

C++プログラミングII

第11回 関数オブジェクトとラムダ式

岡本秀輔

成蹊大学理工学部

共通の関数

- ▶ 前回と同じく共通に使用するヘッダファイル

```
// print.hpp
#include <iostream>
template<typename T>
void print(const T& c)
{
    for (auto& e: c)
        std::cout << e << " ";
    std::cout << "\n";
}

template<typename Itr>
void print(Itr b, Itr e)
{
    for ( ; b != e; ++b)
        std::cout << *b << " ";
    std::cout << "\n";
}
```

関数オブジェクト

関数オブジェクト

- ▶ operator() をメンバ関数として持ち、関数の様に振る舞うオブジェクトを関数オブジェクトと呼ぶ
- ▶ 状態によって結果の変わる関数を表現できる

```
class Functor {  
    int n;  
public:  
    Functor(int a):n{a}{}  
    int operator()(int a) { return a*n; }  
};  
  
int main() {  
    Functor x{3}; // 変数だけど  
    std::cout << x(2) << "\n"; // 6 関数として使える  
    Functor y{4};  
    std::cout << y(2) << "\n"; // 8  
    Functor z{5};  
    std::cout << z(2) << "\n"; // 10  
}
```

関数オブジェクトの利用

- ▶ Even{5}や Even{3}はオブジェクトの右辺値となる

```
class EvenMore {
    int m; // m より大きい偶数
public:
    EvenMore(int x):m{x}{}
    bool operator()(int x) {
        return (x % 2 == 0) && (x > m);
    }
};

int main() {
    std::vector a {1,4,3,2,5,4,7,8,9,6};
    auto it{a.begin()}, b{a.begin()}, e{a.end()};
    it = std::find_if(b, e, EvenMore{5}); // 8
    if (it != a.end()) std::cout << *it << "\n";
    it = std::find_if(b, e, EvenMore{3}); // 4
    if (it != a.end()) std::cout << *it << "\n";
}
```

関数オブジェクトの利用2

- ▶ 初期値を決めて、状態を更新している
- ▶ 右辺値なので何度も使いまわせる

```
#include <algorithm>
#include <vector>
#include "print.hpp"
class F {
    int n;
public:
    F(int a =0):n{a}{}
    int operator()() { ++n; return n*n; }
};

int main() {
    std::vector a(5, 0); // 5要素 int
    std::generate(a.begin(), a.end(), F{0});
    print(a); // 1 4 9 16 25
    std::generate(a.begin(), a.end(), F{2});
    print(a); // 9 16 25 36 49
}
```

標準ライブラリ中の 関数オブジェクト

比較用のライブラリ関数

- ▶ `<functional>`ヘッダファイル
- ▶ 比較演算子を関数オブジェクトにして使う
 - ▶ `bool func(const T& a, const T& b)` の形式
- ▶ 用意されたクラス
 - ▶ `equal_to, not_equal_to,`
`less, less_equal,`
`greater, greater_equal,`

```
#include <functional>
#include <iostream>
int main() {
    auto comp { std::less<int>{} };
    int x{10}, y{20};
    if (comp(x, y))
        std::cout << "ok\n";
}
```


std::sort() の利用

▶ 関数オブジェクトで昇順/降順を決める

```
#include <functional> // greater<>
#include <algorithm>
#include <vector>
#include "print.hpp"

int main() {
    std::vector x {5,3,2,4,6,1,7};
    auto b{x.begin()}, e{x.end()};
    std::sort(b, e); // less<int>{} がデフォルト, 昇順
    print(x); // 1 2 3 4 5 6 7

    std::sort(b, e, std::greater<int>{}); // 降順
    print(x); // 7 6 5 4 3 2 1
}
```

四則演算クラス

- ▶ `<functional>` ヘッダファイル
- ▶ 加算／乗算などのクラスライブラリの使用
 - ▶ `plus`, `minus`, `multiplies`, `divides`, `modulus` など
- ▶ 一見すると使い道が分からない

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

int main() {
    auto op { std::plus<int>{} };
    int x{10}, y{20}, z{0};
    z = op(x, y);
    std::cout << z << "\n";
}
```

accumulate 操作

- ▶ accumulate: 蓄積させる, ためる
- ▶ 値の集約: 複数の値をまとめて一つにする操作
- ▶ 単純には合計を取れば良いがバリエーションもある
 - ▶ 加算ではなく乗算したいなど

