C++プログラミング II

第 8 回 STL の概要とイテレータ

岡本秀輔

成蹊大学理工学部

STL: Standard Template Library の概要

STL とは

- C++を支える標準ライブラリ
- ISO C++標準となっている
- ジェネリック（包括的）プログラミングに基づく
  - 特定の型に依存せず、抽象的／汎用的にコードを記述
  - 包括的なライブラリの組み合わせで具体的なコード作成
- 基本的なテクニックがライブラリ化されている
  - 結果プログラムの性能と安定性の双方が高くなる
  - 「車輪の再発明」を防げる
    - ただし学修にはしばしば必要
- 四つの構成要素
  - コンテナ、イテレータ、アルゴリズム、関数オブジェクト

STL の構成要素

- コンテナ（入れ物）
  - オブジェクトの集まりを管理する（データ構造）
  - 用途別にいくつかの入れ物が用意されている
- イテレータ（反復子）
  - コンテナの中身をループで１個ずつ使用する枠組み
  - コンテナとアルゴリズムの組み合わせの接合部となる
  - ポインタと類似のインタフェース
- アルゴリズム
  - コンテナに対する処理を提供する
  - 例：探索、整列、修正など
  - イテレータを通してコンテナ要素にアクセス
- 関数オブジェクト
  - アルゴリズムの調整に使う
  - 関数のように呼び出せるオブジェクト
  - ラムダ式の値も関数オブジェクト

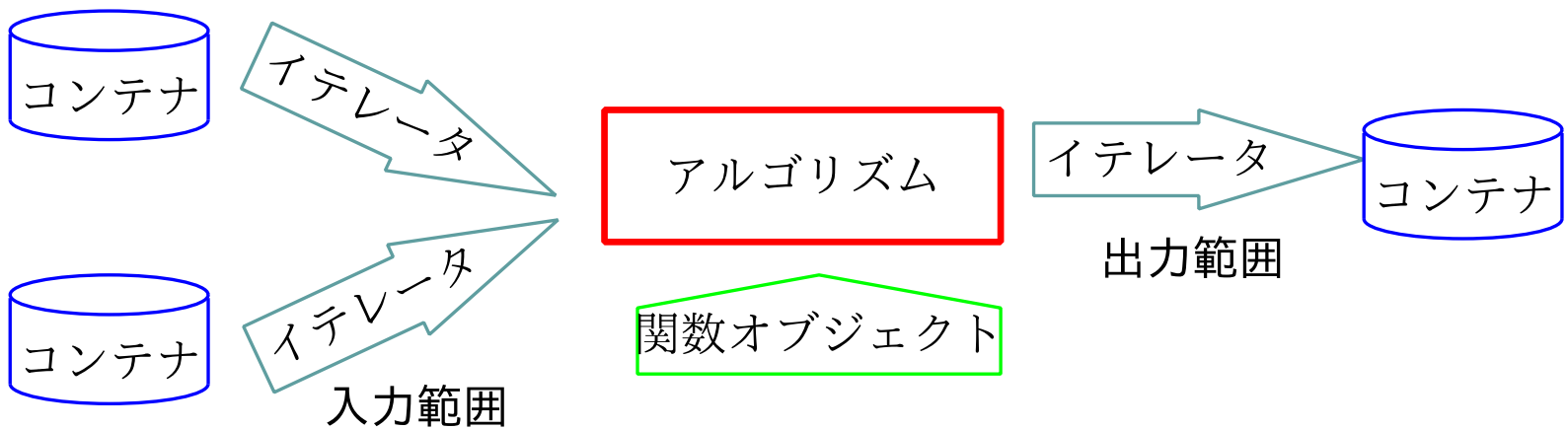
これまでに見たSTLの構成要素

std::の付く多くクラステンプレートや関数テンプレート

- コンテナ
  - std::vector, std::deque, std::list, ...
  - std::set, std::map, ...
- イテレータ
  - コンテナ変数 v の v.begin() や v.end() の戻り値
  - std::find(), マップ変数 m の m.find(x) の戻り値
- アルゴリズム
  - std::sort(), std::find()
- 関数オブジェクト
  - なし

