第 8 章

配列

TODO: 似た処理を書き並べて、変化する部分を配列に置き換える

TODO: 配列の必要数が事前に判明しなければ、多いめに確保する

駐車場には駐車スペースが多数あるでしょう。駐車開始時刻を変数で覚えるとしたら、駐車スペースの数だけの変数が必要です。

隣の駐車スペースでも、同じ駐車場内であれば料金計算などの方法は共通でしょう。プログラム上でも共通の処理で表現したいところです。しかし、変数名が違うものであれば、何度も似たような処理を書き並べることになって煩雑です。

駐車場をいくつも管理するなら、駐車場ごとに駐車スペースの数は異なることでしょう。プログラムを駐車場ごとに修正するとしたら、これまた手間がかかります。

この章では、複数並んだ変数を定義する配列について説明します。配列は、同種のデータを規則的に扱えることに意味があります。

キーワード

- 配列の定義・初期化子
- 要素数
- 添字（要素番号）
- マクロ定数
- 0 始まり・1 始まり
- 多次元配列・配列の配列
- 参照渡し

8.1 配列の定義と初期化

複数の変数をまとめて扱うための仕組みが**配列** (array) です。配列を定義するには、変数定義のときの変数名に続けて、角括弧 [ ] と、変数の個数を書きます。右下の例のように [3] と書けば 3 個の変数ができあがります。できあがった個々の変数を、配列の**要素** (element) と呼びます。[ ] の中に書いた数は**要素数** (number of elements) です。

```
/* 文法 */
型名 変数名[要素数];
```

```
/* 実例 */
int a[3];
```

配列の要素 1 つを指すには、何番目の要素かを示す必要があります。このときにも [ ] と要素の番号を書きます。この要素番号を**添字** (subscript) とか**インデックス** (index) といいます。添字は、C 言語では **0 から**数え始めることになっているので、3 個の要素を用意したら、有効な添字は 0, 1, 2 です。

```
/* 普通の変数（定義と代入） */
int a0, a1, a2;
a0 = 0;
a1 = 10;
a2 = 20;
```

```
/* 配列変数（定義と代入） */
int a[3];   // [3]は要素数
a[0] = 0;   // [0]は添字
a[1] = 10;  // [1]は添字
a[2] = 20;  // [2]は添字
```

関数内で定義した**配列要素の初期値**は、普通の変数と同じく**不定**ですから、代入してから使います。規則正しい値ならループ処理で代入できます。規則性のないデータを列挙するなら**初期化**（変数定義と同時の代入）が便利です。値を書き並べてブロックで囲います。この列挙された個々の値を**初期化子** (initializer) といいます。配列の要素数を初期化子の個数に合わせるなら、**要素数を省略**できます。

```
/* 普通の変数（初期化） */
int a0 = 0, a1 = 10, a2 = 20;
```

```
/* 配列変数（初期化） */
int a[3] = { 0, 10, 20 };
int b[]  = { 0, 10, 20 }; //省略可
```

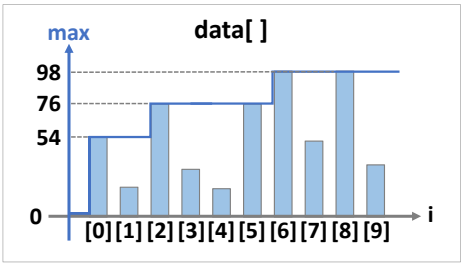
頻出ミス

(1) 初期化子は、変数定義と分離できません。(2) 配列同士の代入もできません。配列全体をコピーするには、要素を一つずつ代入します。（☞8.4.3 項）

```
int a = 1, b;
b = 1; // OK
b = a; // OK
```

```
int a[3] = {1,2,3}, b[3];
// b[3] = {1,2,3}; // NG(1)
// b = a;          // NG(2)
```

