

プログラミング言語論

第7回

C++における関数型プログラミング
ラムダ式と高階関数、const修飾子

□手続き型

- 変数に代入して値をメモリに覚えさせて変化しながら処理を進めていく方法

□関数型

- まさに数学の関数として計算していく方法で、原則入力を入れたら出力が絶対に決まる方法

- グローバル変数が悪なら、代入を禁止してしまえばいいじゃない

「パンがなければケーキを食べればいいじゃない」



□関数型プログラミングの特徴

□参照透過性：副作用を持つ関数を使えない

- ・戻り値以外に変化が何もないことを保証すること

□不変性：変数への代入の禁止

□関数をオブジェクトとして扱う（高階関数）

□遅延評価

□ループはない→再帰関数の利用

□最近，並行処理（マルチスレッドなど）を含むプログラムを安全確実に開発するための技術として人気上昇中

□現代的なプログラミング言語は，「関数型プログラミング言語」の要素技術は広く取り入れられている．

- higher-order function
- 「関数」を引数として持ったり, また「返り値」
として関数を返したりできる関数のこと
- 汎用性と柔軟性が高い関数を書くために役立つ
- 例 : `std::sort(p.begin(), p.end(), cmp)`
 - `cmp`は比較関数
 - この例の`cmp`は, 関数オブジェクト, またはラムダ式
 - ・ラムダ式 (次ページ) は実は関数オブジェクトです

- ラムダ式は, 「名前のない関数オブジェクト」
- ラムダ式の例 (C++)

```
[](int v) { return v * 2; }
```

環境

引数

定義本体

キャプチャ リスト

環境キャプチャはラムダ式が定義されたスコープの中の変数で定義本体で使いたいものを列挙したり、`=`、`&`の指示をします

operator()メソッドが定義されているクラス=関数オブジェクト
別名：ファンクタ(functor)

```
class FunctionObject
{
public:
void operator () ()
{
    std::cout<<"Hello world"<<std::endl;
}
};
```

```
int main(int argc, char const* argv[]) {
    FunctionObject func;
    func();
}
```

状態を保持しない場合，ラムダ式が使われることが多く，
あまり上の形で利用されることは少ないかも。

```
#include <iostream>
```

```
int main()
```

```
{
```

```
    auto f = [](int v) { return v * 2; };
```

```
    std::cout << f(2) << std::endl;
```

```
    std::cout << [](int v) { return v * 2; }(2) << std::endl;
```

```
}
```

ラムダ式の型はややこしいので
autoを使う

関数無くても

関数呼び出しができる！

たまに便利

注意：今日の例では、コンパイルオプション「-std=c++11」
を忘れないように。ラムダ式はC++11で採用

- この問題は必須ではないですが、良い練習になります（というか実はここまでやっておかないとstd::sortが使いこなせたことにならない）。
- 名前と年齢をレコードとして持つPersonクラスを定義し、年齢をキーとしてstd::sortでソートするプログラムを作成せよ。
 - ヒント：std::sort(p.begin(), p.end(), cmp)
cmpは比較関数（関数オブジェクトまたはラムダ式）
 - STLではこのように関数を関数に渡すことがよくある

- 年齢をキーとしてソートしたい

- つぎのラムダ式を準備する

```
auto cmp = [](Person x, Person y) {
```

```
    return x.get_age() < y.get_age();
```

```
};
```

条件式なので真理値型(bool)

の値(true, false)が返ることに注意

ラムダ式の表す関数の返り値の型を指定する書き方もあるが、
本体がreturn文だけのときは省略できるので、ここでは省略

- 省略せずにラムダ関数の戻り値型を明示する書き方は次の通り：

```
[](int v)->int { return v * 2; }
```

```
int main()
{
    std::vector<Person> p(3);
    p[0].set_name("Ichiro");  p[0].set_age(25);
    p[1].set_name("Jiro");    p[1].set_age(21);
    p[2].set_name("Saburo");  p[2].set_age(32);
    std::sort(p.begin(), p.end(), cmp); ← 第3引数として比較関数を渡す
    for (auto x: p) x.print();
}
```

- std::sort関数は**高階関数**の一例（引数として「関数」を受け取る
- 関数を渡すメリット
 - テンプレートにより要素の型（Personなど）は任意にしておける
 - ソートアルゴリズムでは「要素の比較」が必須．ただし，要素の型により比較の方法は異なる．その部分は，「比較関数」をユーザから受け取る形で実装
 - 非常に柔軟性の高いソート関数を書くことができ，この例の場合，比較関数を変更することで、「名前順」のソートも容易で同率時2番目をキーにしたソートも可能

▣ 直接ラムダ式をstd::sortに渡す書き方もよくします.

```
std::sort(p.begin(), p.end(), [](Person x, Person y) {  
    return x.get_age() < y.get_age();});
```

コレをやりすぎるとわかりにくいプログラムになるので、ご利用はほどほどに.

