

# C++プログラミングII

## 第5回 連続コンテナ

岡本秀輔

成蹊大学理工学部

準備

# コマンド引数

- ▶ argc はコマンドに対する引数の数
- ▶ argv の使い方: コマンド引数の取得
  - ▶ argv[0] : コマンド名 (実行ファイル名)
  - ▶ argv[n] : n 個目のコマンドへの引数 (ただし n>0)
  - ▶ 要素は文字列を指すポインタ (以下の使い方を推奨)

```
int main(int argc, char *argv[]) {  
    // 1 個目のコマンドライン引数を整数で取り出す  
    int n { argc>1 ? std::atoi(argv[1]):0 };  
    // コマンド名以外のすべてを vector に取り込む  
    std::vector<std::string> a{argv+1, argv+argc};  
    std::cout << n+1 << "\n";  
    for (auto e : a) std::cout << e << " ";  
    std::cout << "\n";  
}
```

```
$ ./a.out 123 45 6 7  
124  
123 45 6 7
```

# 一様乱数クラス

- ▶ サイコロのように出現確率の等しい乱数
- ▶ 使用方法:
  - ▶ `UniDist a{1,6};` // 範囲を指定して変数宣言
  - ▶ `std::cout << a.get();` // `get` で乱数を得る

ヘッダファイル : `random.hpp`

```
#include <random>
class UniDist {
    std::random_device seed;
    std::mt19937 engine;
    std::uniform_int_distribution<int> udist;
public:
    UniDist(int first, int last) // [first, last]
        :seed{}, engine{seed()}, udist{first,last}{}
    auto get(){ return udist(engine); }
};
```

MT19917 は日本人が作った世界で最も使われている疑似乱数です。

# 指数分布クラス

- ▶ 指数分布（事象が起こる時間間隔）の乱数
- ▶ `ExpDist a{3.0, 60};` // 60 分に 3 回起こる条件
- ▶ `std::cout << a.get();` // 平均 20 を期待する乱数

ヘッダファイル : `random.hpp`

```
#include <random>
class ExpDist{ // unit 時間に times 回発生する条件の乱数
    std::random_device seed;
    std::mt19937 engine;
    std::exponential_distribution<double> edist;
    double unit;
public:
    ExpDist(double lambda, double u =1.0)
        :seed{},engine{seed()},edist{lambda},unit{u}{}
    // 次に起こるまでの時間
    auto get(){ return edist(engine)*unit; }
};
```

# コンテナの概要

# コンテナの種類

- ▶ オブジェクトの集まりを管理する（データ構造）
- ▶ 連続コンテナ
  - ▶ データを一行に並べた入れ物
  - ▶ データの置き場所が挿入した時や指定順に依存する
  - ▶ 汎用 : `vector`, `deque`, `list`
  - ▶ 省メモリ : `array`, `forward_list`
  - ▶ 限定操作 : `stack`, `queue`
- ▶ 連想コンテナ（第6回目の講義で扱う）
  - ▶ データを素早く探すための入れ物
  - ▶ 整列連想コンテナ
    - ▶ `set`, `multiset`, `map`, `multimap`
  - ▶ 順序無し連想コンテナ
    - ▶ `unordered_set`, `unordered_multiset`,  
`unordered_map`, `unordered_multimap`

# ヘッダファイル

- ▶ 連続コンテナはコンテナ名のヘッダファイル
- ▶ 連想コンテナはいくつかが共通のヘッダファイル

インクルードの指定	コンテナ名
<code>&lt;vector&gt;</code>	<code>std::vector</code>
<code>&lt;deque&gt;</code>	<code>std::deque</code>
<code>&lt;list&gt;</code>	<code>std::list</code>
<code>&lt;set&gt;</code>	<code>std::set</code> , <code>std::multiset</code>
<code>&lt;map&gt;</code>	<code>std::map</code> , <code>std::multimap</code>
<code>&lt;unordered_set&gt;</code>	<code>std::unordered_set</code> , <code>std::unordered_multiset</code>
<code>&lt;unordered_map&gt;</code>	<code>std::unordered_map</code> <code>std::unordered_multimap</code>
<code>&lt;stack&gt;</code>	<code>std::stack</code>
<code>&lt;queue&gt;</code>	<code>std::queue</code>
<code>&lt;array&gt;</code>	<code>std::array</code>
<code>&lt;forward_list&gt;</code>	<code>std::forward_list</code>



