

# vectorクラス

- コンテナクラスのひとつ
- コンテナとは？  
C言語の配列のように複数の値を格納できるもの  
**STL(Standard Template Library)**の中で  
定義されている、
- vector以外にlist, map, array, bitset, stack, queue 等さまざまなコンテナがある

# vectorクラス

- コンテナクラス

- **vector**: 動的配列(自由にサイズを変更可能な配列)
- **array** : 静的配列(一旦決めたサイズは変更不可)
- **list** : リスト構造を実現するクラス
- **map** : 連想配列という特殊な配列クラス
- **bitset**: 2進数値を容易に扱うためのクラス
- **stack** : スタック(後入れ先出し)を実現するクラス
- **queue** : キュー(先入れ先出し)を実現するクラス

# vectorクラス

- vectorを使う利点
  - (例)敵クラスのインスタンス生成

```
Enemy enemy1;  
Enemy enemy2;  
Enemy enemy3;
```

バラバラの変数で管理するより配列化したほうがよい

```
Enemy enemy[100];
```

しかしこれでは要素数(100体)までしか対応できない...

# vectorクラス

- vectorを使う利点
  - (例)敵クラスのインスタンス生成

```
Enemy enemy[100];
```



```
Enemy enemy[10000];
```

要素数100から10000に増やすと、最初に10000体分のメモリを確保しようとするため、敵が数体しか登場しないときは**メモリの無駄遣い**になる...

# vectorクラス

- vectorを使う利点

- (例) 敵クラスのインスタンス生成

vectorを使うことで、必要なときに必要なぶんだけ  
配列要素を確保することができる！

また要素が不要になれば削除することもできる！

このような配列を動的配列もしくは可変長配列という

# vectorクラス

- vectorを使う利点

- **動的な配列**を実現できる  
配列の要素数を最初に定めずに、必要になったときにその都度追加や削除ができる！
- クラスのメンバ関数によって、要素数をカウントしたり、要素をすべてクリアしたりといった機能があり、C言語では容易でなかったことができる

# vectorクラス

- vectorクラスのメンバ関数

- `size()` : 配列の全要素数をカウント
- `push_back()` : 配列末尾に要素を付け加える
- `emplace_back()` : 配列末尾に要素を付け加える
- `pop_back()` : 配列末尾のデータを消去する
- `erase()` : 指定された場所の要素を削除する
- `insert()` : 指定された場所へ要素を追加する
- `empty()` : 配列要素が空なら`true`を返す
- `clear()` : 配列要素をすべて削除
- `front()` : 配列の先頭の値を取得
- `back()` : 配列の末尾の値を取得

# vectorクラス

- 教科書P227~228 Sample604
- C++作業フォルダ内にSample604フォルダを作成  
`mkdir Sample604`  
`cd Sample604`
- main.cppを作成  
`copy nul main.cpp`



# vectorクラス

- vectorの宣言方法

- `#include <vector>` が必須!

- `std::vector<型名>` 配列名

※ `using namespace std`を記述している場合  
「 `std::` 」は省略できる


※ `型名`は基本データ型以外に `クラス`も指定可能

# vectorクラス

## main.cpp (Sample604)

```
#include <vector>
#include <string>
using namespace std;
int main(){
    vector<int> v1;
    vector<string> v2;
    v1.push_back(1);
    v1.push_back(2);
    v1.push_back(3);
    v2.push_back("ABC");
    v2.push_back("DEF");
    for (int i = 0; i < v1.size(); i++) {
        cout << "v1[" << i << "]= " << v1[i] << endl;
    }
    for (int i = 0; i < v2.size(); i++) {
        cout << "v2[" << i << "]= " << v2[i] << endl;
    }
    return 0;
}
```

vectorも配列なので  
v1[i]という  
配列名[ 添え字 ]  
という表記が使用できる



# vectorクラス

## main.cpp (Sample604)

```
#include <vector>
#include <string>
using namespace std;
int main(){
```

```
→ vector<int> v1 { 10, 9, 8 }; //初期値10,9,8を設定
vector<string> v2;
v1.push_back(1);
v1.push_back(2);
v1.push_back(3);
v2.push_back("ABC");
v2.push_back("DEF");
for (int i = 0; i < v1.size(); i++) {
    cout << "v1[" << i << "]= " << v1[i] << endl;
}
for (int i = 0; i < v2.size(); i++) {
    cout << "v2[" << i << "]= " << v2[i] << endl;
}
return 0;
}
```

