

高エネルギー宇宙物理学 のための ROOT 入門

－ 第 4 回 －

奥村 暁

名古屋大学 宇宙地球環境研究所

2016 年 6 月 1 日

最新版に git pull してください

A terminal window with a dark gray background and a light gray title bar. The title bar has three circular window control buttons on the left and a maximize button on the right. The terminal displays two lines of text: "\$ cd RHEA" and "\$ git pull".

```
$ cd RHEA
$ git pull
```

補足 (1)

```
$ root
root [0] .x script.C
root [1] .x script.C

root [2] .q
$ root
root [0] .x script.C
```

- ① 実行後に script.C の中身を変更
- ② ROOT 5 だと新しい script.C が実行されるが、ROOT 6 だと古いものが再実行される
- ③ 面倒ですが、一度終了して再実行してください

- ❖ ROOT 6 には厄介な既知のバグがあり、script を修正したものを新規実行し直せない
- ❖ PyROOT では発生しないのでそっちを使うか
- ❖ ROOT を終了して再実行

補足 (2)

- 宇宙地球環境研究所の参加学生にいくつか統計の質問をしたところ、第2回、3回の内容がちゃんと頭に残っていないようなので、よく復習してください
- 本当は練習問題を出したほうが良いのですが、そこまで手が回っていません

C++ と ROOT と Python

- コンパイルという作業が必要→コンパイラ型言語
- Python に比べると色々面倒くさい
 - ▶ 使う側も面倒くさい（機能が少ない、書く量が多いなど）
 - ▶ 教える側も面倒くさい（メモリの処理、ポインタなど）
- 「簡単なデータ解析しかしません」「修士で就職します」の場合、C++ を学ぶ必要性は近年は低い
- C/C++ を学んだほうがよい学生
 - ▶ ハードウェア制御を実行速度重視で行う
 - ▶ ROOT や Geant4 をガリガリ使う
 - ▶ ソフトウェアの開発側に回る（ユーザに終わらない）

Python

- スクリプト型言語、コンパイルの必要がない
- テキスト処理などを初め、C++ より豊富な機能を標準で備える
 - ▶ 自分で色々と機能を実装する必要がない
 - ▶ 間違いが混入しにくく、ソフト開発も素早くできる
- C/C++ より実行速度が遅い場合が多い
 - ▶ ボトルネックの箇所だけ C/C++ で書いたりすることもある
 - ▶ Python の標準ライブラリなどに含まれる機能のほうが自作 C/C++ プログラムより最適化されていて早い場合もある
- 理解が簡単、教えるのも簡単
- 修士で卒業する、データ解析しかしないなら Python だけでも生きていける

基本的な流れ

```
$ g++ hello_world.cxx  
$ ./a.out  
Hello World!
```

- ① コンパイラでコンパイルし、実行ファイルを生成する
- ② 実行ファイルを実行する

```
$ g++ hello_world.cxx -O2  
$ ./a.out  
Hello World!
```

- ③ 最適化オプションをつける
※単純なプログラムだと変化ないが、一般的には速度が向上

```
$ g++ hello_world.cxx -O2 -o hello_world  
$ ./hello_world  
Hello World!
```

- ④ a.out はダサいので、実行ファイル名を変更

```
$ clang++ hello_world.cxx -O2 -o hello_world  
$ ./hello_world  
Hello World!
```

- ⑤ OS X だと Clang を使用する
※g++ と打っても同じコンパイラが走る

コンパイラとは

- 人間の読めるコードを計算機を読める形式（機械語）に変換する
- コンパイルしないと動かない
 - ▶ ただし ROOT は特殊で、コンパイルしていない C++ を実行することができる（後述）
- Linux では GNU Compiler Collection (GCC)、OS X では Clang 使用するのが一般的
- 実際の大規模なソフトウェアでは多数のオプション指定が必要
- CMake や autotools で自動化が可能

