C++プログラミング II

第 9 回 STL アルゴリズム 1

岡本秀輔

成蹊大学理工学部

目的

学習の目的

プログラミングで大切なこと

- ある程度のパターンを覚える
- 自分のやりたいことをパターンに当てはめる
  - パターンはプログラミングの慣用表現とも呼ばれる
- パターンが見つからない時に自分なりの工夫をする

ライブラリを使っていて良いこと

- 実装内容に信頼がおける
- 時代とともに良くなる方向の変化が期待できる
- 第 3 者がプログラムを読んだときに理解しやすい

STL アルゴリズムの構成

STL アルゴリズムの構成

- ジェネリック関数からなる
  - generic: 一般的な、包括的な、総称的な
  - データ型を後で指定する
  - 提供される関数が複数のコンテナに対応する
  - オブジェクト指向プログラミングとの関係
    - オブジェクトがデータの内部を管理し、ジェネリック関数でオブジェクトどうしの関係を処理する
- 注意点
  - コンテナとアルゴリズムで悪い組み合わせがある
  - イテレータの種類で確認する

ヘッダファイル

- `#include <algorithm>`
  - ほとんどのアルゴリズム
- `#include <numeric>`
  - 乱数／分数などの数値関連
- `#include <functional>`
  - 関数オブジェクト, 関数アダプタ
  - 11 回目の授業でとりあげる

アルゴリズムの名前

- `xxxx_if()`
  - `_if` なし:引数で値を指定する
  - `_if` あり:`bool` を返す関数を指定する
    - 条件を決定するための関数
    - 述語と呼ぶことがある（数学用語）
    - 関数以外も指定できる (講義 12 回)
- `xxxx_copy()`
  - 要素が対象範囲で複製される
  - 例：
    - `reverse()`:逆順,
    - `reverse_copy()`:逆順の複製
- `xxxx_n()`
  - 最初の `n` 個

すべての関数がこの規則に従うわけではない

分類

修正なし : 要素を変更しない (探索や数え上げ)
修正あり : 要素の値を変更する
削除 : 不要な要素をまとめる（後ろにどかす）
変化 : 値を変更せずに順序を変更する (逆順や回転)
整列 : 要素を整列させる
整列範囲 : 整列後に行える操作群
数値 : 基本的な数値計算

指定する範囲

- 有効な範囲
  - `begin()` から `end()` に到達できること
- 半開区間（正確には左閉右開区間）
  - `[B, E)` = { x | B $\leqq$ x $<$ E }
  - `B` は含むが `E` は含まない範囲
  - `E` は末尾要素の次の場所を意味する
  - 要素無しの場合を簡単に扱える (`B == E`)

対象イテレータ

- 入出力はコンテナではないがアルゴリズムの対象
- 入力と出力イテレータは前方イテレータの部分機能を持つ
- 入力と出力は共通でない機能を持つ

イテレータ	能力	関係クラス
出力	書いて進む	ostream, inserter
入力	一度読んで進む	istream
前方	進む	foward_list, unordered_*
双方向	進む/戻る	list, *set, *map
ランダム	進む/戻る/直接	array, vector, deque

修正なしアルゴリズム

一覧

- 入力／前方イテレータを範囲として指定する

名前	効果
count	要素の数
count_if	条件にあう要素の数
min_element	最小値
max_element	最大値
minmax_element	最大値と最小値の対
find	値で指定した最初の要素
find_if	指定した基準を満たす最初の要素
find_if_not	指定した基準以外の最初の要素
search_n	指定値が n 個連続する最初の要素
search	範囲 A 中に最初に現れる部分範囲 B の先頭要素
find_end	範囲 A 中に最後に現れる部分範囲 B の先頭要素
find_first_of	範囲 A 中の現れる範囲 B のどれかと同じ最初の要素
adjacent_find	同値で隣り合うという条件の最初の要素

std::count の使い方

- 指定値の要素を数える
- ==演算子で比較
- 入力イテレータ

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <list>
using std::cout;

int main()
{
   std::vector a {1,2,3,1,2,2,3,4};
   cout << std::count(a.begin(), a.end(), 2) <<"\n";

   std::list b {1,2,3,1,2,2,3,4};
   cout << std::count(b.begin(), b.end(), 2) <<"\n";
}
```

count の処理

- 処理内容をイメージしてライブラリを使用すると良い
- 実際のテンプレートでは汎用向けに様々な設定がある
- it の代わりに b を更新する実装が一般的

```
#include <iostream>
#include <vector>

template<typename It, typename T>
int count(It b, It e, const T& v)
{
    int c{0};
    for (It it = b; it!=e; ++it)
        if (*it == v) ++c;  // == 演算子で等値比較
    return c;
}

int main() {
    std::vector a {1,2,3,1,2,2,3,4};
    std::cout << count(a.begin(), a.end(), 2) <<"\n";
}
```

std::count_if の使い方

- bool を返す関数を作り、関数名を指定する

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <list>

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

int main() {
    std::vector a {1,2,3,1,2,6,3,4};
    std::list   b {1,2,3,1,2,6,3,4};
    std::cout
        << std::count_if(a.begin(), a.end(), is_even)
        <<" "    // 4
        << std::count_if(b.begin(), b.end(), is_even)
        <<"\n"; // 4
}
```

count_ifの処理

- テンプレート引数は関数の型も指定できる

```
#include <iostream>
#include <vector>

template<typename It, typename P>
int count_if(It b, It e, P func)
{
    int c{0};
    for (It it = b; it!=e; ++it)
        if (func(*it)) ++c;
    return c;
}

bool is_even(int x) { return x % 2 == 0; }
int main() {
    std::vector a {1,2,3,1,2,2,3,4};
    std::cout <<
        count_if(a.begin(), a.end(), is_even) <<"\n";
}
```

