C++プログラミング II

第 10 回 STL アルゴリズム 2

岡本秀輔

成蹊大学理工学部

共通の関数

- 今回のプログラムで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";
}
```

修正ありアルゴリズム

一覧

- 前提知識
  - 連想コンテナは修正の対象に使用できない
  - 全体が範囲ならば範囲 `for` 文で良い場合が多い
- 関数の名前
  - `_n`:範囲中の最初の `n` 個
  - `_if`:述語が `true` の要素
  - `_copy`:結果を複製

名前	効果
`fill(_n)`	要素に指定値を代入
`iota`	++の増分値の並びで要素値を割り当て
`generate(_n)`	関数の結果で要素の値を置き換え
`replace(_if)`	範囲中の値を置き換え
`replace_copy(_if)`	範囲中の値を置き換えて複製
`copy(_n, _if, _backward)`	複製 (4 バージョンある)
`transform`	要素に関数を適用し、別範囲に保存
`for_each(_n)`	各要素に関数を適用

fill, iota, generate

- コンテナ要素に特定の値を代入するための関数
  - `fill(b,e,x)`: 指定値 `x`
    内容：`for (T it=b; it!=e; ++it) *it = x;`
  - `iota(b,e,x)`:指定値 `x` から++で増えた値
    内容：`for (T it=b; it!=e; ++it){ *it=x; ++x; }`
  - `generate(b,e,f)`: 指定関数 `f` の結果
    内容：`for (T it=b; it!=e; ++it) *it = f();`
- すべて void 関数
- `iota()` のみ<numeric>ヘッダファイル

fill/iota/generateの使用例

```cpp
#include <algorithm>
#include <numeric> // iota
#include <vector>
#include "print.hpp"

int n { 0 };
int f() { ++n; return n*n; }

int main() {
    std::vector<int> a(5);
    std::fill(a.begin(), a.end(), -1);
    print(a); // -1 -1 -1 -1 -1
    std::iota(a.begin(), a.end(), 1);
    print(a); // 1 2 3 4 5
    std::generate(a.begin(), a.end(), f);
    print(a); // 1 4 9 16 25
}
```

置き換えと複製

- 範囲を一つだけ指定して値を更新する
  - `replace(b, e, v1, v2)`
  - `replace_if(b, e, pred1, v)`
  - `for_each(b, e, op1)` 更新しない場合もあり
- 更新せずに結果を複製する (b2 が複製先)
  - `replace_copy(b, e, b2, v1, v2)`
  - `replace_copy_if(b, e, b2, pred1, v)`
  - `copy(b, e, b2)`
  - `copy_if(b, e, b2, pred1)`
  - `copy_n(b, n, b2)`
  - `copy_backward(b, e, e2)` これだけ e2 違う
  - `transform(b, e, b2, op1)`
  - `transform(b, e, b1, b2, op2)`
  - 戻り値：
    - 複製先 `[b2,e2)` の イテレータ `e2` を返す
    - `copy_backward` のみ最終書き込み要素のイテレータ

記号の意味

- 引数の省略記号は以下を意味する (この資料のみ)

記号	意味
b, b1, b2	先頭イテレータ
e	[b, e) の入力範囲となるイテレータ
e2	[b2, e2) の出力範囲となるイテレータ
v, v1, v2	任意の値
n	個数を表す整数
pred1	1 引数の述語となる関数
op1	引数 1 個の関数
op2	引数 2 個の関数

replace/replace_if の使用例

- _if の有無で第 3 引数が変わる

```
#include <algorithm>
#include <list>
#include "print.hpp"

bool is_even(int x) { return x%2 == 0; }

int main()
{
    std::list x {1,2,3,5,3,7,3,8,3};
    print(x); // 1 2 3 5 3 7 3 8 3
    std::replace(x.begin(), x.end(), 3, 0);
    print(x); // 1 2 0 5 0 7 0 8 0
    std::replace_if(x.begin(), x.end(), is_even, 0);
    print(x); // 1 0 0 5 0 7 0 0 0
}
```

replace_copy/replace_copy_if の使用例

- 第 3 引数は複製先の先頭イテレータ
  - 複製先には十分な場所が必要，複製元との重なりは不可
- _if の有無で第 4 引数が変わる

