:\Users\tokoi\source\repos\sample\Debug\sample.exe (プロセス 3660) は、コード 0 で終了しました。  
デバッグが停止したときに自動的にコンソールを閉じるには、[ツール] -> [オプション] -> [デバッグ] -> [デバッグの停止時に自動的にコンソールを閉じる] を有効にします。  
このウィンドウを閉じるには、任意のキーを押してください...



# ソフトウェアを作るのは色々大変

ちゃんとしたソフトウェアを作るなら道具から揃えよう

# メディアを扱うプログラミング

- プログラミング言語 자체は**メディアを扱う機能がない**
- オペレーティングシステムの機能を呼び出す
  - 音声, 音楽, 画像, 映像, グラフィックス
  - いろんなことができるが複雑で手間がかかることが多い
- ライブラリを組み合わせる
  - 音声入出力, 画像入出力, 映像入出力, 3Dモデル入力
  - 音声処理, 画像処理, 映像処理, グラフィックス処理
  - 物理シミュレーション, フォント処理, ...
  - 一通り揃えるのは大変



# ミドルウェアを利用する

- 特定用途向けのソフトウェア開発環境のパッケージ
  - シーディングラフ／レンダリングエンジン
    - [OpenSceneGraph](#), [OGRE](#), [Delta3D](#), ...
  - ツールキット
    - [openFrameworks](#), [Cinder](#), [P5.js](#), [Three.js](#), [freeglut](#), [GLFW](#), [FLTK](#), [Qt](#), ...
  - プログラミング環境
    - [Max/MSP](#), [PureData](#), [SuperCollider](#), [vvvv](#), [TouchDesigner](#), [Processing](#), ...
  - ゲームエンジン
    - [Unity](#), [Unreal Engine](#), [CRYENGINE](#), [Lumberyard](#), [OROCHI4](#), ...
    - [Irrlicht Engine](#), [Armory Engine](#), [Godot](#), [Xenko](#), ...



# クリエイティブコーディング

## ■ 映像・音響による**表現のための**プログラミング

### ■ メディアアート, インタラクティブアート

- “メディアアートの教科書” (多摩美術大学)

- 関連企業

- teamLab (チームラボ)

- Rhizomatiks (ライゾマティクス)

- BACKSPACE Productions Inc. (バックスペースプロダクションズインク)

- Takram (タクラム)

- MontBlanc Pictures (モンブラン・ピクチャーズ)

- THE EUGENE Studio (ザ・ユージーン・スタジオ)

- 株式会社白



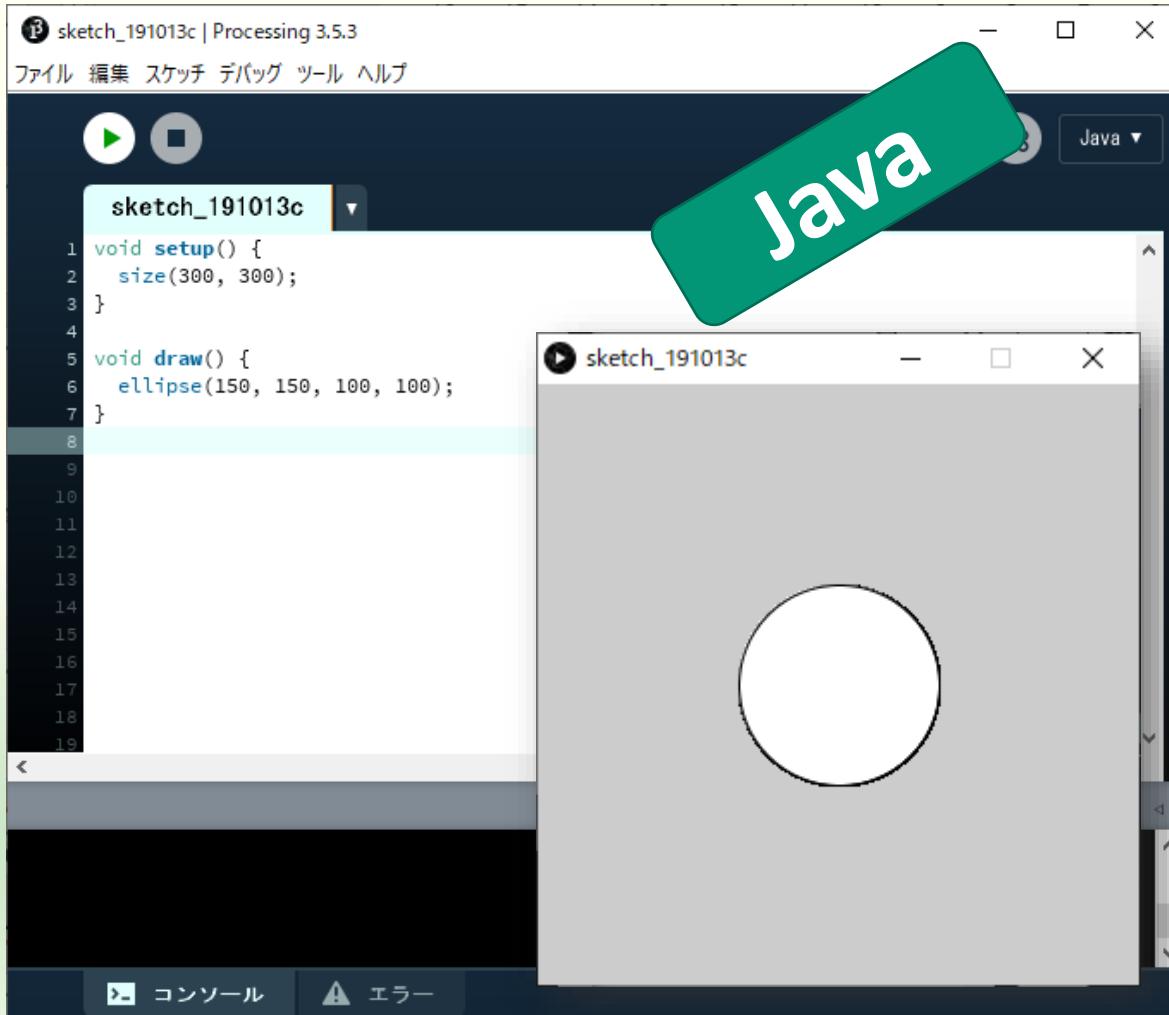
# Processing

---

- プログラミングによる創作を行うためのツール
  - 誰でも簡単に視覚表現を行うプログラムを作れる
  - MIT Media Lab. の John Maeda の下で “[Design by Numbers](#)” の開発に携わった二人の大学院生 Casey Reas と Ben Fry によりオープンソースプロジェクトとして開発
- Java で実装されている
  - プログラムは Java でも Python でも書ける
  - プログラミングで絵を描く



# Processing の sketchbook (開発環境)

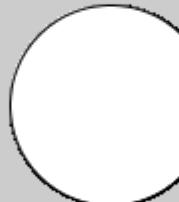


sketch\_191013c | Processing 3.5.3

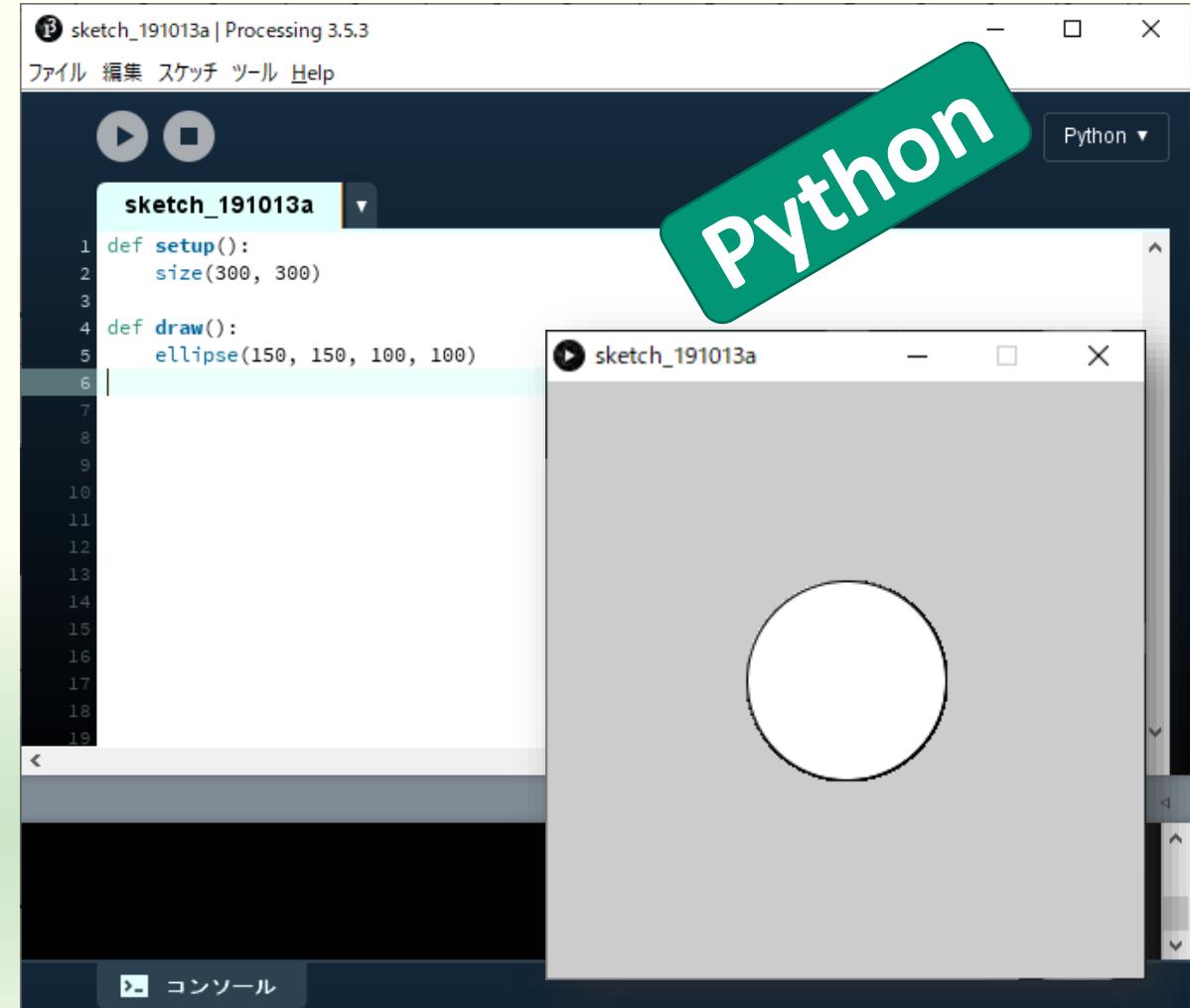
Java

```
sketch_191013c
1 void setup() {
2   size(300, 300);
3 }
4
5 void draw() {
6   ellipse(150, 150, 100, 100);
7 }
```

sketch\_191013c



コンソール エラー

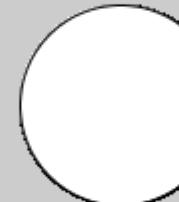


sketch\_191013a | Processing 3.5.3

Python

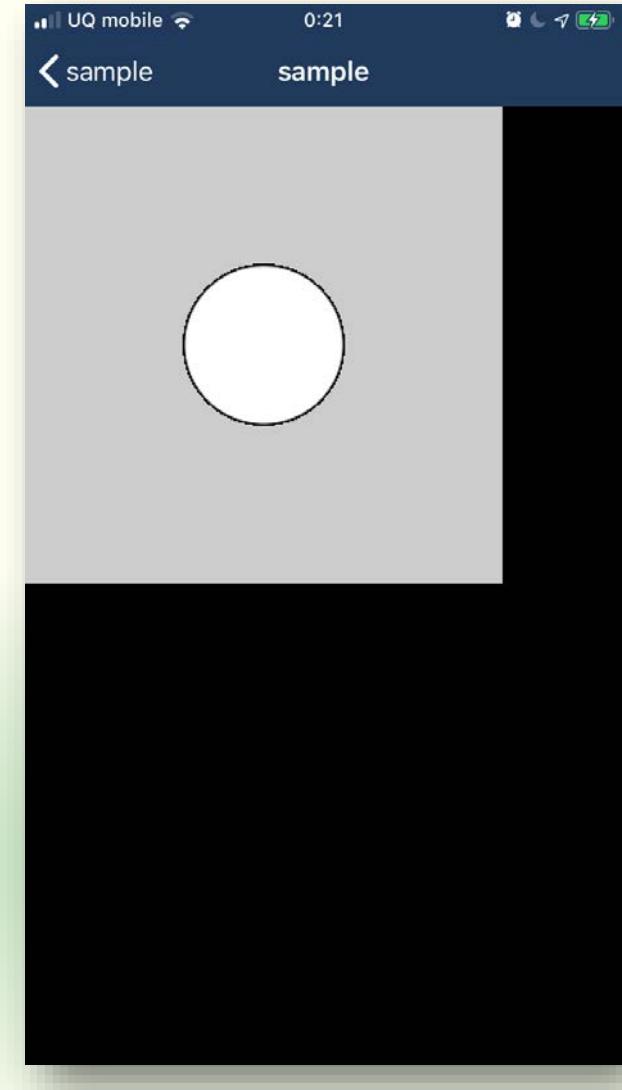
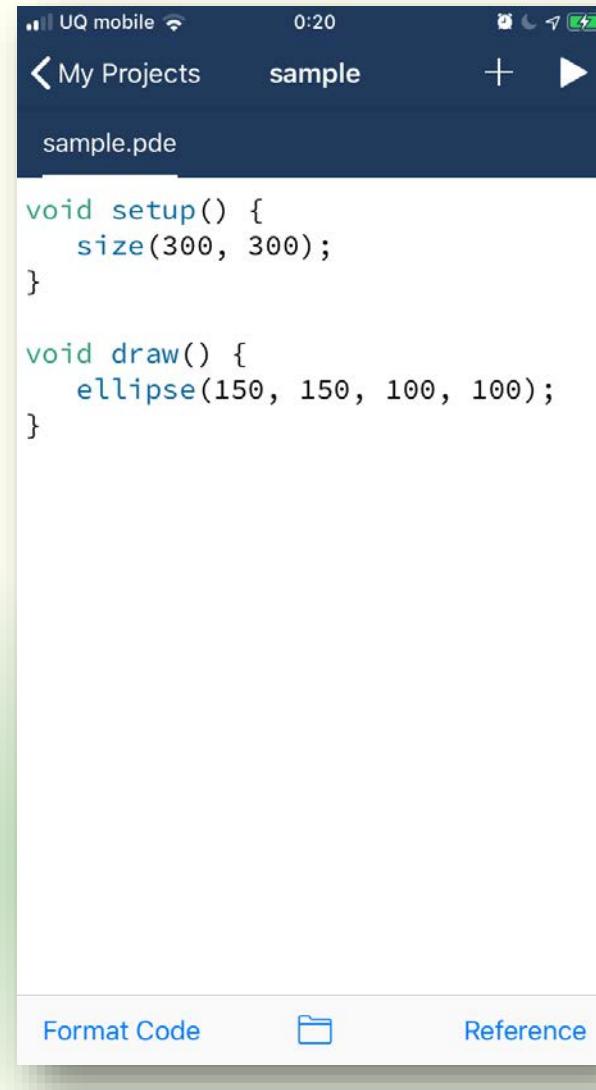
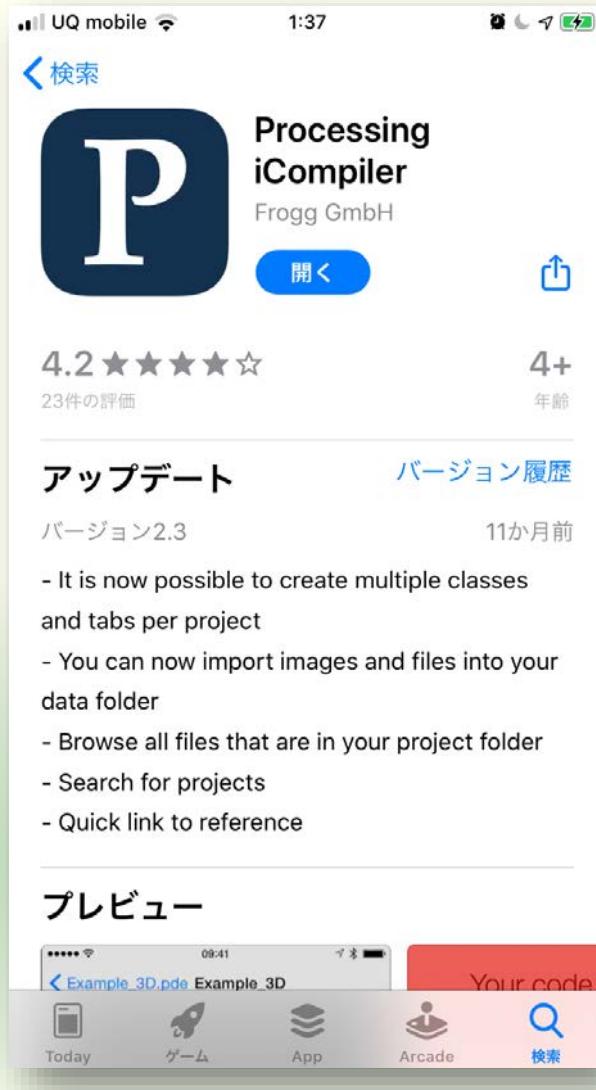
```
sketch_191013a
1 def setup():
2   size(300, 300)
3
4 def draw():
5   ellipse(150, 150, 100, 100)
6 |
```

sketch\_191013a



コンソール

# iPhone でも動く



しかし、この演習では  
openFrameworks を使います  
メディアプログラミング演習



# openFrameworks

---

- クリエイティブコーディングのための C++ 言語によるオープンソースのフレームワーク
- 様々なライブラリを統合
  - グラフィクス：[OpenGL](#), [GLEW](#), [GLUT](#), [libtess2](#), [cairo](#)
  - オーディオ：[rtAudio](#), [PortAudio](#), [OpenAL](#), [Kiss FFT](#) または [FMOD](#)
  - フォント：[FreeType](#)
  - イメージの読み込みと保存：[FreelImage](#)
  - 動画の再生と取込：[Quicktime](#), [GStreamer](#), [videoInput](#)
  - 様々なユーティリティー：[Poco](#)
  - コンピュータビジョン：[OpenCV](#)
  - 3Dモデルの読み込み：[Assimp](#)