ライブラリの設計概念

- ジェネリックプログラミング
  - データ型と操作の切り離し
    - 操作は抽象的なオブジェクトに対する指示
    - オブジェクトには共通のメンバ関数や演算子が必要
    - オブジェクト指向プログラミングと矛盾しない
  - 具体化する際に組み合わせられる
- 全体構成と役割
  - データはコンテナで管理
  - 操作はアルゴリズムで提供, 関数オブジェクトで調整
  - イテレータでコンテナとアルゴリズムをつなぐ



イテレータ

イテレータの役割と基本操作

- iterate: 繰り返す、反復する
- コンテナ要素を個別に操作するためのオブジェクト
  - 各コンテナ用のクラス
  - 要素の挿入や削除で重要な役割を果たす
- コンテナ上の要素の場所を表す
  - ポインタと同じ概念
- イテレータ変数 `it` に対する基本操作

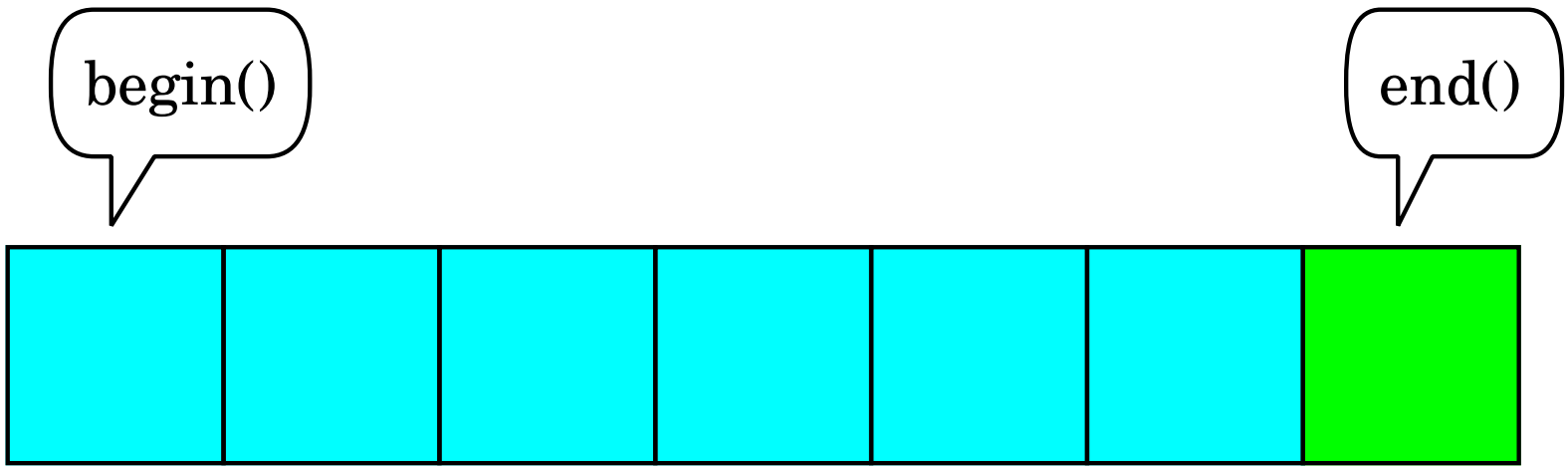
`*it`	it の指す場所のオブジェクトを返す
`it->x`	ポインタと同じく `(*it).x` を意味する
`++it, it++`	次の要素の場所を指すよう更新
`==, !=`	同じ場所を指しているかどうか（比較）
`=`	指し示す場所の更新（代入）