ただ、std::sortの場合はラムダ式で書く人が多いと思います.

```
#include <iostream>
#include <vector>
#include <algorithm>
int main()
{
    std::vector<int> v = {1, 2, 3, 4};
    std::for_each(v.begin(), v.end(), [](int i) {std::cout << i;});
    std::cout << std::endl;
}
```

全要素に対してこの処理をしますというのが非常に明確で最適化がかかりやすいので高速に動く可能性が高い

```
#include <iostream>
#include <functional>

std::function<int(int)> f(int n) {
    return [=](int v) { return v * n; };
}

int main() {
    auto f2 = f(2); auto f3 = f(3);
    std::cout << f2(2) << std::endl;
    std::cout << f3(2) << std::endl;
}
```

□ 返り値としてラムダ式を返す関数も書くことができる

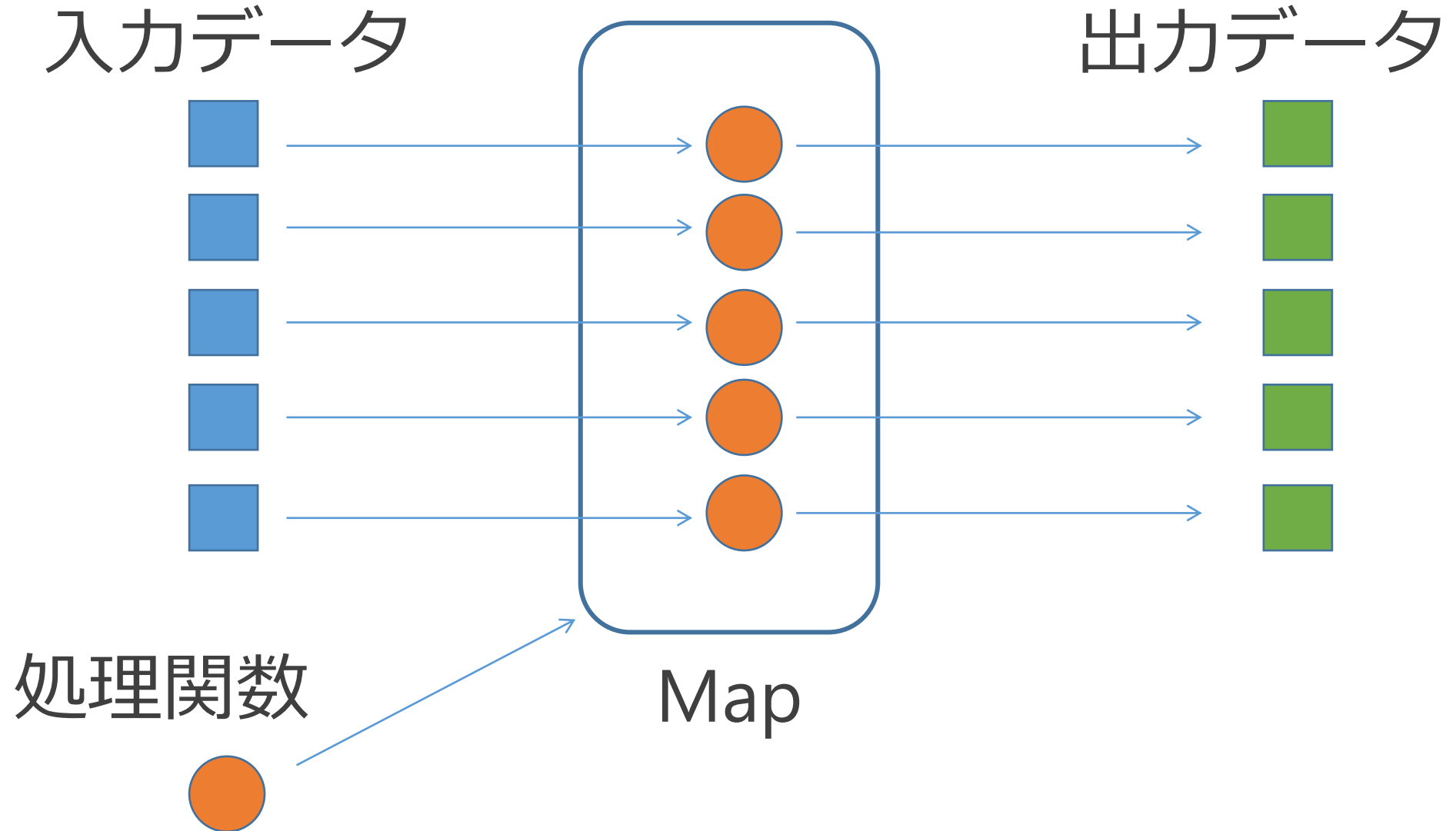
高階関数「関数」を引数として持ったり、また「返り値」として関数を返したりできる関数のこと