# 実は何を使うかすごく悩んだ

---

- Processing

- Java ベースで簡単・処理系も軽い・Python でも書ける

- Python cgkit

- Python で 3D CG を扱うパッケージやプラグインを集めた

- Three.js

- JavaScript で WebGL を使うグラフィックスライブラリ

- Unity

- Unreal Engine と並んでゲーム開発ミドルウェアの標準

# 目標

---

プログラミングを  
知ること

アプリケーション  
プログラムを  
作ること

# openFrameworks の特徴

- openFrameworks は C++ 用のツールキット
  - 既存のライブラリや SDK がそのまま使える
    - openFrameworks 自体はそれらをくっつける糊 (glue) の役割
  - 開発環境には一般のもの (Visual Studio, Xcode など) を使う
    - Processing は Processing 自体が開発環境 (なので手軽ではある)
- 同様なものとして、他に Cinder がある
  - より新しい機能を使って作られている
    - 他に C# 用の OpenTK というツールキットがある



# なぜ openFrameworks か

- 一般的な開発環境に追加して使える
  - Visual Studio, Xcode, eclipse など
- 他のライブラリや SDK と組み合わせやすい
  - デバイスの SDK は C や C++ で用意されていることが多い
  - 研究用としても使える？
- C++ ベースである
  - プログラミングの**初心者**には向いていない
  - まあ何とかなるでしょう？





# 課題 1 – 1

openFrameworks のパッケージのダウンロードと展開

# <https://openframeworks.cc/ja/> を開く



about download documentation learning gallery community development

> forum > github > addons > slack > blog > donations

English 한국어 简体中文



openFrameworksは創造的なコーディングのためのC++のオープンソースツールキットです

ダウンロード

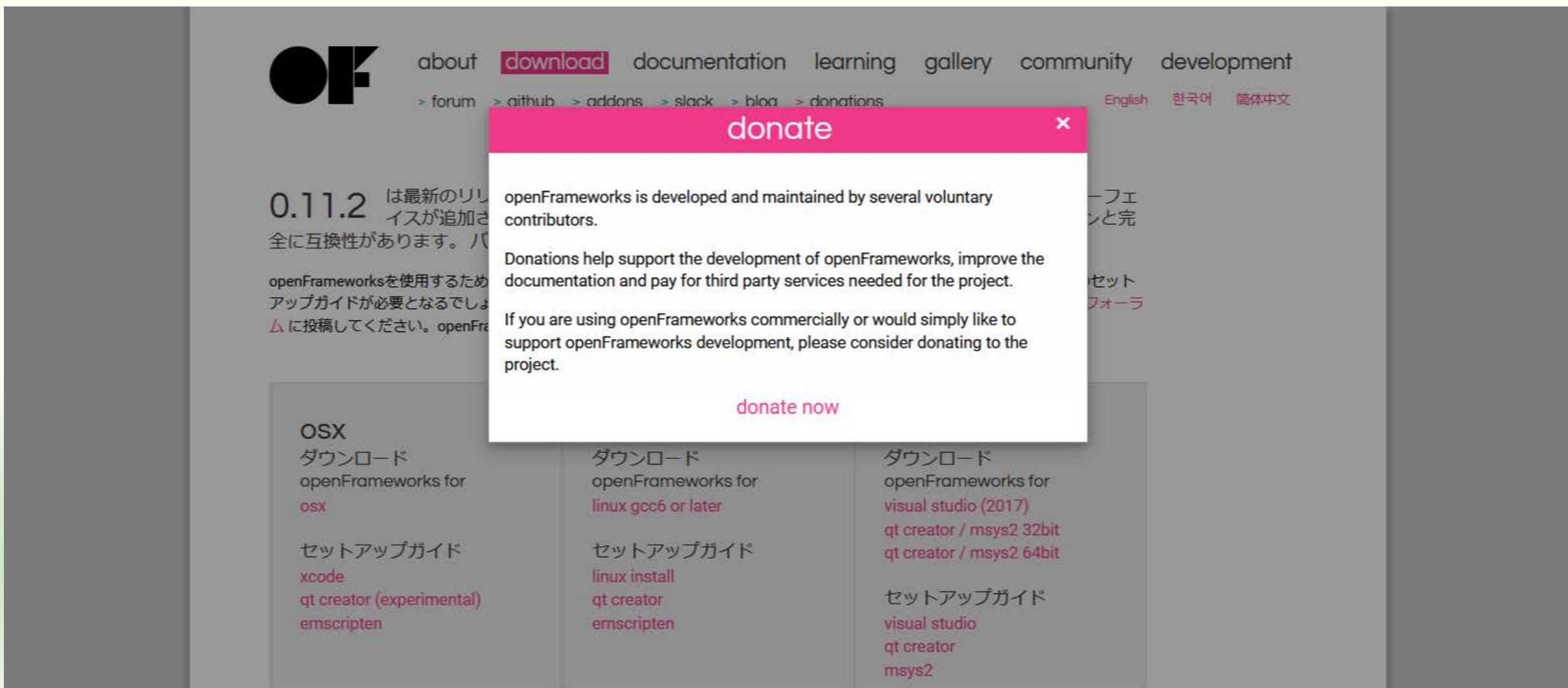
最新のリリース(0.11.2)の入手と、openFrameworksを作動させるためのセットアップガイド。

ドキュメント

openFramewroksのクラス、関数、アドオンのリファレンス資料。ガイドやチュートリアルは、tutorialを参照してください。

フォーラム

# Download (寄付募集がポップアップする)



The screenshot shows the openFrameworks website with a donation modal overlay. The modal is titled "donate" and contains text explaining the project's voluntary contributors and the purpose of donations. It also encourages commercial users to consider donating. A "donate now" button is at the bottom of the modal. The background shows the website's navigation bar, a forum post about version 0.11.2, and download links for OSX, Linux, and Visual Studio.

openFrameworks is developed and maintained by several voluntary contributors.

Donations help support the development of openFrameworks, improve the documentation and pay for third party services needed for the project.

If you are using openFrameworks commercially or would simply like to support openFrameworks development, please consider donating to the project.

donate now

OSX

ダウンロード  
openFrameworks for  
osx

セットアップガイド  
xcode  
qt creator (experimental)  
emscripten

ダウンロード  
openFrameworks for  
linux gcc6 or later

セットアップガイド  
linux install  
qt creator  
emscripten

ダウンロード  
openFrameworks for  
visual studio (2017)  
qt creator / msys2 32bit  
qt creator / msys2 64bit

セットアップガイド  
visual studio  
qt creator  
msys2

# 様々なプラットフォームのアプリが作れる

## osx / linux / windows に対応している



0.11.2 は最新のリリースです。これはマイナーバージョンです。様々な新機能や新しいインターフェイスが追加されています。ですので、このバージョンは0.11.0やさらに新しいバージョンと完全に互換性があります。バージョン間の違いの一覧は、[changelog](#)を参照してください。

openFrameworksを使用するためにはIDE(統合開発環境)が必要です。また、実際に試していくにはプラットフォームごとのセットアップガイドが必要となるでしょう。もしバグをみつけたら [問題点](#)のページに投稿してください。その他質問があれば、[フォーラム](#)に投稿してください。openFrameworksは、[MITライセンス](#)で配布されています。

osx ダウンロード openFrameworks for osx  セットアップガイド xcode qt creator (experimental) emscripten	linux ダウンロード openFrameworks for linux gcc6 or later  セットアップガイド linux install qt creator emscripten	windows ダウンロード openFrameworks for visual studio (2017) qt creator / msys2 32bit qt creator / msys2 64bit  セットアップガイド visual studio qt creator msys2
--	--	---

## スマートフォンや Raspberry Pi にも対応している

mobile  モバイル版のopenFrameworks は、デスクトップ版と同等の機能 に加えて、加速度系、コンパス、 GPSなど、モバイル端末固有の機 能をサポートしています。	ios  osx only  ダウンロード openFrameworks for xcode  セットアップガイド xcode	android  ダウンロード openFrameworks for android  セットアップガイド android studio
--	--	---

linux arm  Raspberry Pi, Beaglebone (black), Pandaboard, BeagleBoardといった、 Linuxの作動するARMベース のボードのための openFrameworksです。 セットアップガイドは主要なボ ードのみしか用意されていません が、ARM6かARM7のボードであ れば作動するはずです。	linux armv6  ダウンロード openFrameworks for linux armv6  セットアップガイド raspberry pi	linux armv7  ダウンロード openFrameworks for linux armv7  セットアップガイド pandaboard generic armv7
--	---	--

# 使用する PC に合ったものをダウンロード

## Windows 版のダウンロード

windows

ダウンロード

openFrameworks for  
visual studio (2017)

qt creator / msys2 32bit

qt creator / msys2 64bit

セットアップガイド

Visual Studio  
2019 にも対応

## macos 版のダウンロード

osx

ダウンロード

openFrameworks for  
osx

セットアップガイド

xcode

qt creator (experimental)

# ダウンロードした ZIP ファイルを展開

- 展開先のフォルダの条件
  - フォルダのパスに**空白文字が含まれていないこと**
    - ユーザ名に空白文字を含んでいると「ドキュメント」や「デスクトップ」のパスに空白文字が含まれることがある
  - フォルダのパスに**全角文字が含まれていないこと**
    - ユーザ名が日本語だと「ドキュメント」や「デスクトップ」のパスに全角文字が含まれることがある
- **個人所有の PC なら C: ドライブの直下 (C:¥) が確実**
  - 条件を満たすことができるなら他の場所でも可

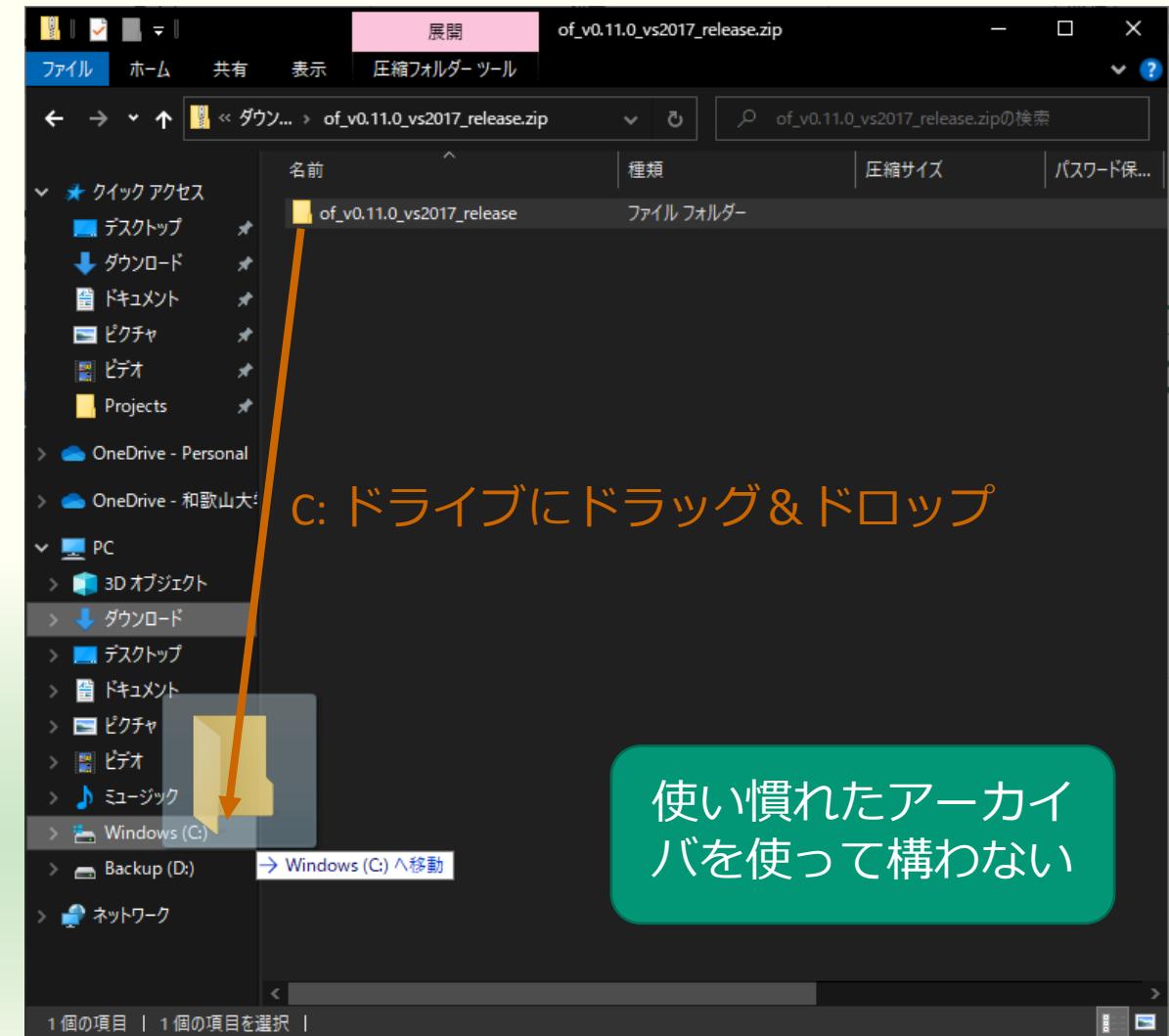
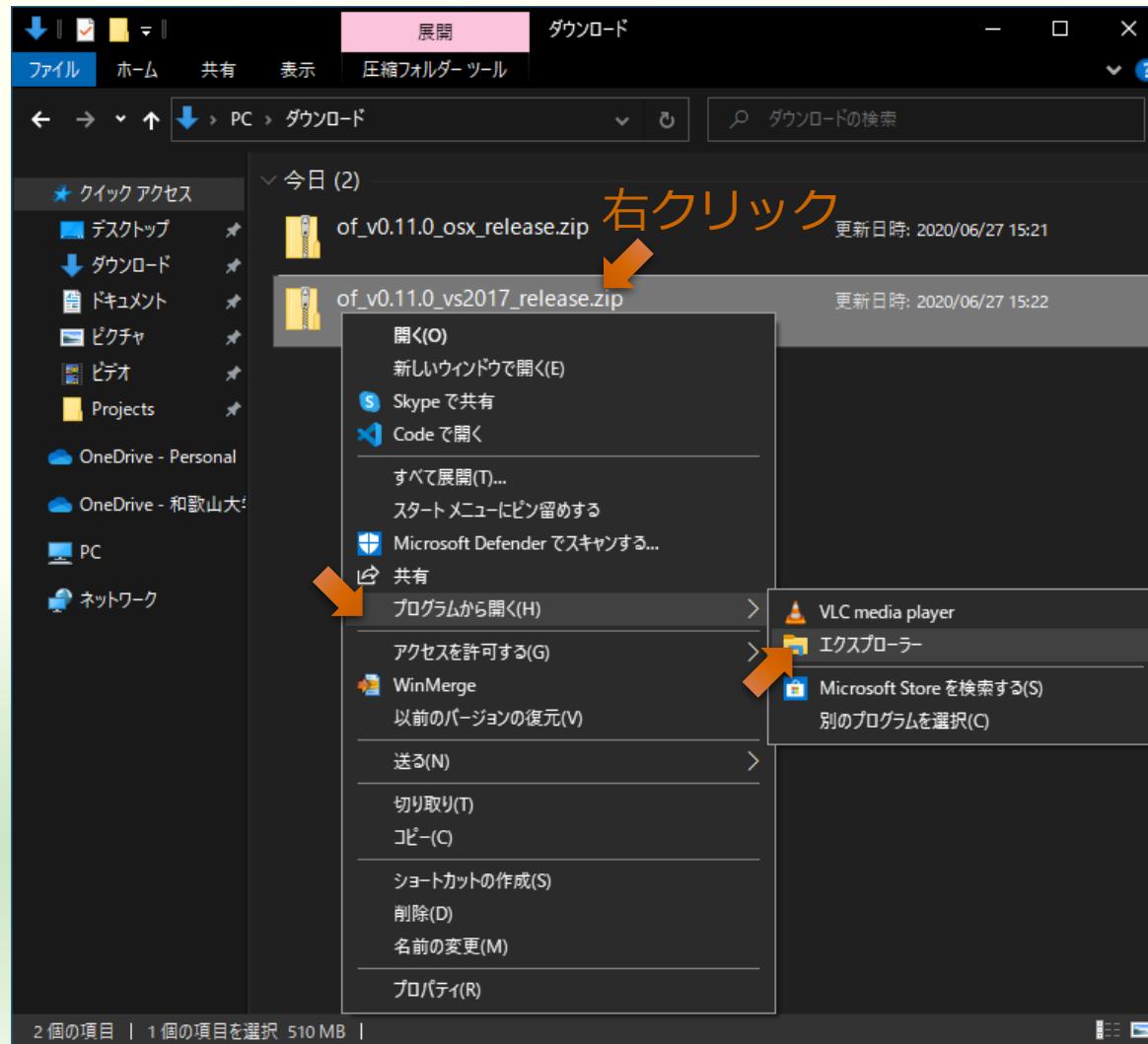