これらに加えてデフォルト Ctor とコピー Ctor がある

`--,<,<=,>,>=`はイテレータの種類によって使用の可否が変わる

コンテナとイテレータ

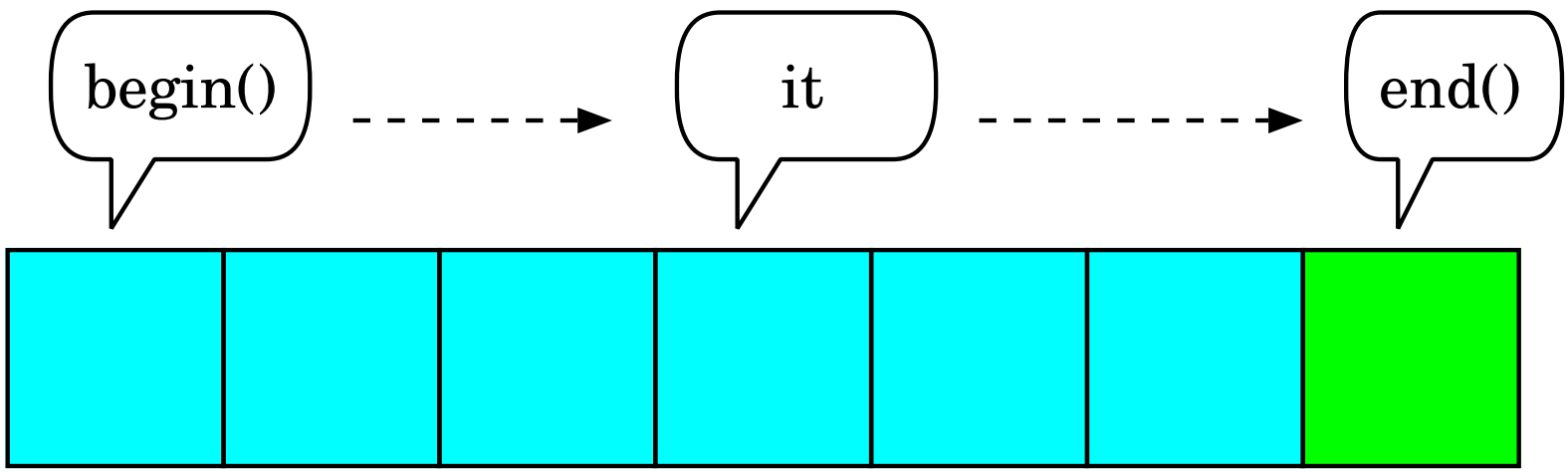
- 全てのコンテナは以下のメンバ関数を提供する
  - `begin()`：先頭要素の場所のイテレータを返す
  - `end()`：末尾要素の次の場所のイテレータを返す
- `end()` が存在しない要素の場所を指す理由
  - コンテナが空の場合には `begin()==end()` となる
  - `it != end()` で要素をあることを示す



イテレータの基本ループ

- ループの終わりのチェックは!=を使う
- イテレータの更新は前置きの++を使う
  - 後ろ置きの it++はわずかに無駄が生じる
- すべてのコンテナがこの形でループが指定できる

```
for (auto it = x.begin(); it != x.end(); ++it)
   cout << *it << "\n";
```



範囲 for 文

- 範囲 for 文はイテレータの基本ループを簡易に書くために C++11 から導入された
- 以下のループはほぼ同じ内容

```
std::list c {1,2,3,4,5};

for (auto e : c)
    cout << e <<"\n";

for (auto it = c.begin(); it != c.end(); ++it) {
    auto e = *it;
    cout << e <<"\n";
}
```

イテレータの型

- コンテナごとにイテレータの型がある
- コンテナの型を T とすると、次の 2 つ
  - T::iterator
  - T::const_iterator
- auto を使うならば深く考えなくても

```
auto it { c.begin() };

for (auto& e : c) e += 2;

for (const auto& e : c) cout << e;
```

