C++プログラミング II

第 13 回 C 言語配列

岡本秀輔

成蹊大学理工学部

学習の背景

- C++の元となったC言語では配列が重要な役割を持つ
  - コンピュータの基本機能を直接とり扱う
  - 高性能、省メモリのプログラムが作れる
- 使い方を誤るとプログラムが異常終了する
  - コンピュータウィルスのターゲットとなった
  - ちょっとした不注意でも脆弱なプログラムとなる

- C++はC言語を拡張して作られている
- C配列を`vector`で安全に置き換える工夫がなされた
  - ほとんどのアプリでは`vector`を使用すべき
  - しかし置き換え機能は完璧ではない
- C配列を学ぶ理由:
  - C配列を使うC++プログラムが未だ多数ある
  - システムプログラムの一部では必須

学んでほしいこと

- 配列を扱うには次の 2 点を分けて考えると良い
  - 連続領域を意識した各要素の操作（要素の複製/更新）
  - データ全体を扱う領域の確保と解放（メモリ管理）
- `vector` はメモリ管理をほぼ自動化するライブラリ
  - 連続領域と添字の関係は学習してきたはず
  - メモリ管理以外の `vector` の内部構造も学習済み

- 連続メモリ領域を意識したプログラミング
  - 要素数固定の配列
  - ポインタと関係
- メモリ管理の基本（次週）

C配列の基本

C 配列の概要

- [] 演算子で要素へアクセスする連続領域
- std::vector と大きく異なる点
  - 静的領域 or スタック領域に要素が割り当てられる
  - コンパイル時に要素数が確定し変更は不可
  - 配列変数への代入や比較ができない
  - size() などのメンバ関数がない
- ポインタと混ぜた使い方が想定される
  - 配列の引数はポインタへの変換となる
  - ポインタ＋連続領域を配列のように使う

```
int a[5] {6,2,3,4,5}; // 要素数 5 の C 配列，サイズ固定

a[0] = 1;  // [] 演算子のみ使用可能
for (int i = 0; i < 5; i++) // a.size() がない
   cout << a[i] <<" ";
cout <<"\n";
```

2 次元配列

- 2 次元配列は 1 次元配列の配列

```
int a[3][4] {{1,2,3,4},{5,6,8,8},{9,10,11,12}};

a[1][2] = 7;  // 次元ごとに添字を指定
for (int i = 0; i < 3; i++) { // 二重ループで処理
    for (int j = 0; j < 4; j++)
        cout << a[i][j] <<" ";
    cout <<"\n";
}
cout <<"\n";
```

- メモリは連続領域として確保される

```
// これは確認用で通常の使い方ではない
for (int i = 0; i < 3*4; i++)
    cout << a[0][i] <<" "; // i>=4 でもエラーではない
cout <<"\n";
```

3 次元配列

- 3 次元配列は 2 次元配列の配列
- 同様に連続メモリ領域に割り当てられる

```
int b[2][3][4] {
    { {1,2,3,4},{5,6,8,8},{9,10,11,12} },
    { {13,14,15,16},{17,18,19,20},{21,22,23,24}}};

b[0][1][2] = 7;
for (int i = 0; i < 2; i++) { // 三重ループで処理
    for (int j = 0; j < 3; j++) {
        for (int k = 0; k < 4; k++)
            cout << b[i][j][k] <<" ";
        cout <<"\n";
    }
    cout <<"\n";
}
```