```
#include <algorithm>
#include <vector>
#include "print.hpp"

bool is_even(int x) { return x%2 == 0; }

int main() {
    std::vector x {1,2,3,4,3,6,3,8,3};
    std::vector y(x.size(), 0); // x と同じサイズ
    auto b{x.begin()}, e{x.end()};
    print(x); // 1 2 3 4 3 6 3 8 3
    std::replace_copy(b, e, y.begin(), 3, 0);
    print(y); // 1 2 0 4 0 6 0 8 0
    std::replace_copy_if(b, e, y.begin(), is_even, 0);
    print(y); // 1 0 3 0 3 0 3 0 3
}
```

copy/copy_if/copy_n の使用例

- 複製先には十分な容量が必要

```
#include <algorithm>
#include <vector>
#include "print.hpp"
bool is_even(int x) { return x%2 == 0; }

int main() {
    std::vector x {1,2,3,4,3,6,3,8,3};
    std::vector y(x.size(), 0); // 同じサイズ
    auto  b{x.begin()},  e{x.end()};
    auto b2{y.begin()}, e2{y.end()};
    e2 = std::copy(b, e, b2);
    print(b2, e2); // 1 2 3 4 3 6 3 8 3
    e2 = std::copy_if(b, e, b2, is_even);
    print(b2, e2); // 2 4 6 8
    e2 = std::copy_n(b, 3, b2); // 先頭から 3 要素
    print(b2, e2); // 1 2 3
}
```

copy_backward の使用例

- 複製先には十分な容量が必要
- 後ろから複製するが順序は保たれる

```
#include <algorithm>
#include <vector>
#include "print.hpp"

int main()
{
    std::vector x {1,2,3,4,5};
    std::vector y(8, 0); // 8個
    auto  b{x.begin()},  e{x.end()};
    auto b2{y.begin()}, e2{y.end()};
    auto it { std::copy_backward(b, e, e2) };
    print(b2, e2); // 0 0 0 1 2 3 4 5
    print(it, e2); // 1 2 3 4 5
}
```

transform の使用例

- 引数 1 個関数 (p) と 2 個関数 (f)

```
#include <algorithm>
#include <vector>
#include "print.hpp"
int p(int a)         { return a*a; }
int f(int a, int b) { return 2*a + b; }

int main() {
   std::vector x {1,2,3,4,5,6,7,8};
   std::vector y(x.size(), 0); // x と同じサイズ
   std::vector z(x.size(), 0); // x と同じサイズ
   auto b{x.begin()}, e{x.end()};

   std::transform(b, e, y.begin(), p);
   print(y); // 1 4 9 16 25 36 49 64

   std::transform(b, e, y.begin(), z.begin(), f);
   print(z); // 3 8 15 24 35 48 63 80
}
```

for_each の使用例

- 第 3 引数に `void f(T&)` または `void f(const T&)` に相当する関数を指定する
- 範囲 for 文が導入される前に作られた

```
#include <algorithm>
#include <iostream>
#include <list>
void update(int& x) { x = x*x; }
void put(int x)     { std::cout<< x <<" "; }

int main()
{
   std::list x {1,2,3,4,5};
   auto b{x.begin()}, e{x.end()};