# vectorクラス

## main.cpp (Sample604)

```
#include <vector>
#include <string>
using namespace std;
int main(){
```

```
→ vector<int> v1(5); //あらかじめ5つの要素を確保して0で初期値
vector<string> v2;
v1.push_back(1);
v1.push_back(2);
v1.push_back(3);
v2.push_back("ABC");
v2.push_back("DEF");
for (int i = 0; i < v1.size(); i++) {
    cout << "v1[" << i << "]= " << v1[i] << endl;
}
for (int i = 0; i < v2.size(); i++) {
    cout << "v2[" << i << "]= " << v2[i] << endl;
}
return 0;
}
```

# vectorクラス

## main.cpp (Sample604)

```
#include <vector>
#include <string>
using namespace std;
int main(){
```

```
→ vector<int> v1(3,1); //あらかじめ3つの要素を確保して1で初期値
vector<string> v2;
v1.push_back(1);
v1.push_back(2);
v1.push_back(3);
v2.push_back("ABC");
v2.push_back("DEF");
for (int i = 0; i < v1.size(); i++) {
    cout << "v1[" << i << "]= " << v1[i] << endl;
}
for (int i = 0; i < v2.size(); i++) {
    cout << "v2[" << i << "]= " << v2[i] << endl;
}
return 0;
}
```

# vectorクラス

- `vector`の初期化方法
- `vector<型名> 配列名 { 値1, 値2, ... }`  
すべて異なる値で初期値を与える
- `vector<型名> 配列名 (要素数)`  
指定の要素数を確保して、0で全要素を初期化
- `vector<型名> 配列名 (要素数, 値)`  
指定の要素数を確保して、指定値で全要素を初期化

# vectorクラス

- vectorの末尾にデータを追加
- vector<int> v1 の配列に対して

v1.push\_back(10)

or

v1.emplace\_back(10)

引数として型名と同じデータを指定すると要素数が増え  
ていく

# vectorクラス(要素の追加と要素数)

main.cpp (Sample604)

```
v1.push_back(3);  
v2.push_back("ABC");  
v2.push_back("DEF");
```

→ **v1.push\_back(4);** //末尾に要素追加

→ **cout << "v1の要素数:" << v1.size() << endl;**  
for (int i = 0; i < v1.size(); i++) {  
 cout << "v1[" << i << "]= " << v1[i] << endl;  
}

→ **v2.push\_back("G");** //末尾に要素追加

→ **cout << "v2の要素数:" << v2.size() << endl;**  
for (int i = 0; i < v2.size(); i++) {  
 cout << "v2[" << i << "]= " << v2[i] << endl;  
}  
return 0;

```
}
```



# vectorクラス(要素の削除)

main.cpp (Sample604)

```
v1.push_back(3);  
v2.push_back("ABC");  
v2.push_back("DEF");
```

➡ **v1.pop\_back();** //末尾要素の削除

```
v1.push_back(4);  
cout << "v1の要素数:" << v1.size() << endl;  
for (int i = 0; i < v1.size(); i++) {  
    std::cout << "v1[" << i << "]= " << v1[i] << endl;  
}
```

➡ **v2.pop\_back();** //末尾要素の削除

```
v2.push_back("G");  
cout << "v2の要素数:" << v2.size() << endl;  
for (int i = 0; i < v2.size(); i++) {  
    cout << "v2[" << i << "]= " << v2[i] << endl;  
}  
return 0;  
}
```

# vectorクラス(挿入と削除)

- vectorの末尾だけでなく、指定した場所に要素を追加するメンバ関数
  - **insert**(場所, 挿入値)
  - **erase**(場所)

が存在するが、挿入する場所、削除する場所はどちらも配列の添え字の番号ではない...