4, 6が出力される

- 自分でstd::sort のように関数を引数として受け取る関数を書くためには、std::function を利用する
- 引数として次の形で書く

```
std::function<int(int)> f
```

受け取る関数fの型が「返り値int, 引数int
一個」であることを表す



Map関数を実装してみる

```
#include <iostream>
```

```
#include <vector>
```

```
#include <functional>
```

参照渡し：後述



```
void my_map(std::vector<int> & v, std::function<int(int)> f) {
```

```
    for (auto &x : v) x = f(x);
```

```
}
```

```
int main() {
```

```
    std::vector<int> vec = {1, 2, 3, 4};
```

```
    my_map(vec, [](int i){return i*i;});
```

```
    for (auto x: vec) std::cout << x << std::endl;
```

```
}
```

1, 4, 9, 16

が表示される

```
void swap(int &a, int &b) {  
    int tmp = a;  
    a = b;  
    b = tmp;  
}  
  
int main() {  
    int a = 10;  
    int b = 20;  
    swap (a, b);  
}
```

- C++はコピー渡し（値渡し）、ポインタ渡しに加えて、**参照渡し**が使える
 - 参照渡しはポインタ渡しとほぼ同義（ポインタ渡しのほうはもっと無茶なことができるが）
 - コピー渡しは名前の通り
 - クラスのインスタンスを関数に渡すときは参照渡しを使うことが多い。（コピーの手間を避けたい）
 - 基本的にコピー渡しは、intやdoubleなどの基本型に限られると理解しておけばよい
 - 関数内で値を触らない場合は、constをつける（後述）

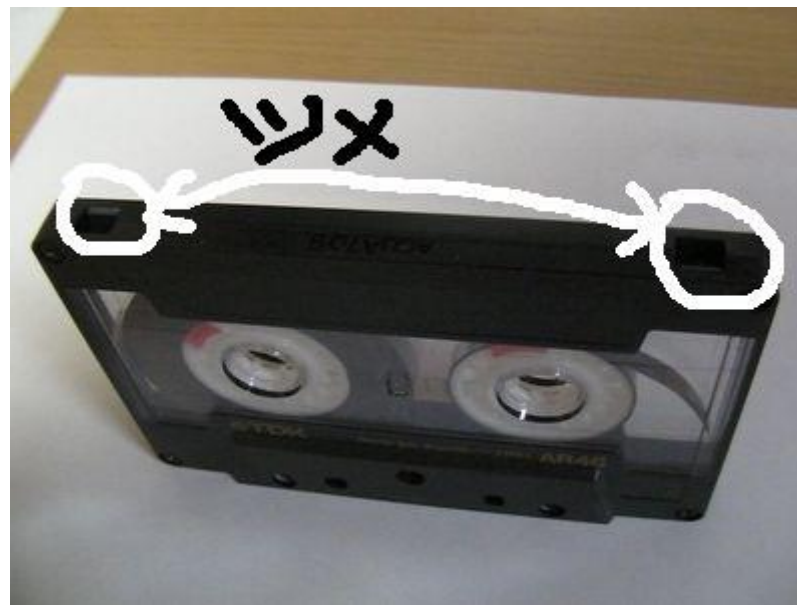
- 不変性 (immutable) : 変数は定義可能で、代入は不可能
- 参照透過性 (Referential transparency) : 副作用を持つ関数 (状態の変化を伴う関数, 例えば前ページのswap) はではないこと.
- 参照透過な関数は, 入力が同じなら必ず同じ答えを返す
- これらを完全には, C++では実現できない (マルチパラダイムと言ってもC++は基本的にはオブジェクト指向型言語)。
- **const 修飾子**を適切に利用することで、不変でない部分, 参照透過でない部分を明示的に**限定する**ことはできる (使い方は簡単)

- 変数の宣言にconstをつけることで、その変数が変更できなくなる

```
const double pi = 3.14159;
```

メリット：誤って定数を書き換えてしまうバグを防げる

デメリット：ちょっとプログラムを試したくて値を代入しなおすことがめんどくさい（この行為自体がバグの温床）



まだビデオ使ったことあるよね？ . . .