   std::for_each(b, e, update);
   std::for_each(b, e, put);
   std::cout <<"\n"; // 1 4 9 16 25
}
```

条件のまとめ

アルゴリズム	イテレータ	その他の指定
fill	前方	値
fill_n	前方	個数と値
iota	前方	値
generate	前方	引数なし関数
generate_n	出力	個数と引数なし関数
replace	前方	新旧の値
replace_if	前方	1 引数述語と値
replace_copy	入力	新旧の値
replace_copy_if	入力	1 引数述語と値
copy	入力	
copy_if	入力	1 引数述語
copy_n	入力	個数
copy_backward	双方向	
transform	入力	1or2 引数関数
for_each	入力	1 引数関数
for_each_n	入力	個数と 1 引数関数

イテレータアダプタ

イテレータアダプタ

- イテレータのように振る舞うクラス・オブジェクト
  - イテレータとインタフェースが同じ
  - STL アルゴリズムで使用できる
  - コンテナと無関係な場合もある
  - `<iterator>`ヘッダファイル
- 挿入イテレータ: 生成関数の利用
  - `std::back_inserter(c)`
  - `std::front_inerter(c)`
  - `std::inserter(c, p)`
- ストリームイテレータ: クラス変数を作る
  - `std::istream_iterator<T>`
  - `std::ostream_iterator<T>`
- 逆（リバース）イテレータ: コンテナメンバ関数
  - `.rbegin()`, `.rend()`
- move イテレータ（この授業では割愛)

挿入イテレータ

- it = std::back_inserter(c)
  - *it = x; で c.push_back(x) が行われる
- it = std::front_inerter(c)
  - *it = x; で c.push_front(x) が行われる
- it = std::inserter(c, p)
  - *it = x; で c.insert(p, x) が行われる
  - 連想コンテナは p をヒントとして扱う (無視する)

```
#include <algorithm>
#include <iterator> // back_inserter
#include <list>
#include "print.hpp"
bool is_odd(int x) { return x%2!=0; }
int main() {
    std::list<int> x{1,2,3,4,5}, y{};
    auto b{x.begin()}, e{x.end()};
    std::copy_if(b, e, std::back_inserter(y), is_odd);
    print(y); // 1 3 5
}
```

ストリームイテレータ

- istream_iterator のデフォルトコンストラクタはストリームの終わりを表すオブジェクトとなる

```
#include <algorithm>
#include <iostream>
#include <iterator>
bool is_odd(int x) { return x%2 != 0; }

int main() {
   std::istream_iterator<int>  b{std::cin}, e{};
   std::ostream_iterator<int> b2{std::cout, " "};
   std::copy_if(b, e, b2, is_odd);
   std::cout <<"\n";
}
```

```
 $ echo 1 2 3 4 5 6 |./a.out
1 3 5
```

逆イテレータ

- 逆順にたどるイテレータ
- 双方向またはランダムイテレータが対象
- コンテナの rbegin(), rend() メンバ関数

```
#include <algorithm>
#include <iostream>
#include <list>
bool is_even(int x) { return x%2 == 0; }

int main() {
   std::list a {1,2,3,4,5};
   for (auto it = a.rbegin(); it!=a.rend(); ++it)
      std::cout << *it <<" ";
   std::cout <<"\n"; // 5 4 3 2 1

   auto it{ std::find_if(a.rbegin(),
                         a.rend(), is_even) };
   if (it != a.rend())
      std::cout << *it <<"\n"; // 4
}
```

削除アルゴリズム

削除アルゴリズム

- 要素削除の準備をするアルゴリズム
- 必要な要素だけを前方に残す、要素数の変更なし
- 無効となる要素の先頭のイテレータを返す
  - 無効となる要素の値はどうなるか分からない
  - 元の値の場合と複製された値の場合がある
- 実際の削除はコンテナのメンバ関数が行う

名前	効果
`remove`	指定値を削除
`remove_if`	基準にあう要素を削除
`remove_copy`	指定値を取り除いて複製
`remove_copy_if`	基準あう要素を除いて複製
`unique`	連続する値を削除（唯一にする）
`unique_copy`	連続する値を除いて複製

removeの使用例

- 形式
  - remove(b, e, x)
  - remove_if(b, e, pred1)
  - remove_copy(b, e, b2, x)
  - remove_copy_if(b, e, b2, pred1)
- a.erase(remove(b,e,x), e)としても良い

```
#include <algorithm>
#include <list>
#include "print.hpp"

int main() {
    std::list a {1,2,3,2,4,5,2,6};
    auto it {a.begin()};
    it = std::remove(a.begin(), a.end(), 2);
    a.erase(it, a.end());
    print(a); // 1 3 4 5 6
}
```

unique の使用例

- ==演算子で等値となる連続値を取り除く
- 引数 2 個の述語の関数を指定しても良い

