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

```
#include <iostream>
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}はオブジェクトの右辺値となる

```
#include <algorithm>
#include <iostream>
#include <vector>
class EvenMore { // (ましてや)
    int m;
public:
    EvenMore(int x):m{x}{}
    bool operator()(int x) {
        return (x > m) && (x % 2 == 0);
    }          // m より大きい偶数
};

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";
}
```

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 =1):n{a}{}
    int operator()() { auto x{n*n}; ++n; return x; }
};

int main() {
    std::vector a(5, 0); // 5 要素 int
    std::generate(a.begin(), a.end(), F{1});
    print(a); // 1 4 9 16 25
    std::generate(a.begin(), a.end(), F{3});
    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
- 用意されているクラス
  - equal_to, not_equal_to,
    less, less_equal, greater, greater_equal,

```
#include <functional>
#include <iostream>
int main() {
    int x{10}, y{20};
    std::less<int> comp0; // テンプレートクラス
    if (comp0(x,y)) std::cout << "ok\n";

    auto comp { std::less<int>{} };
    if (comp(x,y)) std::cout << "ok\n";
}
```

選択ソートの例

```
#include <functional>
#include <vector>
#include "print.hpp"
template<typename T, typename C>
void sort(T& a, C comp) { // selection sort
   for (size_t i = 0; i < a.size()-1; i++) {
      size_t min = i;
      for (size_t j = i+1; j < a.size(); j++)
         if (comp(a[j], a[min])) min = j;
      std::swap(a[i], a[min]);
   }
}
int main() {
   std::vector x {5,3,2,4,6,1,7};
   sort(x, std::less<int>{}); // int 用の < 演算子
   print(x); // 1 2 3 4 5 6 7

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

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>
    print(x); // 1 2 3 4 5 6 7

    std::sort(b, e, std::greater<int>{});
    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() {
    int x{10}, y{20}, z{0};

    auto op { std::plus<int>{} };
    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
}
```

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一般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ほしい！

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

```
#include <iostream>
int main(){
 // 関数を作って変数として保存
 auto f1 { []{ std::cout <<"hello lambda\n"; } };
 f1(); // 通常の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
 []{ std::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; } };
 std::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

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

int main() {
    test( []{ std::cout <<"world\n"; } ); // 実引数

    auto x { []{std::cout<<"again\n"; } };
    test( x ); // 変数に入れて実引数に
}
```

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を auto できる

```
#include <algorithm>
#include <vector>
#include "print.hpp"
using std::string;
int main() {
    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::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

```
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
    std::vector x {1,3,4,5,8,9,11,12,14};
    auto b{x.begin()}, e{x.end()};
    while (true) {
        b = std::find_if(b, e, [](int x){return x%2==0;});
        if (b == e) break;
        std::cout << *b <<" ";
        ++b;
    }
    std::cout <<"\n"; // 4 8 12 14
}
```

キャプチャ

ラムダ式で使用できる変数

- ラムダ式は関数なので大域変数が利用可能
- ラムダ式内の仮引数を含む局所変数も通常と同じ扱い
  - `[](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基本的にアクセス不可

```
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;        // compile 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>`
- 対象の指定範囲は整列済みを前提とする

名前	ltr	効果
`binary_search`	前方	二分探索により要素を探す
`lower_bound`	前方	指定より小さくない最初の要素
`upper_bound`	前方	指定より大きい値の最初の要素
`equal_range`	前方	指定と同じ値の要素の範囲
`merge`	入力	2 つの範囲の結合
`inplace_merge`	双方向	同一コンテナ内での結合
`includes`		
`set_union`		
`set_intersection`		集合演算関連今回は割愛
`set_difference`		
`set_symmetric_difference`		

binary_search

- < 演算子による二分探索
- 第 4 引数に比較関数を指定可

```
#include <functional>
#include <algorithm>
#include <iostream>
#include <list>
int main() {
    std::list s {13,12,9,7,4,3};
    s.sort();
    auto b{s.begin()}, e{s.end()};
    if (std::binary_search(b, e, 7))
        std::cout << "found\n";

    s.reverse();
    b = s.begin(); e = s.end();
    if (std::binary_search(b, e, 7, std::greater<int>{}))
        std::cout << "found\n";
}
```

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

```
#include <algorithm>
#include <vector>
#include "print.hpp"
int main() {
    std::vector x {1,3,5,7,9,11,13}; // 昇順
    auto b{x.begin()}, e{x.end()};
    auto lo1 {std::lower_bound(b, e, 2)};
    auto up1 {std::upper_bound(b, e, 9)};
    print(lo1, up1); // 3 5 7 9

    std::reverse(b, e);                    // 降順
    auto cmp { [](int a, int b){ return a > b; } };
    auto lo2 { std::lower_bound(b, e, 9, cmp) };
    auto up2 { std::upper_bound(b, e, 2, cmp) };
    print(lo2, up2); // 9 7 5 3
}
```

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> // search.hpp

// 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 eqrange(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)};
}
```

```cpp
#include <functional>
#include <vector>
#include "print.hpp"
#include "search.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] {eqrange(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