```
template<typename T, typename F>
T accumulate(const std::vector<T>& v, T ini, F func)
{
    for (auto& e : v)
        ini = func(ini, e); // 演算結果を ini に蓄積
    return ini;
}

int main() {
    std::vector v {1,2,3,4,5,6,7,8,9,10};
    std::cout
        << accumulate(v, 0, std::plus<int>{}) <<"\n" // 55
        << accumulate(v, 1, std::multiplies<int>{})
        <<"\n"; // 3628800
}
```

std::accumulate

- ▶ accumulate は STL アルゴリズムの関数でもある

```
#include <functional>
#include <numeric>
#include <iostream>
#include <vector>

int main() {
    std::vector v {1,2,3,4,5,6,7,8,9,10};
    auto b{v.begin()}, e{v.end()};
    std::cout
        << std::accumulate(b, e, 0) <<"\n" // 55
        << std::accumulate(b, e, 1,
            std::multiplies<int>{}) <<"\n"; // 3628800
}
```

ラムダ式の基本

背景

- ▶ 関数テンプレートとクラスの組み合わせ
 - ▶ 組み込み型とクラスを同じアルゴリズムで扱える
 - ▶ 使用する演算子が `<` などに固定
- ▶ 関数テンプレートと関数の引数の組み合わせ
 - ▶ 関数テンプレートをカスタマイズできる
 - ▶ 引数にしたい関数は単純な場合多い
- ▶ 関数テンプレートと関数オブジェクトの組み合わせ
 - ▶ より高い自由度で処理が指定できる
 - ▶ `operator()` の処理は単純な場合多い
- ▶ 関数定義や関数オブジェクトのクラス定義が、実際にそれらを使う場所から離れすぎる
- ▶ 一時的に使う関数定義ができないか？

ラムダ式

- ▶ 無名の関数オブジェクトを作るための式
- ▶ 式なので右辺値として使う
 - ▶ 変数に代入できる
 - ▶ 実引数に指定できる
 - ▶ 無名の関数として使える
- ▶ 形式1(引数なし):
 - ▶ `[] { 処理 }`
 - ▶ `[] { std::cout<<"hello"; return 1; }`
- ▶ 形式2(引数あり):
 - ▶ `[] (引数リスト) { 処理 }`
 - ▶ `[] (int x, int y) { return x < y; }`
- ▶ 戻り値の型は return 文から推測される
 - ▶ void 関数でも良い
 - ▶ 戻り値の型を明示する方法もある

基本的な例

- ▶ ラムダ式用の変数の型は複雑なので `auto` を指定する

```
// 関数を作って変数として保存
```

```
auto f1 { []{ cout <<"hello lambda\n"; } };
```

```
f1(); // 通常関数と同じように呼び出し
```

```
// 作ってその場で呼び出すこともできる
```

```
[] { cout <<"direct call\n"; }();
```

```
// 引数と戻り値を持つ場合 (戻り値の型は推定される)
```

```
auto f2 { [](int x){ return x; } };
```

```
// 戻り値の型の明示指定 -> 戻り値の型
```

```
auto f3 { [](int x) -> double { return x; } };
```

```
cout << f2(9)/2 << ", " << f3(9)/2 << "\n";
```


ラムダ式と関数テンプレート

- ▶ ラムダ式の型はテンプレート引数にもできる

```
template<typename T>
void test(T func) {
    std::cout << "hello, ";
    func();
}

int main() {
    // ラムダ式を実引数にする,      hello, world
    test( []{ std::cout <<"world\n"; } );

    // ラムダ式の変数を実引数にする, hello, again
    auto x {[]{ std::cout<<"again\n"; }};
    test( x );
}
```

ラムダ式と auto 引数

- ▶ 引数の型を呼び出し時に決定できる
- ▶ 関数テンプレートと同じ使い方ができる

```
#include <iostream>
int main()
{
    auto f { [](auto x){ return x*x;} };
    std::cout << f(3)    <<"\n"    // 9
               << f(3.2) <<"\n"; // 10.24

    auto g { [](auto x, auto y){ return x+y;} };
    std::string a{"hello, "}, b{"world"};
    std::cout << g(3, 4) <<"\n"    // 7
               << g(a, b) <<"\n"; // hello, world
}
```

ラムダ式の利用

std::sort とラムダ式

- ▶ 自前の比較関数を簡単に作成できる
- ▶ 推論可能ならばラムダ式の引数の型は auto も可

```
#include <algorithm>
#include <vector>
#include "print.hpp"
int main() {
    using std::string;
    std::vector<string> x {"if", "while", "for", "do"};
    auto b{x.begin()}, e{x.end()};
    std::sort(b, e, // 文字比較の降順
        [](string a, string b){ return a > b; } );
    print(x); // while if for do

    std::sort(b, e, // 文字数の昇順
        [](auto a, auto b){ return a.size() < b.size(); } );
    print(x); // if do for while
}
```

std::accumulate

▶ 演算内容をラムダ式で指定する

