

C++プログラミングII

第4回 vector の機能と構造

岡本秀輔

成蹊大学理工学部

学修に当たって

- ▶ 自動車の運転:
 - ▶ 自動車の運転の仕方は免許をとる際に習う
 - ▶ ガソリン車、電気自動車、ハイブリット車で運転の仕方は同じ
 - ▶ それぞれに特徴があり、仕組みを知れば特徴をいかせる
- ▶ 今日学ぶ内容
 - ▶ `std::vector` の様々な機能
 - ▶ `vector` 類似クラス作成での内部動作の確認
 - ▶ `vector` の使い方だけでなく、内部の仕組みを知ることによって特徴をいかせるようにする

vector の機能と構造

C++で使う四つのメモリ領域

- ▶ **コード領域**：関数やメンバ関数の機械語が置かれる。
- ▶ **静的領域**：大域変数や文字列用。コンパイル時にメモリサイズが決まり、実行開始時に割り当てられる。
- ▶ **スタック領域**：引数を含む局所変数と関数呼び出しの制御に使う。コンパイル時に関数ごとのメモリサイズが決まり、関数呼び出しごとに割り当て、呼び出しから戻る際に解放がなされる。
- ▶ **ヒープ領域**：プログラムの実行状況に応じて割り当てと解放を行う。vector の本体はこの領域に関係する。

コード領域

```
int main(){ }  
int Point::get(){ }
```

静的領域

global

スタック領域

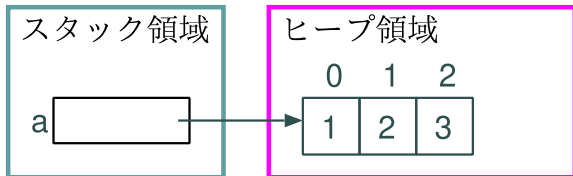
param

local

ヒープ領域

std::vector の使うメモリ

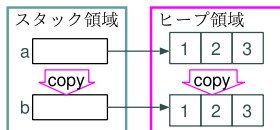
- ▶ 変数の制御情報部分は静的領域またはスタック領域
- ▶ 数が増減する要素の値を保存する本体はヒープ領域
- ▶ その本体は固定サイズの値が隣り合う形
 - ▶ 添字指定で n 番目の要素の場所が計算可能
 - ▶ 要素の場所 = 先頭の場所 + $n \times$ 要素のサイズ
- ▶ `vector<int> a {1,2,3};` のイメージ:



代入と関数呼び出し

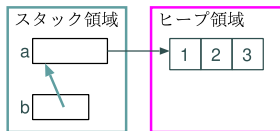
- ▶ 代入と引数の値渡しでは制御情報と本体の双方がコピーされる

- ▶ `b=a;`
 - ▶ `void f(vector<T> b);`



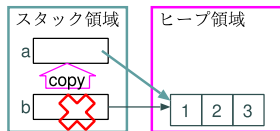
- ▶ 参照渡しでの引数では場所情報（変数のアドレス）が渡される

- ▶ `void f(vector<T>& b);`
 - ▶ `void f(const vector<T>& b);`



- ▶ 局所変数を関数から返すと制御情報のみコピー (move) される

```
auto f(){ vector<T> b(3);  
           return b;      }  
a = f();
```

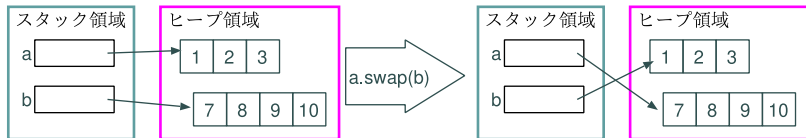


- ▶ さらに効率化も行われる (RVO)

swap メンバ関数

- ▶ `void vector<T>::swap(vector<T>& other);`
- ▶ 同型の他の変数と内容（制御情報と本体）を交換する

```
int main() {  
    vector<int> a{1,2,3}, b{7,8,9,10};  
    print(a); // (1,2,3)  
    print(b); // (7,8,9,10)  
    a.swap(b);  
    print(a); // (7,8,9,10)  
    print(b); // (1,2,3)  
}
```