イテレータの使用例

std::find() 関数

- iterator を返す標準関数
  - #include<algorithm>
- 引数：探索範囲の開始位置, 終了位置, 探す値
- 終了位置を返すと “ not found ” の意味

```
std::list x {3,5,2,8,9,6,4};

auto it { std::find(x.begin(), x.end(), 9) };
if (it != x.end()) {
   std::cout << *it <<"\n";
   x.insert(it, 10); // 9 の位置に 10 を挿入
}

for (auto e : x) // 確認
   std::cout << e <<" ";
std::cout <<"\n";
```

insert メンバ関数

- 挿入でイテレータが無効となるコンテナがある
  - vector, deque, unordered_*は無効の可能性がある
- insert() は挿入要素のイテレータを返し、続行できる

```
std::vector x {10,3,10,8,10,10,4};

// すべての 10 の前に 5 を挿入
for (auto it = x.begin(); it != x.end(); ++it) {
   if (*it == 10) {
      it = x.insert(it, 5); //  5 のイテレータ
      ++ it;                // 10 のイテレータ
   }
}

for (auto e : x) // 確認
   std::cout << e <<" ";
std::cout <<"\n";
```

erase メンバ関数

- 削除で以前のイテレータが無効になるコンテナがある
- erase() は削除対象の次の要素のイテレータを返す
  - vector, deque は割り当て直されたイテレータ

```
std::vector x {3,5,3,8,3,6,3};

// すべての 3 を削除
auto it { x.begin() };
while (it != x.end()) {
   if (*it == 3)
      it = x.erase(it); // 削除してその次を返す
   else
      ++it;             // 次の要素へ
}

for (auto e : x)
   std::cout << e <<" ";
std::cout <<"\n";
```

equal_range メンバ関数

- set, map, multimap, unordered_*のメンバ関数
- 同じキーの要素の範囲を
  std::pair<iterator,iterator> で返す
  - multiset, multimap, unordered_multimap で有用

```
std::multimap<std::string,int> x
{ {"xn",10},{"ya",5},{"xn", 5},{"zn",3},
  {"xn",36},{"sa",5},{"xn",24},{"tn",3}};

auto r {x.equal_range("xn")};
for (auto it = r.first; it != r.second; ++it) {
   auto [k,v] {*it};
   std::cout <<"["<< k <<","<< v <<"]";
}
std::cout <<"\n"; // [xn,10][xn,5][xn,36][xn,24]
```

lower_bound/upper_bound メンバ関数

- set, map, multimap, unordered_*のメンバ関数
- キーの下限と上限のイテレータを返す
  - lower_bound: 指定キーより前でない要素
  - upper_bound: 指定キーより後でない要素

```
std::map<std::string,int> x
{ {"e",10},{"a",5},{"i", 5},{"h",3},
  {"f",36},{"b",5},{"d",24},{"g",3}};

auto l {x.lower_bound("c")}; // "c"はない
auto u {x.upper_bound("f")};
for (auto it = l; it != u; ++it) {
   auto [k,v] {*it};
   std::cout <<"["<< k <<","<< v <<"]";
}
std::cout <<"\n"; // [d,24][e,10][f,36]
```

イテレータの分類

概要

- コンテナごとにできる範囲が異なる
- 例：
  - `vector`,`deque` はランダムに要素にアクセスできる
  - `list` は先頭または末尾から一つずつ見るのみ
  - `set`, `map`, `unordered_*`は基本的に要素の探索
  - `forward_list` は前から探すのみ
- イテレータの分類

  前方 :前から順にアクセス
  双方向 :前からと後ろからの双方向のアクセス
  ランダム :どの要素でも自由にアクセス
- 機能の包含関係
  - 前方 $\subset$ 双方向 $\subset$ ランダム

厳密には連続もあるがここでは省略

分類表

- イテレータが持つ演算子による分類

	==, != ++, *, ->	--	[],+,-,+=,-= <,<=,>,>=	コンテナ
前方	○	×	×	unordered_*, foward_list
双方向	○	○	×	list, set, map multi*
ランダム	○	○	○	vector, array deque

