

# C++プログラミングI

- 第6回 vectorによる多次元配列
- 担当：二瓶芙巳雄

# 多次元配列とは

- 一次元の情報：
  - 一つの座標で表す長さなどの情報
- 多次元の情報：
  - 絵の情報はxy座標上の点の集まり（2次元）
  - 動画の情報はxyとフレーム番号（3次元）
- **C++による多次元情報の扱い**
  - 1次元配列：配列
  - 2次元配列：**配列を要素とする配列**
  - 3次元配列：2次元配列を要素とする配列
- `vector` の2次元配列
  - `vector<T>` を要素とする `vector`
  - ※ `<T>` の `T` (Template)は任意の型. `int` や `double` など.

```
1 std::vector<std::vector<T>>> x;
```

# vector の2次元配列

- 2次元配列：1次元配列の配列
  - 1次元 int 配列： `vector<int>`
  - 2次元 int 配列： `vector< vector<int> >`
- `vector<vector<int>> x;` は要素数0
  - ※ 配列の入れ物だけ作ったイメージ
- 追加後の `x` と `a`, `b`, `c` は無関係

|      | [0] | [1] |
|------|-----|-----|
| x[0] | 1   | 2   |
| x[1] | 3   | 4   |
| x[2] | 5   | 6   |

```
1 // 3x2の2次元配列
2 using std::cout, std::vector;
3
4 int main() {
5     vector<vector<int>> x;
6     vector a{1,2}, b{3,4}, c{5,6};
7
8     x.push_back(a);
9     x.push_back(b);
10    x.push_back(c);
11
12    cout << x[0][0] << " " << x[0][1] << "\n"; // 1 2
13    cout << x[1][0] << " " << x[1][1] << "\n"; // 3 4
14    cout << x[2][0] << " " << x[2][1] << "\n"; // 5 6
15 }
```

% ./a.out

```
1 2
3 4
5 6
```

# 2次元配列へのアクセス

- `x[0]` や `x[1]` は1次元配列
- `x[0][0]` や `x[1][0]` は `int`
- `x[i][j]` は左辺値として指定できる

```
1  int main() {
2      vector<vector<int>> x;
3      vector a{1,2}, b{3,4}, c{5,6};
4      x.push_back(a); x.push_back(b); x.push_back(c);
5
6      cout << x[0][0] << " " << x[0][1] << "\n"; // 1 2
7      cout << x[1][0] << " " << x[1][1] << "\n"; // 3 4
8      cout << x[2][0] << " " << x[2][1] << "\n"; // 5 6
9      cout << "\n";
10
11     x[0][0] = 10; // 先頭要素を変更
12     std::cin >> x[0][1]; // 2 番目の要素に入力
13     cout << "\n";
14
15     cout << x[0][0] << " " << x[0][1] << "\n"; // 1 2
16     cout << x[1][0] << " " << x[1][1] << "\n"; // 3 4
17     cout << x[2][0] << " " << x[2][1] << "\n"; // 5 6
18 }
```

```
% ./a.out
1 2
3 4
5 6

3000

10 3000
3 4
5 6
```

|      | [0] | [1] |
|------|-----|-----|
| x[0] | 1   | 2   |
| x[1] | 3   | 4   |
| x[2] | 5   | 6   |

|      | [0] | [1]  |
|------|-----|------|
| x[0] | 10  | 3000 |
| x[1] | 3   | 4    |
| x[2] | 5   | 6    |

# 2次元配列の初期化

- 中括弧 `{}` による初期化リストを並べる

```
1  vector<vector<int>> y {{3,4,5},{5,6,7}};  
2  
3  // 改行するとわかりやすい  
4  vector<vector<int>> y {  
5      {3,4,5},  
6      {5,6,7}  
7  };
```

- 要素がすべて同じ場合、丸括弧 `()` を使う
  - 50x100の2次元配列, 値は1

```
1  vector<vector<int>> z(50, vector<int>(100, 1));  
2  または  
3  vector z(50, vector(100, 1)); // 省略形
```

|      | [0] | [1] | [2] |
|------|-----|-----|-----|
| y[0] | 3   | 4   | 5   |
| y[1] | 5   | 6   | 7   |