```
#include <algorithm>
#include <list>
#include "print.hpp"
bool eq(double x, double y)
     { return std::abs(x-y) < 0.01; }

int main() {
   std::list a {1,2,2,3,3,4,5,5,6};
   a.erase(std::unique(a.begin(), a.end()), a.end());
   print(a); // 1 2 3 4 5 6

   std::list b {1.0, 1.001, 2.0, 2.002, 3.0};
   b.erase(std::unique(b.begin(), b.end(), eq),
           b.end());
   print(b); // 1 2 3
}
```

条件のまとめ

アルゴリズム	イテレータ	その他の指定
`remove`	前方	値
`remove_if`	前方	1 引数述語
`remove_copy`	入力	値
`remove_copy_if`	入力	1 引数述語
`unique`	前方	なし or 2 引数述語
`unique_copy`	入力	なし or 2 引数述語

順序変更アルゴリズム

順序変更アルゴリズム

名前	効果
reverse	要素を逆順に変更
reverse_copy	要素の逆順を複製
rotate	指定した要素が先頭となるように循環移動
rotate_copy	指定で分かれた前半と後半を入れ替えて複製
partition	基準にあう要素が前側になるように順序変更
stable_partition	移動で相対的な位置関係を変えない partition
partition_copy	partition の複製版, 出力先を 2 つ指定する
next_permutation	辞書順で次にあたる順列を生成
prev_permutation	辞書順で前にあたる順列を生成
shuffle	要素の順序をランダムに入れ替える
sample	指定個数の要素をランダムに取り出す

reverse, rotate, partition の使用例

- rotate は指定値を第 2 引数のイテレータで指定
- partition は区切りのイテレータを返す

```
#include <algorithm>
#include <list>
#include "print.hpp"
bool is_odd(int x) { return x%2 != 0; }

int main() {
    std::list a {1,2,3,4,5};
    std::reverse(a.begin(), a.end());
    print(a); // 5 4 3 2 1
    auto it {std::find(a.begin(), a.end(), 2)};
    std::rotate(a.begin(), it, a.end()); // 2 が先頭に
    print(a); // 2 1 5 4 3
    it = std::partition(a.begin(), a.end(), is_odd);
    print(a); // 3 1 5 4 2   奇数が前側に
    std::cout << *it <<"\n"; // 4 が境界
}
```

partition_copy の使用例

- 条件に合う結果と合わない結果用の二つの先頭を指定
- 戻り値は std::pair でそれぞれの end

```
#include <algorithm>
#include <vector>
#include "print.hpp"
bool is_even(int x) { return x % 2 == 0; }

int main() {
   std::vector x {1,2,3,4,5,6,7};
   std::vector y(x.size(), 0), z(x.size(), 0);
   auto  b{ x.begin() }, e{ x.end() };
   auto b2{ y.begin() }; // 結果1(述語true)
   auto b3{ z.begin() }; // 結果2(述語false)
   auto [e2,e3] {
     std::partition_copy(b, e, b2, b3, is_even) };
   print(b2, e2); // 6 2 4
   print(b3, e3); // 3 5 1 7
}
```

permutation とは

- 数学では「置換」や「順列」と呼ばれる
- 集合の要素に順序を与えたものと考えてよい
- 例：{1, 2, 3}に順序を与えた並び替えは 3!=6 個ある
  - 1 2 3
  - 1 3 2
  - 2 1 3
  - 2 3 1
  - 3 1 2
  - 3 2 1

next_permutation の使用例

- 次の順列を構成する並びにに並べ替える
- 並べ替えに成功すれば true, 失敗で false

```
#include <algorithm>
#include <list>
#include "print.hpp"
int main()
{
    std::list a {1,2,3};
    auto b{a.begin()}, e{a.end()};
    for (bool f{true}; f;
        f = std::next_permutation(b, e) ){
        print(a);
    }
}
```

実行結果

```
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1
```

shuffleの使用例

- 要素をランダムに入れ替える
- 乱数生成器が必要となる
  - 第5回で作ったrandom.hppを確認せよ