# 共通の操作

- ▶ コンストラクタ (Ctor) の形式, 代入, 比較, 範囲 for 文, swap, empty, size, clear が共通の操作
  - ▶ unordered\_\*には大小比較がない(==, !=もない)
  - ▶ forward\_list に size なし
  - ▶ array は clear なし、Ctor にも制限あり
  - ▶ stack, queue は別名操作ばかり

```
template<typename T>
void common(T a, T b) {
    T c{a};
    if (!a.empty())
        for (auto& e:a)
            std::cout << e << " ";
    a.swap(b);
    c = b;
    if (c == b) std::cout << "ok\n";
    if (a < b) std::cout << "a<b:"; // 除く unordered_*
    std::cout << a.size() << "\n"; // 除く forward_list
    a.clear(); // 除く array
}
```

## 共通の操作 (つづき)

```
int main() {  
    std::vector<int>    v1{1,2},    v2{1,3,5};  
    common(v1, v2);  
    std::deque<int>    d1{1,2},    d2{1,3,5};  
    common(d1, d2);  
    std::list<int>     l1{1,2},    l2{1,3,5};  
    common(l1, l2);  
    std::set<int>      s1{1,2},    s2{1,3,5};  
    common(s1, s2);  
    std::multiset<int> ms1{1,2}, ms2{1,3,5};  
    common(ms1, ms2);  
    std::map<int,int>  m1{{1,3},{2,1}},  
                        m2{{1,4},{3,2},{5,3}};  
    common(m1, m2);  
    std::multimap<int,int> mm1{{1,3},{2,1}},  
                             mm2{{1,4},{3,2},{5,3}};  
    common(mm1, mm2);  
}
```

# 内部構造とインタフェース

## ▶ 内部構造

- ▶ 実装に使用すべきデータ構造には決まりはない
- ▶ 想定される形式
  - ▶ 連続コンテナ：配列または連結リスト
  - ▶ 整列連想コンテナ：2 分探索木
  - ▶ 順序無し連想コンテナ：ハッシュ表

## ▶ インタフェース

- ▶ 共通メンバ関数の使用すれば、コンテナを入れ替えたプログラムがそのまま動作する
- ▶ どれもテンプレートクラスを使って実装されている
- ▶ 動作やメモリ使用量に差がある

# 連続コンテナ

# 基本三種の特徴

vector : 最もよく使われる

- ▶ 要素はメモリ上に並ぶ
- ▶ 要素にランダムなアクセスが可能
- ▶ 先頭や途中への挿入/削除に時間がかかる

deque : 機能が充実し性能もそこそこ良い

- ▶ 要素は**ほぼ**メモリ上に並ぶ
- ▶ 要素にランダムなアクセスが可能
- ▶ 先頭と末尾への挿入/削除の時間が一定

list : 上記二つは異なる場面で使う

- ▶ 要素はメモリ上に並ばないことが前提
- ▶ 先頭または末尾からたどる（双方向）
- ▶ メモリのオーバヘッドが大きい
- ▶ 挿入削除の時間が一定（検索は遅い）
- ▶ 固有のメンバ関数が多数

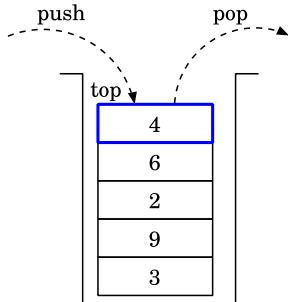
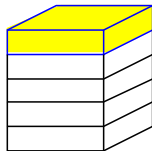
# 変種