C/C++ の基本

```
$ cat hello_world.cxx
```

```
#include <cstdio>
```

```
int main() {  
    printf("Hello World!\n");  
  
    return 0;  
}
```

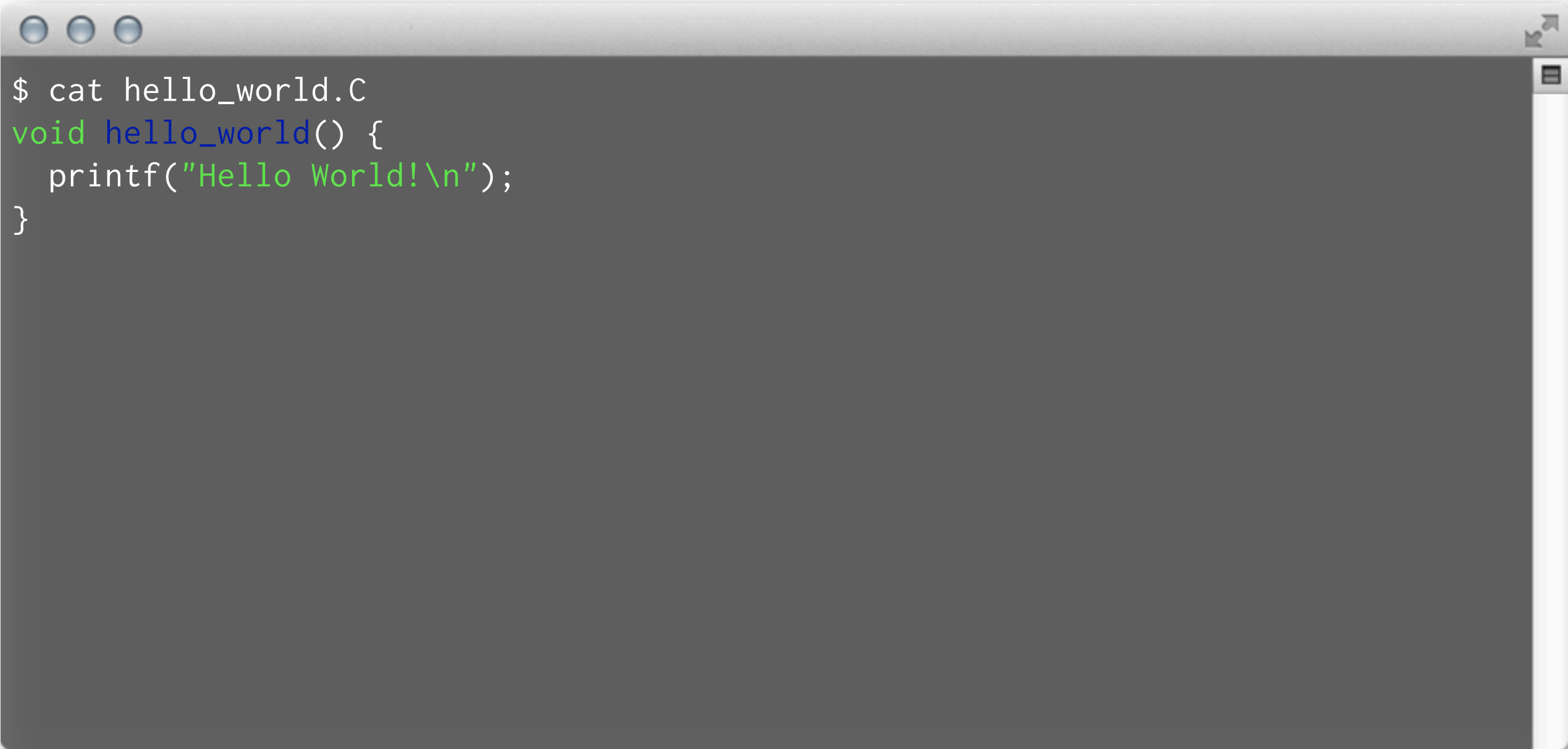
❶ 非常に初歩的なことをする場合以外は、高度な機能を使うためにヘッダーファイルを #include する