# 展開先として不適当なフォルダの例

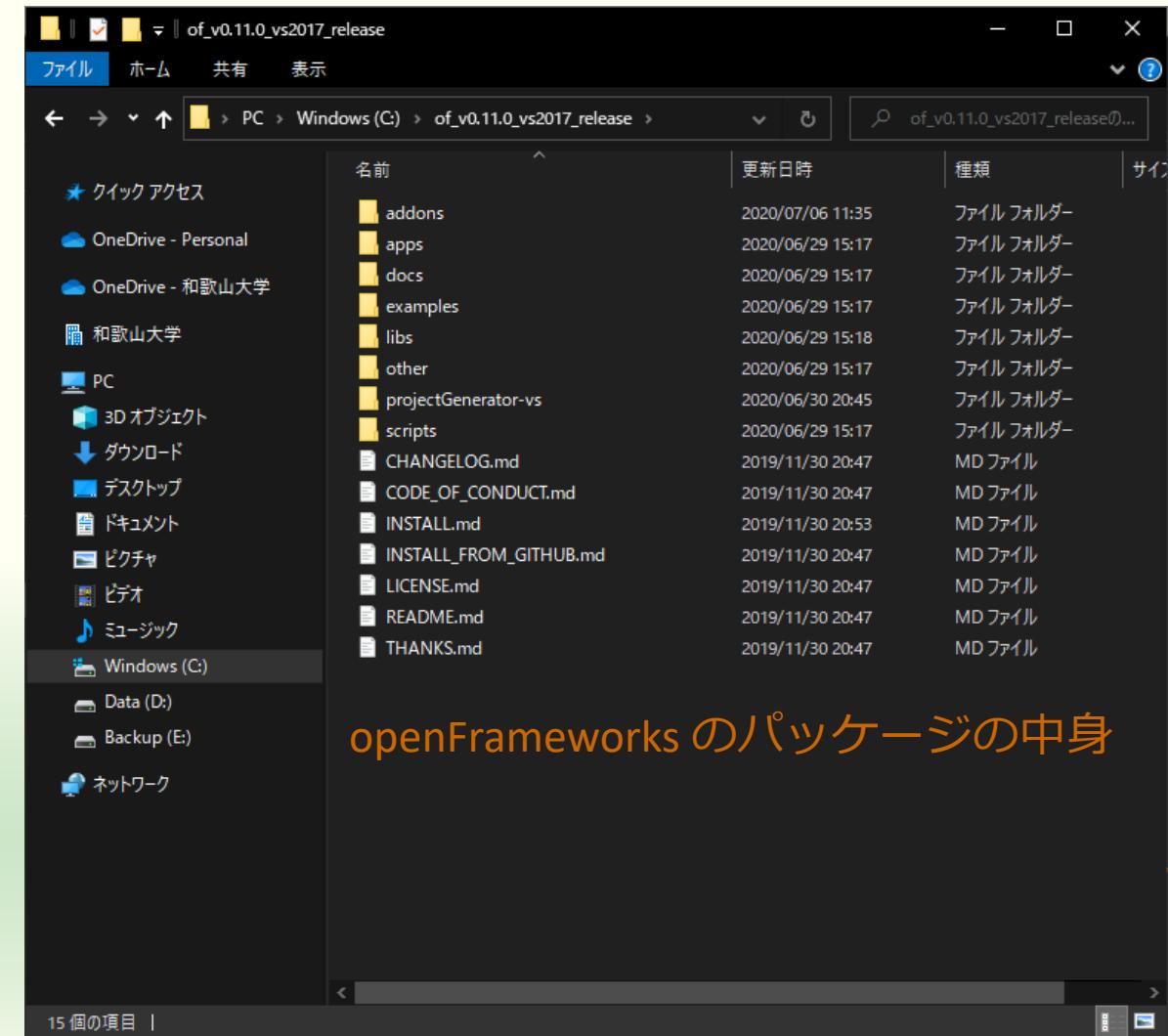
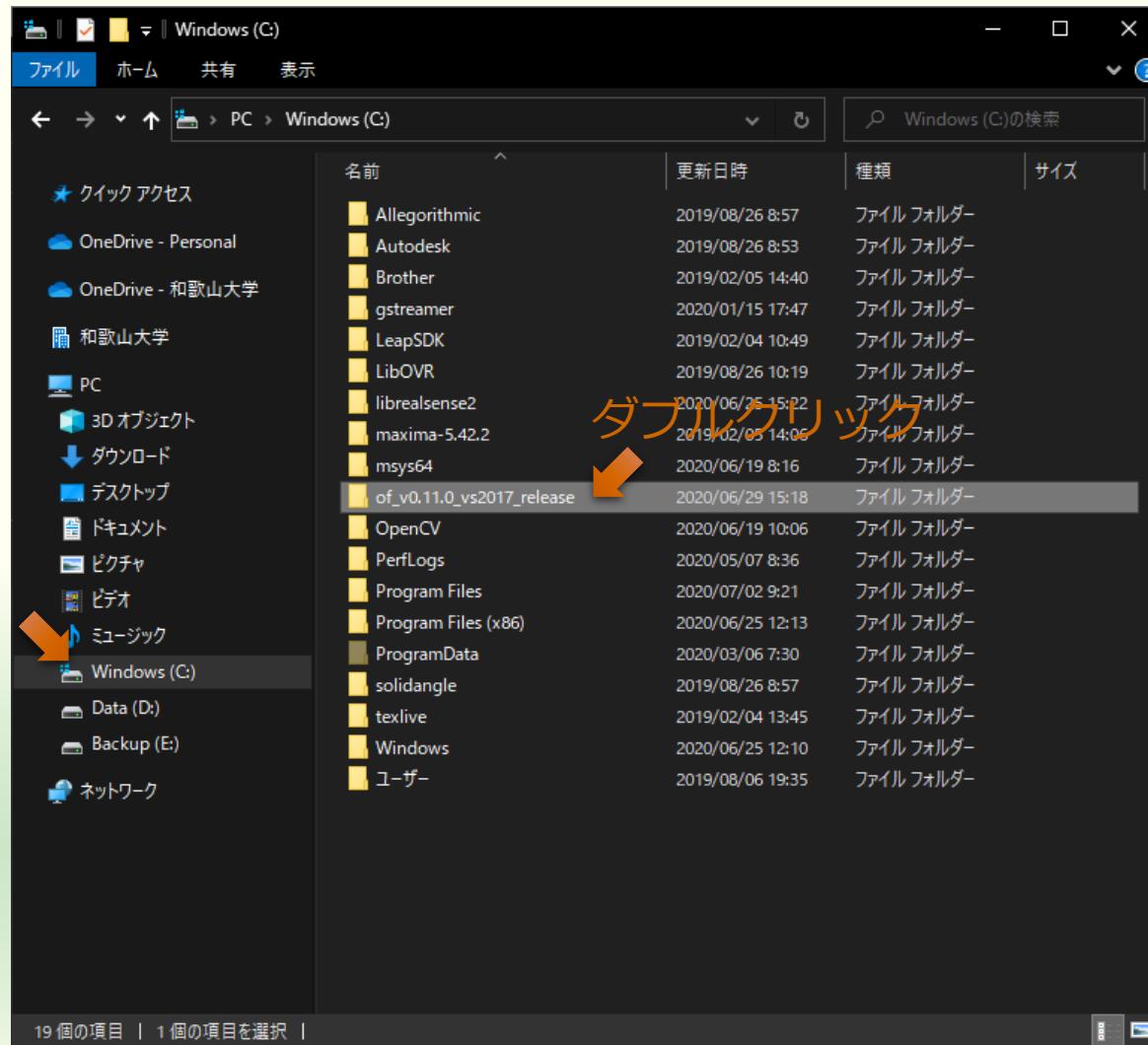


- “ドキュメント” は実体が “Documents” なので可
- “デスクトップ” も実体が “Desktop” なので可
- ユーザ名に**空白文字**を含む場合 (“Taro Yamada” など) 不可
- ユーザ名に**全角文字**を含む場合 (“山田太郎” など) 不可

# ZIP ファイルの中身を C:¥ にコピー



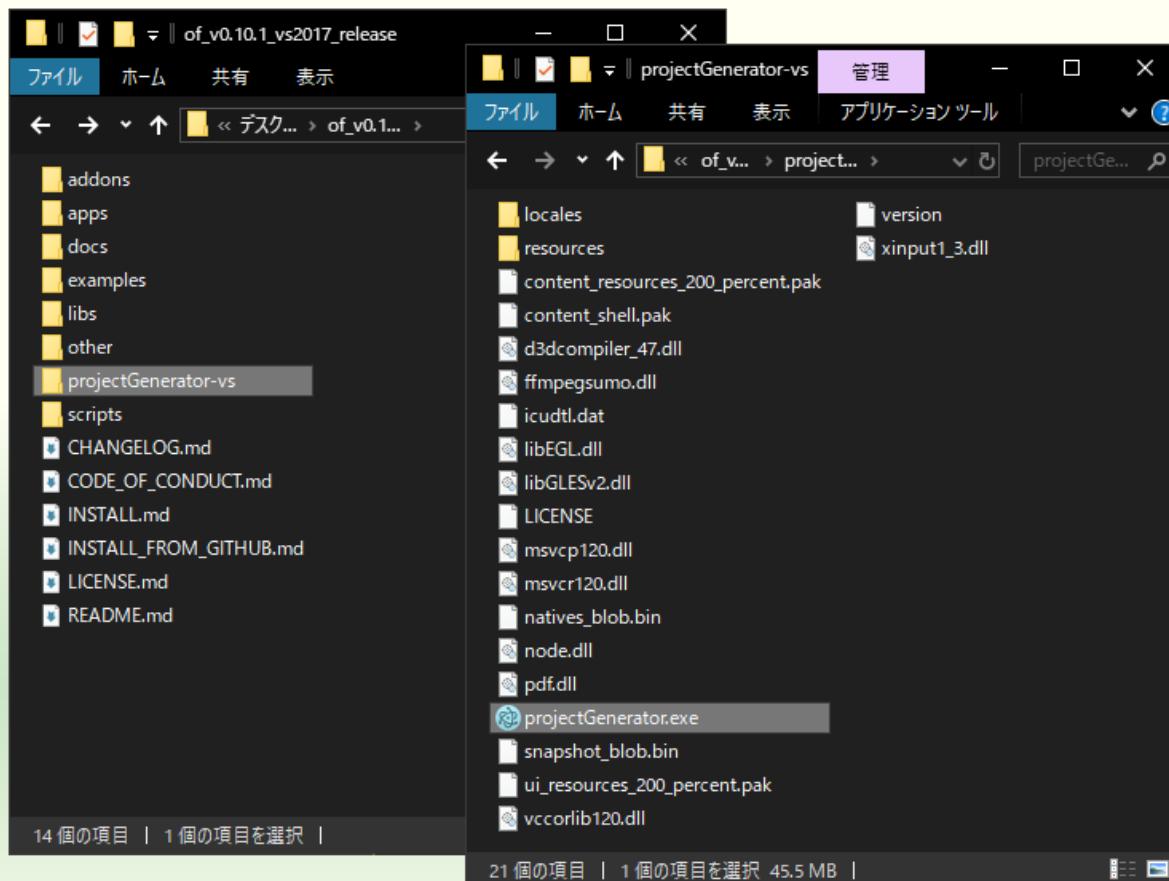
# C: ドライブに展開したパッケージの中身



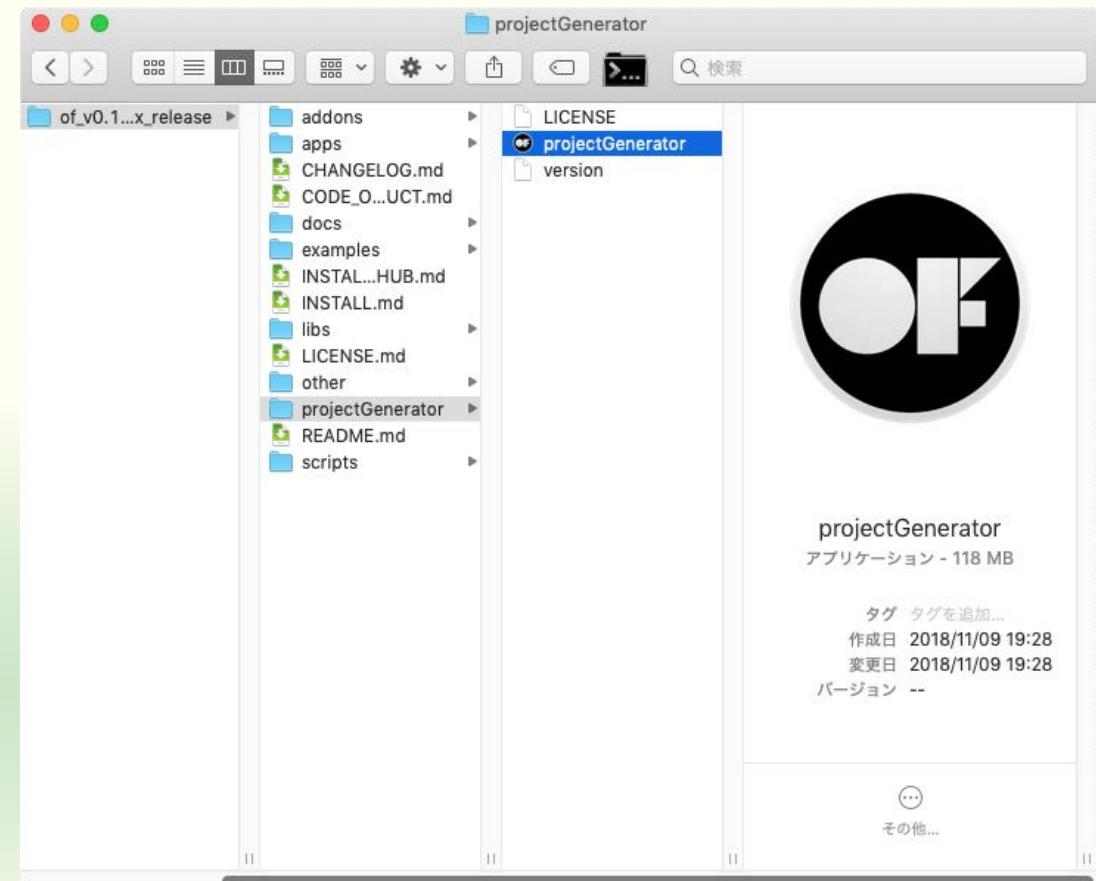
openFrameworks のパッケージの中身

# パッケージの内容

## windows 版のパッケージ



## osx 版のパッケージ



# 配置したパッケージはむやみに移動しない

- openFrameworks を使って作ったプログラム（プロジェクト）は openFrameworks のフォルダ内にある
- 何かのプロジェクトをビルド後にパッケージ自体を別のところに移動すると以降ビルドに失敗するようになる
  - 最初にビルドしたときに作られる openFrameworks のライブラリファイルの場所を絶対パスで記録している