- イテレータの [] 演算子

```
std::vector x {1,2,3,4};
auto it {x.begin()};
std::cout << it[2] <<"\n"; // 3
```

イテレータの演算

イテレータ it に対する演算

- ++it：次, --it：一つ前
- 整数 n との演算
  - it+n：n 個後ろのイテレータ
  - it-n：n 個前のイテレータ
  - it+=n：n 個後ろへ更新
  - it-=n：n 個前へ更新
- イテレータどうしの減算
  - n = it2 - it1：要素間の距離（n は整数)
- 大小比較：順序判定

```
std::vector x {1,2,3,4};
auto it  {x.begin()};
auto it2 { it+2 };
std::cout << *(it+1) <<" "<< *it2 <<"\n"; // 2 3
if (it < it2) std::cout <<"true\n"; // true
```

補助関数

補助関数

- <iterator>ヘッダファイル
- ランダムイテレータ以外はは線形時間の操作
  - 線形時間の操作：一つ一つたどること
- 前方イテレータに対する指示の制限を緩和する

関数 (すべて std::*)	動作	使用条件
advance(p,n)	p+=n	n<0 は双方向のみ
n=distance(p,q)	n=q-p	p から q に到達可能
q=next(p,n)	q=p+n	advance と同じ
q=next(p)	q=next(p,1)	advance と同じ
q=prev(p,n)	q=p-n	双方向のみ
q=prev(p)	q=prev(p,1)	双方向のみ

p，q:イテレータ, n:整数

advance/distance

- 一つ飛ばしの処理

```
#include <iostream>
#include <iterator>
#include <forward_list>
int main()
{
   std::forward_list x {1,2,3,4,5,6,7,8,9,10};
   for (auto it {x.begin()};
        it !=x.end(); std::advance(it, 2)) {
      std::cout << std::distance(x.begin(), it)
                <<":"<< *it <<" ";
   }
   std::cout <<"\n"; // 0:1 2:3 4:5 6:7 8:9
}
```

x.end() を越えない配慮が必要

next

- 10 個飛ばした処理

```
#include <iostream>
#include <list>
int main()
{
   std::list<double> d;
   for (int i = 0; i < 30; i++)
      d.push_back(i*1.5);

   double sum{0};
   for (auto it {std::next(d.begin(), 10)};
        it != d.end(); ++it) {
      sum += *it;
   }
   std::cout << sum <<"\n";
}
```

prev/next

- 両端を除いた処理

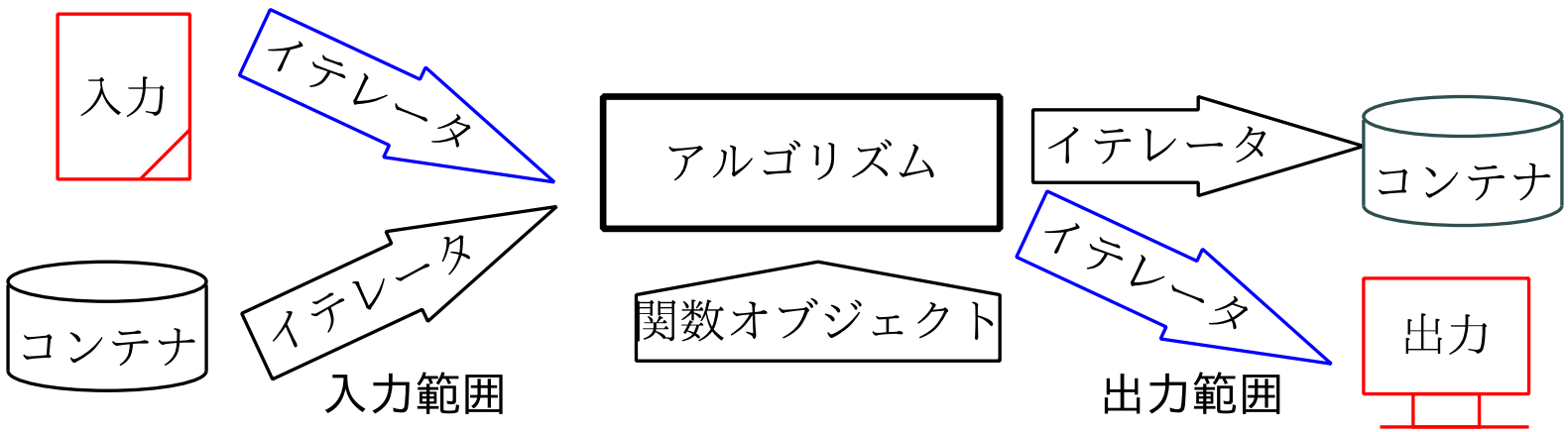
```
#include <iostream>
#include <list>
int main()
{
   std::list<double> d;
   for (int i = 1; i < 11; i++)
      d.push_back(i);

   double sum{0};
   auto start {std::next(d.begin())};
   auto end   {std::prev(d.end())  };
   for (auto it = start; it != end; ++it)
      sum += *it;
   std::cout << sum <<"\n";
}
```