MEMO: 初期化子なしのサンプルが少ないとの指摘あり。



配列は、名前（変数名）ではなく、番号（添字）で変数を区別するものです。これを利用すると、例えば、いくつもの変数の合計が**ループ処理**で求められます。

```
/* 普通の変数（合計） */
int sum = a0 + a1 + a2;
```

```
/* 配列変数（合計） */
int sum = 0;
for (int i=0; i<3; i++) {
    sum = sum + a[i];
}
```

普通の変数では、3 個版、4 個版と、個数に応じて違う式を用意せねばなりませんが、配列なら、ループ回数が違うだけの**同じ処理**ですみます。

頻出ミス

int a[3]; と用意した配列は a[3] の要素を使ってはいけません。a[-1] もダメです。しかし、使ってもコンパイル時には**エラーも警告も出ない**のが普通です。実行時は、動きがおかしくなる可能性があります。（Java なら確実に実行時エラーです。）C 言語では、実行効率を優先し、不正な添字を**確実に検出する手段がありません**。

8.1.1 配列要素の最大値

配列要素の最大値を求めてみましょう。合計を求めるのに、先ほどはループ中に暫定の合計値を更新していきました。同じ発想で、**暫定の最大値を更新**してみます。

ソースコード 8.1 では、100 点満点のテストの点数のデータ 10 個を data[] に用意しました。max を暫定最大値とするので、0 で初期化します。最初は 0 でも −100 でも、小さい値なら何でもよくて、max はループ処理中に徐々に大きくなります。

ループ処理では、data[i] を順に調べて、max より大きいときに max を更新します。変数の値の変化をよく理解してください。

ソースコード 8.1 の実行結果

```
data[0] = 54, max = 54
data[1] = 20, max = 54
data[2] = 76, max = 76
data[3] = 32, max = 76
data[4] = 19, max = 76
data[5] = 76, max = 76
data[6] = 98, max = 98
data[7] = 51, max = 98
data[8] = 98, max = 98
data[9] = 35, max = 98
max = 98
```

ソースコード 8.1　配列の最大値

```
1  #include <stdio.h>
2
3  int main(void) {
4      int data[10] = {          // 0から100の点数のデータ
5          54, 20, 76, 32, 19, 76, 98, 51, 98, 35
6      };
7      int max = 0;              // 暫定最大値
8      for (int i=0; i<10; i++) {
9          if (max < data[i]) { // より大きな値を見つければ
10             max = data[i];   // 暫定最大値を更新する
11         }
12         printf("data[%d] = %d, max = %d\n", i, data[i], max); // 経過
13     }
14     printf("max = %d\n", max); // 最終結果
15 }
```

8.2　要素数とマクロ定数

ところで、ソースコード 8.1 には気になるところがあります。「10」という数値が 2 ヶ所にあります。4 行目の配列要素数と、8 行目のループ回数です。この 2 つが食い違うと、プログラムは正しい動作ができません。もっというと、5 行目のデータの個数を数えて、その値と一致させねばなりません。今は絶妙にバランスをとっていますが、将来、ちょっとしたことで破綻するでしょう。これはプログラムの**構造上の欠陥**です。

この「10」のような、恣意的で、それだけでは意味のわからない数値のことを、**マジックナンバー** (magic number) と呼んで、プログラマは忌み嫌います。**欠陥の隠れているサイン**だからです。マジックナンバーでなくすためには、**意味のわかる名前**をつけます。例えば下の例のように、N_DATA というマクロを定義して、「10」を置き換えます。すると、マクロで 2 ヶ所の数値がいつでも同じになるので、欠陥が解消されます。

```
#define N_DATA 10    /* 要素数をマクロで定義 */
int data[N_DATA] = { 54, 20, 76, 32, 19, 76, 98, 51, 98, 35 };

for (int i=0; i<N_DATA; i++) { ... }
```

配列要素数は**定数式** (constant expression) （コンパイル時に値の決まる式）で指定する必要があるので[*1]、このように**マクロ定数**を使います。N_ の接頭辞は、number of の意味で、個数を表すのによく使われます。