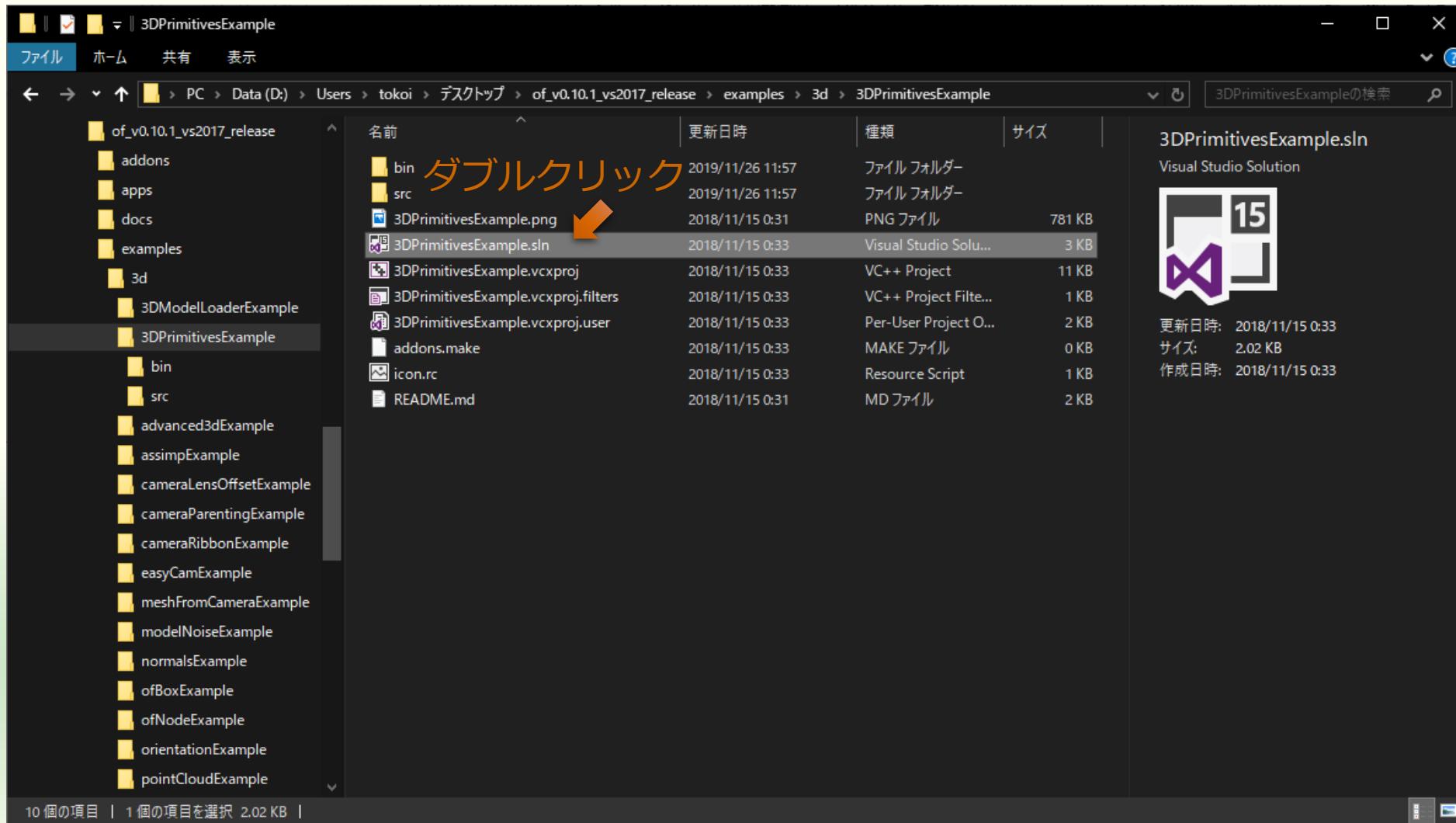




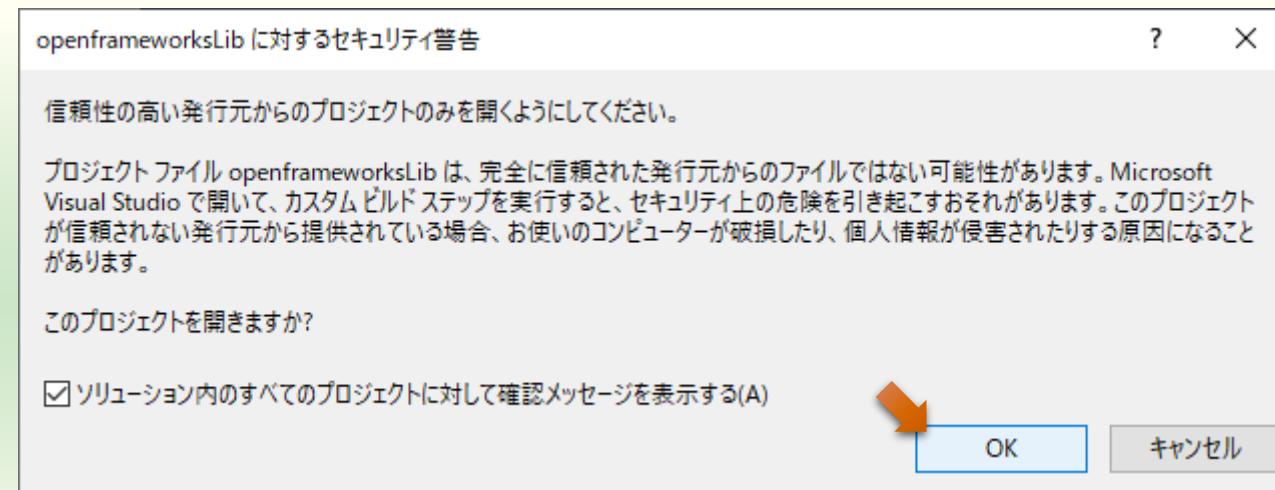
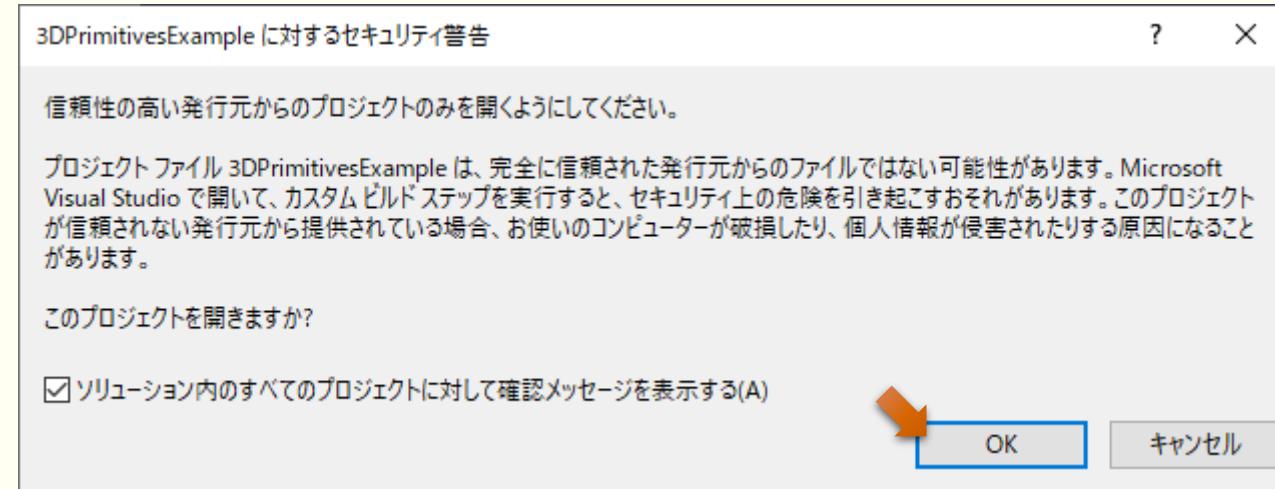
# 課題 1 – 2