```cpp
#include <algorithm>
#include <iterator>
#include <vector>
#include "print.hpp"
int main() {
    std::vector<int> x{1,3,5,8}, y{2,4,6,7,9,10}, 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
}
```

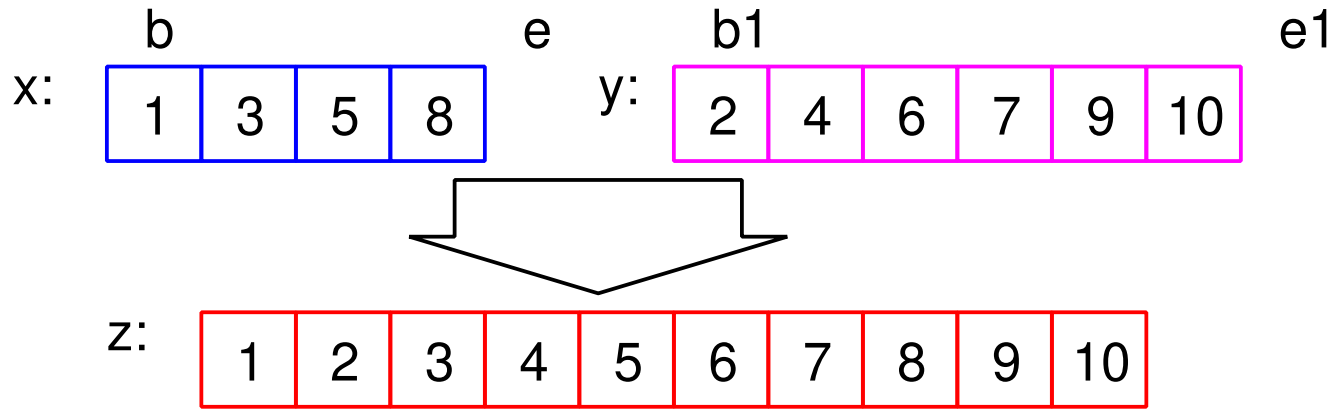
inplace_merge

- 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

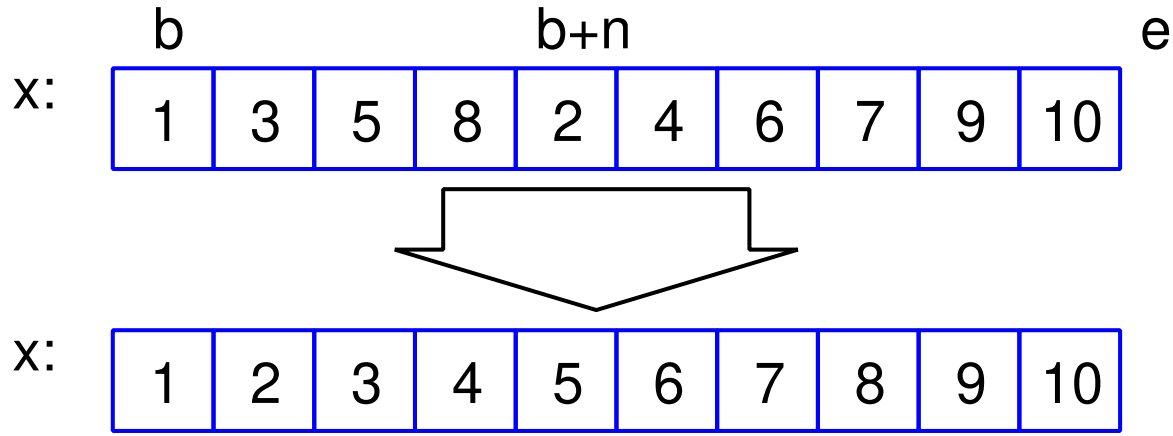
```
#include <algorithm>
#include <iterator>
#include <vector>
#include "print.hpp"
int main(){
    std::vector x{1,3,5,8}, y{2,4,6,7,9,10};
    auto n {x.size()};
    std::copy(y.begin(),y.end(), std::back_inserter(x));
    auto b{x.begin()}, e{x.end()};
    print(b, b+n); // 1 3 5 8
    print(b+n,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

- `merge`



- `inplace_merge`



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
        auto m { b + (e-b)/2 };         // 真ん中で分ける
        merge_sort(b, m);               // 前半を整列
        merge_sort(m, e);               // 後半を整列
        std::inplace_merge(b, m, e); // マージ
    }
}

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
}
```

補足：std::list の merge

- 追加する形でマージを行う
- < 演算子以外で比較する場合は第 2 引数で指定

```
#include <functional>
#include <list>
#include "print.hpp"
int main() {
    std::list x{2,8,10}, y{1,3,4,5,6};
    x.merge(y);
    print(x);

    std::list z{9,7};
    x.reverse();
    x.merge(z, std::greater<int>{});
    print(x);
}
```

数値アルゴリズム

数値アルゴリズム

- `#include <numeric>`

名前	ltr	効果
`iota`	前方	++の増分値の並び
`accumulate`	入力	初期値と要素との蓄積演算
`adjacent_difference`	入力	一つ前の要素との差分
`partial_sum`	入力	前側要素の合計
`inner_product`	入力	2 つの範囲の積和演算（内積）
`reduce`		
`transform_reduce`		
`inclusive_scan`		並列処理に関連する計算（割愛）
`exclusive_scan`		
`transform_inclusive_scan`		
`transform_exclusive_scan`		

accumulate

- 範囲 [b,e) と初期値を指定
- オプションで 2 引数関数を指定する

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

accumulate とラムダ式

- 演算内容をラムダ式で指定する例

```
#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
}
```

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

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

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

    std::vector<int> z(y.size());
    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 ([b,e) と [b2,?) で同一サイズ)
- オプションで二つの演算 (和と積に相当する演算)

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