❷ 必ず main 関数が実行される。他の関数は全て main 関数から呼び出される。

❸ main 関数は int の返り値が必要。ここでエラーコードを返して main を抜ける。0 は正常終了の意味。

- ROOT の場合は特殊で、main 関数は ROOT 自体が既に実行している
- ROOT5 の場合は CINT が、ROOT6 は Cling がスクリプトを呼び出すため、スクリプト内に main は不要
- ROOT を使わない純粋な C++ の場合、コンパイルしないと実行できない

ROOT スクリプトの場合

A terminal window with a dark gray background and a light gray title bar. The title bar has three circular window control buttons on the left and a maximize button on the right. The terminal displays the following C code:

```
$ cat hello_world.C
void hello_world() {
    printf("Hello World!\n");
}
```

```
$ cat hello_world.C
void hello_world() {
    printf("Hello World!\n");
}
```

- ❖ main 関数は定義する必要なし
- ❖ 多くの標準的なヘッダーファイルも #include する必要なし

Python の場合

```
$ cat hello_world.py
#!/usr/bin/env python

def hello_world():
    print("Hello World!")

if __name__ == "__main__":
    hello_world()

$ python hello_world.py
Hello World!
$ ./hello_world.py
Hello World!
$ python
>>> import hello_world
>>> hello_world.hello_world()
Hello World!
```

① スクリプト内部で Python を呼び出すときに必要

② 関数の定義の仕方、def を使う

③ 1 つ目、2 つ目のやり方で必要

④ python コマンドにスクリプトを食わせる

⑤ 実行ファイルとして使う

⑥ module として使う

■ 書き方と実行方法は何通りがある

- ▶ スクリプトを python コマンドに実行させる
- ▶ スクリプト自体を実行し内部で python コマンドを走らせる
- ▶ module として使う方法 (import する)

ROOT スクリプトで main を再定義すると

```
$ cat main.C
int main() {
    return 0;
}
$ root
root [0] .x main.C
Error in <TApplication::TApplication>: only one instance of TApplication allowed
-----
| Welcome to ROOT 6.06/04                http://root.cern.ch |
|                                     (c) 1995-2016, The ROOT Team |
| Built for macosx64                    |
| From tag v6-06-04, 3 May 2016          |
| Try '.help', '.demo', '.license', '.credits', '.quit'/'.'q' |
|                                     |
-----

/Users/oxon/.rootlogon.C:38:7: error: redefinition of 'fontid'
Int_t fontid=132;
    ^
/Users/oxon/.rootlogon.C:38:7: note: previous definition is here
Int_t fontid=132;
    ^
```

- 新しく作られた main ではなく、ROOT が新たに走り出す
- main は特殊な関数なので、ROOT スクリプト内では使わないこと
- ~/.rootlogon.C が 2 回呼び出されてエラーを吐いている

もう少し ROOT っぽい例 (C++)

```
$ cat first_script2.C
void first_script2(int nbins, int nevents) {
    TH1D* hist =
        new TH1D("myhist", "Gaussian Histogram (#sigma = 1)", nbins, -5, 5);
    hist->FillRandom("gaus", nevents);
    hist->Draw();
}
$ root
root [0] .x first_script2.C(500, 100000)
root [1] myhist->GetName()
(const char *) "myhist"
root [2] gROOT->Get("myhist") ① 普通の C++ の教科書的には、TH1D hist("myhist" ...) とする
(TObject *) 0x7fe79d0572d0
root [3] myhist ② ROOT が名前でオブジェクトの管理をしている
(TH1D *) 0x7fe79d0572d0
root [4] delete gROOT->Get("myhist") ③ ROOT では名前を使ってオブジェクトのアドレスを取り出せる
root [5] gROOT->Get("myhist") ④ delete でオブジェクトをメモリ上から消すと、
(TObject *) nullptr ROOT の管理からも外れる
```