# イテレータ(iterator:反復子)

- vectorといったコンテナクラス内の要素の位置を示すポインタのようなもの

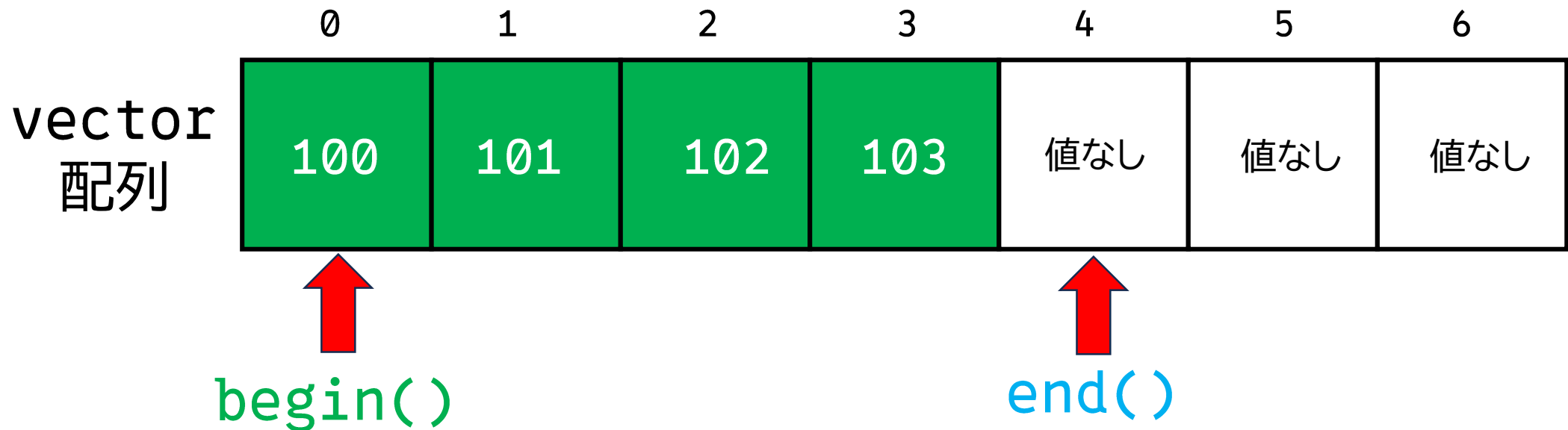
```
vector<型名>::iterator itr
```

```
itr = v1.begin();
```

とすると、`itr`は先頭要素のイテレータとなる

# イテレータ(iterator:反復子)

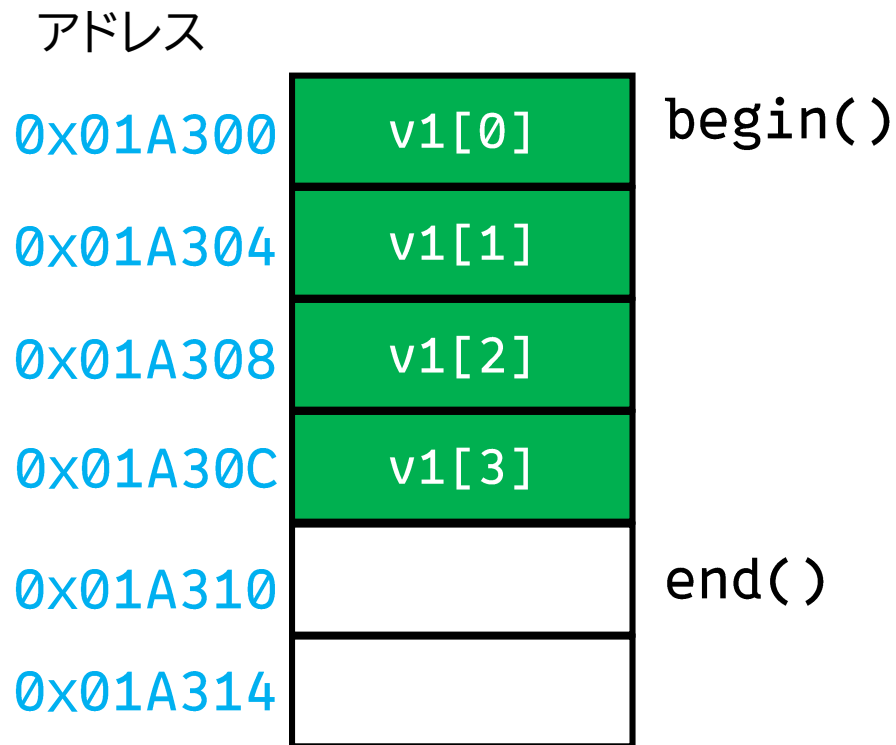
- イテレータ = インスタンス.`begin()`  
コンテナクラスの先頭要素の場所をイテレータに代入
- イテレータ = インスタンス.`end()`  
コンテナクラスの最終要素のひとつ先の場所をイテレータに代入