- 関数の引数に `const` を指定すると、その関数の中では値を書き換えることができなくなる
- 参照渡ししの引数につけることが多い
- 習慣にするのがよい
- メリット：意図しない副作用を防ぐ(見つけにくいバグを抑制)

```
void print(const std::vector<int> & v) {  
    for (auto x: v) std::cout<<x;  
}
```

この関数内ではvの中身を変更しないので、`const`で引数を受け取るのがよい

- メンバ関数の宣言の末尾に `const` 修飾子をつけることで、そのメンバ関数を呼び出したときにオブジェクトが変化しない（読み取り専用関数である）ことを宣言できる
- すなわち`const`メンバ関数はオブジェクトの状態を変えないことが保証されている（参照透過性の保証）
- `const`宣言されたオブジェクトには、`const`メンバ関数だけがアクセスできる
- `const`メンバ関数から非`const`メンバ関数を呼び出すことはできない（コンパイラに怒られる）

```
class A
{
public:
    int val;
    int read() const {
        return val;
    }
};
```

状態が変わらない = 参照透過

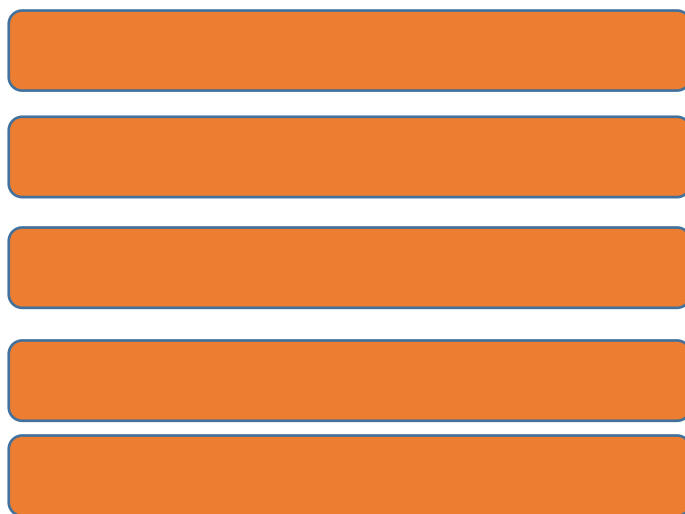
メンバ変数の値を変更しない
メソッドはconstをつけることを
習慣にしよう

Class A

constでないメンバ関数



constなメンバ関数



問題発生：クラスAの
オブジェクトの
状態がオカシイぞ？

ここだけを調べれば
よい

- オブジェクト指向のデメリット：オブジェクトが状態を持ち、そんなオブジェクトがたくさんある→システムの状態の複雑化→プログラミングの困難化
- 参照透過でない部分をなるべく局所化する
 - 状態変数の局所化
 - 関数型言語の利用→強制的な参照透過性の確保（関数型言語の第一のメリットは書かれた関数がすべて副作用を持たないこと）
 - ループではなく、map, reduce を使う
 - C++などのオブジェクト指向言語では、constを利用する、プログラム構造を工夫するようにして、参照透過でない部分をまとめる

- C++には、現代的なプログラミング技術と考え方が詰まっています。（しかも、まだまだ進化中）
- あと調べてみるとよいと思われるおすすめ事項（ネットで例を見ていって写経するだけでも当面はOK!）
- 知らなくてもプログラムは書けるが知っていると少し幸せになれるし、自分のプログラミングスキルも伸びる！
 - スマートポインタ：メモリ管理がラクラク。
 - 静的アサーション：防衛的プログラミング
 - 例外処理
 - コンパイル時計算（c++14）：constexpr
 - 数学ライブラリ
 - STL, boostのライブラリ探検
 - 並行処理：asyncなど
 - C++におけるユニットテスト

速いプログラミング
がしたいなら**C++**！

- ラムダ式（無名関数）について，C++以外の言語でどのような言語で使えるか調べその言語名をかけ．（使い方とかはいらない）

- (1) 2つの整数を引数としてとり、その和を返すラムダ式を定義し、そのテストコードを書け.
- (2) $f(\text{int } n)$ はラムダ式を返す関数である。返ってくるラムダ式は"A"をn回表示するものである。関数fを実装し、そのテストコードを書け.

- `std::for_each`で何かプログラムを書いてみよ。ただし、ラムダ式を使う場合と使わない場合の両方を試すこと。
- `std::find_if`で何かプログラムを書いてみよ。ただし、ラムダ式を使う場合と使わない場合の両方を試すこと。

□ `int n` と関数 `double f(double a)` を引数として取り、
関数 `f(...f(f(f(.))))...`

(ただし `f` の適用は `n` 回) を返す関数とテストコードを書け.