[*1] 変数のように、実行時まで値の決まらない式でも、可変長配列（☞5 ページのコラム）によってコンパイルエラーにならない場合もありますが、今後のことを考えると、使用は控えたほうがよいでしょう。

鋭意作成中

コラム： マクロの定数式

マクロ定数には、定数を用いた式を定義しても構いません。ただし、マクロはソースコード上の**文字の置き換え**を指示するものなので、式の中では優先順位が考慮されません。ですから、定義する**式の全体をカッコで囲う**のが安全です。

```
#define SIZE 1+1 // ×
int a[SIZE * 2]; // 1+1*2 = 3
```

```
#define SIZE (1+1)//○
int a[SIZE * 2];  //(1+1)*2 = 4
```

頻出ミス

マクロ定義の**行末のセミコロン** (;) は、文字の置き換え指示としては正当です。しかし、このマクロが配列の要素数に現れると**コンパイルエラー**です。「']' より前に';' が現れた」のようなメッセージからは原因がわかりにくいので要注意です。

```
#define SIZE 10; // 原因
// int a[SIZE]; // ここでエラー
```

↔

```
// int a[10;]; // NG
```

コラム： 可変長配列

C99 では、配列がローカル変数ならば、要素数に**実行時に決まる式（変数）**を指定しても大丈夫です。**可変長配列** (variable-length array) という機能です。

```
/* 配列がグローバル変数 */
int n = 10;
// int a[n]; //コンパイルエラー
```

```
/* 配列がローカル変数 */
int func(int n) {
    int a[n]; // C99のみOK
```

ただし、C11 でオプション機能に格下げになりました。Visual C++ は C99 に準拠しておらず、この機能も意図的にサポートしていません。C++ 規格にも取り込まれず、**セキュリティ上の懸念**もあるため、将来的には消え去る機能でしょう。

8.2.1　初期化子の個数

先ほどのソースコード 8.1 には、まだ問題が残っています。マジックナンバーをマクロで置き換えたとしても、データを追加あるいは削除すると、個数を数えなおさねばなりません。手作業だと間違えそうなので、プログラム（コンパイラ）にやらせましょう。

配列要素数を省略すると、初期化子の個数ぴったりの配列になります。この配列の要素数を計算する慣用句があります。sizeof 演算子で、**変数の占める領域の大きさ**（バイト数）がわかります[*2]。「sizeof(data)」が配列全体の大きさ、「sizeof(data[0])」が要素 1 個の大きさなので、この **2 つの割り算が要素数**になります。8 行目のループの回数をこの要素数で置き換えると、個数を気にせずデータを書き並べてよくなります。

```
int data[] = { 54, 20, 76, 32, 19, 76, 98, 51, 98, 35 };
int num = sizeof(data)/sizeof(data[0]);  /* 要素数を計算で求める */

for (int i=0; i<num; i++) { ... }
```

ところで、配列の**最後の初期化子の後にコンマ**は不要ですが、書いてもエラーになりません。初期化子を行単位で入れ替えたり追加削除するような場面では、すべての行末にコンマを書いた方が統一的ですので、気にせず書いておきましょう。

```
int data[] = {
    54, 20, 76, 32, 19,
    28, 38, 51, 98, 35, // 最後のコンマは余分だが、書いてもよい
//  0,  // ここを有効にしたときに、上の行の最後のコンマが役立つ
};
```

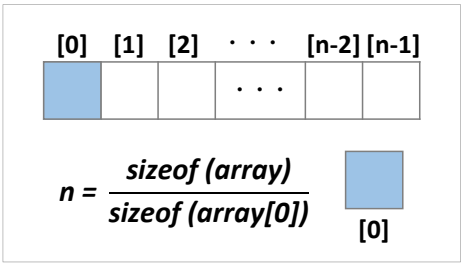
コラム：初期化子の過不足

配列変数の定義で、**要素数**と**初期化子**は、どちらか片方を指定すれば十分ですが、両方とも指定して初期化子が足りなければ、必要なだけ**末尾に 0 が補われます**。ただし、初期化子を空にすることは、言語規格ではなぜか許容されていません[a]。したがって、**すべての配列要素を 0 で初期化**する慣用句は、初期化子をひとつだけ書く「int a[100] = {0};」[b]です。