# イテレータ(iterator:反復子)

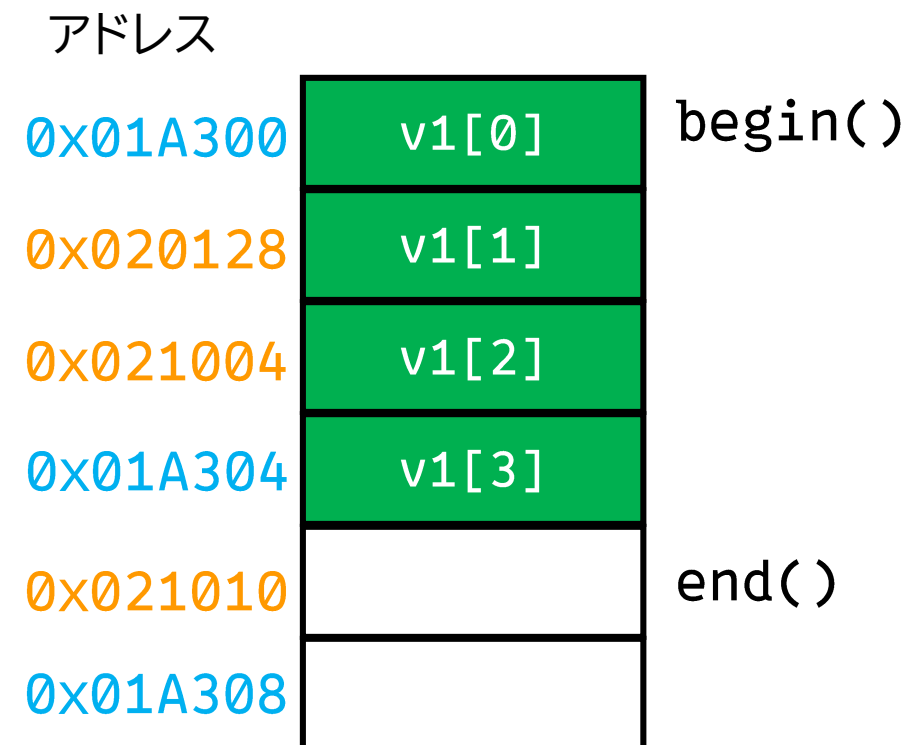
- イテレータの必要性

## 静的配列



連続したアドレスに配置

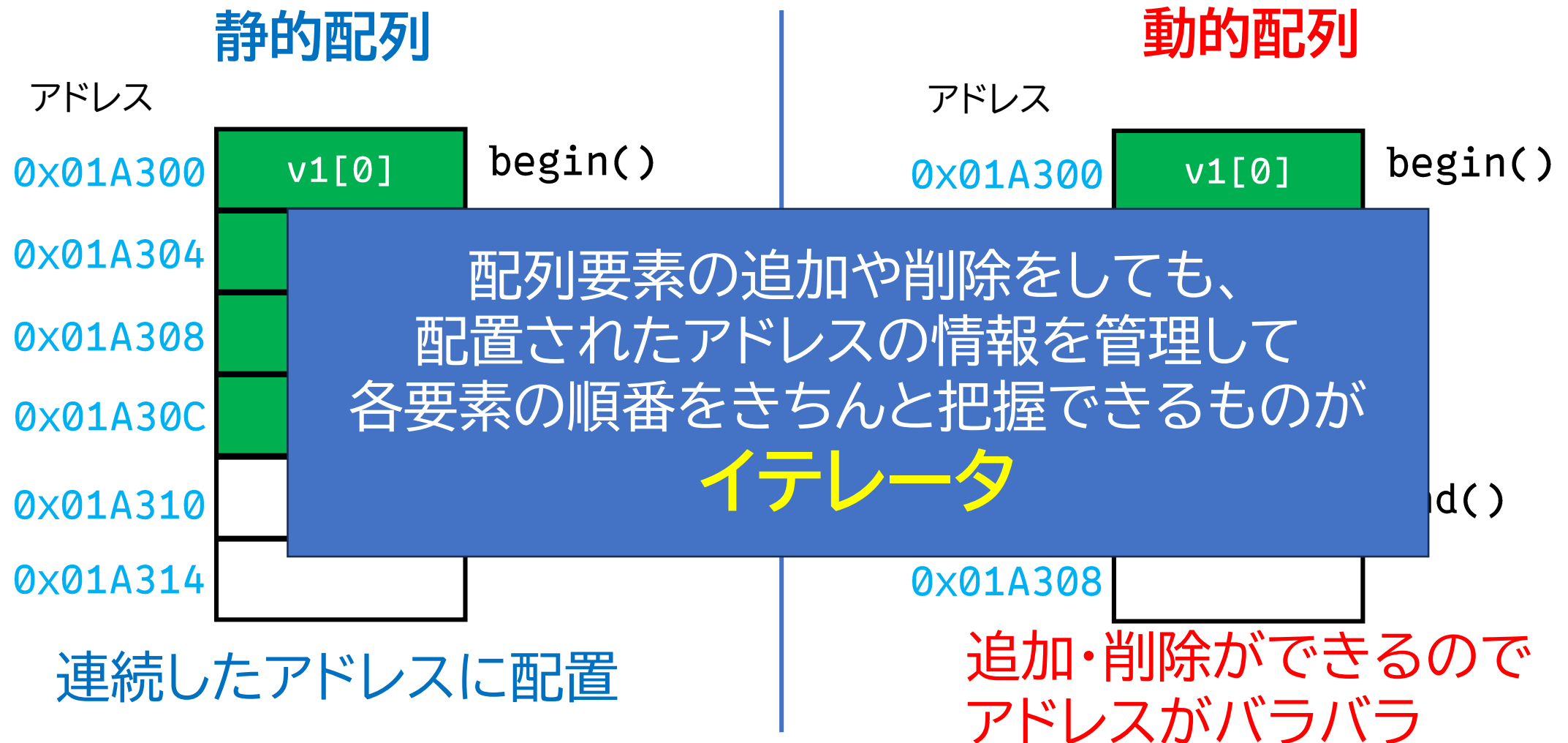
## 動的配列



追加・削除ができるため  
アドレスが不連続

# イテレータ(iterator:反復子)

- イテレータの必要性



# イテレータ(iterator:反復子)

main.cpp (Sample604)

```
v1.push_back(3);  
v2.push_back("ABC");  
v2.push_back("DEF");  
v1.pop_back();  
v1.push_back(4);  
cout << "v1の要素数:" << v1.size() << endl;
```

➡ **vector<int>::iterator itr = v1.begin();**  
➡ **cout << "イテレータが指す要素の値:" << \*itr << endl;**

```
for (int i = 0; i < v1.size(); i++) {  
    cout << "v1[" << i << "]=" << v1[i] << endl;  
}
```