- ▶ 特定のメンバ関数に絞ったもの
  - ▶ コンテナアダプタ: 他のコンテナが土台
    - stack : 積み上げ型データ構造 (LIFO)
    - queue : 待ち行列型データ構造 (FIFO)
- ▶ 用途を限ってメモリ使用量を削減
  - array : **vector の固定サイズ版**
    - ▶ 要素数はコンパイル時に固定 (変更不可)
    - ▶ メモリ使用量が最小
  - forward\_list : **単方向の list**
    - ▶ 先頭からたどることに特化している
    - ▶ メモリ使用量が `std::list` より少ない
    - ▶ `std::list` と同じ固有のメンバ関数
    - ▶ 先頭以外の挿入削除操作が独自形式

`std::stack`

# スタックとは

- ▶ 英単語 stack : (干し草、本などの) 山、積み重ね
- ▶ データ構造としての特徴
  - ▶ LIFO: Last In First Out
  - ▶ 「後入れ先出し」で使う
  - ▶ 積み上げて一番上から処理する
- ▶ `std::stack` の操作
  - ▶ `push` : データを一番上に積む
  - ▶ `pop` : 一番上のデータを取り去る
  - ▶ `top` : 一番上のデータにアクセス
  - ▶ `empty` : スタックが空かどうか
  - ▶ `size` : スタック中のデータ数
- ▶ `#include<stack>` ヘッダファイル





# 入力の逆順出力

- ▶ 入力データをスタックに積み上げ
- ▶ 上から出力と取り出しを繰り返す

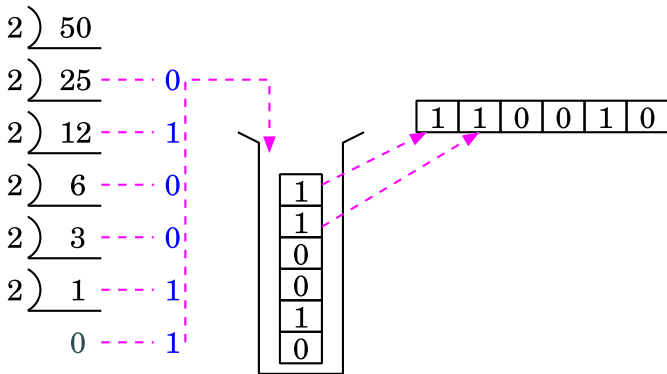
```
#include <iostream>
#include <stack>
int main() {
    std::stack<int> s;
    for (int x; std::cin >> x; ) s.push(x);

    while (!s.empty()) {
        std::cout << s.top() << " ";
        s.pop();
    }
    std::cout << "\n";
}
```

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

# 10進2進変換

- ▶ 10進数の数を2進数に変換する
- ▶ 手順
  - ▶ 2で割ったあまりを push していく
  - ▶ pop しながら出力していく



## 10進2進変換のコード

```
void dec2bin(size_t dec) {
    std::stack<size_t> s;
    while (dec != 0) {
        s.push(dec % 2);
        dec /= 2;
    }

    while (!s.empty()) {
        std::cout << s.top();
        s.pop();
    }
    std::cout << "\n";
}

int main() {
    for (size_t x; std::cin >> x; )
        dec2bin(x);
}
```

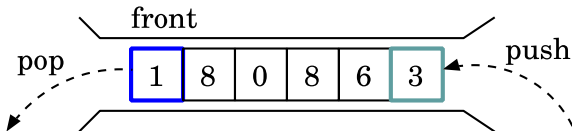
# 実行例

```
$ echo 50 158 224 329 | ./a.out  
110010  
10011110  
11100000  
101001001
```