最大最小

- 最大/最小を探しイテレータを返す
- minmax_element はイテレータの対 (std::pair)

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
int main() {
   std::vector a {3,5,4,8,7,1,2};

   auto it { std::min_element(a.begin(), a.end()) };
   std::cout << *it <<" "; // 1

   it = std::max_element(a.begin(), a.end());
   std::cout << *it <<"\n"; // 8

   auto p { std::minmax_element(a.begin(),a.end())};
   auto [min,max] {p}; // p は std::pair<It,It>
   std::cout << *min <<" "<< *max <<"\n"; // 1 8
}
```

min_element の処理

- max_element, minmax_element もほぼ同じ

```
#include <iostream>
#include <vector>

template<typename It>
It min_element(It b, It e)
{
    if (b == e) return e;
    It m{b};
    ++b;
    for (It it=b; it!=e; ++it)
        if (*it < *m) m = it;
    return m;
}

int main() {
    std::vector a {3,5,4,8,7,1,2};
    auto it { min_element(a.begin(), a.end()) };
    std::cout << *it <<"\n";
}
```

std::find, std::find_if, std::find_if_not

- 引数は count/count_if と同じ、結果はイテレータ
- it==a.end() ならば見つからなかったことになる

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

int main()
{
  std::vector a {2,2,5,1,2,2,3,4};
  auto i1 { std::find(a.begin(), a.end(), 3) };
  auto i2 { std::find_if(a.begin(),a.end(), is_odd)};
  auto i3 { std::find_if_not(a.begin(), a.end(),
                                          is_odd)};
  using std::cout;
  if (i1 != a.end()) cout <<"i1:"<< *i1 <<"\n";
  if (i2 != a.end()) cout <<"i2:"<< *i2 <<"\n";
  if (i3 != a.end()) cout <<"i3:"<< *i3 <<"\n";
}
```

条件に合うすべての要素（応用）

- std::find は最初に見つけた要素のイテレータを返す
- 条件に合う要素をすべて探すには、見つけた要素の次のイテレータを開始位置にする

```
// 値 2 の要素のすべての添字
std::vector a {2,2,5,1,2,2,3,4};
auto it { std::find(a.begin(), a.end(), 2) };

while (it != a.end()) {
    std::cout << it - a.begin() <<" ";
    ++it; // 次の要素
    it = std::find(it, a.end(), 2);
}
std::cout <<"\n";
```

イテレータどうしの減算はランダムイテレータのみ

find/find_if/find_if_not の処理

- 条件にあう要素を発見した時点でイテレータを返す
- find_if_not は if 文の条件が!func(*it) となる

```
template<typename It, typename T>
It find(It b, It e, const T& x)
{
   for (It it=b; it!=e; ++it)
      if (*it == x) return it;
   return e;
}

template<typename It, typename P>
It find_if(It b, It e, P func)
{
   for (It it=b; it!=e; ++it)
      if (func(*it)) return it;
   return e;
}
```

メンバ関数 count と find

set, multiset, map, multimap, unordered_* は
count, find メンバ関数を持つ

- どれも key に対する結果を返す
- 重複なしコンテナの count は 0,1 を返す

メンバ関数と std::count/std::find の双方が使用可能

- 対象範囲
  - std::count/std::find:指定範囲
  - メンバ関数：要素全体
- 性能
  - std::count/std::find：範囲の個数の比例
  - メンバ関数 count：要素数の対数と重複数に関係
  - メンバ関数 find：要素数の対数に関係

count/find の使用例比較

- 探索範囲が全体ならばメンバ関数の指定がシンプル
- 性能もメンバ関数の方が良い

```
#include <algorithm>
#include <iostream>
#include <set>
int main()
{
   std::multiset x {1,2,3,1,2,2,3,2};
   std::cout<< std::count(x.begin(),x.end(),2)<<"\n";
   std::cout<< x.count(2) <<"\n";

   auto it {x.begin()};
   it = std::find(x.begin(), x.end(), 3);
   if (it != x.end()) std::cout << *it <<"\n";
   it = x.find(3);
   if (it != x.end()) std::cout << *it <<"\n";
}
```

search_n の使い方

- 連続する要素の探索
- it search_n(b, e, c, v)
- [b, e):探索範囲, c:連続回数, v:要素値
- 戻り値はイテレータ
  - 見つけた並びの先頭, c<=0 ならば b, それ以外は e

```
#include <algorithm>
#include <iostream>
#include <vector>
int main()
{
   std::vector a {1,2,3,2,2,2,3,4};
   int c{3}, v{2};
   auto it { std::search_n(a.begin(), a.end(),c,v) };

   if (it != a.end())
      std::cout <<"found\n";
}
```