サンプルプロジェクトのビルド

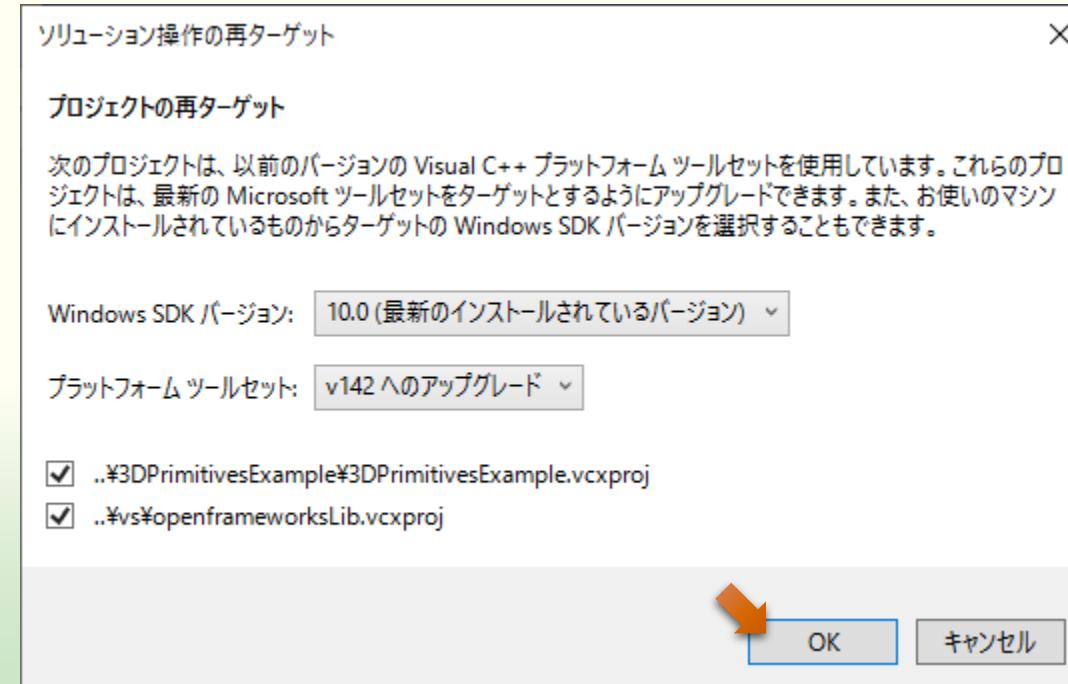
# (windows) 3DPrimitivesExample.sln



# セキュリティ警告が出ても「OK」

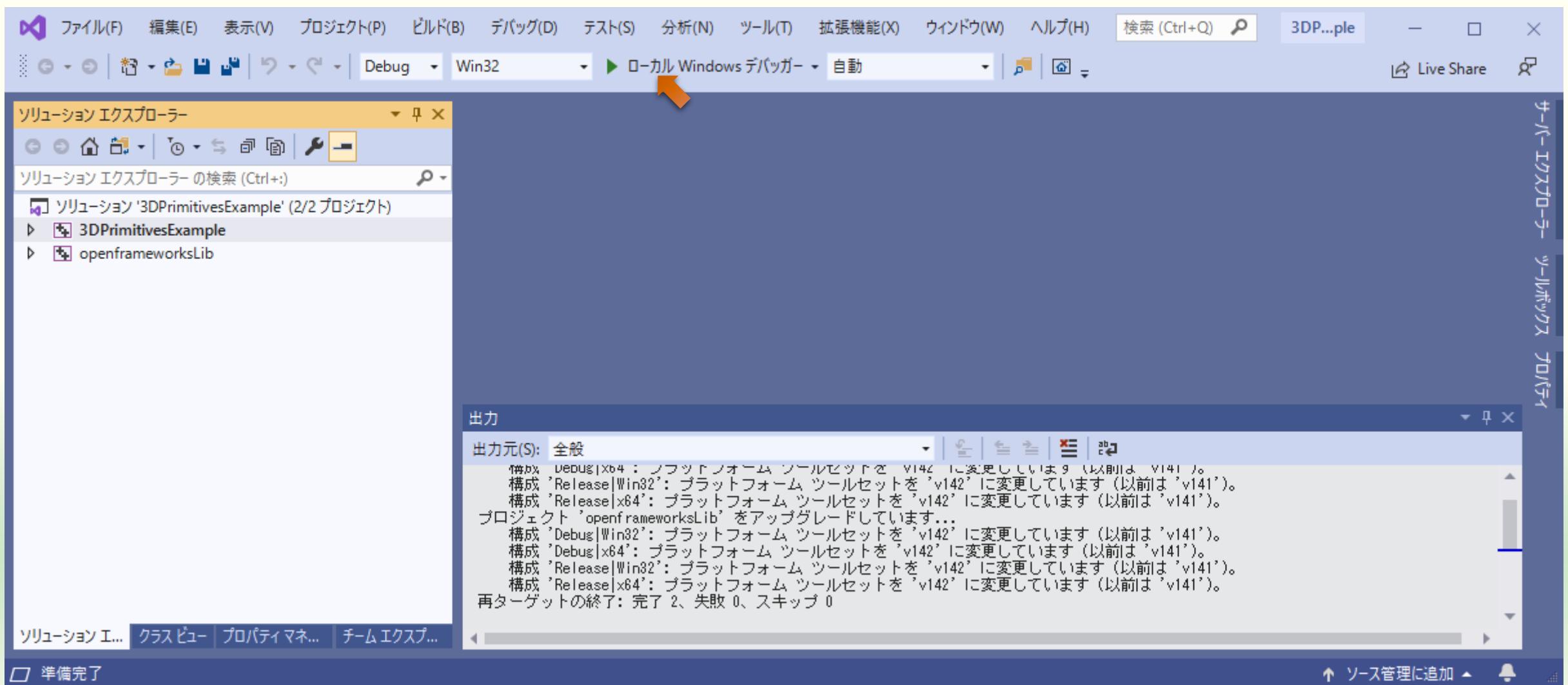


# ソリューションの再ターゲット

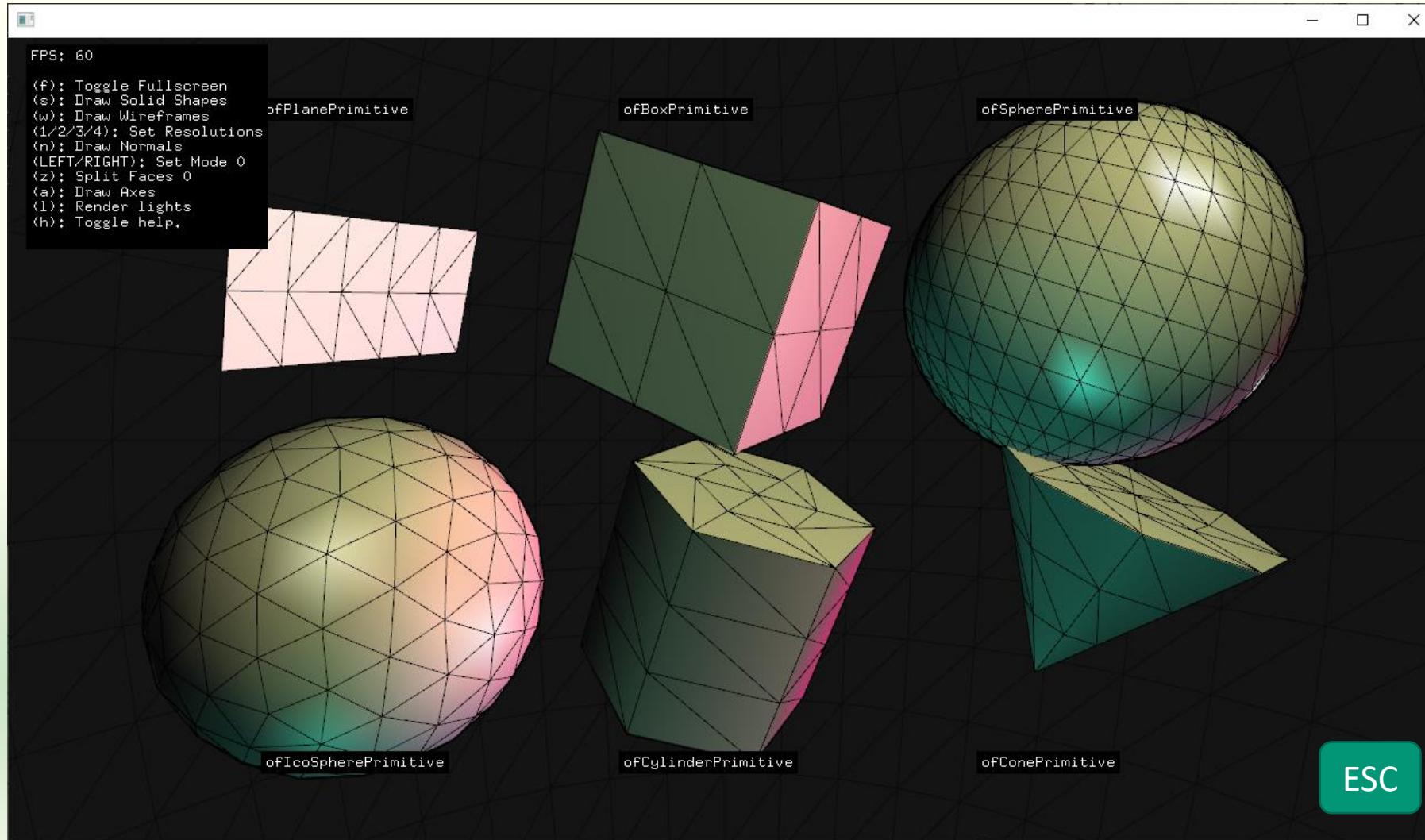


Visual Studio は頻繁に更新しているので皆さんのがお使いの Visual Studio SDK のバージョンと合わない場合がある

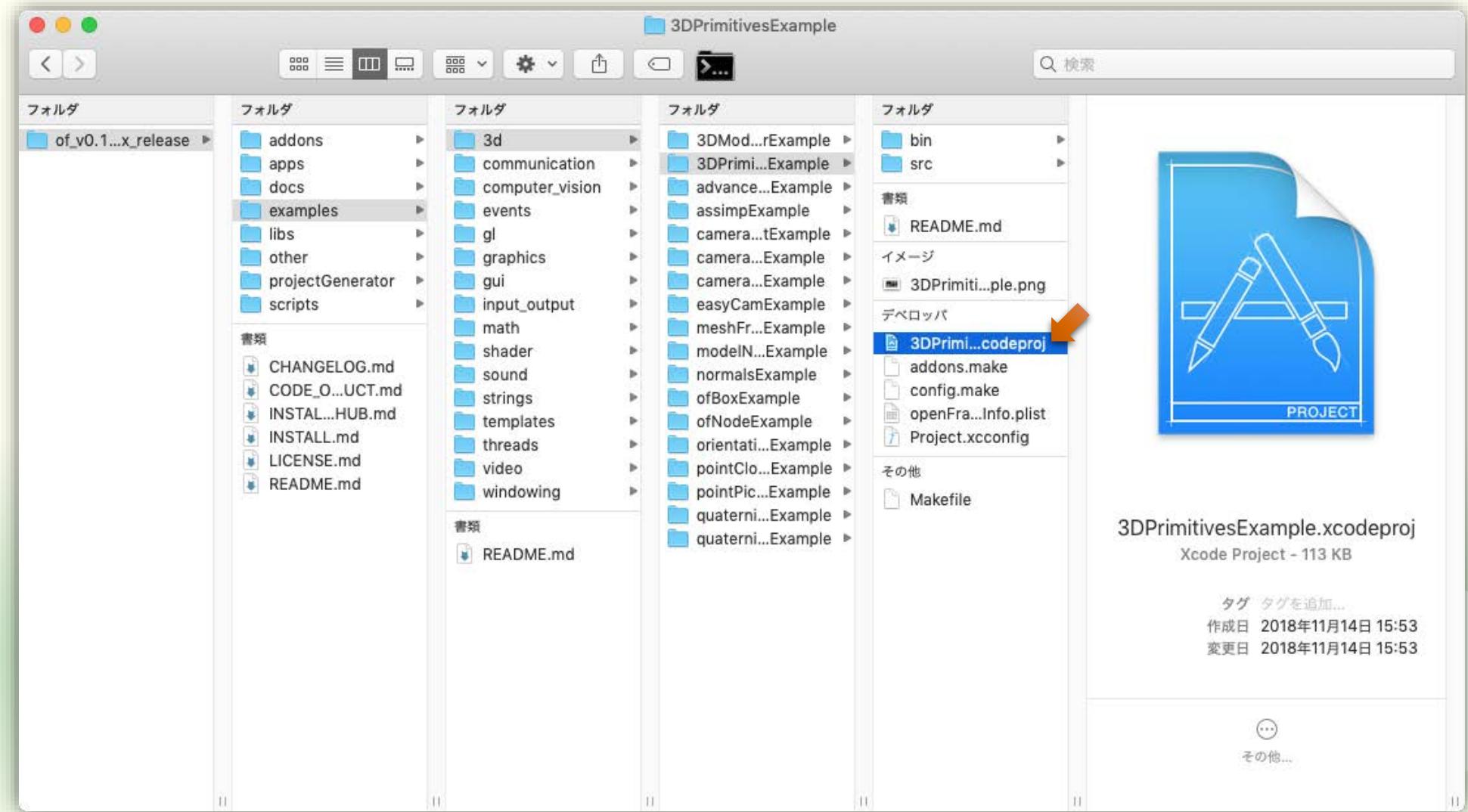
# (windows) デバッグを開始する



# (windows) 実行中

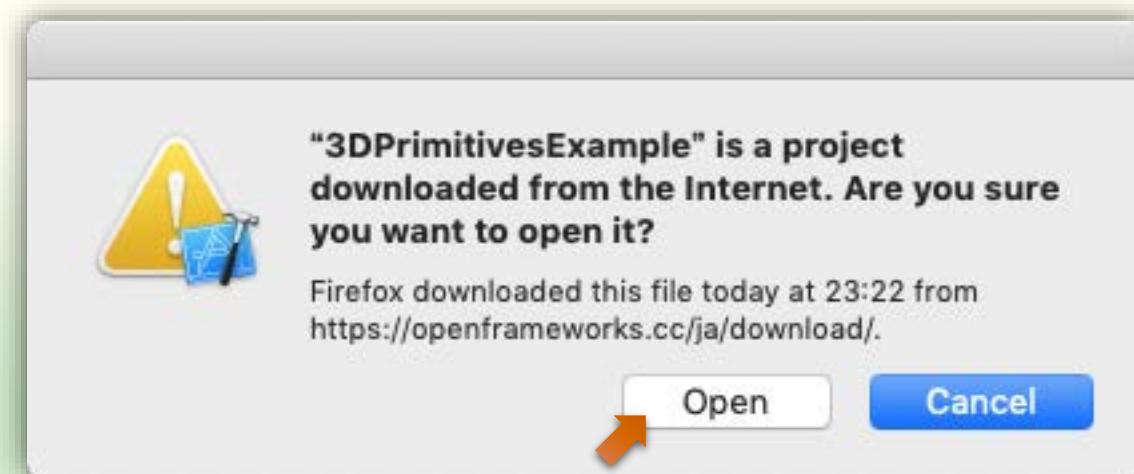


# (macOS) 3DPrimitivesExample.xcodeproj

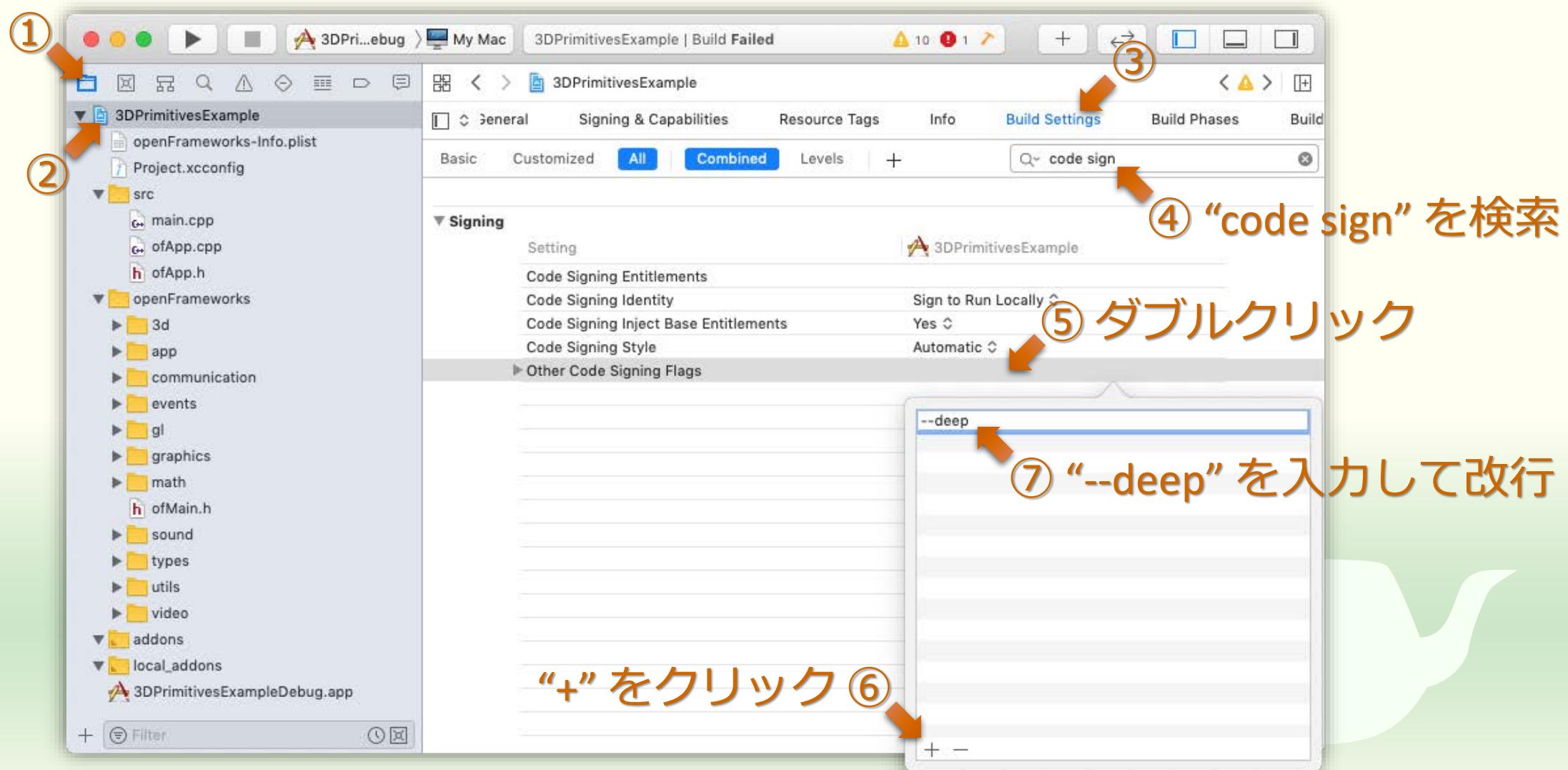


# (macOS) “Open” する

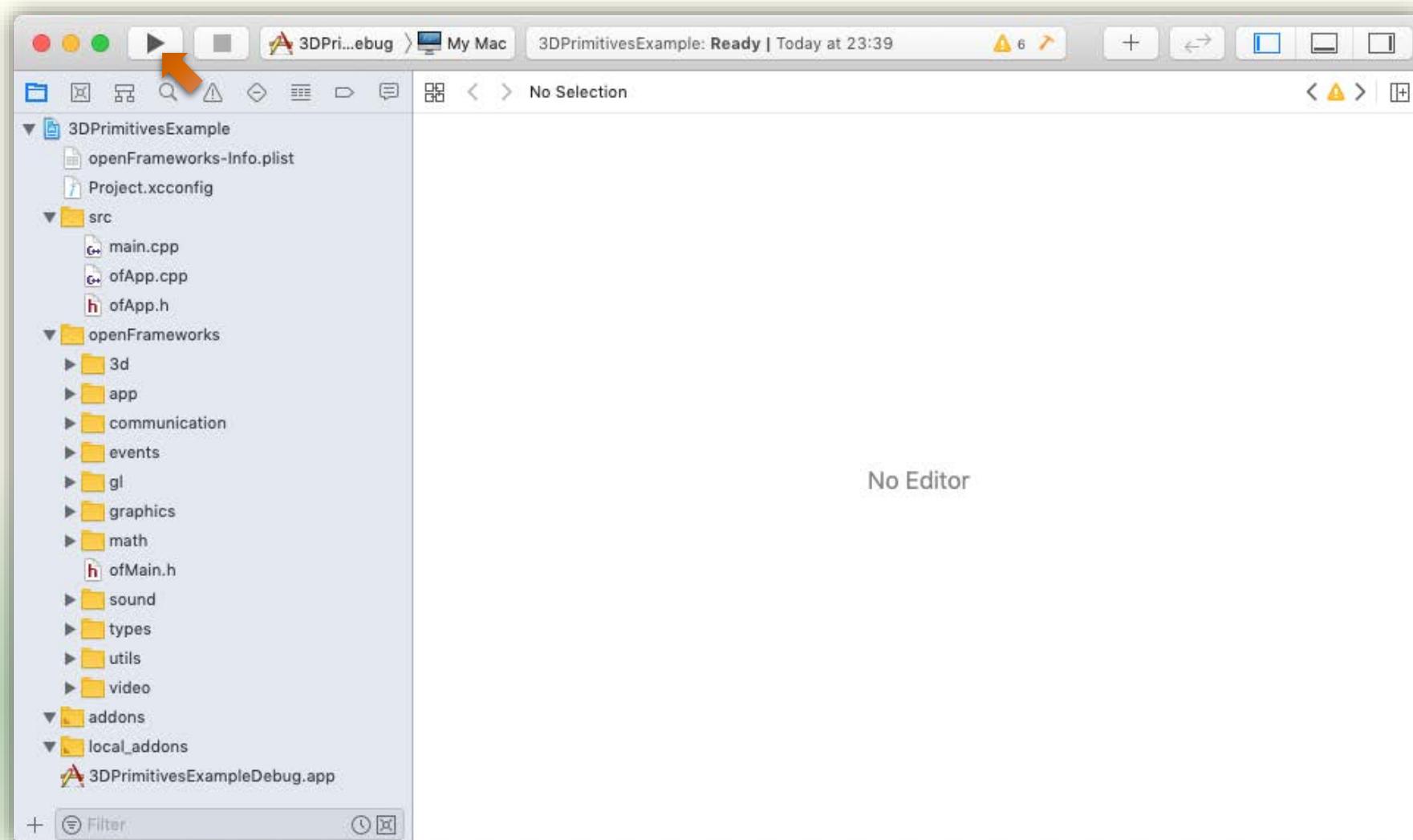
- 「インターネットからダウンロードしてきたものだけど本当に開いてよいか」って聞かれるので “Open” をクリック



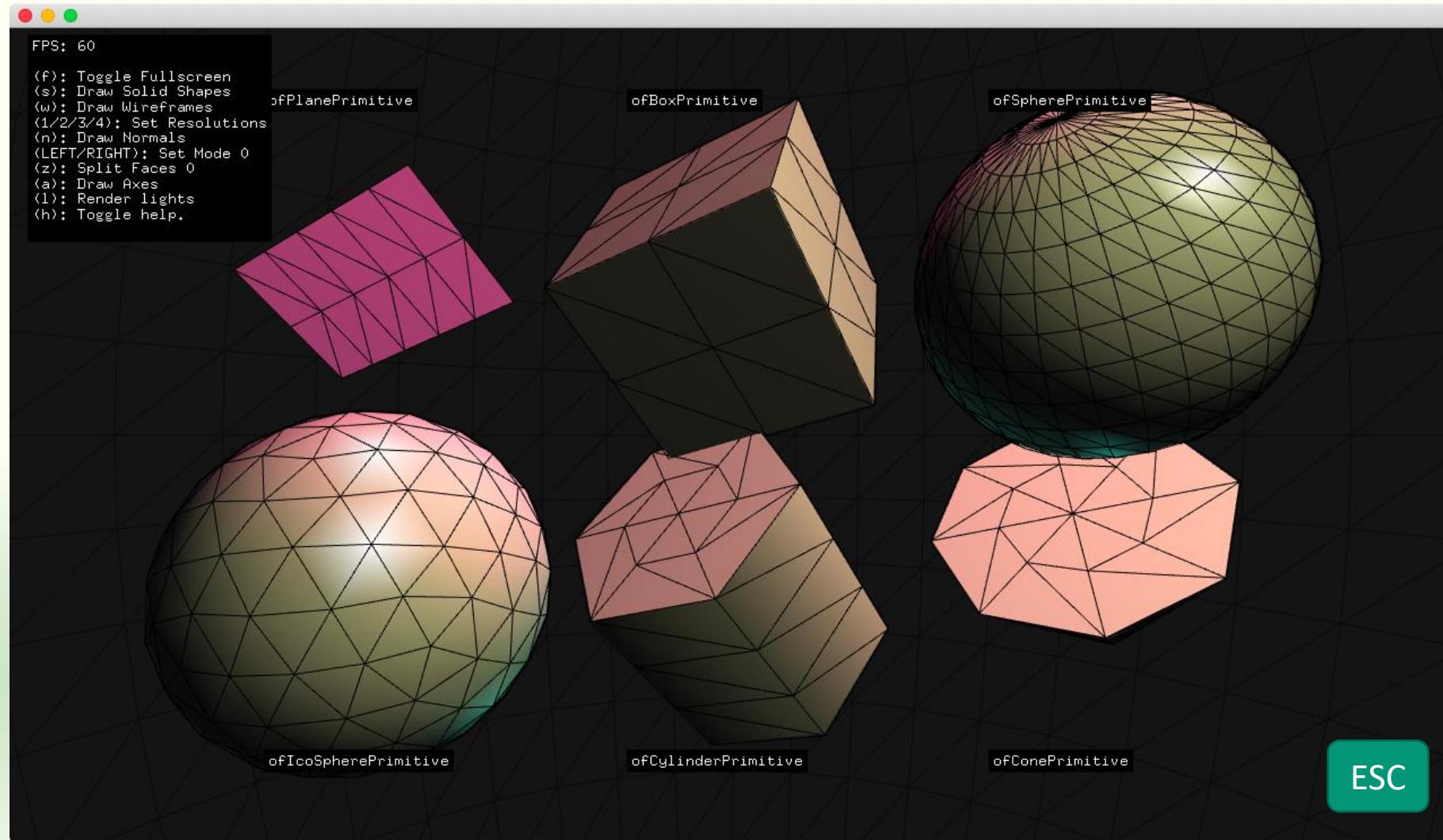
# (macOS) Other Code Signing Flags に “--deep”