ヒープ領域の割り当て

▶ 概要

- ▶ `vector` の本体部分のサイズは増減する
- ▶ 必要に応じて適当なサイズを確保する
- ▶ 少し多めに確保して割り当て回数を減らす
- ▶ 確保する量に決まりはない（実装の仕方に依存）

▶ 確保したメモリサイズの確認

- ▶ `capacity` メンバ関数（容量の意味）
- ▶ `size` は挿入された要素の数

▶ 確保するタイミング

- ▶ コンストラクタ, `push_back`, `insert` など

▶ 解放するタイミング

- ▶ 変数の使用が終了した時点（デストラクタ）など
- ▶ `pop_back`, `erase`, `clear`, `resize` では解放しない実装が許される

push_back と capacity

- ▶ 容量が変化した時点でそのサイズを出力してみる

```
const size_t N {200};  
std::vector<int> v;  
size_t cap { v.capacity() };  
  
cout << cap << "\n";  
for (size_t i=0; i<N; i++){  
    v.push_back(i);  
    if (cap != v.capacity()){  
        cap = v.capacity();  
        cout << cap << "\n";  
    }  
}  
  
cout << "size      = "  
    << v.size() << "\n"  
    << "capacity = "  
    << v.capacity() << "\n";
```

出力結果：

```
0  
1  
2  
4  
8  
16  
32  
64  
128  
256  
size      = 200  
capacity = 256
```

末尾削除／全削除

- ▶ 要素削除では容量は変化しない（実装の仕方に依存）

```
void print(const std::vector<int>& a) {  
    cout << a.size() << ", " << a.capacity() << ": ";  
    for (auto e : a)  
        cout << e << " ";  
    cout << "\n";  
}
```

```
int main() {  
    std::vector<int> a {1,2,3};  
    print(a);      // 3, 3:  1 2 3  
    a.pop_back();  
    print(a);      // 2, 3:  1 2  
    a.pop_back();  
    print(a);      // 1, 3:  1  
    a.clear();  
    print(a);      // 0, 3:  
}
```

先頭と末尾

- ▶ `front()` : 先頭要素へのアクセス
- ▶ `back()` : 末尾要素へのアクセス
- ▶ どちらも左辺値として使える
 - ▶ 要素のリファレンスを返す

```
using std::cout;
std::vector a {3,2,3,4,8};
a.front() = 1;
a.back() = 5;
cout << a.front() <<" " << a.back() <<"\n"; // 1 5
for (auto e : a)
    cout << e <<" ";
cout <<"\n"; // 1 2 3 4 5
```

vector のイテレータ

- ▶ イテレータ（反復子）とは
 - ▶ 要素の場所情報を保持し、ループで使うためのクラス
 - ▶ 他のコンテナ（複数データを管理するクラス）と操作を揃えるためのもの
 - ▶ ポインタに類似の `*` や `->` 演算子を使用する
 - ▶ `vector` 用のイテレータは加算減算や比較が可能
- ▶ イテレータの例
 - ▶ `begin()`：要素の先頭を表すイテレータを返す
 - ▶ `end()`：末尾要素の次を表すイテレータを返す

```
std::vector a {1,2,3,4};  
auto it { a.begin() };    // イテレータ  
cout << *it << " " << *(it + 1) << "\n"; // 1 2  
++it;  
cout << *it << "\n";      // 2  
cout << a.end() - a.begin() << "\n";    // 4
```

末尾以外の挿入削除

- ▶ 末尾以外の挿入削除の指定にはイテレータを使用する。
添字を使った挿入削除はできない。
- ▶ `push_front()`, `pop_front` は提供されない。
理由は操作時間が要素数に比例するため。

```
void print(const std::vector<int>& a) {  
    for (auto e : a) cout << e << " ";  
    cout << "\n";  
}  
  
int main() {  
    std::vector a {8,6,5,3,2,1};  
    a.insert( a.begin(), 9 );    // push_front() に相当  
    print(a); // 9 8 6 5 3 2 1  
    a.insert( a.begin()+2, 7 ); // a[2] に7を挿入  
    print(a); // 9 8 7 6 5 3 2 1  
    a.erase( a.begin()+3 );      // a[3] を削除  
    print(a); // 9 8 7 5 3 2 1  
}
```