初期化のルール

- 初期値が要素数に満たない場合に残りは 0 となる
- 多次元配列でも 1 次元配列と同じ初期化が許される
- 初期値だけで要素数を省略する宣言も許される

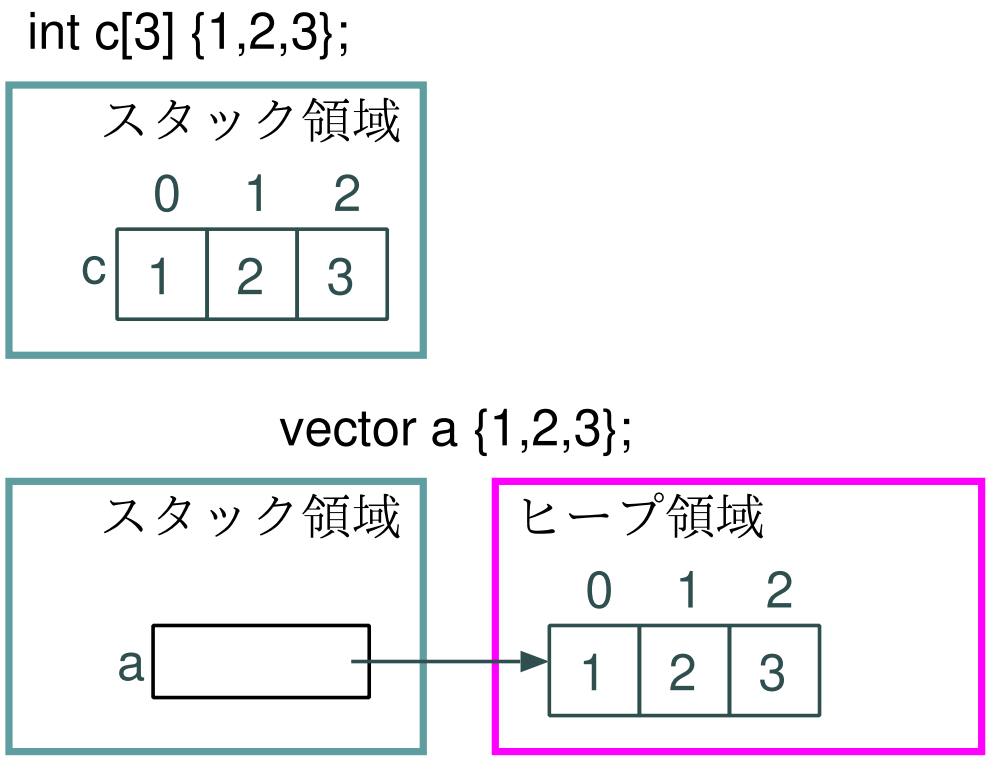
```
int a[5] {1,2};                 // 1 2 0 0 0
int b[2][3] {{1,2},{2,3}}; // 1,2,0,2,3,0

int c[2][2] {1,2,3,4};          // 1 次元で指定
int d[2][4] {0};                // すべて 0, {} でも良い

int e[] {1,2,3,4,5};            // 5 要素
```

- 初期値指定なし配列の初期値は、局所変数なら不定、大域変数なら 0 となる (組み込み型共通)

```
double x[100]; // 不定 または 0.0
```

メモリ割り当てとデフォルト値

- C 配列が割り当てられるメモリ領域 (組み込み型共通)
  - 局所変数ならスタック領域 (関数呼出し時の割り当て)
  - 大域変数なら静的領域 (プログラム開始時の割り当て)
  - ポインタと組み合わせればヒープ領域も可能（次回）
- `vector` との比較（局所変数の場合）



C 配列への入力

- 要素ごとに入力を行う（まとめた入力指定はない）
- 入力数不明でも要素数が増やせない
- 添字の上限チェックが必要

```
    int a[5] {0}; // 5 要素の初期値が 0
    int n {0};    // 入力した個数を覚えておく必要がある

    while (n < 5 && cin >> a[n]) // 添字チェック＋入力
        ++n;

    for (int i = 0; i < n; i++)
        cout << a[i] <<" ";
    cout <<"\n";
}
```

```
$ echo 5 6 7 8 9 10 | ./a.out          要素数より多い入力
5 6 7 8 9
$ echo 8 9 10 | ./a.out                要素数より少ない入力
8 9 10
```