# イテレータ(iterator:反復子)

```
v1.push_back(3);
```

**auto**による**型推論**を使って、右辺値から型名を自動的に割り当て  
**auto** → `std::vector<int>::iterator`

```
cout << "v1のサイズ: " << v1.size() << endl;
```

➡ **auto** itr = v1.begin();

```
cout << "イテレータが指す要素の値:" << *itr << endl;
```

```
for (int i = 0; i < v1.size(); i++) {  
    cout << "v1[" << i << "]=" << v1[i] << endl;  
}
```



# イテレータ(iterator:反復子)

- イテレータを使うと、コンテナクラス内の指定場所への要素の追加や削除を行うことが可能

インスタンス.**insert**(イテレータ, 挿入値)  
イテレータが指し示す場所に値を挿入する

インスタンス.**erase**(イテレータ)  
イテレータの示す場所の要素を削除する

# イテレータ(iterator:反復子)

main.cpp (Sample604)

```
v1.push_back(3);  
v2.push_back("ABC");  
v2.push_back("DEF");  
v1.pop_back();  
v1.push_back(4);  
cout << "v1の要素数:" << v1.size() << endl;
```

```
auto itr = v1.begin();  
cout << "イテレータが指す要素の値:" << *itr << endl;
```

```
→ v1.insert(itr + 2, 20); //指定した場所(先頭から+2番目)へ要素(20)を追加  
→ itr = v1.begin();      // 挿入で配列のイテレータが変更になったので再取得  
→ v1.erase(itr + 4);     //指定した場所(先頭から+4番目)の要素を削除
```

```
for (int i = 0; i < v1.size(); i++) {  
    cout << "v1[" << i << "]= " << v1[i] << endl;  
}
```

# イテレータ(iterator:反復子)

```
v1.pop_back();  
v1.emplace_back(4);  
cout << "v1の要素数:" << v1.size() << endl;
```

main.cpp (Sample604)

```
auto itr = itr.begin();  
cout << "イテレータが指す要素の値:" << *itr << endl;  
v1.insert(itr + 1, 20);  
itr = v1.begin();  
v1.erase(itr + 4);  
// for (int i = 0; i < v1.size(); i++) {  
//     cout << "v1[" << i << "]= " << v1[i] << endl;  
// }
```

```
→ for (auto itr = v1.begin(); itr != v1.end(); ++itr) {  
→     cout << *itr << endl; //itrV1.end()は最終要素のひとつ後  
→ }
```

イテレータを用いた要素の表示

# イテレータ(iterator:反復子)

- イテレータを用いたループ処理

```
for (auto itr = v1.begin()  
      ; itr != v1.end()  
      ; ++itr)
```

v1の先頭要素からv1の最終要素まで  
イテレータitrをひとつずつ進めながら、  
ループ処理を行う

# イテレータ(iterator:反復子)

- 範囲for文

イテレータを用いたfor文を、より簡略化したもの

```
for (auto d : v1){ // 範囲forを使うとdに  
    cout << d << endl; // v1の要素が順に格納  
}
```

v1の先頭要素から最終要素まで、すべての要素  
ぶんループ

(※ ただし指定場所からの開始・終了は不可)

# イテレータ(iterator:反復子)

main.cpp (Sample604)

```
v2.push_back("DEF");  
v1.pop_back();  
v1.push_back(4);  
cout << "v1の要素数:" << v1.size() << endl;
```