逆に、初期化子が余ると、コンパイルエラーか警告になります。

[a] 今でも警告しないコンパイラも多く、C23 では言語規格で許容される見込みです。
[b] C 言語の言語規格に強く依存した手法なので、ループ処理で 0 を代入するのもよいでしょう。

[*2] sizeof 演算子は、**??**節では型に使いましたが、変数でも計算できます。型の場合は「sizeof(int)」のように () が必要ですが、変数では「sizeof data」と省略しても構いません。
なお、sizeof は size_t 型（符号なし整数、☞**??**節）なので、演算結果を int 変数に代入する際には、厳密には int へのキャストが必要です。



個数を数えずに、末尾の目印に**番兵** (sentinel) と呼ばれる、データとしては出現するはずのない数値を配置する手法もあります。0 以上のデータを扱っていれば、例えば –1 を末尾に配置して、ループの条件を「データが 0 以上」とするわけです。

```
int data[] = { 54, 20, ,..., 98, 35, -1 };  // 末尾の-1が番兵
for (int i=0; data[i]>=0; i++) { ... }
```

8.3 関数に配列を渡す

配列を、関数の引数に渡すことができます。次のようにします。

仮引数 受け取り側は、関数の仮引数に、変数名の後に **[要素数]** を書きます。変数定義と似てますが、**要素数を省略**できる場合があります。(☞8.3.1 項)

実引数 呼び出し側は、配列の変数を用意して、実引数には**変数名だけ**を書きます。

```
/* 普通の引数 */
void func(int x);

int main(void) {
    int x = 0;
    func(x); // xはint
    ...
```

```
/* 配列の引数 */
void func2(int a[5]);

int main(void) {
    int a[5] = {0,0,0,0,0};
    func2(a); // aはint配列
    ...
```

頻出ミス

配列を渡すのに、変数名に要素数までつけて a[5] のようにすると、要素 1 つを取り出したことになります。(しかも添字の範囲外アクセスです。) [ ] の役目は、変数定義では**要素数**、式に現れると**添字**(何番目)と違うので、要注意です。

```
void func(int x);
...
  int a[5] = {0,0,0,0,0};
  func(a[0]); //OK, a[0]はint
```

```
void func2(int a[5]);
...
  int a[5] = {0,0,0,0,0};
//func2(a[5]); //NG, a[5]はint
```

8.3.1　配列要素数は管理されない

ここで思い出したいのは、（関数と無関係な）C 言語のそもそもの要素数の扱いです。例えば 5 個しか用意していない配列要素の 10 番目を使っても**検出機能がなく**、多くの場合コンパイルエラーにも、実行時エラーにもなりませんでした。（☞3 ページの頻出ミス）

そして関数に配列を渡しても、関数側では**要素数を感知できません**。つまりプロトタイプ宣言での配列要素数は、いくつを指定しても同じことで、**省略すら可能**です[*3]。そのため、要素数が必要なときの慣用句としては、右の例のように、**別の引数**で渡します[*4]。

```
void func3(int num, int a[]);

int main(void) {
    int a[5] = {0,0,0,0,0};
    func3(5, a);
    ...
```

8.3.2　配列は参照渡し

関数とのやりとりに配列を用いると、次のような動作になります。

引数 配列自体をコピーせず、メモリ上のアドレス（ポインタ）が渡されます。参照渡し（call by reference）相当になって、関数で、呼び出し元の配列を変更できます。

戻り値 配列自体は返せません。return 文に配列名を与えると、アドレスが返されます。

これまで、引数や戻り値はコピーされましたが、配列では、全要素の値をコピーすると非効率ですから、このような動作になっています。関数が配列を返すには、本来は入力であるはずの**引数を出力に転用**します。例えば、配列の n 個の要素を 0 で埋める関数は右のようになります。

```
/* n個の要素を0にする */
void clear(int n, int a[]) {
    for (int i=0; i<n; i++) {
        a[i] = 0;
    } // ↑呼び出し元の配列が変化
}
```