配列の引数

1次元配列の引数

- 引数の宣言では `int a[]` と `int *a` が同じ意味
  - 配列の要素をコピーしないという設計
- 値渡しで配列名の指定はアドレスを渡すことになる
  - 配列のサイズ情報が渡されない
  - 配列の末尾を知る手段が必要

```
void print(int a[], int n){ // n で要素数を指定
    for (int i=0; i < n; i++)
        cout << a[i] <<" ";
    cout <<"\n";
}

void print1(int *p) { cout << *p <<"\n"; }

int main() {
    int x[5] {1,2,3,4,5};
    print(x, 5);    // 1 2 3 4 5
    print1(x);      // 1
    print1(&x[1]); // 2
}
```

多次元配列の引数

- 一番左側以外の次元の要素数を指定する
- 仮引数はその次元の配列へのポインタとなる
- 一番左側の次元の要素数は書いても無視される
  - `void f(int a[3])` は `void f(int a[])`
  - `void g(int a[2][3])` は `void g(int a[][3])`
  - `void h(int a[4][2][3])` は `void h(int a[][2][3])`

```
// 引数の型は int[3] の配列へのポインタ: int(*a)[3]
void print2d(int a[][3], int n){ // n で要素数を指定
    for (int i=0; i < n; i++)
        for (int j = 0; j < 3; j++)
            cout << a[i][j] <<" ";
    cout <<"\n";
}

int main() {
    int x[2][3] {{1,2,3},{4,5,6}};
    print2d(x, 2);  // 1 2 3 4 5 6
}
```

C 配列とポインタ

配列名のルール

- 配列名は単独で先頭要素のアドレスとなる
  - ポインタ変数への代入文に使われる
  - 値渡しの実引数もアドレスを渡したことになる
  - 初期値の指定してはリファレンス以外アドレスになる

```
void print(int* p) { cout << *p <<"\n"; }

int main() {
    int a[4] {3,1,2,4}; // 型は int[4]
    print(a);           // a の先頭アドレス

    int* b {nullptr};
    b = a;              // a の先頭アドレスを代入
    int* c {a};         // 初期値は a の先頭アドレス
    cout << *b <<" "<< *c <<"\n";   // 3 3
}
```

比較

- 配列名の比較はアドレスの比較となる
  - 中身の要素値は比較と無関係
- アドレスなのでポインタとの比較ができる
- 連続領域内でなければ大小比較は無意味
  - コンパイルでき、実行時のエラーもない
  - 個々の配列のアドレス順番はコンパイラ次第で変わる

```
int main() {
    int  a[4] {3,1,2,4};
    int  b[4] {3,1,2,4};
    int* p {a};      // p はポインタ
    cout << std::boolalpha
         << (a == b) <<" "    // false
         << (a <  b) <<" "    // ?   (比較の意味無し)
         << (a == p) <<"\n"; // true
}
```

配列とは何か？(確認)

`int a[5];`(初期値なしの宣言) を例に考える

- **同一の型**: 配列は 1 つ以上の同じ型の要素の並び
- **連続領域**: 各要素はメモリ上で隣り合っている
  - `&a[0]` と`&a[1]` はバイト単位で `sizeof(int)` だけ異なる
- 配列名を式で使うと先頭要素のアドレスと解釈される
  - `int a[5];` ならば `a == &a[0]`
  - `a` の型は `int[5]`、`a` を式で使うと `int*`に変換される (型については混乱しそう)
- `a` の要素値は 0(大域変数) または不定 (局所変数)

```
int a[5];
cout << a <<"\n"
     << &a[0] <<" " << &a[1] <<"\n" // アドレス
     << std::boolalpha
     << (a == &a[0]) <<"\n"; // true
for (int i=0; i<5; i++)
   cout << a[i] <<" "; // 何がでるか?
```