- C++ にはスコープ (scope) という概念が存在する
- {} や関数を抜けると、その変数は消えてしまう
- new してオブジェクトのアドレスをポインタ変数として扱うと、delete が呼ばれるまでオブジェクトがメモリ上から消えない (変数 TH1D* hist は消える)
- ROOT が "myhist" という名前のオブジェクトを記憶しているので、後から参照できる

new を使わないと

```
$ cat first_script2_wo_new.C
void first_script2_wo_new(int nbins, int nevents) { ❶ ポインタでない変数にする
    TH1D hist("myhist", "Gaussian Histogram (#sigma = 1)", nbins, -5, 5);
    hist.FillRandom("gaus", nevents);
    hist.Draw(); ❷ メンバ関数の呼び出しは -> ではなく . を使う
}
$ root
root [0] .x first_script2_wo_new.C(500, 100000) ❸ TCanvas に何も表示されない
root [1] myhist->GetName()
input_line_79:2:3: error: use of undeclared identifier 'myhist'
(myhist->GetName()) ❹ オブジェクトが消えているので、ROOT も既に管理していない
^
```

- ❖ この書きかたは教科書的な C++ では普通
- ❖ ROOT の場合、生成したオブジェクトをスクリプト終了後にも引き続き描画させ解析したい
- ❖ ポインタを使わないとこれができない

Python の場合

```
$ cat first_script2.py
import ROOT

def first_script2(nbins, nevents):
    global hist
    hist = ROOT.TH1D('myhist', 'Gaussian Histogram (#sigma = 1)', nbins, -5, 5)
    hist.FillRandom('gaus', nevents)
    hist.Draw()

$ python
>>> import first_script2
>>> first_script2.first_script2(500, 100000)
>>> first_script2.hist.GetName()
'myhist'
>>> import ROOT
>>> ROOT.myhist.GetName()
'myhist'
```

- Python も同様に、関数を抜けるとその変数は消えてしまう
- C++ の delete の相当する機能も働くため、オブジェクト自体も消える
- これを防ぐには global 変数を使う

なぜ ROOT はスクリプト型言語のように動くのか

- ROOT 5 では CINT (シーイント) という C/C++ のインタプリタ (コンパイルしないで機械語に逐次変換する) が使われており、C/C++ を (ほぼ) 実行できる
- ROOT 6 では Clang を使用した Cling というインタプリタが使われるようになった
 - ▶ より C/C++ の文法に則っている
 - ▶ 実行速度の向上
 - ▶ エラーが分かりやすい、読みやすい

型 (Type)

- C/C++ には型がある
 - ▶ 符号あり整数型 : char (8 bit)、short (16)、int (32 or 64)
 - ▶ 符号なし整数型 : unsigned char など
 - ▶ 浮動小数点型 : float (32 bit)、double (64)
 - ▶ 32 bit OS か 64 bit かで int の大きさが違う
- ROOT では環境依存をなくすため、Short_t や Long_t などが定義されている (C の教科書で見たことのない型が ROOT の例で出てくるのはこのため)
- C++11 (新しい規格の C++) では、このような混乱をなくすために int8_t (8 bit) などが追加された

クラス (Class)

- 色々な変数や機能をひとまとまりにした、型の「ような」もの
- 好きなものを自分で追加できる。型は追加できない。
- TGraph や TH1D は ROOT が持つクラス
 - ▶ 内部にデータ点やビン幅などの数値情報
 - ▶ 名前、タイトルなどの文字情報
 - ▶ Draw() や GetStdDev() などのメンバ関数

C++ のクラスの例

```
double x1 = 1.5, y1 = 2.3, z1 = -0.4;  
double x2 = -3.1, y2 = 5.6, z2 = 1.9;  
double x3 = x1 + x2, y3 = y1 + y2, z3 = z1 + z2
```