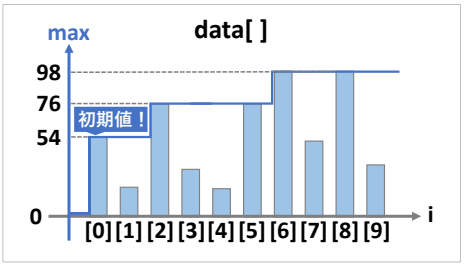
また、**??**項でできなかった、関数でのスワップ（値の入れ替え）が、配列なら可能です。

```
/* a[] の i,j 要素を入れ替える */
void swap(int a[], int i, int j) {
    int tmp = a[i]; a[i] = a[j]; a[j] = tmp;
}
```

もっとも、関数でスワップするには、無理に配列に頼らず、**??**章で学ぶポインタを使います。あくまでも、配列は「ポインタのような性質を持っている」と理解してください。

[*3] 正確には、多次元配列では最初の 1 次元の要素数だけ省略できます。（☞8.6.1 項）

[*4] C23 では `func(int num, int a[num]);` のプロトタイプで、a の要素数を num とした不正アクセスの検査をする提案があります。どのように実現されるのかは、まだ不透明です。



8.4 配列を順番に操作する

8.4.1 配列の最大値を返す関数

4 ページのソースコード 8.1 を関数版に書き換えてみましょう。

ソースコード 8.2 では、最大値を返す関数 max_array() を作ってみます。引数は、**配列サイズ**と **int 配列**です。関数内の処理を見てみましょう。**暫定最大値** max を小さな値で初期化したいのですが、配列がテストの点数とは限らない状況を思い浮かべると、0 でも大きすぎる可能性があります。そこで 5 行目では、配列の**先頭要素**を使ってみました。これは**慣用句**です。6 行目のループは i=0 を除いてよくなります。ただし n=0 だと 5 行目の x[0] が不正アクセスになるので、本来なら検出してエラー処理すべきところです。

ソースコード 8.2 配列の最大値（関数版）

```
1  #include <stdio.h>
2
3  /* x[0]..x[n-1] の最大値を返す（n>0）*/
4  int max_array(int n, int x[]) {
5      int max = x[0];            // 初期値は先頭要素
6      for (int i=1; i<n; i++) { // i=0 は不要
7          if (max < x[i]) {      // より大きな値を見つければ
8              max = x[i];        // 暫定最大値を更新する
9          }
10     }
11     return max;
12 }
13
14 int main(void) {
15     int data[] = {  // 0から100の点数のデータ
16         54, 20, 76, 32, 19, 76, 98, 51, 98, 35, // 末尾のコンマは有益
17     };
18     int num = sizeof(data)/sizeof(data[0]); // 要素数を計算で求める
19     printf("max = %d\n", max_array(num, data));
20 }
```

8.4.2　0 始まり・1 始まり

毎月の日数を配列に保存します。（簡単のため、閏年は除外して平年だけを考えます。）12 ヶ月分のデータが必要ですから、配列の要素数を 12 にしてみます。添字は 0 始まりなので、1 月は 0 番めです。月名と日数を表示しようとすると、ソースコード 8.3 のように、月の数字[*5]と、配列の添字がずれるので、+1 や -1 の**調整項が必要**になります。

ソースコード 8.3　月ごとの日数（0 始まり）

```
#include <stdio.h>

#define N_MONTH 12
int days[N_MONTH] = { /* 0オリジン */
    31, 28, 31, 30, 31, 30, 31, 31, 30, 31,  30,  31
};//1月 2月 3月 4月 5月 6月 7月 8月 9月 10月 11月 12月

int main(void) {
    for (int i=1; i<=N_MONTH; i++) {
        printf("%d月は%d日あります\n", i, days[i-1]); // 調整項あり
    }
}
```