# 1次元配列と2次元配列の初期化の比較

```
1 // 1次元配列の初期化
2
3 // 要素なし
4 vector<double> a;
5
6 // 値の直接指定
7 vector<double> b {1.0, 1.1, 1.2};
8
9
10
11
12 // 要素数5, 初期値が0.0
13 vector<double> c(5);
14
15 // 要素数5, 初期値1.4
16 vector<double> d(5, 1.4);
17
18 // 以下も可
19 vector<int> z { vector<int>(3, 1) };
20 vector<int> y { vector(3, 1) };
```

```
1 // 2次元配列の初期化
2
3 // 要素なし
4 vector<vector<double>> a;
5
6 // 値の直接指定: 2x3
7 vector<vector<double>> b {
8     {1.0, 1.1, 1.2},
9     {2.0, 2.1, 2.2}
10 };
11
12 // 要素数5, 初期値が要素なしvector<double>: 5x0
13 vector<vector<double>> c(5);
14
15 // 要素数5, 初期値が要素数30値1.4のvector<double>: 5x30
16 vector<vector<double>> d(5, vector<double>(30, 1.4));
17
18 // 以下も可: 3x3
19 vector<vector<int>> z {
20     vector<int>(3, 1),
21     vector(3, 1),
22     { 1, 2, 3 }
23 };
```

# 3次元配列の初期化

## ■ 4x5x2の例

```
1  // 要素はすべて0
2  vector xx( 4, vector( 5, vector(2, 0) ) );
3  xx[0][0][0] = 10;
4
5  // それぞれの値を持つ場合
6  vector<vector<vector<int>>> xxa {
7      {{1,2}, {2,3}, {3,4}, {5,6}, {6,7}},
8      {{2,7}, {3,8}, {4,2}, {6,9}, {7,3}},
9      {{3,4}, {4,2}, {5,1}, {7,3}, {8,8}},
10     {{7,3}, {0,8}, {9,6}, {1,4}, {7,2}}
11 };
12 cout << xxa[0][0][0]; // 1 先頭の要素
13 cout << xxa[0][0][1]; // 2
14 // cout << xxa[0][0][2]; // この要素番号はない
15 cout << xxa[0][1][0]; // 2
16 cout << xxa[3][4][1]; // 2 一番最後の要素
```

## 2次元配列とループ

- `a.size()` と `a[i].size()` で要素数を確認
  - `a` は `vector<double>` を要素として持つ `vector`
  - `a[i]` は `double` を要素として持つ `vector`
- `size_t` 型 (0以上の整数型) の利用
  - `size_t`: 配列のインデックスなどを表すために設計された型. 非負. 範囲が環境に依存しない.
  - `int`: 整数いろいろに使える型. 負の範囲がある. 範囲が環境に依存. 配列のインデックスには不適.
- `.size()` の戻り値の型は `size_t`. 暗黙的キャストを避けるため, `size_t` を使うとよい.

```
1  vector<vector<double>> a {  
2      {2.3, 8.4, 4.3, 1.2},  
3      {8.2, 1.3, 8.1, 7.5},  
4      {3.3, 3.1, 9.8, 5.3}  
5  };  
6  
7  for (size_t i = 0; i < a.size(); i++) {  
8      for (size_t j = 0; j < a[i].size(); j++)  
9          cout << a[i][j] << " ";  
10     cout << "\n";  
11 }
```



## 2次元配列の入出力の例

- 要素数が事前に分かる場合は定数を用いる
- `for` 文の添字は `int` でも良い( `.size()` を使わないため)

```
1  int main() {
2      const int n{3}, m{4};
3      vector b(n, vector(m, 0));
4
5      for (int i = 0; i < n; i++)
6          for (int j = 0; j < m; j++)
7              std::cin >> b[i][j];
8
9      for (int i = 0; i < n; i++) {
10         for (int j = 0; j < m; j++) {
11             if (j != 0) cout << ", ";
12             cout << b[i][j];
13         }
14         cout << "\n";
15     }
16 }
```