```
#include <iostream>
#include <numeric>
#include <vector>
int main() {
    std::vector x {1,2,3,4};
    auto b{x.begin()}, e{x.end()};
    int n {x.back()};
    std::cout
        << std::accumulate(b, e, 0, // 二乗和
            [](int i, int e) { return i + e*e; })
        << " " << n*(n+1)*(2*n+1)/6 << "\n" // 30
        << std::accumulate(b, e, 0, // 三乗和
            [](int i, int e) { return i + e*e*e; } )
        << " " << (n*(n+1)/2)*(n*(n+1)/2) << "\n"; // 100
}
```

std::find_if 関数

- ▶ STL の `_if` の名前の関数すべてでラムダ式が利用できる

```
vector a {1,3,4,5,8,9,11,12,14};  
auto itr { std::find_if(a.begin(), a.end(),  
                        [](int x){ return x%2 == 0; }) };  
if (itr != a.end()) {  
    cout << *itr << "\n";    // 4  
}
```

ラムダ式を持つラムダ式

- ▶ ラムダ式を内部に持つラムダ式
- ▶ `find_if` などを自在に拡張して名前をつけられる

```
#include <algorithm>
#include <vector>
#include "print.hpp"
int main(){
    std::vector a {1,3,4,5,8,9,11,12,14};
    auto find_even { // 偶数を探して iterator を返す
        [](auto b, auto e) {
            return std::find_if(b, e,
                [](int x){ return x%2 == 0; });
        }
    };
    for (auto e{a.end()}, b{find_even(a.begin(),e)};
        b != e; b = find_even(std::next(b),e)) {
        std::cout << *b << " ";
    }
    std::cout << "\n"; // 4 8 12 14
}
```

キャプチャ

ラムダ式で利用できる変数

- ▶ ラムダ式は関数なので大域変数が利用可能
- ▶ 仮引数を含む局所変数も通常の関数と同じ扱い
 - ▶ `[] (int x) { int y {x+1}; return y; }`
 - ▶ `x` と `y` はラムダ式の局所変数
- ▶ それ以外の周辺の変数は指定なくアクセスできない

```
int global{10};

int main() {
    int x {0};
    auto f {
        [] (int a){
            int b { global+1 };    // ok
            std::cout << a+b;      // ok
            return x + 1;          // error
        }
    };
}
```

キャプチャとは capture: 捕獲, 捕らえること

- ▶ ラムダ式で周辺の局所変数を取り込む指定
- ▶ 二種類の取り込み指定
 - ▶ 値での取り込み
 - ▶ [=] は外側のスコープの局所変数をすべて値で利用
 - ▶ [x] や [id, x] は指定変数を値で利用
 - ▶ リファレンスでの取り込み
 - ▶ [&] は外側の局所変数をすべてリファレンスで利用
 - ▶ [&id] や [&id, &x] などの変数指定

```
int id{1}, x{3};  
auto f { [=] { return x; } };  
auto g { [id,x]{ return id+x; } };  
auto h { [&] { return x; } };  
auto i { [&id] { return id; } };
```

値取り込みの例

- ▶ [=]: すべての取り込み
- ▶ [id,x]: id と x を明示的に指定
- ▶ 値指定のラムダ式を指定した時点の値が使われる
- ▶ 取り込んだ変数は基本的に読み出し専用

```
int id{1}, x{3};  
auto f {      [=](int y){ return x+y;      } };  
auto g { [id,x](int z){ return id+x+z; } };  
id += 5; // 後で変更してもキャプチャには関係なし  
x  += 5;  
std::cout << f(1) << " " << g(1) << "\n"; // 4 5
```

リファレンス

- ▶ `id` と `x` は呼び出し時の状態が使われる