`std::queue`

# キューとは

- ▶ 英単語 queue : (待っている) 人々, 待ち行列
- ▶ データ構造としての特徴
  - ▶ FIFO: First In First Out
  - ▶ 「先入れ先出し」で使う
  - ▶ 最後尾に入れて、先頭から処理
- ▶ `std::queue` の主な操作
  - ▶ `push` : データを最後尾に入れる (enqueue)
  - ▶ `pop` : 先頭のデータを取り去る (dequeue)
  - ▶ `front` : 先頭のデータにアクセス
  - ▶ `empty` : 空かどうか
  - ▶ `size` : データ数
- ▶ `#include<queue>` ヘッダファイル



# キューが使われる場面

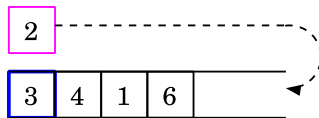
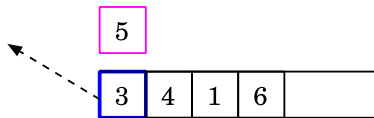
- ▶ 一つの資源を複数の消費者が順番に消費する状況
  - ▶ 待ち行列のシミュレーション
  - ▶ CPU スケジューリング,
  - ▶ プリンタの印刷待ち
- ▶ 連続的な入力に対して、断続的でまとまった出力
  - ▶ 工場からの配送
  - ▶ 入出力バッファ
  - ▶ Unix パイプ



# サイコロの目とキュー

以下のアルゴリズムを作ってみる

1. サイコロを  $n$  回振ってキューに入れる
2. サイコロを振る
3. キューの先頭と偶奇が同じならば先頭を取り出す
4. キューの先頭と偶奇が異なるならば最後尾に挿入
5. キューが空ならば 6 へ, または 2 に戻る
6. 2 以降の繰り返し回数を出力





# コードと実行例

```
#include <iostream>
#include <queue>
#include "random.hpp"
int main(int argc, char *argv[]) {
    int n { argc>1 ? std::atoi(argv[1]):3 };
    std::queue<int> q;
    UniDist rnd{1,6}; // 1 から 6 の一様乱数
    for (int i = 0; i < n; i++)
        q.push( rnd.get() );
    n = 0;
    while (!q.empty()) {
        ++ n;
        auto x { rnd.get() };
        if (x%2 == q.front()%2)
            q.pop();
        else
            q.push( x );
    }
    std::cout << n << "\n";
}
```

実行例

```
$ ./a.out
17
$ ./a.out 5
13
$ ./a.out 5
99
$ ./a.out 10
496
$ ./a.out 10
142
```

# クリニックの待ち行列シミュレーション

- ▶ 想定する状況
  - ▶ 患者が平均 10 分間隔でランダムに来る
  - ▶ 一人の医者が患者一人を平均 8 分で診察する
  - ▶ ある程度時間が経った後に何人待ちが想定されるか?
- ▶ M/M/1 待ち行列モデル
  - ▶ 確率論で扱われるシンプルかつ重要なモデル
  - ▶ 解析的に何人分の待ちが必要かを計算できる
  - ▶ 上記は指数分布を仮定すると 4 人分の待ち (32 分待つ)
- ▶ シミュレーションプログラム
  - ▶ 指数分布の乱数 (ExpDist クラス)
  - ▶ 単位時間当たりのイベント回数を指定して、イベントが次に起こる時間を乱数で得る
  - ▶ 「平均 10 分間隔で来る」 → 「1 時間で 6 人来る」
  - ▶ 「一人平均 8 分で診察」 → 「1 時間で 7.5 人診察」

## simulate関数

```
// 引数の数の患者の診察が終了した時点の待ち人数
int simulate(int num_patient =100) {
    ExpDist next_patient {6.0}; // 次の患者到着時間
    ExpDist clinical_time{7.5}; // 診察時間
    double arrival{ next_patient.get() };
    std::queue<double> q; // 先頭は診察中，他が待ち
    while (num_patient > 0) {
        if (q.empty() || arrival < q.front()) {
            if (!q.empty()) q.front() -= arrival;
            q.push( clinical_time.get() ); // 到着
            arrival = next_patient.get();
        } else {
            arrival -= q.front();
            q.pop(); // 診察終了
            -- num_patient;
        }
    }
    return q.size();
}
```