① 型だけでやると見づらく煩雑

```
Vector3D v1(1.5, 2.3, -0.4);  
Vector3D v2(-3.1, 5.6, 1.9);  
Vector3D v3 = v1 + v2;
```

② クラスにすることでより直感的に

- 情報をクラスにまとめることで扱いやすくなる
- 数値データに限らず、なんでもクラスにできる

簡単なクラスの例 (Vector3D.h)

```
#ifndef VECTOR_3D
#define VECTOR_3D

class Vector3D {
private:
    double fX;
    double fY;
    double fZ;

public:
    Vector3D();
    Vector3D(double x, double y, double z);
    Vector3D(const Vector3D& other);
    virtual ~Vector3D();

    virtual double X() const { return fX; }
    virtual double Y() const { return fY; }
    inline virtual double Z() const;
    virtual void Print() const;
};
```

- ❖ 「宣言」はヘッダーファイルに、定義はソースファイルに書くのが一般的
- ❖ 拡張子はそれぞれ .h/.hpp/.hxx/.hh などか、.cc/.cpp/.cxx など

使用例 (Vector3D_main.cxx)

```
#include <cstdio>
#include "Vector3D.h"

int main() {
    Vector3D v0;           // default constructor
    Vector3D v1(1.5, 2.3, -0.4); // constructor with arguments
    Vector3D v2 = Vector3D(-3.1, 5.6, 1.9); // operator=, constructor
    Vector3D v3 = v1 + v2;   // operator=, operator+
    Vector3D v4(v1 - v2);    // copy constructor, operator-
    double product = v1 * v2; // operator*

    v0.Print();
    v1.Print();
    v2.Print();
    v3.Print();
    v4.Print();
    printf("v1*v2 = %f\n", product);

    return 0;
}
```

- 自分で作ったヘッダーファイルを #include することで、新たな機能として使えるようになる

実行例

```
$ g++ -c Vector3D.cxx
$ g++ -c Vector3D_main.cxx
$ g++ Vector3D.o Vector3D_main.o -o Vector3D
$ ./Vector3D
(x, y, z) = (0.000000, 0.000000, 0.000000)
(x, y, z) = (1.500000, 2.300000, -0.400000)
(x, y, z) = (-3.100000, 5.600000, 1.900000)
(x, y, z) = (-1.600000, 7.900000, 1.500000)
(x, y, z) = (4.600000, -3.300000, -2.300000)
v1*v2 = 7.470000
```

- 各ファイルを順次コンパイルし、オブジェクトファイル (.o) を生成する
- 最後にオブジェクトファイルを結合し、実行ファイルを作る

Python のクラスの例 (vector.py)

```
class Vector3D(object):
    def __init__(self, x = 0., y = 0., z = 0.):
        self.x = x
        self.y = y
        self.z = z

    def __str__(self):
        return "(x, y, z) = (%f, %f, %f)" % (self.x, self.y, self.z)

    def __add__(self, other):
        return Vector3D(self.x + other.x, self.y + other.y, self.z + other.z)

    def __sub__(self, other):
        return Vector3D(self.x - other.x, self.y - other.y, self.z - other.z)

    def __mul__(self, other):
        return self.x*other.x + self.y*other.y + self.z*other.z
```

- C++ とは書き方がかなり違うので、よく見比べてください

実行例

```
$ python vector.py
(x, y, z) = (0.000000, 0.000000, 0.000000)
(x, y, z) = (1.500000, 2.300000, -0.400000)
(x, y, z) = (-3.100000, 5.600000, 1.900000)
(x, y, z) = (-1.600000, 7.900000, 1.500000)
(x, y, z) = (4.600000, -3.300000, -2.300000)
v1*v2 = 7.470000
```

第 4 回のまとめ

- C++ と Python の違い
 - ROOT でのいくつかの注意点
 - クラス
-
- 分からなかった箇所は、各自おさらいしてください