範囲 for 文

- ▶ 範囲 for 文はイテレータのループを簡潔に書くために C++11 から導入された
- ▶ `begin()` と `end()` を備えることが条件
- ▶ 以下の 2 つの for 文はほぼ同じ内容

```
std::vector a {1,2,3,4,5};  
  
for (auto it=a.begin(); it!=a.end(); ++it) {  
    auto e { *it };  
    std::cout << e << "\n";  
}  
  
for (auto e : a)  
    std::cout << e << "\n";
```

MyVec(vector 類似クラス) の実装

MyVec の概要

- ▶ `vector` の最小限の機能で、その他の機能を実装する
- ▶ 作ってみることで `vector` のメンバ関数の動きを学ぶ
 - ▶ ただし要素は基本データ型のみのサポートとする
 - ▶ クラスオブジェクトを要素にした場合には制限がある
- ▶ 使用する `vector` の機能（後の授業で置き換える）：
 - ▶ ヒープ領域に連続領域を割り当ててアクセスする機能
 - ▶ サイズ指定コンストラクタ `vector<int> a(5);`
 - ▶ `[]` 演算子による要素へのアクセス `a[0]`
 - ▶ `swap()` メンバ関数 `a.swap(b);`
- ▶ 実装する機能（`vector` と同じ名前のメンバ関数）
 - ▶ `empty`, `size`, `capacity`, `swap`
 - ▶ `push_back`, `pop_back`, `operator[]`, `clear`
 - ▶ `begin`, `end`, `insert`, `erase`

ファイル構成

- ▶ MyVec クラス定義は `myvec.hpp` ファイルに保存
- ▶ 使用するプログラムは `#include "myvec.hpp"` を書く

ソースコード 1: `myvec.hpp`

```
template<typename T>
class MyVec { ... };
```

ソースコード 2: `test.cpp`

```
#include <iostream>
#include "myvec.hpp"
int main() { MyVec<int> x; ... }
```

```
$ g++ -std=c++17 test.cpp
$ ./a.out
```

クラス定義

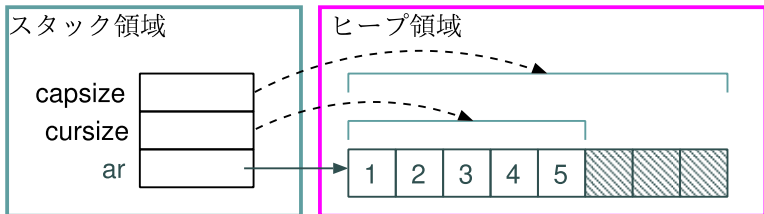
▶ データメンバとメンバ関数の宣言

ヘッダファイル 3: myvec.hpp

```
#include <vector>
template<typename T>
class MyVec {
    size_t capsize{}; // 割り当てサイズ
    size_t cursize{}; // 使用サイズ
    std::vector<T> ar;
public:
    bool    empty()      const { return cursize == 0; }
    size_t  size()       const { return cursize; }
    void    clear()      { cursize = 0; }
    size_t  capacity()   const { return capsize; }
    T       operator[] (size_t i) const
        { return ar[i]; }
    T&      operator[] (size_t i) { return ar[i]; }
    // この後に作るメンバ関数が続く
};
```

データメンバの役割

- ▶ `capsize` : 未使用を含めて割り当てたサイズ
- ▶ `cursize` : 挿入済みの現在の要素数
- ▶ `ar` : 要素を保持するためだけに使用
 - ▶ 割り当てサイズを `ar` に任せない
 - ▶ `ar.capacity() == capsize` にする
 - ▶ `cursize=0;` で `MyVec::size() == 0`



MyVecの[]演算子

添字を使った要素へのアクセス

- ▶ `T operator[](size_t i) const;`
 - ▶ `y = myvec[x];` の形式で使用する (右辺値)
- ▶ `T& operator[](size_t i);`
 - ▶ `myvec[x] = y;` の形式で使用する (左辺値)
 - ▶ `T&`を返すことで要素を変更できる

```
MyVec<int> x;
```

```
// 後で作成する要素を挿入するコード
```

```
// 要素へのアクセス
```

```
x[0] = 1; // 左辺値
```

```
x[1] = 5;
```

```
x[2] = 3;
```

```
for (size_t i = 0; i < x.size(); i++)
```

```
    std::cout << x[i] << " "; // 右辺値
```

```
std::cout << "\n"; // 1 5 3
```