```
#include <algorithm>
#include <random>
#include <vector>
#include "print.hpp"
int main()
{
   std::vector a {1,2,3,4,5,6,7,8,9};
   std::random_device seed;
   std::mt19937 engine{seed()};

   std::shuffle(a.begin(), a.end(), engine);

   print(a); // 毎回異なる結果
}
```

sample の使用例

- 指定数の要素をランダムに取り出す
- 乱数生成器が必要となる

```
#include <algorithm>
#include <iterator>
#include <random>
#include <list>
#include "print.hpp"
int main()
{
   std::list<int> a{1,2,3,4,5,6,7,8,9}, b;
   std::random_device seed;
   std::mt19937 engine{seed()};

   std::sample(a.begin(), a.end(),
               std::back_inserter(b), 5, engine);

   print(b); // 毎回異なる 5 要素の結果
}
```

条件のまとめ

アルゴリズム	イテレータ	その他の指定
`reverse`	双方向	
`reverse_copy`	双方向	
`rotate`	前方	
`rotate_copy`	前方	
`partition`	双方向	1 引数述語
`stable_partition`	双方向	1 引数述語
`partition_copy`	入力	1 引数述語
`next_permutation`	双方向	
`prev_permutation`	双方向	
`shuffle`	ランダム	一様乱数生成器
`sample`	入力	結果はランダム itr, 個数, 一様乱数生成器

整列アルゴリズム

整列アルゴリズム

名前	効果
sort	全要素を整列
stable_sort	同値要素の相対関係を変更しないで整列
partial_sort	[b,e) の間の m で指定した場所まで整列
partial_sort_copy	[b, e) を整列して [b2,e2) にいれる。 min(e-b, e2-b2) 個のみが対象。
nth_element	n 番目に大きい要素を n 番目に配置し、それより 小さい値を n の前側、大きい値を後ろ側に配置
make_heap	ヒープの作成（最大値を先頭に移動）
push_heap	ヒープに値を挿入（値を適切な場所へ移動）
pop_heap	ヒープから値を削除（最大値を最後尾に移動）
sort_heap	ヒープ範囲を整列

「アルゴリズムとデータ構造」の第 11 回目で扱うのでヒープは割愛

sort: 指定範囲の整列

- この例はメンバ x だけで比較する構造体
- sort_stable の引数指定はこれと同じ

```
struct T { int x, y; };
using CT = const T; // const T に CT という名をつける
bool operator<(CT& a, CT& b) { return a.x < b.x; }
bool   greater(CT& a, CT& b) { return a.x > b.x; }

auto& operator<<(std::ostream& o, CT& t) {
   return o <<"("<< t.x <<","<< t.y <<")";
}

int main() {
   std::vector<T> a {{3,1},{3,2},{1,1},{4,2},{2,3}};
   print(a); // (3,1) (3,2) (1,1) (4,2) (2,3)
   std::sort(a.begin(), a.end());
   print(a); // (1,1) (2,3) (3,1) (3,2) (4,2)
   std::sort(a.begin(), a.end(), greater);
   print(a); // (4,2) (3,1) (3,2) (2,3) (1,1)
}
```

partial_sort: 部分的な整列

- 引数 (b, m, e) に対して、[b,e) を整列した際の部分結果が [b,m) となるように要素を入れ替える

```
#include <algorithm>
#include <random>
#include <vector>
#include "print.hpp"
int main()
{
  std::vector a {1,2,3,4,5,6,7,8,9};
  std::mt19937 engine{std::random_device{}()};
  std::shuffle(a.begin(), a.end(), engine);
  print(a); // 毎回異なる結果

  std::partial_sort(a.begin(), a.begin()+4, a.end());
  print(a); // 1 2 3 4 が前半になる
}
```

partial_sort_copy: 部分整列を複製

- 引数 (b,e,b2,e2) に対して、[b,e) を整列対象として, [b2,e2) を出力対象とする
- 整列する個数は二つの範囲の狭い方で決まる