# main と実行結果

```
int main() {  
    const int N{30}; // N回試行  
    double sum{};  
    for (int i = 0; i < N; i++) {  
        auto x {simulate()};  
        std::cout << x << " ";  
        sum += x;  
    }  
    std::cout << "\n" << sum/N << "\n";  
}
```

\$ ./a.out

0 2 3 4 4 6 16 2 1 3 2 0 3 4 0 0 4 10 0 4 4 2 2 3 4 5 4 3 6 0  
3.36667

\$ ./a.out

1 2 9 3 5 3 4 2 2 1 12 6 2 2 3 1 5 0 2 0 6 0 0 2 4 2 4 8 0 5  
3.2

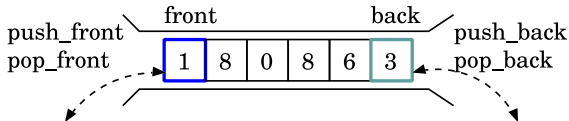
\$ ./a.out

1 17 5 10 8 1 19 1 6 0 10 1 2 1 13 3 1 1 3 3 6 0 0 7 7 4 2 0 1 6  
4.63333

`std::deque`

# デックとは

- ▶ deque : double-ended queue
  - ▶ 両端キュー (プログラミング技術上の造語?)
- ▶ データ構造としての特徴
  - ▶ 先頭末尾の挿入削除が一定時間、その他は時間がかかる
  - ▶ 添字を使ったアクセスが可能
- ▶ `std::deque` の主な操作 (vector+アルファ)
  - ▶ `push_front/push_back` : 挿入
  - ▶ `pop_front/pop_back` : 削除
  - ▶ `operator[]` : 任意の場所のアクセス
  - ▶ `front/back` : 先頭/末尾のアクセス
  - ▶ `empty, size` 他
- ▶ `#include<deque>` ヘッダファイル



# デックが使われる場面

- ▶ アプリの undo-redo 機能（回数に上限あり）
  - ▶ undo 用と redo 用の二つのスタック
  - ▶ undo: コマンドの取り消し
  - ▶ redo: コマンドの再実行
  - ▶ どちらもコマンド履歴を保存し、ある回数まで達したら古いものから消していく
  - ▶ サイズ無制限のスタックを用意すれば undo スタックは最終結果を保持する
- ▶ ブラウザの履歴
  - ▶ undo-redo と同じくある回数まで保存したところで古い履歴を消していく
- ▶ マルチプロセッサスケジューリング
  - ▶ CPU ごとにキューを持つ
  - ▶ キューが空の CPU は他の CPU キューの末尾からもらう
  - ▶ A-Steal アルゴリズムと呼ばれる

# 回文チェック

- ▶ 回文：前から読んでも後ろから読んでも同じ
  - ▶ 例：しんぶんし, たけやぶやけた, civic, level, radar
- ▶ すべて挿入した後に、両端をチェックしながら削除

```
bool is_palindrome(std::string s) {  
    std::deque<char> d;  
    for (auto ch : s)  
        d.push_back(ch); // 一文字ずつ挿入  
    while (d.size() > 1) {  
        if (d.front() != d.back()) // 両端の文字を比較  
            return false;  
        d.pop_front();  
        d.pop_back();  
    }  
    return true;  
}
```

再帰呼び出しの例があったことを思い出そう



## まとめ

# まとめ

- ▶ コンテナの概要
  - ▶ 連続コンテナ
  - ▶ 連想コンテナ
- ▶ `std::stack` 後入れ先出し
- ▶ `std::queue` 先入れ先出し
- ▶ `std::deque` 両端キュー