メンバ関数の作成

以下の操作を加えていく

- ▶ `push_back`
- ▶ `pop_back`
- ▶ `swap`
- ▶ `begin(),end()`
- ▶ `erase`
- ▶ `insert`

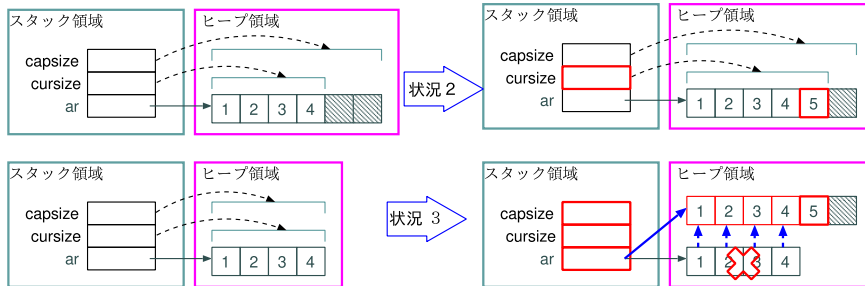
push_back と割り当て数の関係

- ▶ 2^n の要素を持てるように割り当て数を増やしていく
- ▶ capsize と cursize を変更していく。
- ▶ capsize の変更に応じて ar も変更する

```
MyVec<int> myvec; // 割り当て数と要素数ともに 0
myvec.push_back(1); // 割り当て数 1 要素数 1
myvec.push_back(2); // 割り当て数 2 要素数 2
myvec.push_back(3); // 割り当て数 4 要素数 3
myvec.push_back(4); // 割り当て数 4 要素数 4
myvec.push_back(5); // 割り当て数 8 要素数 5
myvec.push_back(6); // 割り当て数 8 要素数 6
myvec.push_back(7); // 割り当て数 8 要素数 7
myvec.push_back(8); // 割り当て数 8 要素数 8
myvec.push_back(9); // 割り当て数 16 要素数 9
```

MyVec::push_back 操作の概要

- ▶ 末尾に要素を挿入する
- ▶ 状況により三種類の対応が必要
 - ▶ 状況 1: 未だ割り当てがない
 - ▶ 割り当て数の決定, 割り当て, 代入
 - ▶ 状況 2: 割り当て数に余裕がある
 - ▶ 代入
 - ▶ 状況 3: 割り当て数に余裕がない
 - ▶ 割り当て数の決定, 割り当て, 現在要素のコピー, 代入



MyVec::push_back の実装

- ▶ 新しいサイズは空なら 1 それ以外は容量 2 倍
- ▶ 要素の代入と cursize の増加は共通

ヘッダファイル 4: myvec.hpp

```
class MyVec {  
    ...  
    void push_back(const T& x) {  
        if (cursize == capsize) { // 空 or 満杯  
            capsize = (capsize==0) ? 1 : 2*capsize;  
            std::vector<T> n(capsize); // 新しい割り当て  
            for (size_t i=0; i<cursize; i++) // コピー  
                n[i] = ar[i];  
            ar.swap(n); // 入れ替え  
        }  
        ar[cursize] = x; // 代入  
        ++cursize;  
    }  
};
```


MyVec::pop_back 操作

▶ 操作

- ▶ 末尾要素を削除する（使えなくする）
- ▶ 割り当て数を変更する必要はない
- ▶ `cursize` を調整する
- ▶ `size()` と `empty()` の結果に影響する

▶ vector との違い

- ▶ 削除要素のデストラクタが実行されない。
ややこしいので考えないことにする。

ヘッダファイル 5: myvec.hpp

```
class MyVec {  
    ...  
    void pop_back()          { -- cursize; }  
};
```