連続領域内のポインタ演算

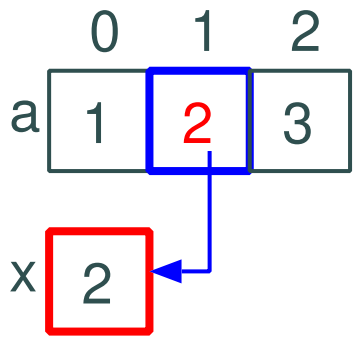
- 連続領域を指すポインタには加減算が使用可
  - 配列を指すポインタに使用する
  - 演算子の多重定義ではなく機械語に対応する
- ポインタ p と整数 n の加減算：p ± n
  - p の指す要素の型で n 要素先のアドレス
  - p の指す要素の型で n 要素前のアドレス
- ポインタどうしの減算：q-p
  - 半開範囲 [p,q) の要素の個数
  - p+n == q の関係

```
int  a[4] {0,1,2,3};
int* p{a};
cout << std::boolalpha
     << *(p+2)              <<" "  // 2
     << ((p+2) == &a[2]) <<" "; // true
int* q { a + 4 }; // 末尾要素の次
cout << q - p <<"\n"; // 4
```

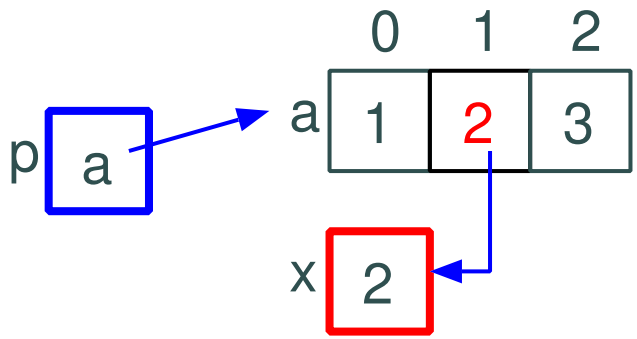
配列に見せるポインタの演算子

- 連続領域を指すポインタには [] 演算子が使用可
  - 演算子の多重定義ではなく機械語に対応する
- ポインタ p において p[n] は*(p+n) を意味する
  - コードの見た目では配列と区別が付かなくなる
  - 配列アクセスとは異なる機械語が生成される

```
int a[3] {1,2,3};
x = a[1];
```




```
int a[3] {1,2,3};
int* p { a };
x = p[1];  // x = *(p + 1)
```



類似と相違

- ポインタは配列のように振る舞うが配列ではない
- a には要素数の情報がある
- p にはない（要素の型とアドレスのみ）

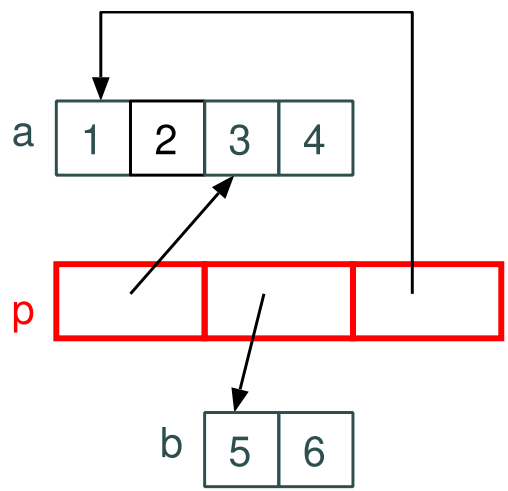
```
int a[3] {1,2,3};
int *p {a};
cout << a[2] <<" "<< p[2] <<"\n"; // 同じ
cout << sizeof(a) <<"\n"; // int のサイズ*3
cout << sizeof(p) <<"\n"; // ポインタのサイズ
```

ポインタ配列

- ポインタ配列は多次元配列に見える

```
int  a[2][2] {{1,2},{3,4}};
int  b[2]    {5,6};
int* p[3]    {a[1], b, a[0]};
for (int i = 0; i < 3; i++)
    cout << p[i][0] <<" "
         << p[i][1] <<"\n";
```