```
std::vector a {4,1,3,2};
std::vector b {9,8};
std::vector c {8,7,10,12,13,9};
print(a); // 4 1 3 2
auto itb { std::partial_sort_copy(
                a.begin(), a.end(),
                b.begin(), b.end()) };
print(b.begin(), itb); // 1 2       itb == b.end()

auto itc { std::partial_sort_copy(
                a.begin(), a.end(),
                c.begin(), c.end()) };
print(c.begin(), itc); // 1 2 3 4   itc != c.end()
print(itc, c.end());   // 13 9
```

nth_element: n 番目と前後の入れ替え

- 第 2 引数に n 番目のイテレータ m を指定する
- m には整列したときと同じ値が入る
- m より前側には、*m 以下の値が入る

```
#include <algorithm>
#include <random>
#include <vector>
#include "print.hpp"
int main()
{
    std::vector a {1,2,3,4,5,6,7,8,9};
    std::mt19937 engine{std::random_device{}()};
    std::shuffle(a.begin(), a.end(), engine);
    print(a); // 毎回異なる結果

    std::nth_element(a.begin(), a.begin()+5, a.end());
    print(a); // 6 は必ず同じ位置にくる
}
```

条件のまとめ

アルゴリズム	イテレータ	その他の指定
sort	ランダム	<, 2 引数述語
statlbe_sort	ランダム	<, 2 引数述語
partial_sort	ランダム	< , 2 引数述語
partial_sort_copy	入力	<, 2 引数述語
		複製先ランダム itr
nth_element	ランダム	<, 2 引数述語
make_heap	ランダム	<, 2 引数述語
push_heap	ランダム	<, 2 引数述語
pop_heap	ランダム	<, 2 引数述語
sort_heap	ランダム	<, 2 引数述語

整列チェックアルゴリズム

整列チェックアルゴリズム

名前	効果
`is_sorted`	指定範囲が整列しているか？
`is_sorted_until`	整列を乱す最初の要素
`is_partitioned`	述語の基準で 2 つのグループに分かれているか?
`partition_point`	述語の基準で分かれた後半グループの先頭要素
`is_heap`	ヒープを構成しているか?
`is_heap_until`	ヒープの構成を乱す最初の要素

「アルゴリズムとデータ構造」の第 11 回目で扱うのでヒープは割愛

is_sorted と is_sorted_until の使用例

- is_sorted(b, e) は [b,e) の整列の有無を bool 値で返す
- is_sorted_until(b, e) は [b,e) で整列を乱す最初の要素のイテレータを返す

```
std::vector a {1,2,3,0,5,4};
auto b{a.begin()}, e{a.end()};

if (!std::is_sorted(b, e))
    std::cout <<"not sorted\n";

auto it { std::is_sorted_until(b, e) };
print(b, it); // 1 2 3

std::sort(b, e);
if (std::is_sorted(b, e))
    std::cout <<"sorted\n";
```

is_partitioned の使用例

- partition によって偶数が前側に移動する
- is_partition でそれを確認する

```
bool is_even(int x) { return x % 2 == 0; }

template<typename Itr, typename F>
void check(Itr b, Itr e, F func) {
   std::cout
     << std::boolalpha // <iomanip>
     << std::is_partitioned(b, e, func)
     <<"\n";
}

int main() {
   std::vector a {1,2,3,4,5,6,7};
   auto b{a.begin()}, e{a.end()};
   check(b, e, is_even); // false
   std::partition(b, e, is_even);
   check(b, e, is_even); // true
}
```

partition_point の使用例

- 条件を満たさないグループの先頭
- std::partitoin の返すイテレータと同一

```
bool is_even(int x) { return x % 2 == 0; }

int main()
{
    std::vector a {1,2,3,4,5,6,7};
    auto b{a.begin()}, e{a.end()};
    auto it0 { std::partition(b, e, is_even) };
    auto it1 { std::partition_point(b, e, is_even) };
    if (it0 == it1) std::cout <<"ok\n"; // ok
    print(b, it1); // 6 2 4
    print(it1, e); // 3 5 1 7
}
```

条件のまとめ

アルゴリズム	戻り値	イテレータ	その他
is_sorted	bool	前方	<,2 引数述語
is_sorted_until	前方	前方	<,2 引数述語
is_partitioned	bool	入力	1 引数述語
partition_point	前方	前方	1 引数述語
is_heap	bool	ランダム	<,2 引数述語
is_heap_until	ランダム	ランダム	<,2 引数述語