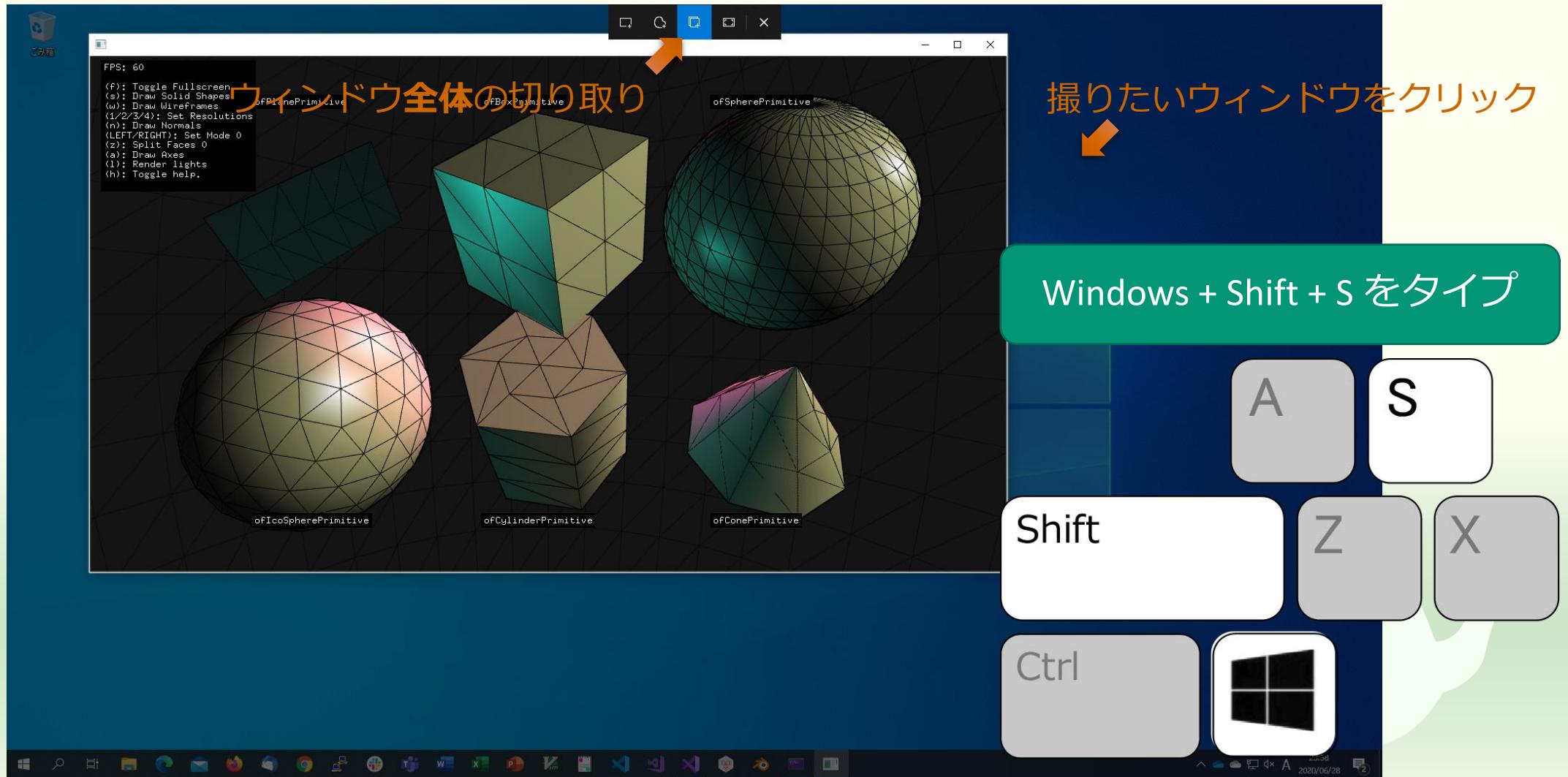
# (macOS) “Run” する



# (macOS) 実行中

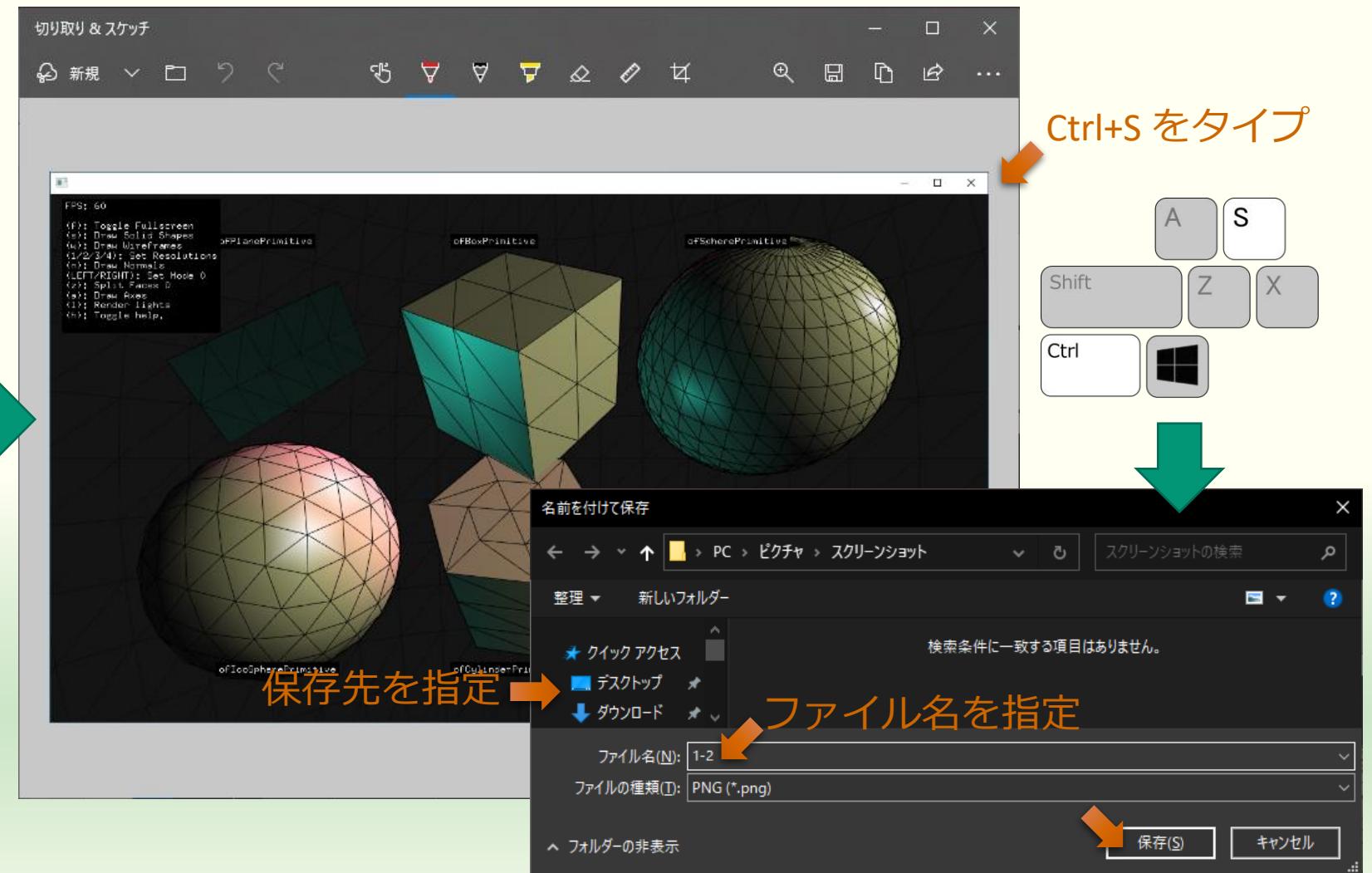


# 実行結果のスクリーンショットを撮る



# “1-2.png” というファイル名で保存

通知をクリック



# スクリーンショットのアップロード

- 実行結果のスクリーンショットを **1-2.png** というファイル名で Moodle の第 1 回課題にアップロードしてください





# 課題 1 – 3

他のサンプルプロジェクトのビルド

# 他のサンプルプロジェクトをビルドする

- example のサンプルプロジェクトは openFrameworks で何か作ろうとしたときに必ず参考になります
- 他のサンプルプロジェクトのプロジェクト名を一通り見てください
  - 何をするものか考えてください
- これらの中から 3 つ以上のサンプルプロジェクトをビルド・実行してみてください
  - カメラやマイクが必要なものもあります