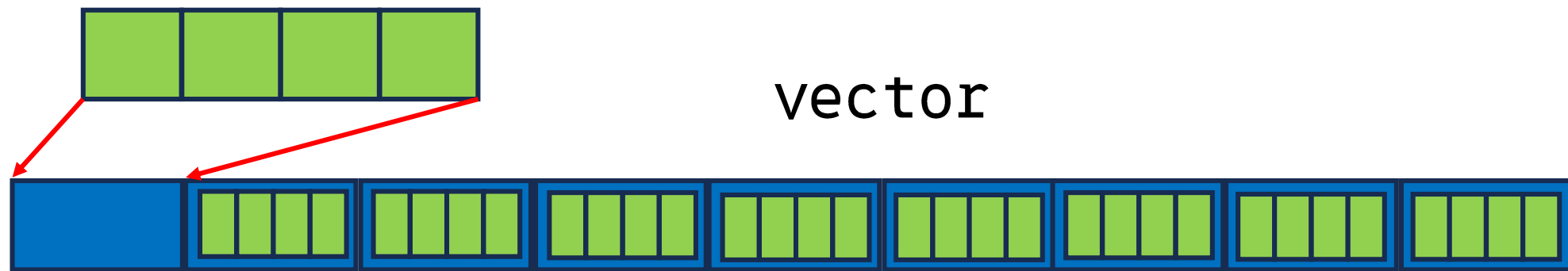
```
auto itrV1 = itrV1.begin();  
cout << "イテレータが指す要素の値:" << *itrV1 << endl;  
v1.insert(itrV1 + 1, 11);  
v1.erase(itrV1 + 2);  
// for (int i = 0; i < v1.size(); i++) {  
//     cout << "v1[" << i << "]=" << v1[i] << endl;  
// }  
// for (auto itrV1 = v1.begin(); itrV1 != v1.end(); ++itrV1) {  
//     cout << *itrV1 << endl;  
// }
```

```
for (auto d : v1) { //範囲for文にすると  
    cout << d << endl; //すべての要素を順次取り出す形となる  
}
```

範囲forを用いた要素の表示

# vectorクラス(二次元配列)

- C言語の配列と同様にvectorクラスを使って二次元配列を定義することが可能



vectorコンテナの中に、さらにvectorのコンテナを格納しているイメージ

# vectorクラス(二次元配列)

- 二次元配列を定義する記述方法
  - 行数と列数を定めずにint型の二次元配列v1を宣言  
`vector<vector<int>> v1`
  - 2行2列のint型の二次元配列v2を宣言  
`vector<vector<int>> v2(2, vector<int>(2))`  
                        行数                        列数
  - 2行2列のint型の初期値ありの二次元配列v3を宣言  
`vector<vector<int>> v3{ {1,2}, {3,4} }`



# vectorクラス(二次元配列)

- C言語の配列と同様に行番号と列番号を添え字で指定することで配列要素にアクセス可能

```
vector<vector<int>> v3{{1,2}, {3,4}};
```

と宣言したあと、

```
cout << v3[1][0] << endl;
```

を実行すると、1行0列目の「3」が表示される

# vectorクラス(二次元配列)

- vectorクラスは動的配列のため、行数や列数がプログラム中に変わることもある

そのため、1行目は列数が2列、2行目は3列...  
というように不揃いに可能性があるため注意

v[0]	[0][0]	[0][1]		
v[1]	[1][0]	[1][1]	[1][2]	
v[2]	[2][0]			
v[3]	[3][0]	[3][1]	[3][2]	[3][3]

v.size() : 行数を取得  
v[0].size() : 0行目の列数を取得  
v[1].size() : 1行目の列数を取得  
...

# vectorクラス(二次元配列)

- 要素が空(empty)な配列を宣言した直後は値を追加することができないので注意

```
vector<vector<int>> v{};  
v.push_back(100); //コンパイルエラー
```

- 要素を追加していく場合はresize関数を使用する

```
vector<vector<int>> v{};  
v.resize(1);           //0行目を作成  
v.push_back(100);      //v[0][0]に100が入る  
v.resize(2);           //1行目を作成
```

# vectorクラス

## • vectorまとめ

- **動的配列**を実現するコンテナクラス
- 最初に要素数を指定する必要がなく、適宜増減可能
- C言語の配列と同様に**添え字番号**で要素にアクセス
- 末尾に要素を追加、末尾の要素を削除可能
- 末尾以外の場所にも追加・削除可能だが**イテレータ**による場所の指定が必要
- 基本データ型以外のクラスインスタンスも格納可能