```
int id{1}, x{3};  
auto h {      [&](int a){ return a+id;   } };  
auto i { [&id,&x](int b){ return b+id+x; } };  
id += 5; x += 5;  
// 変更後の id, x が使われる (更新もできる)  
std::cout << h(1) << " " << i(1) << "\n"; // 7 15
```

キャプチャの注意点

- ▶ 値とリファレンスは混在可能 [id,&x]
- ▶ [&] は指定が楽だがバグを持ち込む元となる
- ▶ 組み合わせによってはエラーになる [&,&i]
 - ▶ &の指定に&i が含まれている
- ▶ ラムダ式の値は取り込んだ変数のスコープ外でも利用できてしまうので、リファレンスの際には変数が廃棄されていないかの確認が必要。

整列範囲用アルゴリズム

整列範囲用アルゴリズム

- ▶ `#include <algorithm>`
- ▶ 指定範囲は整列していること前提とする

名前	効果
<code>binary_search</code>	二分探索により要素を探す
<code>lower_bound</code>	指定値より小さくない最初の要素
<code>upper_bound</code>	指定値より大きい値の最初の要素
<code>equal_range</code>	指定値と同じ値を持つ要素の範囲
<code>merge</code>	2つの範囲の結合
<code>inplace_merge</code>	同一コンテナ上の2つの整列範囲の結合
<code>includes</code>	範囲 a の要素が範囲 b の要素を含むか
<code>set_union</code>	2つの整列範囲の和集合
<code>set_intersection</code>	2つの整列範囲の積集合（共通部分）
<code>set_difference</code>	2つの整列範囲の差集合
<code>set_symmetric_difference</code>	和集合から積集合を除いた整列範囲

下段は集合演算（量の問題で今回は割愛）

binary_search

- ▶ 二分探索
- ▶ 比較は無指定ならば< 演算子を使う
- ▶ 以下は関数オブジェクトを指定した例

```
#include <functional>
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    std::vector s {13,12,9,7,4,3}; // 降順
    std::vector t {4,8,12};
    for (auto x:t) {
        if (std::binary_search(s.begin(), s.end(),
                               x, std::greater<int>{}))
            std::cout << x << " ";
    }
    std::cout << "\n"; // 4 12
}
```


lower_bound, upper_bound

- ▶ lower_bound : 指定値より小さくない最初の要素
- ▶ upper_bound : 指定値より大きい値の最初の要素
- ▶ 値のみの指定と比較関数指定の二種がある

```
std::vector x {1,2,3,4,5,6,7,8,9,10}; // 昇順
auto low {std::lower_bound(x.begin(), x.end(), 3)};
auto up {std::upper_bound(x.begin(), x.end(), 8)};
std::cout <<"["<< *low <<" , "<< *up <<"): ";
print(low, up); // [3,9): 3 4 5 6 7 8
```

```
std::vector y {10,9,8,7,6,5,4,3,2,1}; // 降順
auto cmp { [](int e, int v) { return e>v; } };
auto l {std::lower_bound(y.begin(), y.end(), 7, cmp)};
auto u {std::upper_bound(y.begin(), y.end(), 2, cmp)};
if (l == y.end() || u == y.end()) return 1;
std::cout <<"["<< *l <<" , "<< *u <<"): ";
print(l, u); // [7, 1): 7 6 5 4 3 2
```

equal_range

- ▶ 指定値と同じ値を持つ要素の範囲
- ▶ デフォルトでは < 演算子で比較
- ▶ 範囲を `std::pair` で返す

```
#include <algorithm>
#include <list>
#include "print.hpp"
int main() {
    std::list x {1,2,3,3,3,3,4,5,6};
    auto b{x.begin()}, e{x.end()};
    auto [b2, e2] { std::equal_range(b, e, 3) };
    print(b2, e2); // 3 3 3 3

    std::list y {6,5,4,2,2,2,2,1,0};
    auto b3{y.begin()}, e3{y.end()};
    auto [b4, e4] { std::equal_range(b3, e3, 2,
        [](int a, int b) { return a > b; }) };
    print(b4, e4); // 2 2 2 2
}
```

実装の詳細

▶ lower_bound, upper_bound の利用

```
#include <algorithm>

// binary_search
template<typename Itr, typename T, typename F>
bool bsearch(Itr b, Itr e, const T& v, F cmp) {
    auto it { std::lower_bound(b, e, v, cmp) };
    return (it != e) && !(cmp(v, *it));
}

// equal_range
template<typename Itr, typename T, typename F>
auto erange(Itr b, Itr e, const T& v, F cmp) {
    return std::pair{std::lower_bound(b, e, v, cmp),
                     std::upper_bound(b, e, v, cmp)};
}
```

```
#include <functional>
#include <vector>
#include "print.hpp"
int main() {
    std::vector a {1,2,3,3,4,5,6};
    auto b{a.begin()}, e{a.end()};
    if (bsearch(b, e, 3, std::less<int>{})) {
        auto [p,q] {erange(b,e,3, std::less<int>{})};
        print(p,q); // 3 3
    }
}
```

merge

- ▶ 二つの整列範囲を結合して整列した結果を得る