そこで、せっかく確保される**0 番めの要素は捨てて**しまって、配列要素を 1 オリジン（1 始まり、☞**??**ページのコラム）で使ってみましょう。確保する要素は 13 個に増えます。ソースコード 8.4 では、0 番めの初期化子は -1 と、明らかに無効な値にしました。1 回だけの配列定義には +1 の調整項がありますが、何度も使う**ループ処理はすっきり**します。

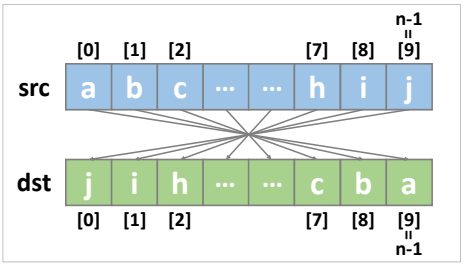
ソースコード 8.4　月ごとの日数（1 始まり）

```
#include <stdio.h>

#define N_MONTH 12
int days[N_MONTH + 1] = { /* 1オリジン */
     -1,   31, 28, 31, 30, 31, 30, 31, 31, 30, 31,  30,  31
};// dummy 1月 2月 3月 4月 5月 6月 7月 8月 9月 10月 11月 12月

int main(void) {
    for (int i=1; i<=N_MONTH; i++) {
        printf("%d月は%d日あります\n", i, days[i]); // 調整項なし
    }
}
```

コメント中の **dummy** は「場所だけ確保して、値に意味のない」くらいの意味のプログラミング用語です。カタカナで「ダミー」とも表記します。

[*5] 英語圏では、月を名前（Jan., Feb., ...）で呼ぶので、0 オリジンで違和感がないようです。そのためか（C 言語に限らず）日付取得のライブラリ関数（☞**??**節）などで、月は 0 オリジン、日は 1 オリジンと、不統一なことがよくあります。

dst[] の添字			src[] の添字		
i の式	n の式	$n=10$	$n=10$	n の式	n, i の式
i	0	0	9	$n-1$	$(n-1)-i$
i	1	1	8	$n-2$	$(n-1)-i$
i	2	2	7	$n-3$	$(n-1)-i$
⋮	⋮	⋮	⋮	⋮	⋮
i	$n-3$	7	2	2	$(n-1)-i$
i	$n-2$	8	1	1	$(n-1)-i$
i	$n-1$	9	0	0	$(n-1)-i$



8.4.3 配列の正順・逆順コピー

配列をコピーしたければ、要素を一つずつ代入します。配列を関数に渡すと参照渡しになるので、ソースコード 8.5 の array_copy() のような関数が作れます。コピー元の src は source（源泉、情報源）、コピー先の dst は destination（目的地）の省略形です。

逆順にコピーするには、添字をうまく計算すれば大丈夫です。まず、$n=10$ のような具体的な値で dst[0]=src[9], ..., dst[9]=src[0] を実行したい、と考えます。次に、dst[] に i を使うことにします。そして、src[] の添字を n と i の式で表せれば、ループ処理の完成です。逆順の処理にどうしても -1 **の調整項**が出てくることは、慣用句とも言えます。

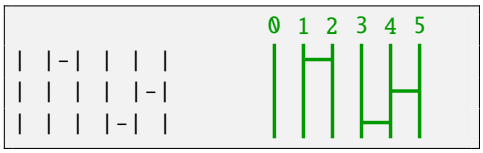
ソースコード 8.5 配列要素のコピー

```
1  #include <stdio.h>
2
3  void array_copy(int n, int dst[], int src[]) {
4      for (int i=0; i<n; i++) {
5          dst[i] = src[i]; // 1要素ずつコピー
6      }
7  }
8
9  void array_reverse(int n, int dst[], int src[]) {
10     for (int i=0; i<n; i++) {
11         dst[i] = src[(n - 1) - i]; // 逆順にコピー
12     }
13 }
14
15 #define NUM ((int)(sizeof(orig)/sizeof(orig[0])))
16
17 int main(void) {
18     int orig[] = { 1, 3, 5, 7, 9, 2, 4, 6, 8 };
19     int copy[NUM];     array_copy(NUM, copy, orig);       // コピー
20     int reverse[NUM];  array_reverse(NUM, reverse, orig); // 逆順
21     for (int i=0; i<NUM; i++) {   // 結果の表示
22         printf("%5d %5d %5d\n", orig[i], copy[i], reverse[i]);
23     }
24 }
```