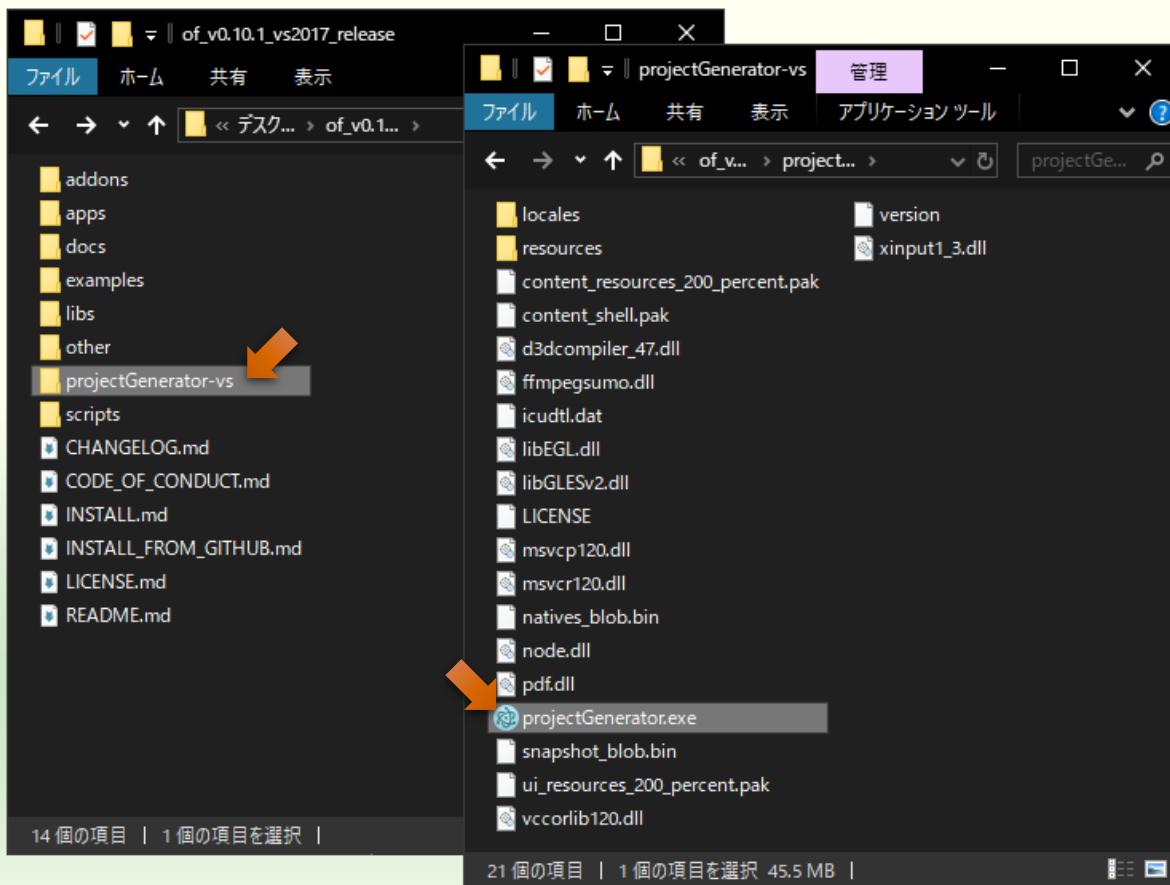


# 課題 1 – 4

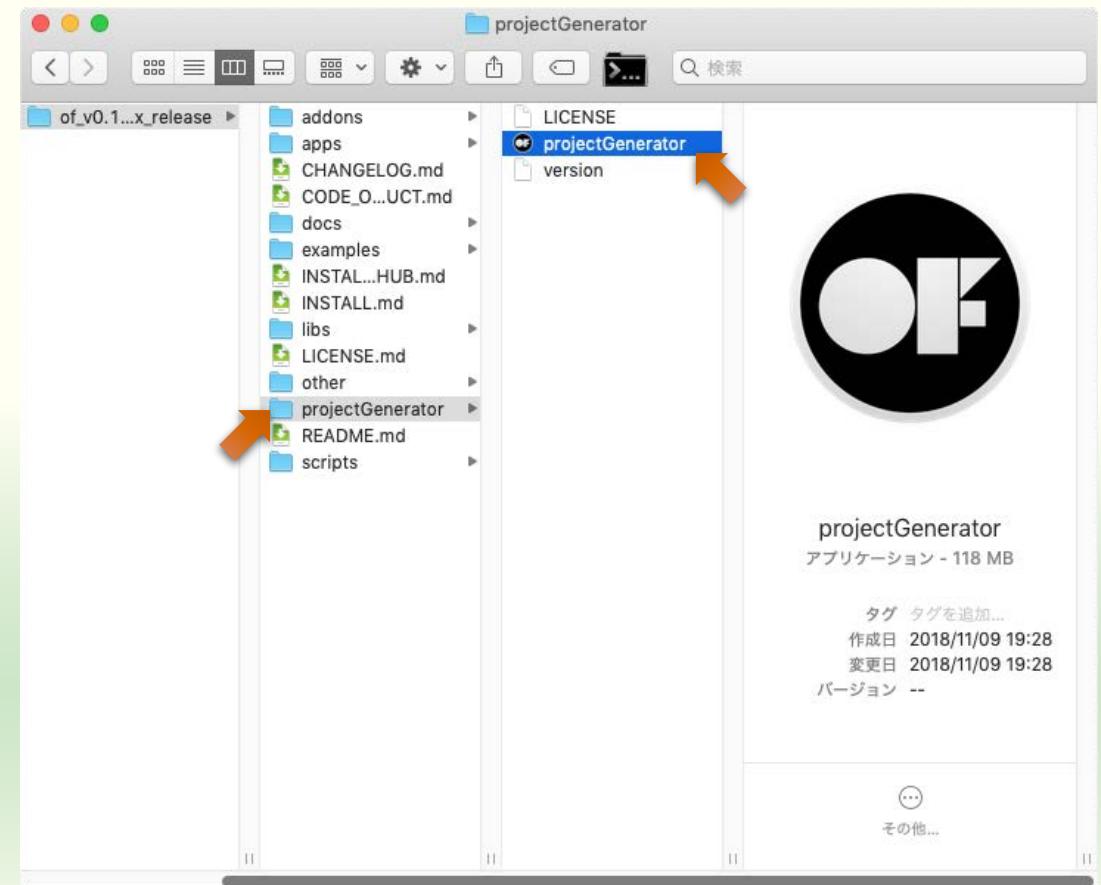
空のプロジェクトを作成してビルド

# projectGenerator を起動する

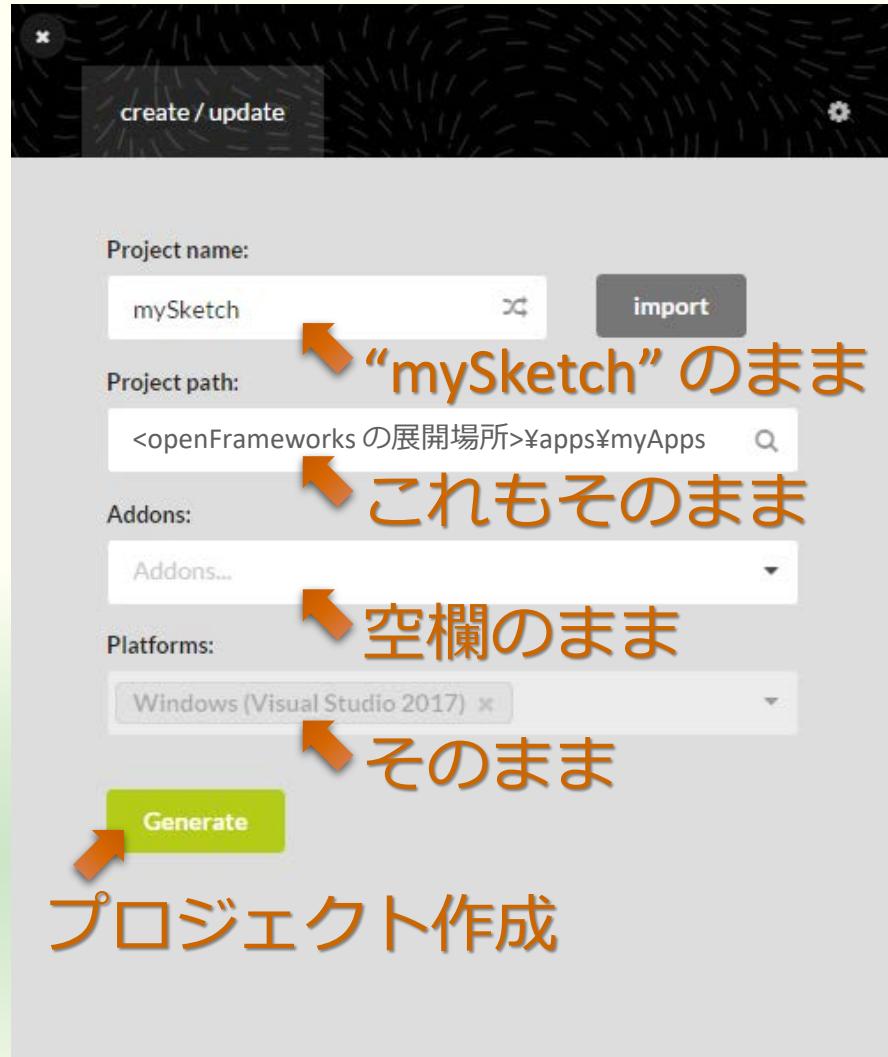
## windows 版のパッケージ



## macos 版のパッケージ



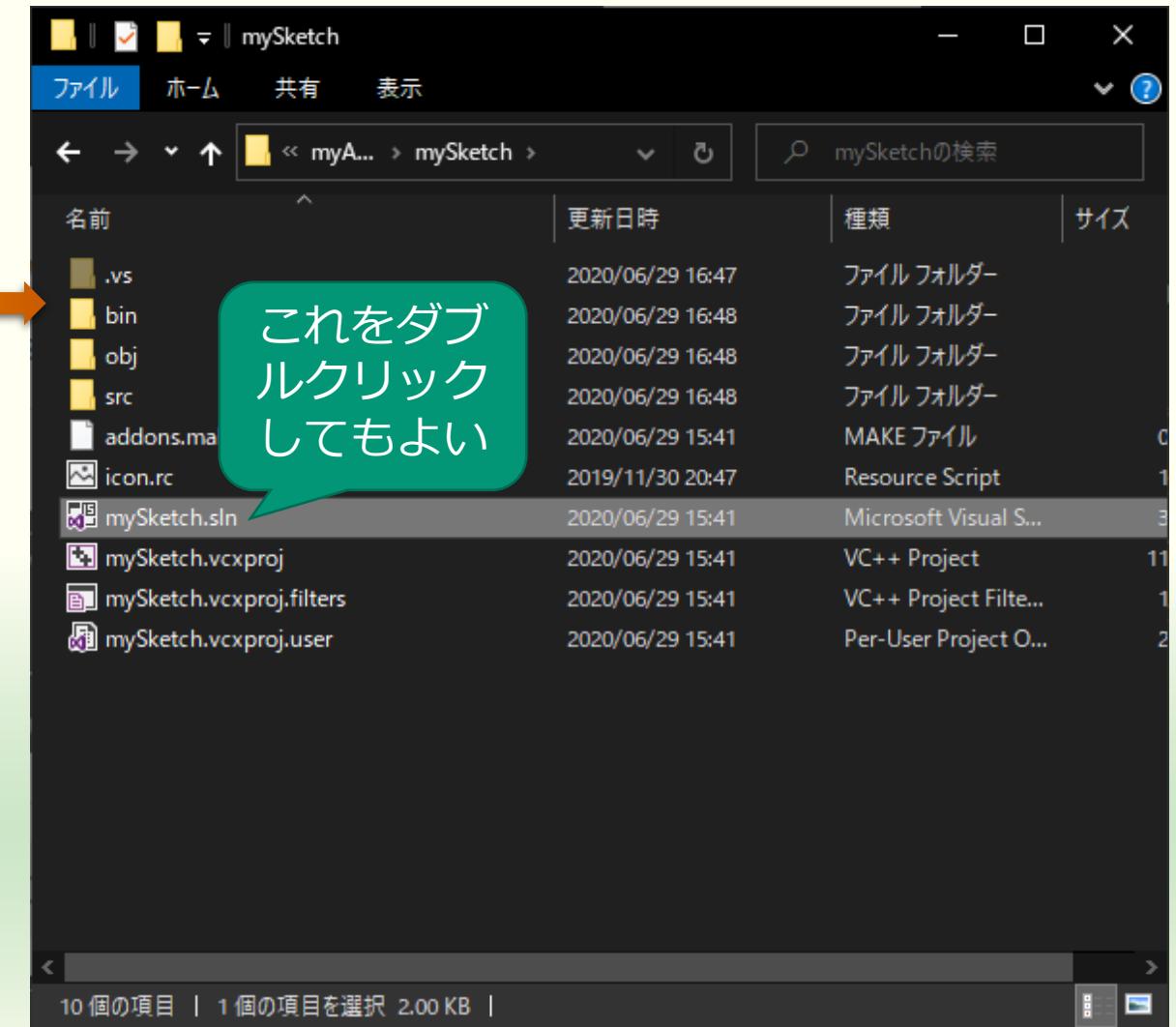
# 空のプロジェクトの作成



- 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



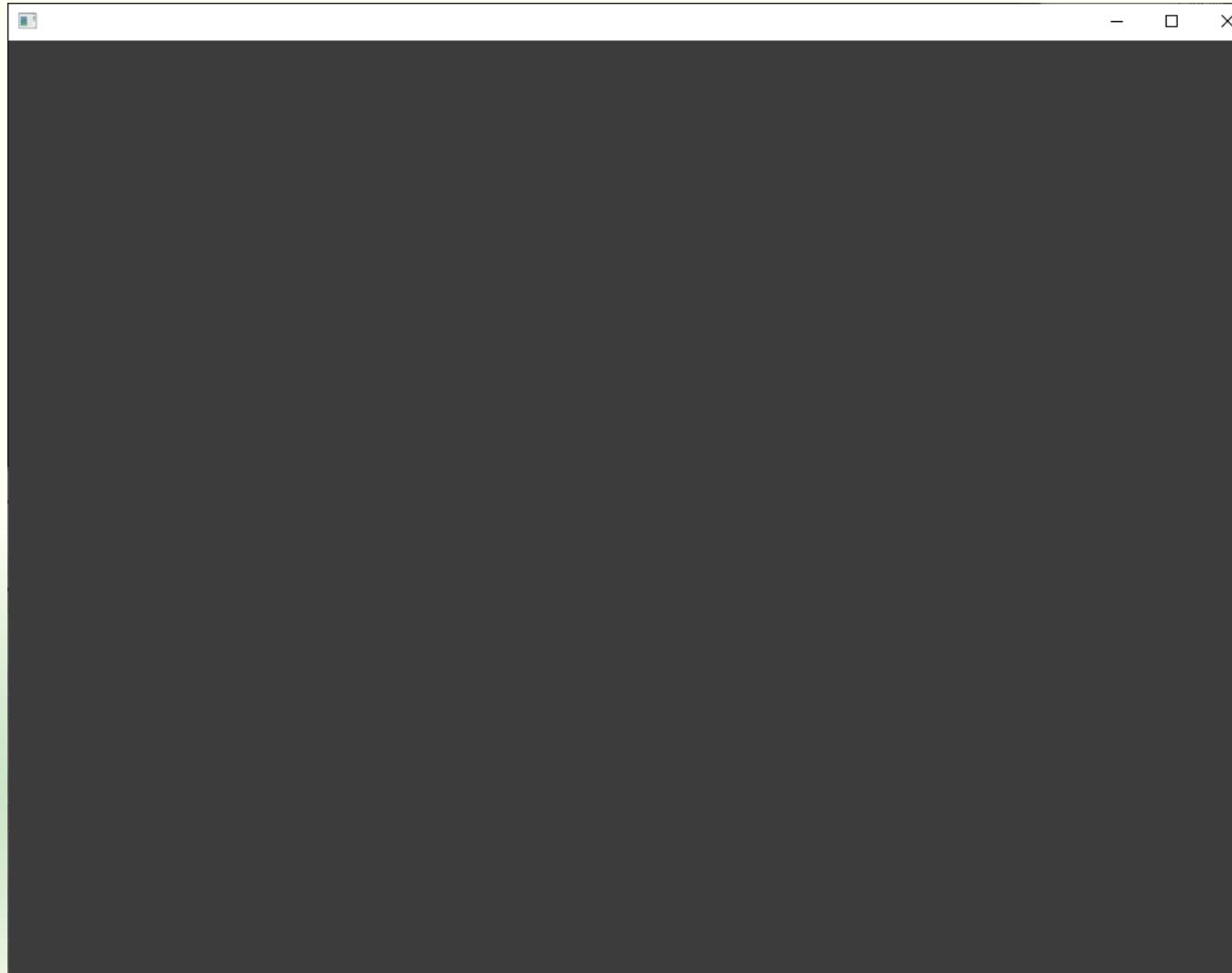
# プロジェクトの作成成功



# IDE (Visual Studio, Xcode) で開く

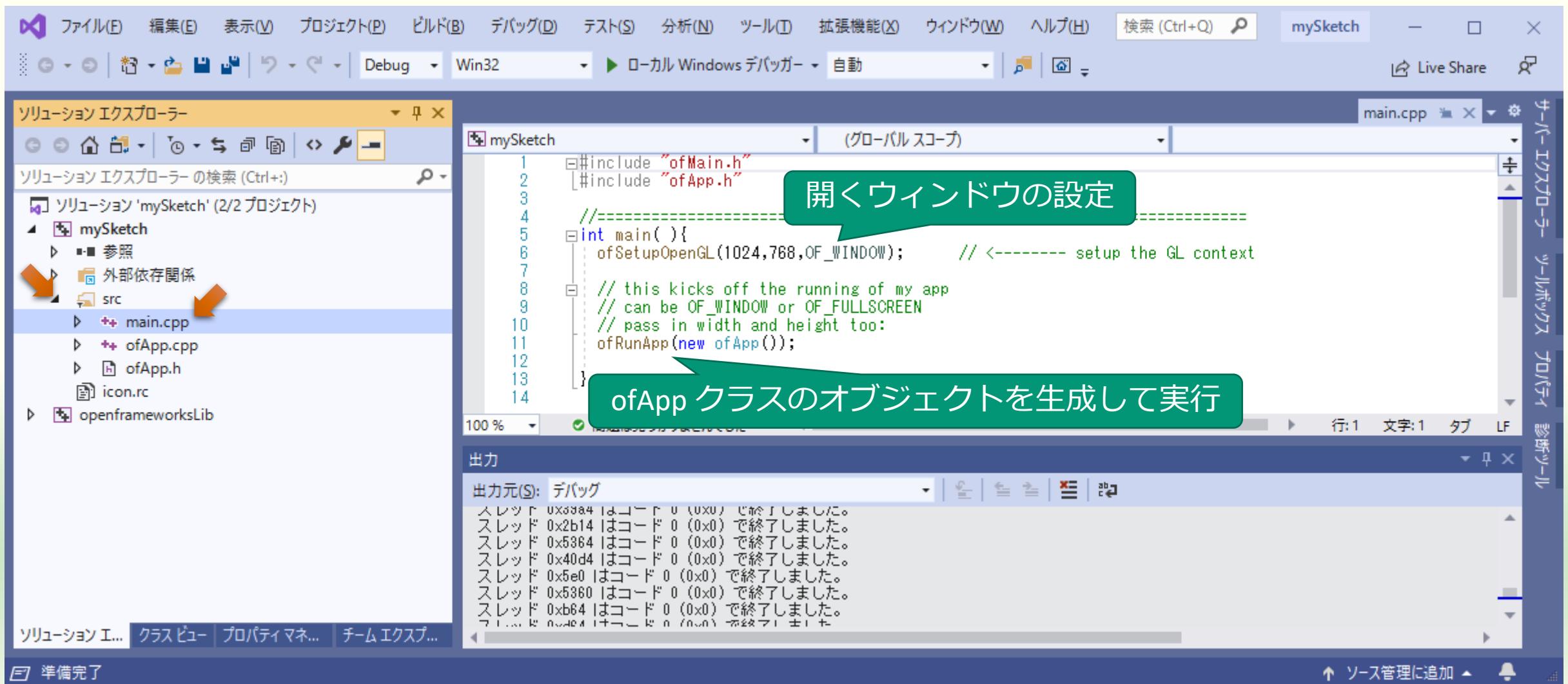


# 実行中のウィンドウ



ESC キーで終了

# main() 関数は（今は）いじらない

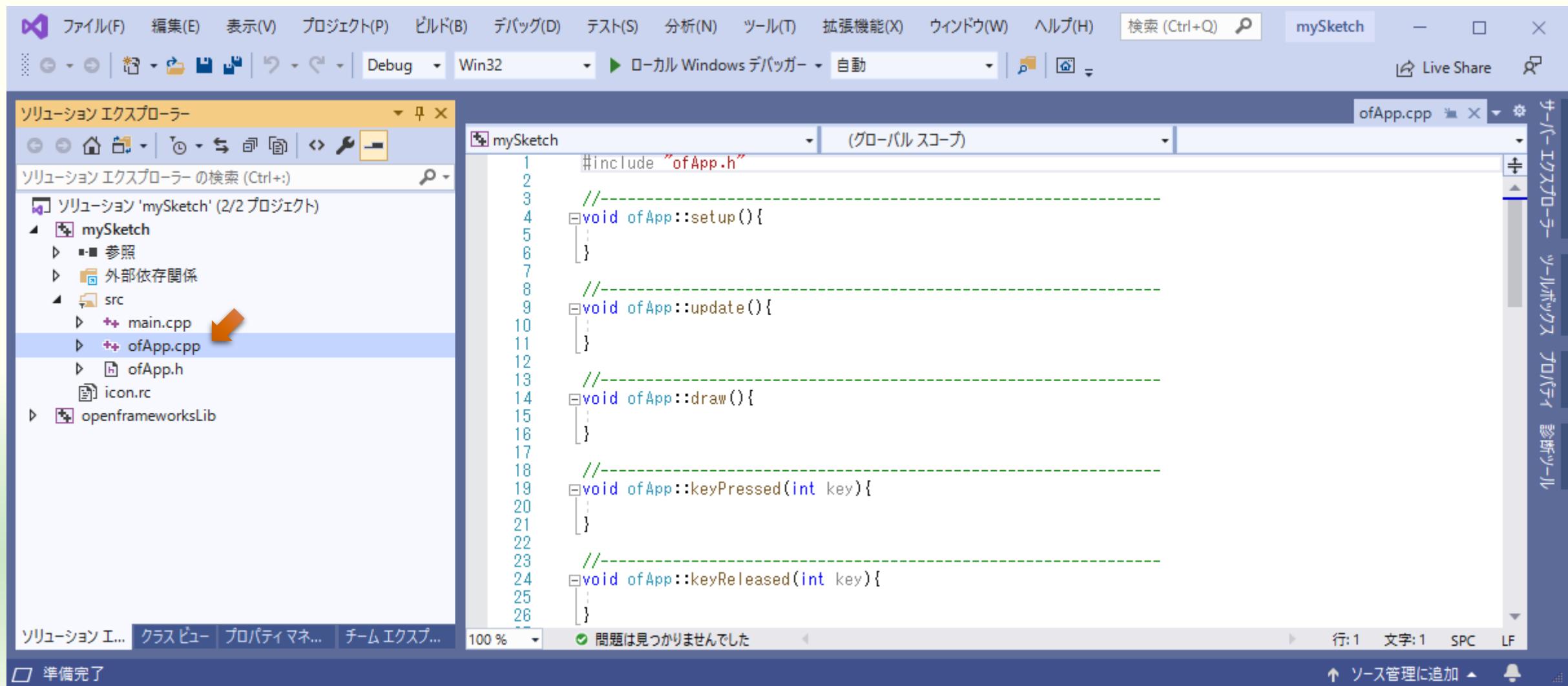


# ofApp クラス (“クラス” の話は後日)



```
1 #pragma once
2
3 #include "ofMain.h"
4
5 class ofApp : public ofBaseApp{
6
7 public:
8     void setup();
9     void update();
10    void draw();
11
12    void keyPressed(int key);
13    void keyReleased(int key);
14    void mouseMoved(int x, int y );
15    void mouseDragged(int x, int y, int button);
16    void mousePressed(int x, int y, int button);
17    void mouseReleased(int x, int y, int button);
18    void mouseEntered(int x, int y);
19    void mouseExited(int x, int y);
20    void windowResized(int w, int h);
21    void dragEvent(ofDragInfo dragInfo);
22    void gotMessage(ofMessage msg);
23
24
25};
```

# ofApp クラスのメンバの実装はまだ空



Visual Studio IDE screenshot showing the project structure and code editor for a C++ project named 'mySketch'.

**Solution Explorer:** Shows the project structure with files: mySketch, src (containing main.cpp and ofApp.cpp), ofApp.h, icon.rc, and openframeworksLib.

**Code Editor (ofApp.cpp):** Displays the following code:

```
#include "ofApp.h"

//-
//void ofApp::setup(){
//}

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

//-
//void ofApp::draw(){
//}

//-
//void ofApp::keyPressed(int key){
//}

//-
//void ofApp::keyReleased(int key){
}
```

The file 'ofApp.cpp' is selected in the Solution Explorer, indicated by a blue selection bar and an orange arrow pointing to it.



# 課題 1 – 5

openFrameworks を使ったプログラムの動作を確認してみよう

# ofApp.cpp の内容 (先頭部分)

```
#include "ofApp.h"

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

}

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

}

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

}

//-----
void ofApp::keyPressed(int key){

}
```

# メンバ関数 (画面表示関連)

- `void ofApp::setup()`
  - アプリケーションを起動したときに一度だけ実行される
- `void ofApp::update()`
  - 画面の表示を行う前に繰り返し実行される
- `void ofApp::draw()`
  - 画面の表示を繰り返し行う

“void” は戻り値を返さないことを表す  
これらの関数は `return` を使わない

# setup() で文字を出力してみる

```
#include "ofApp.h"  
  
//-----  
void ofApp::setup(){  
    std::cout << "setup()\n";  
}  
  
//-----  
void ofApp::update(){  
}  
  
//-----  
void ofApp::draw(){  
}
```

setup()

setup() は  
最初に一度だけ実行される

# update() で文字を出力してみる

```
#include "ofApp.h"

//-----
void ofApp::setup(){
    std::cout << "setup()\n";
}

//-----
void ofApp::update(){
    std::cout << "update()\n";
}

//-----
void ofApp::draw(){
}
```

```
setup()
update()
...
```

update() は  
繰り返し何度も実行されている

# draw() で文字を出力してみる

```
#include "ofApp.h"

//-----
void ofApp::setup(){
    std::cout << "setup()\n";
}

//-----
void ofApp::update(){
    std::cout << "update()\n";
}

//-----
void ofApp::draw(){
    std::cout << "draw()\n";
}
```

```
setup()
update()
draw()
...
```

draw() は update() の  
後に実行される

# メンバ関数 (キー操作関連)

- `keyPressed(int key)`
  - キーを押したときに実行
  - `key` は押したキー
- `keyReleased(int key)`
  - キーを離したときに実行
  - `key` は押していたキー

キーは文字が表示されている**黒い**ウィンドウ (コンソール) ではなくグレーのウィンドウでタイプしてください

# keyPressed(int key) で文字を出力してみる

```
//-----
void ofApp::keyPressed(int key){
    std::cout << static_cast<char>(key)
        << " was pressed\n";
}

//-----
```

```
void ofApp::keyReleased(int key){}
```

```
...
update()
draw()
update()
draw()
c was pressed
update()
draw()
update()
draw()
update()
draw()
d was pressed
update()
draw()
update()
draw()
update()
draw()
...
...
```

このウィンドウ（コンソール）ではなく  
アプリケーションのウィンドウで

キーを押したときに  
draw() の後で実行される

# keyReleased(int key) で文字を出力してみる

```
//-----
void ofApp::keyPressed(int key){
    std::cout << static_cast<char>(key)
        << " was pressed\n";
}

//-----
void ofApp::keyReleased(int key){
    std::cout << static_cast<char>(key)
        << " was released\n";
}
```

```
...
update()
draw()
f was pressed
update()
draw()
update()
draw()
f was released
update()
draw()
g was pressed
update()
draw()
update()
draw()
g was released
update()
draw()
...
...
```

キーを離したときに  
draw() の後で実行される



# 課題 1 – 6

簡単な図形を描いてみよう

# setup() で背景色を指定する

```
#include "ofApp.h"

//-----
void ofApp::setup(){
    ofBackground(0, 0, 0);
}
    (r = 0, g = 0, b = 0) ⇒ 黒
//-----
void ofApp::update(){

}

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

}
```

- `void ofBackground(int r, int g, int b, int a=255)`
  - 背景色を指定する
    - r: 背景色の赤成分の強さ、0～255
    - g: 背景色の緑成分の強さ、0～255
    - b: 背景色の青成分の強さ、0～255
    - a: 背景色の不透明度、0（透明）～255（不透明）、省略時は255
  - `ofBackground(255, 255, 255)` なら白
  - マニュアル
    - [https://openframeworks.cc/documentation/graphics/ofGraphics/#show\\_ofBackground](https://openframeworks.cc/documentation/graphics/ofGraphics/#show_ofBackground)

# 背景色が黒になる

---



# マニュアルの見方

## global functions

- > `ofBackground()`
- > `ofBackgroundGradient()`
- > `ofBackgroundHex()`
- > `ofBeginSaveScreenAsPDF()`
- > `ofBeginSaveScreenAsSVG()`
- > `ofBeginShape()`
- > `ofBezierVertex()`
- > `ofClear()`
- > `ofClearAlpha()`
- > `ofCurveVertex()`
- > `ofCurveVertices()`
- > `ofDisableAlphaBlending()`
- > `ofDisableAntiAliasing()`
- > `ofDisableBlendMode()`
- > `ofDisableDepthTest()`
- > `ofDisablePointSprites()`
- > `ofDisableSmoothing()`

### `ofBackground(...)`

`void ofBackground(const ofColor &c)`

### `ofBackground(...)`

`void ofBackground(int brightness, int alpha=255)`

### `ofBackground(...)`

`void ofBackground(int r, int g, int b, int a=255)`

*Documentation from code comments*

Sets the background color.