MyVec::swap 操作

- ▶ 制御に使う変数の内容を入れ替える
- ▶ クラスのデータメンバをすべて入れ替える

ヘッダファイル 6: myvec.hpp

```
class MyVec {  
    ...  
    void swap(MyVec<T>& x) {  
        ar.swap(x);  
        std::swap(capsize, x.capsize);  
        std::swap(cursize, x.cursize);  
    }  
};
```

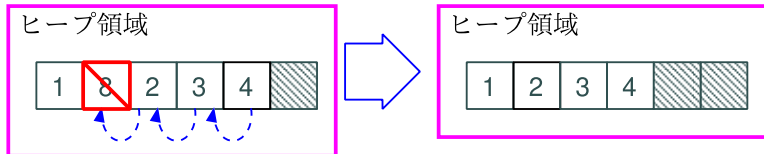
末尾以外の挿入削除

- ▶ insert, erase にはイテレータが必要
- ▶ イテレータはやや複雑なクラスなので単純化する
 - ▶ 添字を指定した挿入と削除だけできるようにする
 - ▶ iterator クラスを作らずに size_t を使う
- ▶ begin, end を作り vector と使い方を似せる
 - ▶ size_t begin() { return 0; }
 - ▶ size_t end() { return cursize; }

```
// 想定する使い方 (これならば vector でも同じ)
for (int i = 0; i < 5; i++) {
    x.insert(x.begin(), i);
    print(x);
}
x.insert(x.begin()+3, 10);
print(x);
x.erase(x.begin()+2);
print(x);
```

MyVec::erase 操作

- ▶ 添字で指定して要素を削除する
- ▶ 削除要素より後ろの要素を前にシフト (容量変更なし)

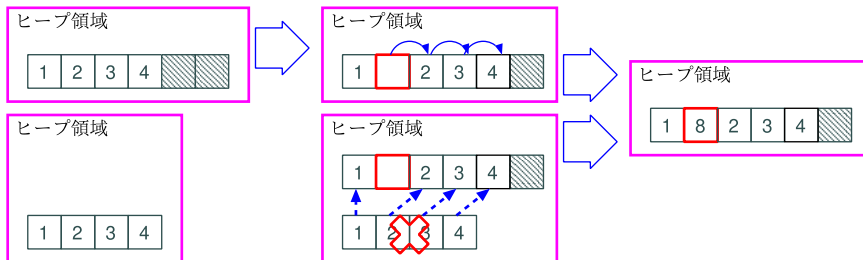


ソースコード 7: myvec.hpp

```
size_t erase(size_t pos) { // メンバ関数
    if (pos >= cursize) return cursize;
    for (size_t i = pos; i < cursize-1; i++) // 前シフト
        ar[i] = ar[i+1];
    -- cursize;
    return pos;
}
```

MyVec::insert 操作の概要

- ▶ 指定方法と動作
 - ▶ `size_t insert(size_t i, const T& v);`
 - ▶ 添字 `i` の要素の前に値 `v` を挿入し、その `i` を返す
 - ▶ MyVec が空ならば `push_back` と同じ動作
 - ▶ `i >= cursive` の時も `push_back` と同じ動作
- ▶ 容量が足りる 場合：場所を空けて代入
- ▶ 容量が足りない場合：新たに割り当てて、コピーと代入



MyVec::insert の実装

ソースコード 8: myvec.hpp

```
size_t insert(size_t pos, const T& v) { // メンバ関数
    if (pos > cursize) pos = cursize;
    if (cursize == capsize) { // 空 or 満杯の場合は
        capsize = (capsize==0) ? 1 : 2*capsize;
        std::vector<T> n(capsize);
        for (size_t i = 0; i < pos; i++) // 前半
            n[i] = ar[i];
        for (size_t i = pos; i < cursize; i++) // 後半
            n[i+1] = ar[i];
        ar.swap(n); // 入れ替え
    } else {
        for (size_t i=cursize; i>pos; i--) // 後シフト
            ar[i] = ar[i-1];
    }
    ar[pos] = v;
    ++cursize;
    return pos;
}
```