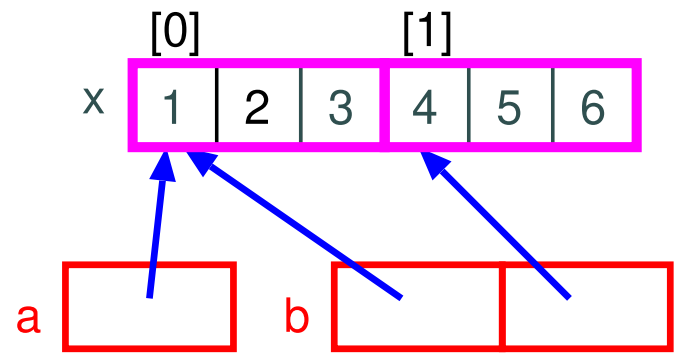
```
3 4
5 6
1 2
```



確認

```
int x[2][3] {{1,2,3},{4,5,6}};
int (*a)[3] {x};           // a は int[3] 配列へのポインタ
int* b[2]   {x[0],x[1]}; // b はポインタの配列

for (int i = 0; i < 2; i++) // 使い方は同じ
   for (int j = 0; j < 3; j++)
      cout << a[i][j] <<" "<< b[i][j]<< "\n";
```

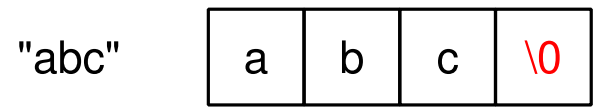
- a は int[3] を指すが、その個数は不明
  - f(int a[][3]) と同じ状況 (引数だけこの形式が OK)
- b は 2 個のポインタ、その先の要素数は不明



文字配列

文字列リテラル

- ヌル文字で終端する char 型の配列を C 文字列と呼ぶ
- 文字列リテラルは C 文字列である
  - 要素の末尾にヌル文字 ('\0') が加わる
  - 静的領域に割り当てられ、値の変更不可 (const)
  - 例："abc"の型は const char[4]



- 初期値に使う場合には注意が必要 (15 頁と同じルール)

```
std::cout << "abc";       // abc は静的領域に割り当て
char  a[4]     {"abc"}; // a は変数に対応する領域
const char* b  {"abc"}; // b は静的領域を指す変数
                        // 局所変数ならスタック領域
std::string c  {"abc"}; // 静的領域の文字列で初期化
```

C 文字列と std::cout

- cout への C 文字列の指定は、ポインタ指定に変わる
  - operator<<関数の呼び出しで実引数はアドレス
- cout に char*のアドレスを指定すると、
  - 先頭要素からヌル文字の手前まで出力する
  - ヌル文字がない場合には暴走する可能性がある
  - 他のポインタ型の指定ではアドレス値が出力される

```
#include <iostream>
int main() {
    char a[4]      {"abc"};
    const char* b {"abc"};
    std::cout << a <<" "<< b <<"\n"; // abc abc
}
```

C 文字列と std::cin

- cin への C 文字列の指定はリファレンスで渡される
  - 扱いは char*と同じで、要素数の情報が使用されない
  - 要素数以上の入力があれば暴走する可能性が高い
- 入力には配列の要素数以下の文字数を必ず指定する
  - setw() 指定で要素数-1 まで入力を受け付ける

```
#include <iostream>
#include <iomanip>
int main() {
    char a[5];
    // std::cin >> a;  は脆弱
    std::cin >> std::setw(sizeof(a)) >> a;
    std::cout << a <<"\n";
}
```

```
$ echo abc | ./a.out
abc
$ echo abcde | ./a.out
abcd
```

C文字列用ライブラリ関数

- <cstring>ヘッダファイル
- 以下はその一部 (引数 size_t n は処理する文字数の上限)