今回使用したもの

引数の指定方法は  
3種類ある

```
void ofApp::setup(){
    ofBackground(255,0,0); // Sets the background color to red
}
```

使用例

# 関数の定義の見方

void ofBackground(int r, int g, int b, int a=255)

戻り値のデータ型  
void は値を返さない場合

第1引数は int 型

第2引数は int 型

第3引数は int 型

第4引数は int 型  
省略したときは 255

使用例

ofBackground(0, 0, 0)

背景色は不透明の黒

ofBackground(255, 0, 0, 51)

背景色は不透明度20%の赤



# draw() で円を描く

```
#include "ofApp.h"

//-----
void ofApp::setup(){
    ofBackground(0, 0, 0);
}

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

}

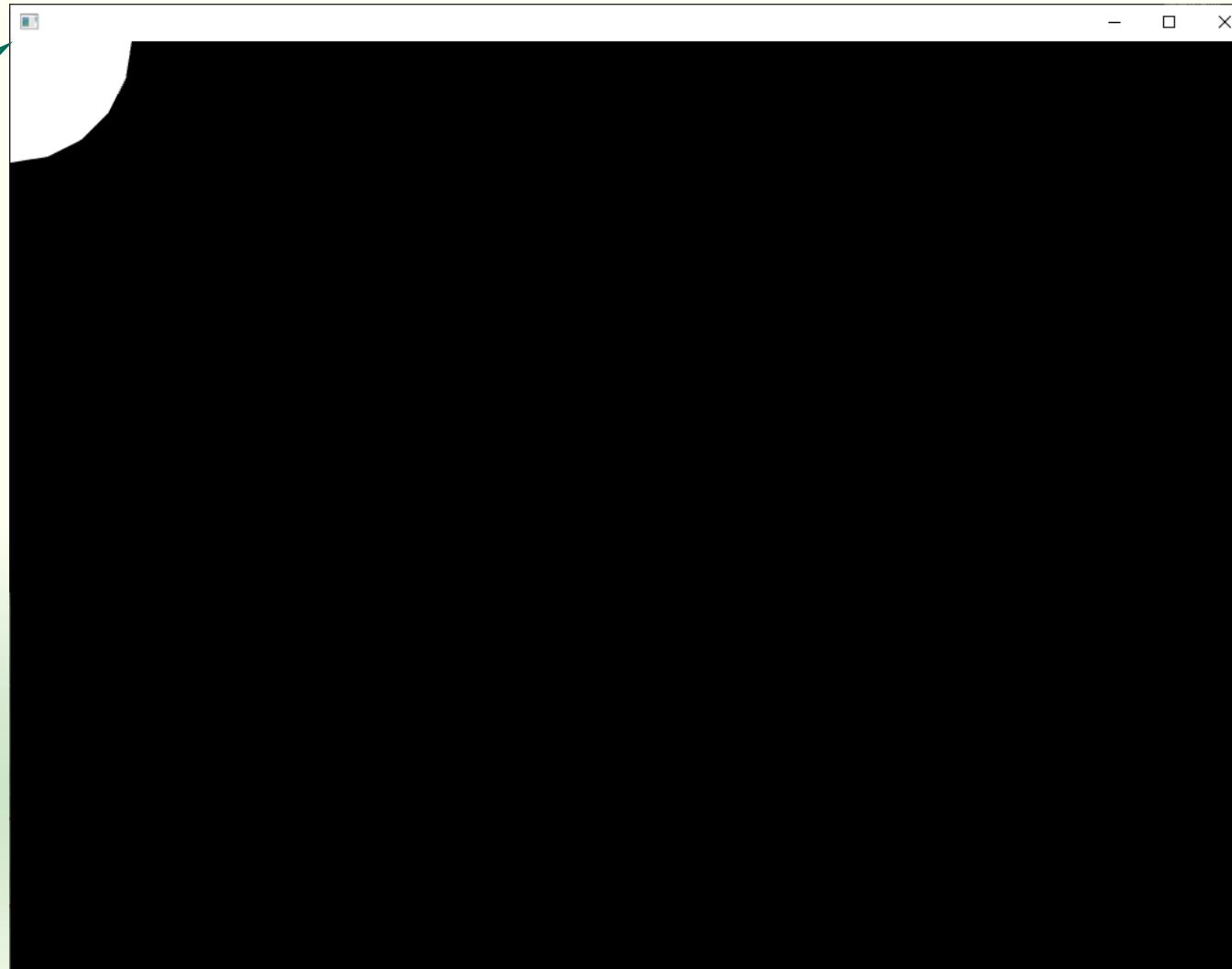
//-----
void ofApp::draw(){
    ofDrawCircle(0.0f, 0.0f, 100.0f);
}
```

中心 (0, 0), 半径 100

- void ofDrawCircle(float x, float y, float radius)
  - 円を描く
    - x: 円の中心の x 座標値
    - y: 円の中心の y 座標値
    - radius: 円の半径
  - ofDrawCircle(30.0f, 40.0f, 100.0f) なら (30, 40) を中心に半径 100 の円
- マニュアル
  - [https://openframeworks.cc/documentation/graphics/ofGraphics/#!show\\_ofDrawCircle](https://openframeworks.cc/documentation/graphics/ofGraphics/#!show_ofDrawCircle)

# 左上に円が描かれる

左上が原点 (0, 0)



# 円の色を変える

```
#include "ofApp.h"

//-----
void ofApp::setup(){
    ofBackground(0, 0, 0);
}

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

}

//-----
void ofApp::draw(){
    (r = 255, g = 0, b = 0) ⇒ 赤
    ofSetColor(255, 0, 0);
    ofDrawCircle(0.0f, 0.0f, 100.0f);
}
```

- void ofSetColor(int r, int g, int b)
- void ofSetColor(int r, int g, int b, int a)
- 以後描画するものの色を指定する
  - r: 背景色の赤成分の強さ、0～255
  - g: 背景色の緑成分の強さ、0～255
  - b: 背景色の青成分の強さ、0～255
  - a: 背景色の不透明度、0（透明）～255（不透明）、省略時は255
- マニュアル
  - [https://openframeworks.cc//documentation/graphicsofGraphics/#!show\\_ofSetColor](https://openframeworks.cc//documentation/graphicsofGraphics/#!show_ofSetColor)

# 円の色が変わる



# draw() で矩形を描く

```
#include "ofApp.h"

//-----
void ofApp::setup(){
    ofBackground(0, 0, 0);
}

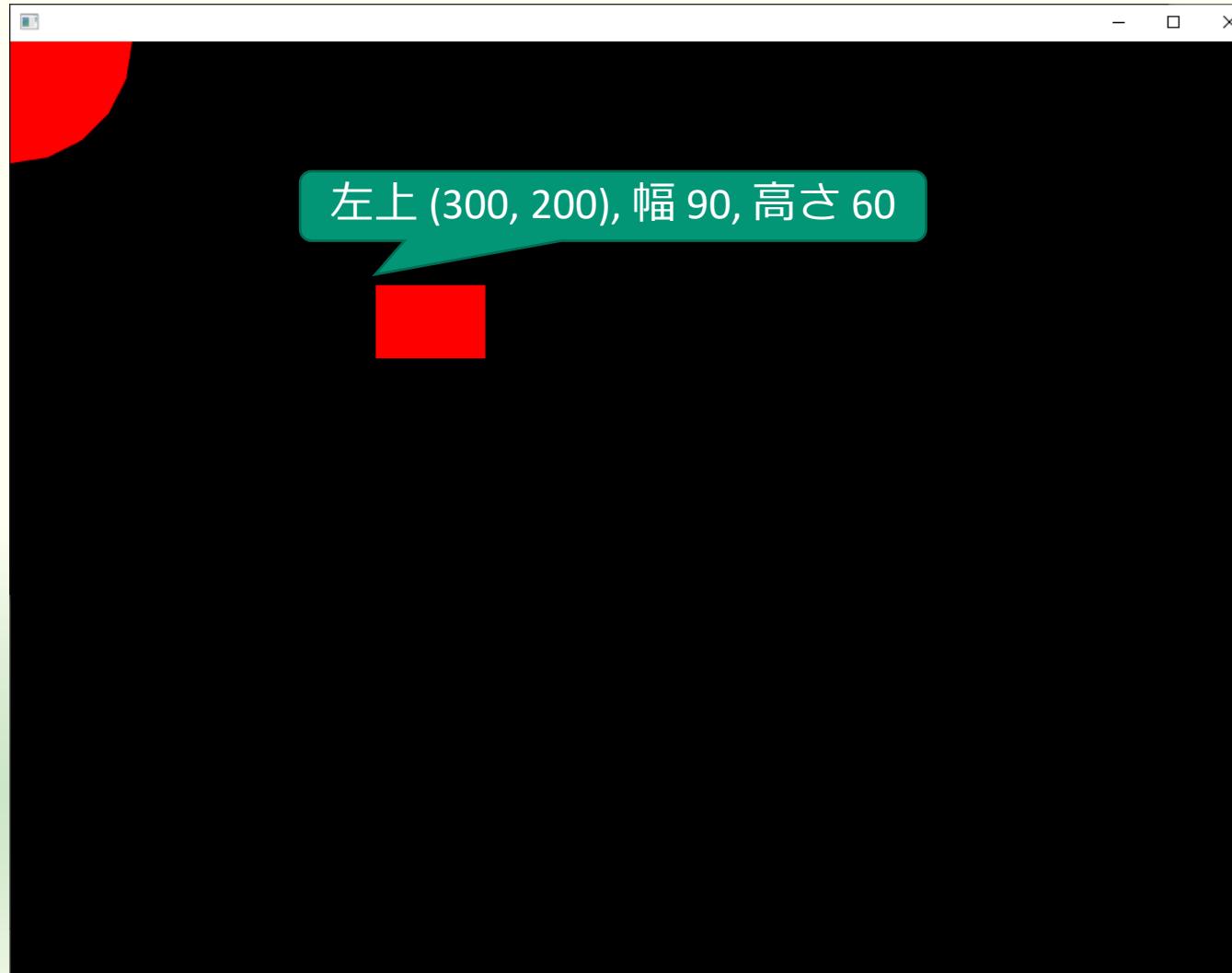
//-----
void ofApp::update(){

}

//-----
void ofApp::draw(){
    ofSetColor(255, 0, 0);
    ofDrawCircle(0.0f, 0.0f, 100.0f);
    ofDrawRectangle(300.0f, 200.0f, 90.0f, 60.0f);
}
```

- **void ofDrawRectangle(float x1, float y1, float w, float h)**
  - 矩形を描く
    - x1: 矩形の左端の x 座標
    - y1: 矩形の上端の y 座標
    - w: 矩形の幅
    - h: 矩形の高さ
  - ofDrawRectangle(10.0f, 20.0f, 100.0f, 200.0f) なら (10, 20) を左上にして 幅 100 高さ 200 の矩形
- マニュアル
  - [https://openframeworks.cc//documentation/graphicsofGraphics/#!show\\_ofDrawRectangle](https://openframeworks.cc//documentation/graphicsofGraphics/#!show_ofDrawRectangle)

# 矩形が追加される



# 矩形の色を変える

```
#include "ofApp.h"

//-----
void ofApp::setup(){
    ofBackground(0, 0, 0);
}

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

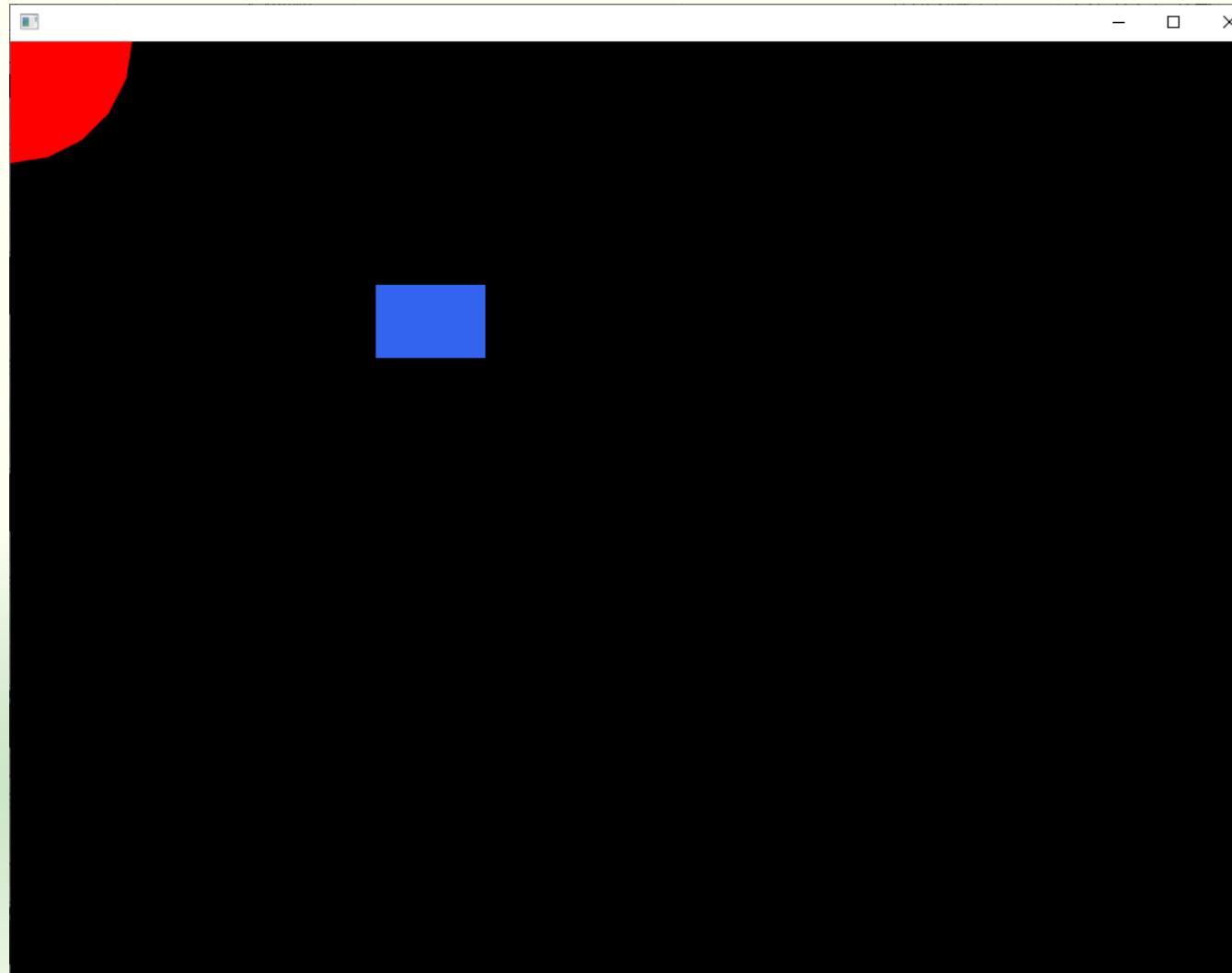
}

//-----
void ofApp::draw(){
    ofSetColor(255, 0, 0);
    ofDrawCircle(0.0f, 0.0f, 100.0f);
    ofSetColor(50, 100, 240);
    ofDrawRectangle(300.0f, 200.0f, 90.0f, 60.0f);
}
```

- `ofSetColor()` で指定した色は `ofSetColor()` の呼び出し以降に描画する図形に適用される



# 矩形の色が変わる





# 課題 1 – 7

自分で図形を作ってみよう

# 二次元図形を描く

- 以下の openFrameworks の関数を使って何か二次元の図形を描いてください

- ofDrawCircle()
  - [https://openframeworks.cc/documentation/graphics/ofGraphics/#!show\\_ofDrawCircle](https://openframeworks.cc/documentation/graphics/ofGraphics/#!show_ofDrawCircle)
- ofDrawCurve()
  - [https://openframeworks.cc/documentation/graphics/ofGraphics/#!show\\_ofDrawCurve](https://openframeworks.cc/documentation/graphics/ofGraphics/#!show_ofDrawCurve)
- ofDrawEllipse()
  - [https://openframeworks.cc/documentation/graphics/ofGraphics/#show\\_ofDrawEllipse](https://openframeworks.cc/documentation/graphics/ofGraphics/#show_ofDrawEllipse)
- ofDrawLine()
  - [https://openframeworks.cc/documentation/graphics/ofGraphics/#show\\_ofDrawLine](https://openframeworks.cc/documentation/graphics/ofGraphics/#show_ofDrawLine)
- ofDrawRectRounded()
  - [https://openframeworks.cc/documentation/graphics/ofGraphics/#show\\_ofDrawRectRounded](https://openframeworks.cc/documentation/graphics/ofGraphics/#show_ofDrawRectRounded)
- ofDrawRectangle()
  - [https://openframeworks.cc/documentation/graphics/ofGraphics/#show\\_ofDrawRectangle](https://openframeworks.cc/documentation/graphics/ofGraphics/#show_ofDrawRectangle)
- ofDrawTriangle()
  - [https://openframeworks.cc/documentation/graphics/ofGraphics/#show\\_ofDrawTriangle](https://openframeworks.cc/documentation/graphics/ofGraphics/#show_ofDrawTriangle)

# 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 **1-7.png** というファイル名で保存し、Moodle の第 1 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 1 回課題にアップロードしてください



# ofApp.cpp の保存場所