./a.out

```
1 2 3 4 5 6 7 8 9 10 11 12
1, 2, 3, 4
5, 6, 7, 8
9, 10, 11, 12
```

# 範囲 for 文の利用

- 添字情報が不要なら場合には記述が簡潔になる
- リファレンスの利用が性能に影響を与える
- 2次元配列 `a` の中から1次元配列を `x` に一つずつ取り出しながら繰り返し、内側のループでは `x` の中から `double` 型の要素を `y` に取り出して出力

```
1  int main() {
2      vector<vector<double>> a{
3          {2.3, 8.4, 4.3, 1.2},
4          {8.2, 1.3, 8.1, 7.5},
5          {3.3, 3.1, 9.8, 5.3}
6      };
7
8      // 範囲 for 文の利用
9      for (const vector<double>& x : a) {
10         for (double y : x)
11             cout << y << " ";
12         cout << "\n";
13     }
14     // 上下は同じ挙動
15     for (int i = 0; i < 3; i++) {
16         for (int j = 0; j < 4; j++)
17             cout << a[i][j] << " ";
18         cout << "\n";
19     }
20 }
```

# auto の利用

- 初期値から型を推定する
- 複雑な型名に有効
  - 単純な型には使用しない方がよい
- 型指定の面倒さは `auto` を使って解決できる
  - が、`auto` の乱用はプログラムを読みにくくする

```
./a.out
```

```
1 2.1
```

```
1 1 1
```

```
2.1 2.1 2.1 2.1
```

```
2.1 2.1 2.1 2.1
```

```
2.1 2.1 2.1 2.1
```

```
1  int main() {
2      auto i{ 1 };    // int
3      auto d{ 2.1 };  // double
4      cout << i << " " << d << "\n\n";
5
6      auto v{ vector(3, 1) }; // vector<int>
7      for (auto x : v)        // int
8          cout << x << " ";
9      cout << "\n";
10
11     // vector<vector<double>>
12     auto t{ vector(3, vector(4, 2.1)) };
13     // const vector<double>&
14     for (const auto& x : t) {
15         for (auto y : x) // double
16             cout << y << " ";
17         cout << "\n";
18     }
19 }
```

# 3次元配列とループ

- 各次元の要素数に注意
- 添字の変数名には `i`, `j`, `k` を用いる習慣がある

```
1  vector<vector<vector<int>>> a {  
2      {{0,-1,2,3}, {-4,5,6,7}, {8,-9,10,11}},  
3      {{0,1,-2,3}, { 4,5,-6,7}, {8,9,-10,11}}  
4  };  
5  
6  // 負の要素を数える  
7  int n {0};  
8  for (size_t i = 0; i < a.size(); i++)  
9      for (size_t j = 0; j < a[i].size(); j++)  
10         for (size_t k = 0; k < a[i][j].size(); k++)  
11             if (a[i][j][k] < 0) ++n;  
12  std::cout << n << "\n"; // 6
```

# 3次元配列と範囲for文

- 添字は不要だが各次元用の変数名が必要
- リファレンス指定の有無が性能に影響する

```
1  vector<vector<vector<int>>> a {  
2      {{0,-1,2,3}, {-4,5,6,7}, {8,-9,10,11}},  
3      {{0,1,-2,3}, { 4,5,-6,7}, {8,9,-10,11}}  
4  };  
5  
6  n = 0;  
7  for (const auto& x : a)  
8      for (const auto& y : x)  
9          for (auto z : y)  
10             if (z < 0) ++ n;  
11  std::cout << n << "\n"; // 6
```

# n x m行列と転置行列

- 入力は $n \rightarrow m$ の順, 出力は $m \rightarrow n$ の順
- 添字の変化に注意

```
% ./a.out
1 2 3 4
5 6 7 8
9 10 11 12

1 5 9
2 6 10
3 7 11
4 8 12
```

```
1  int main() {
2      const int n{3}, m{4};
3      auto data { vector(n, vector(m, 0)) };
4
5      // n x m の 2 次元配列に入力
6      for (int i = 0; i < n; i++) // 行方向
7          for (int j = 0; j < m; j++) // 列方向
8              std::cin >> data[i][j];
9      cout << "\n";
10
11     // m x n の転置行列を出力
12     for (int j = 0; j < m; j++) { // 転置の行方向
13         for (int i = 0; i < n; i++) // 転置の列方向
14             cout << data[i][j] << " ";
15         cout << "\n";
16     }
17     cout << "\n";
18 }
```