関数宣言	処理内容
`size_t strnlen(const char*, size_t n)`	ヌル文字までの文字数
`int strncmp(const char* s1, const char* s2, size_t n)`	文字列比較
`char* strncpy(char* d, const char*s, size_t n)`	文字列の複製
`char* strncat(char* d, const char*s, size_t n)`	文字列の連結
`void* memchr(const char*s, int c, size_t n)`	文字位置の特定
`void* memrchr(const char*s, int c, size_t n)`	逆順文字位置の特定

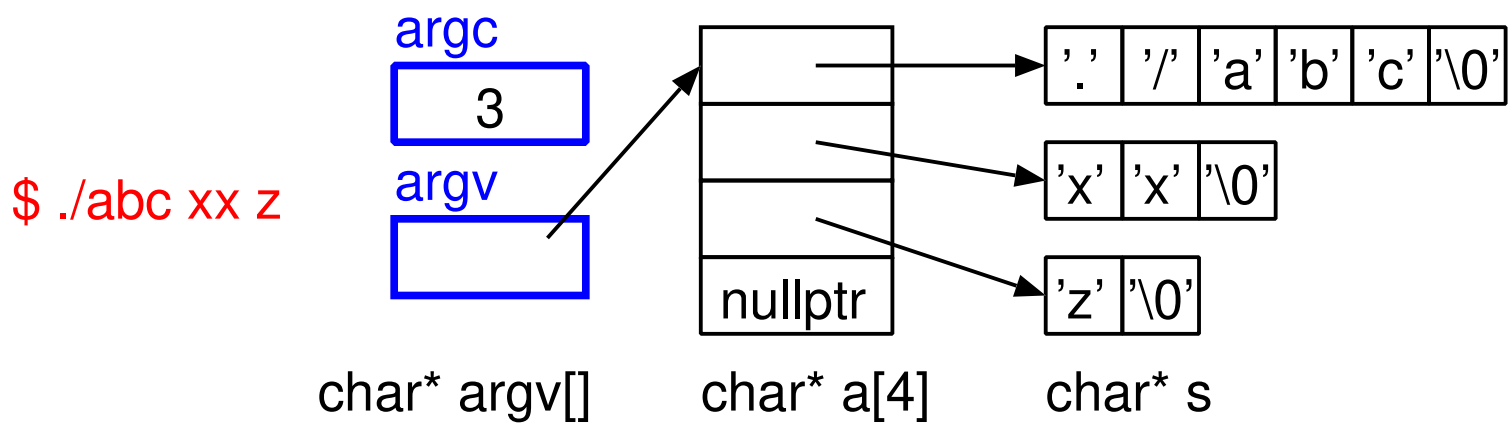
- C 言語では単独の文字は int 型として扱う
- 元々strXXX() という文字数上限なしの関数群であったが、安全のために strnXXX() という文字数を指定する関数が追加された
- std::string の size(), 比較演算, 代入, +=, find(), rfind() に相当する

main 関数の引数

第 1 引数 コマンド名＋コマンド引数の個数
第 2 引数 コマンド名とコマンド引数の文字列情報

- 文字列へのポインタ配列を指すポインタ
- argv[0] はコマンド名
- argv[argc] は nullptr

```
int main(int argc, char *argv[])
{
   for (int x = 0; x < argc; x++)
      cout <<"argv["<< x <<"]="<< argv[x] <<"\n";
}
```



std::string と C 文字列

- `std::string` の ctor は文字列リテラルが引数
- メンバ関数 `c_str()` はオブジェクト内部の C 文字列の先頭アドレスを返す
  - `const char*`型
  - 必ずヌル文字で終端する
    - `c_str()+size()` にはヌル文字がある
    - 有効範囲は `[c_str(), c_str()+size()]`

```
std::string x("abc");
cout << x << "\n"; // abc
const char* p { x.c_str() };
char ch { *(p + x.size()) };
cout << p <<"\n"  // abc;
     << std::boolalpha
     << (ch == '\0') // true
     << "\n";
```