MyVec::insert の動作確認

```
#include <iostream>
#include "myvec.hpp"
void print(const MyVec<int>& a) {
    std::cout <<a.capacity()<<" ";
    for (size_t i = 0; i < a.size(); i++)
        std::cout <<(i==0?" ":" ")<< a[i];
    std::cout <<"]\n";
}

int main() {
    MyVec<int> x;
    x.insert(x.begin(), 1);
    print(x); // 1 [1]
    x.insert(x.begin()+1, 3);
    print(x); // 2 [1 3]
    x.insert(x.begin()+1, 2);
    print(x); // 4 [1 2 3]
    x.erase(x.begin()+1);
    print(x); // 4 [1 3]
}
```

push_back との違い

- ▶ `push_back` は `insert` で実装してもほぼ同じ

```
void push_back(const T& x) {  
    insert(cursize, x);  
}
```

- ▶ 性能
 - ▶ `insert` は挿入位置より後方のシフトが性能を悪くする
 - ▶ `push_back` は割り当てをうまくすれば最後の代入のみ
- ▶ 結論
 - ▶ `insert` を多用する使い方は性能悪化に注意

まとめ

まとめ

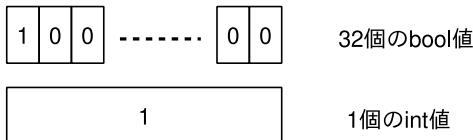
- ▶ `vector` とメモリ
 - ▶ 要素本体はヒープ領域に割り当てられる
 - ▶ 要素数の変化に対応する
 - ▶ すべては連続領域に割り当てられる
- ▶ `MyVec` 類似クラスの実装
 - ▶ 連続領域の割り当て数を管理する
 - ▶ `push_back` は 2^n の要素を持つようにした
 - ▶ 要素の増加では割り当て直しと要素のコピーが必要
 - ▶ `pop_back` は見た目のサイズの調整のみ
 - ▶ `swap` は制御情報の入れ替えのみ
 - ▶ `erase` では要素を前側へのシフトが必要
 - ▶ `insert` は割り当て直しまたは後ろ側へのシフトが必要

参考

`std::vector<bool >`

std::vector<bool>とは

- ▶ bool 型の特殊性
 - ▶ bool は true/false の 2 値なので 1 ビットで表現可能
 - ▶ 現代のコンピュータは以下が苦手
 - ▶ 1 ビット単位のメモリ割り当て
 - ▶ 1 ビット単位の代入
- ▶ ビットアレイ（ビットベクター）とは
 - ▶ 32 ビットの整数を使えば 32 個の bool 値が表現可能
 - ▶ 省メモリ化とアクセス高速化の実装技術
- ▶ bool のみクラステンプレートの特殊化で実装される
 - ▶ イテレータは使用可
 - ▶ 要素へのポインタ/リファレンスは使用不可



エラトステネスのふるい法

- ▶ 最大 n までの素数を探すアルゴリズム
- ▶ 素数とは：2 以上で 1 とそれ自身のみが約数となる整数
- ▶ アルゴリズムのステップ
 1. 2 から n までの数列を作る。
 2. p を 2 として、それを最小の素数とする。
 3. p 以外の p の倍数 ($2 \times p, 3 \times p \dots$) を数列から削除する。
 4. p の次に大きな数を新たな p とする。無ければ 6 に進む。
 5. 3 に戻る。 ($p > \sqrt{n}$ ならば 6 に進んでもよい)
 6. 残った数列を n 以下の素数として終了する。

2	3	4	5	6	7	8	9	10	11	12
2	3	4	5	6	7	8	9	10	11	12
2	3	4	5	6	7	8	9	10	11	12
2	3	4	5	6	7	8	9	10	11	12

ふるい法のプログラム

- ▶ 配列の添字を調べる数に対応させる
- ▶ 要素の値が true の添字を素数にする

```
std::vector<bool> p(n+1, true); // n+1 個すべてが true
// 配列 p は添字が 0 から n まで有効

// p[0] = p[1] = false;
for (int i = 2; i*i <= n; i++) {
    if (!p[i]) continue;
    for (int j = 2*i; j <= n; j+=i){
        p[j] = false;
    }
}

for (int i = 2; i <= n; i++)
    if (p[i]) cout << i << " ";
cout << "\n";
```