## 2x2の逆行列

- `abs()` には `<cmath>` が必要
- `ia` の宣言に注意 (空の `vector<double>` が2個)
- `ia[0] = { 値1, 値2 };`
  - `=` の左辺 `ia[0]` は `vector<double>` なので, 右辺を `{ 値1, 値2 }` とすると, `ia[0][0]` が 値1, `ia[0][1]` が 値2 になる

```
% ./a.out
1 3
5 7

-0.875 0.375
0.625 -0.125
```

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, A^{-1} = \frac{1}{ad - bc} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

```
1  int main() {
2      vector<vector<double>> a { {1.0,3.0}, {5.0,7.0} };
3
4      // a の逆行列が存在するかを判定
5      const double epsilon {1.0e-8};
6      const double det {
7          a[0][0] * a[1][1] - a[0][1] * a[1][0]
8      };
9
10     // double型 == 0 の比較を避ける
11     if (abs(det) < epsilon) {
12         cout << "no inverse\n"; return 1;
13     }
14
15     // a の逆行列を計算
16     vector<vector<double>> ia(2);
17     ia[0] = { a[1][1]/det, -a[0][1]/det };
18     ia[1] = { -a[1][0]/det, a[0][0]/det };
19
20     cout << ia[0][0] << " " << ia[0][1] << "\n"
21          << ia[1][0] << " " << ia[1][1] << "\n\n";
22 }
```

## 2次元配列の引数

- `vector` 配列は何次元でも引数として指定できる
- 基本は値渡し（要素のコピー）なので参照渡しが無効的
- `const vector<T>&` または `vector<T>&`
- ※ 参照渡し（リファレンス）については第5回関数について参照
- ※ 関数呼び出しの実引数の型と、関数の宣言における仮引数の型を意識する

```
1 void incr(vector<vector<int>>& x) {  
2     /* 配列の各要素を +1 する処理 */  
3 }  
4  
5 void print(const vector<vector<int>>& x){  
6     /* 配列のすべての要素を出力する処理 */  
7 }  
8  
9 int main() {  
10     vector<vector<int>> a {  
11         {1,2,3,4,5},  
12         {2,3,4,5,6},  
13         {3,4,5,6,7}  
14     };  
15  
16     incr(a);  
17     print(a);  
18 }
```



## 2次元配列の一部を実引数にする例

- 2次元配列は配列の配列である点を利用

```
% ./a.out  
2 3 4 5 6
```

```
1  // int の 1 次元配列を出力する  
2  void print_int_array(const vector<int>& x) {  
3      for (size_t i = 0; i < x.size(); i++) {  
4          if (i != 0) cout << " ";  
5          cout << x[i];  
6      }  
7      cout << "\n";  
8  }  
9  
10 int main() {  
11     vector<vector<int>> a {  
12         {1,2,3,4,5},  
13         {2,3,4,5,6},  
14         {3,4,5,6,7}  
15     };  
16     print_int_array( a[1] );  
17 }
```

# string配列との類似性

- `string` は1次元配列と同じ扱い
- `string` の `vector` 配列は2次元配列に見える

```
% ./a.out  
e 2 g
```

```
1 // 文字列を 1 文字ずつ間隔をあけて出力する  
2 void print_capital(string s) {  
3     for (size_t i = 0; i < s.size(); i++) {  
4         if (i != 0) cout << " ";  
5         cout << s[i];  
6     }  
7     cout << "\n";  
8 }  
9  
10 int main() {  
11     vector<string> b {"a1cd", "e2g", "hi"};  
12     print_capital( b[1] );  
13 }
```

# 配列を返す関数

- 基本としてはすべての要素をコピーで返す
- 返す配列が局所変数の場合はコピーをせず再利用

```
% ./a.out
3 2
10 11 20 21 30 31
10 11
20 21
30 3
```

```
1  vector<int> input(int n) {
2      vector<int> v(n);
3      for (auto& e: v) std::cin >> e;
4      return v;
5  }
6
7  int main() {
8      int row{}, col{};
9      std::cin >> row >> col; // n x mを入力
10
11     vector<vector<int>> t(row);
12     for (auto& e: t) // 2次元配列の入力
13         e = input(col);
14
15     for (const vector<int>& x : t) {
16         for (int y : x) cout << y << " ";
17         cout << "\n";
18     }
19 }
```