C++固有の機能とC配列

STL で使うために

- C 配列はメンバ関数を持たない
- size(), begin(), end() 関数が提供される
  - begin(),end() があるので範囲 for 文も利用可
  - これらの関数は vector などのコンテナにも適用可能

```
int a[5] {1,2,3,4,5};

// 要素数の取得
for (size_t i = 0; i < std::size(a); i++)
   cout << a[i] <<" ";
cout <<"\n"; // 1 2 3 4 5
// 範囲 for 文
for (auto e:a) cout << e <<" ";
cout <<"\n"; // 1 2 3 4 5
// STL アルゴリズム
auto b { std::begin(a) }, e{ std::end(a) };
auto it{ std::find(b, e, 3) };
cout << *it <<"\n"; // 3
```

std::array

std::array は C 配列と std::vector の間の子

- C 配列のメモリ割り当てと初期化ルール (一部)
  - サイズ変更不可
  - 静的領域またはスタック領域に割り当て
  - 初期値なしは不定値
- コンテナのインタフェース, ランダム・イテレータ

```
#include <algorithm>
#include <iostream>
#include <array>
int main() {
    std::array a{1,2,3,4,5};
    std::array<int,8> b;     // 要素数 8, 初期値不定
    size_t n{a.size()};
    std::copy(a.begin(), a.end(), b.begin());
    for (size_t i = 0; i < n; i++)
        std::cout << b[i] <<" ";
    std::cout <<"\n"; // 1 2 3 4 5
}
```

リファレンス引数

- リファレンス引数とすれば型情報をすべて引き継げる
  - この print2d は 2x3 の配列専用の関数
  - 同じ 2 次元でも要素数が異なる配列では使えない
  - この形の関数を作ることはほとんどない

```
void print2d(int (&a)[2][3]){ // 自由度が低い
    for (auto& d1: a)           // d1 の型は int[3]
        for (int e: d1)
            cout << e <<" ";
    cout <<"\n";
}

int main() {
    int x[2][3] {{1,2,3},{4,5,6}};
    print2d(x);  // 1 2 3 4 5 6
}
```

関数テンプレートとリファレンス引数

- 関数テンプレートならば vector と兼用に作れる
- 機械語は呼び出しの種類数だけ作られる

```
template<typename T>
void print2d(T& a) {  // 比較のために const なし
   for (auto& d1: a)
      for (auto e: d1) // ここも auto にする価値あり
         cout << e <<" ";
   cout <<"\n";
}

int main() {
   int x[2][3] {{1,2,3},{4,5,6}};
   print2d(x);    // 1 2 3 4 5 6

   std::vector<std::vector<int>> y {{1,2,3},{4,5,6}};
   print2d(y);    // 1 2 3 4 5 6
}
```

ランダム・イテレータとの比較

- 頁 18 との比較
- ランダム・イテレータは配列のポインタがモデル
- *演算子とイテレータの加減算で使い方が同じ
- イテレータ p において p[n] は*(p+n) と同じ結果
  - &a[2] は要素の (ヒープ領域上の) アドレス
  - (p+2) はイテレータであってポインタではない
  - *(p+2) は添字 2 の要素のリファレンス
  - &(*(p+2)) は添字 2 の要素のアドレス

```
std::vector a {0,1,2,3};
auto p {a.begin()};
cout << std::boolalpha
     << *(p+2)                    <<" "  // 2
     << (&(*(p+2)) == &a[2]) <<" "; // true
auto q {a.end()}; // 末尾要素の次
cout << q - p <<"\n"; // 4
```

まとめ

自分の言葉でまとめましょう。

- C 配列の基本
- 配列の引数
- C 配列とポインタ
- 文字配列
- C++固有の機能と C 配列