search_n の処理

- 先頭を見つけてから c 回だけ数える
- 繰り返しが 1 回の場合もある
- 二重ループで実装してもよい

```
template<typename It, typename T>
It search_n(It b, It e, int c, const T& v) {
    if (c <= 0) return b;
    int count{0};
    It head{b};
    for (It it=b; it!=e; ++it) {
        if (*it == v) {
            if (count == 0) head = it; // 先頭
            ++count;
            if (count == c) return head; // 条件を満たす
        } else {
            count = 0; // やり直し
        }
    }
    return e;
}
```

search の使い方 1

- 部分範囲の探索
- `it search(b1, e1, b2, e2)`
- 探索範囲 `[b1, e1)` から部分範囲となる `[b2,e2)` を探す
- 見つけた範囲の先頭のイテレータ, または `e1` を返す
- `==` 演算子で判定

```
int main() {
   std::vector a{3,2,4,6,8,1,2,3,4}, s{1,2,3};

   auto it { std::search(a.begin(), a.end(),
                         s.begin(), s.end()) };
   if (it != a.end())
      std::cout <<"found at "
                << it-a.begin() <<"\n"; // 5
}
```

search の使い方 2

- it search(b1, e1, b2, e2, p)　　接尾辞が_if ではない
- 探索範囲 [b1, e1) から部分範囲となる [b2,e2) を探す
- 見つけた範囲の先頭のイテレータ, または e1 を返す
- p(述語) 関数で判定

```
bool pred(int x, int y) { return x % y == 0; }

int main()
{
   std::vector a{3,2,4,6,8,1,2,3,4}, s{1,2,3};

   auto it {std::search(a.begin(), a.end(),
                        s.begin(), s.end(), pred) };

   if (it != a.end())
      std::cout <<"found at "
                << it-a.begin() <<"\n"; // 1
}
```

search の処理

- 二重ループで探す方法が直感的
- もっと効率の良い方法がある
  - Boyer-Moore 法, Boyer-Moore-Horspool 法
  - C++17 でも利用できる（この講義では省略）

```
template<typename T, typename K>
T search(T b, T e, K b2, K e2) {
   for (T it=b; it!=e; ++it) {
      T s1{it}; // it は候補の先頭
      for (K s2=b2; ; ++s1, ++s2) {
         if (s1 == e)  return e;  // なし
         if (s2 == e2) return it; // あり
         if (!(*s1 == *s2)) break;
      }
   }
   return e;
}
```

find_end の使い方

- 最後尾にある部分範囲の先頭
- `it find_end(b1, e1, b2, e2)`,
  `it find_end(b1, e1, b2, e2, p)`　　接尾辞が`_if`ではない

```
std::vector a{3,5,1,2,3,8,1,2,3}, s{1,2,3};

auto it { std::find_end(a.begin(), a.end(),
                        s.begin(), s.end()) };
if (it != a.end())
   std::cout <<"found at "
             << it-a.begin() <<"\n"; // 6
```

find_first_of の使い方

- 範囲中のどれかを見つける
- `it find_first_of(b1, e1, b2, e2)`,
  `it find_first_of(b1, e1, b2, e2, p)`
- 探索範囲 `[b1, e1)` から別範囲 `[b2,e2)` のどれかを探す
- 見つけた要素のイテレータ, または `e1` を返す
- == 演算子または関数 `p`(述語) で判定

```
std::vector a{3,5,1,2,3,8,1,2,3}, s{7,8,9};

auto it { std::find_first_of(
                a.begin(), a.end(),
                s.begin(), s.end()) };

if (it!=a.end())
   std::cout << *it <<"\n"; // 8
```

adjacent_find の使い方

- 同値で隣接する要素を探す
- `it adjacent_find(b, e)`,
  `it adjacent_find(b, e, p)`
- 見つけた要素のイテレータ, または `e` を返す
- ==演算子または関数 `p`(述語) で判定

```
std::vector a {1,2,3,4,5,5,6,7,8};

auto it { std::adjacent_find(a.begin(), a.end()) };

if (it != a.end())
  std::cout << *it <<"\n"; // 5
```

adjacent_find の処理

- 範囲に要素がない場合を先に調べる
- 二つ 2 のイテレータを更新して比較する

```
template<typename T>
T adjacent_find(T b, T e) {
    if (b == e) return e;
    T prev {b};
    ++b;
    for (T it=b; it!=e; ++prev, ++it)
        if (*prev == *it) return prev;
    return e;
}
```

条件のまとめ

アルゴリズム	イテレータ	比較演算
count	入力	==
count_if	入力	指定関数
min_element	前方	<
max_element	前方	<
minmax_element	前方	<
find	入力	==
find_if	入力	指定関数
find_if_not	入力	指定関数
search_n	前方	==
search	前方	==, 指定関数
find_end	前方	==, 指定関数
find_first_of	前方	==, 指定関数
adjacent_find	前方	==, 指定関数