ストリーム イテレータ

コンテナ以外のイテレータ

- 標準入出力、ファイル、string はストリーム
  - `std::istream cin; std::ifstream fin; std::istringstream iss;`
  - `std::ostream cout; std::ofstream fout; std::ostringstream oss;`
- ストリーム用のイテレータがあれば、アルゴリズムが入出力とコンテナを区別しなくても良くなる。



ストリーム イテレータ

前方イテレータをさらに制限したイテレータ

```
#include <iterator>
```

- 入力イテレータ
  - ++, ==, !=演算子
  - *演算子による一度だけの読み出し（入力となる）
  - コピー Ctor(ストリームの種類を決める）
  - デフォルト Ctor が入力ストリームの終わりの印となる
- 出力イテレータ
  - ++演算子
  - *演算子の結果へ代入（出力となる）
  - コピー Ctor(ストリームの種類を決める）
  - 出力には終わりがないので比較はしない

厳密にはちょっと異なるが詳細は省略

入力イテレータの例

- 入力処理がコンテナと同じループで指定できる
- デフォルト Ctor の eos がストリームの終わりを示す
  end of stream の意味の変数名

```
#include <iostream>
#include <iterator>
int main()
{
    std::istream_iterator<int> it{std::cin};
    std::istream_iterator<int> eos;

    int sum{0};
    for (; it != eos; ++it)
        sum += *it;                    // 入力
    std::cout << sum <<"\n";
}
```

```
$ echo 1 2 3 4 5 | ./a.out
15
```

出力イテレータの例

- Ctor の第 2 引数は出力の区切り文字列
- この例は int 専用なので最後の改行には別手段が必要

```
#include <iostream>
#include <iterator>
int main()
{
    std::ostream_iterator<int> o{std::cout, ", "};
    *o = 1;
    ++o;
    *o = 2;
    ++o;
    *o++ = 3;
    *o++ = 4;
}
```

```
$ ./a.out
1, 2, 3, 4,
```

補足

前置きと後置き++の違い

- 前置きは更新してからその値を返す
- 後置きは更新するが、以前の値を返す
  - 一時変数を必要とする
- コード:
  - `++x：{ x=x+1; return x; }`
  - `x++：{auto a{x}; x=x+1; return a; }`

Python 風 Range クラス

```
class Itr {
    int val;
public:
    Itr(int v):val(v){}
    int  operator*()  { return val; }
    void operator++() { ++val; }
    bool operator!=(const Itr& e){return val!=e.val;}
};
class Range {
    Itr bv, ev;
public:
    Range(int b,int e) :bv{b}, ev{e}{}
    auto begin() { return bv; }
    auto end()   { return ev; }
};

int main() {
    for (auto e : Range{1,10}) // 使用例
        std::cout << e <<"\n";
}
```