```
#include <algorithm>
#include <iterator>
#include <vector>    // list はメンバ関数 merge を持つ
#include "print.hpp"
int main() {
    std::vector x{1,3,5,8};
    std::vector y{2,4,6,7,9,10};
    std::vector<int> z;
    auto b {x.begin()}, e {x.end()};
    auto b1{y.begin()}, e1{y.end()};
    std::merge(b, e, b1, e1, std::back_inserter(z));
    print(z); // 1 2 3 4 5 6 7 8 9 10
    std::reverse(b, e);
    std::reverse(b1, e1);
    std::merge(b, e, b1, e1, z.begin(),
               [](auto p, auto q){ return p > q; });
    print(z); // 10 9 8 7 6 5 4 3 2 1
}
```

inplace_merge

- ▶ 余計な場所を使わずにマージを行う
- ▶ 第 2 に引数に切り替わりの先頭を指定する
- ▶ < 演算子以外を使う場合は第 4 引数に指定

```
#include <algorithm>
#include <iterator>
#include <vector>
#include "print.hpp"
int main() {
    std::vector x{1,3,5,8};
    size_t n {x.size()};
    std::vector y{2,4,6,7,9,10};
    std::copy(y.begin(), y.end(),
              std::back_inserter(x));
    auto b{x.begin()}, e{x.end()};
    print(b, b+n); // 1 3 5 8
    print(b+n, e); // 2 4 6 7 9 10
    std::inplace_merge(b, b+n, e);
    print(x); // 1 2 3 4 5 6 7 8 9 10
}
```

マージソート

- ▶ 分割後に `inplace_merge` を繰り返す

```
#include <algorithm>
#include <vector>
#include "print.hpp"
template<typename Itr>
void merge_sort(Itr b, Itr e) {
    if (e - b > 1) { // 2個以上なら
        Itr m { b + (e - b) / 2 }; // 真ん中で分ける
        merge_sort(b, m);
        merge_sort(m, e);
        std::inplace_merge(b, m, e);
    }
}

int main() {
    std::vector x {3,5,2,7,4,9,1,8,6};
    merge_sort(x.begin(), x.end());
    print(x); // 1 2 3 4 5 6 7 8 9
}
```

数値アルゴリズム

数値アルゴリズム

▶ `#include <numeric>`

名前	効果
<code>iota</code> <code>accumulate</code> <code>adjacent_difference</code> <code>partial_sum</code> <code>inner_product</code>	++の増分値の並びで要素値の割り当て 初期値と要素の加算または指定演算を繰り返す 一つ前の要素との差分を順に計算 前側要素の合計を順に計算 2つの範囲の組み合わせの合計（内積）
<code>reduce</code> <code>transform_reduce</code> <code>inclusive_scan</code> <code>exclusive_scan</code> <code>transform_inclusive_scan</code> <code>transform_exclusive_scan</code>	並列処理に関連する計算（割愛）

adjacent_difference, partial_sum

- ▶ どちらも演算を変更できる
- ▶ 以下は変更なし基本的な例

```
#include <iterator>
#include <numeric>
#include <vector>
#include "print.hpp"
int main() {
    std::vector x {1,4,9,16,25,36};
    std::vector y(x.size(), 0), z(x.size(),0);
    auto b {x.begin()}, e {x.end()};
    auto b2{y.begin()}, e2{y.end()};

    std::adjacent_difference(b, e, b2);
    print(y); // 1 3 5 7 9 11

    std::partial_sum(b2, e2, z.begin());
    print(z); // 1 4 9 16 25 36
}
```

inner_product

- ▶ 二つの範囲と初期値
- ▶ オプションで二つの演算

```
#include <functional>
#include <iostream>
#include <numeric>
#include <vector>
int main() {
    std::vector x{1,2,3,4,5};
    std::vector y{5,2,2,4,1};
    auto b {x.begin()}, e{x.end()};
    auto b2{y.begin()};

    int r { std::inner_product(b, e, b2, 0) };
    std::cout << r << "\n"; // 36 内積

    r = std::inner_product(b,e,b2,0,std::plus<int>{},
        [](int a, int b) { return (a==b)?1:0; });
    std::cout << r << "\n"; // 2 対応する同値の数
}
```

まとめ

それぞれについて簡単に説明を加えましょう。

- ▶ 関数オブジェクト
- ▶ 関数オブジェクトのライブラリクラス
- ▶ ラムダ式
- ▶ キャプチャ
- ▶ 整列範囲用アルゴリズム
- ▶ 数値アルゴリズム