8.4.4　あみだくじ

あみだくじの道を表示するプログラムです。ソースコード 8.6 では、6 人の行き先を示す縦の道を 6 本と、隣の人の入れ替えを示す横の道を 3 ヶ所描きました。

ソースコード 8.6 の実行結果



ソースコード 8.6　あみだくじの道を表示する

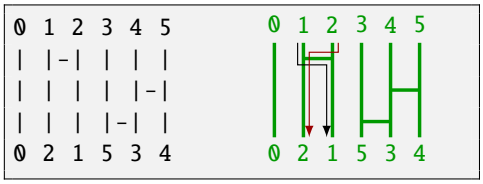
```
#include <stdio.h>
#define N_PEOPLE 6

/* 左から pos 番目と pos+1 番目の人の間には - を描く */
void bar_print(int pos) {
    for (int i=0; i<N_PEOPLE; i++) {
        if (i == pos) { printf("|-"); }
        else          { printf("| "); }
    }
    printf("\n");
}

int main(void) {    // ハードコーディング
    bar_print(1);   // 左から1番目と2番目の間に横線
    bar_print(4);   // 左から4番目と5番目の間に横線
    bar_print(3);   // 左から3番目と4番目の間に横線
}
```

横の道は、**1 段につき 1 ヶ所、隣同士**の入れ替えに制限したので、この例では 1 – 4 – 3 と簡単に表現できます。（0 オリジンで数えています。）ただし、ハードコーディングになっているので編集には不向きです。次のように、配列とループ処理に置き換えましょう。

```
    int bar[] = { 1, 4, 3 };
    int n_bar = sizeof(bar)/sizeof(bar[0]);
    for (int j=0; j<n_bar; j++) {
        bar_print(bar[j]);  // 左からbar[j]番目とbar[j]+1番目の間に横線
    }
```

次は、人の行き先も計算で求めましょう。ソースコード 8.7 に作りかけのプログラムを載せておきます。

ソースコード 8.7 の出力目標



人の並び順を記憶する people[] を用意して、**初期化**する people_init() と、**表示**する people_print() を作っておきました。

隣同士を入れ替える people_swap() の処理を作り、うまく main() から呼び出して完成させてください。

ソースコード 8.7 あみだくじの行き先を表示する（未完成）

```c
#include <stdio.h>
#define N_PEOPLE 6
int people[N_PEOPLE]; // 人の並び順を記憶する

/* i 番目の人を i で初期化 */
void people_init() {
    for (int i=0; i<N_PEOPLE; i++) { people[i] = i; }
}

/* 人を表示する */
void people_print() {
    for (int i=0; i<N_PEOPLE; i++) { printf("%d ", people[i]); }
    printf("\n");
}

/* pos 番目と pos+1 番目の人を入れ替える */
void people_swap(int pos) {
    // ←ここを作る
}

/* pos 番目と pos+1 番目の人の間には - を描く */
void bar_print(int pos) {
    for (int i=0; i<N_PEOPLE; i++) {
        if (i == pos) { printf("|-"); }
        else          { printf("| "); }
    }
    printf("\n");
}

int main(void) {
    int bar[] = { 1, 4, 3 }; // ←ここでコースを変える
    int n_bar = sizeof(bar)/sizeof(bar[0]);

    people_init();
    people_print();
    for (int j=0; j<n_bar; j++) { // j段め
        bar_print(bar[j]);        // bar[j]番目に横線
        people_swap(0);       // ←ここを直す
        // people_print();   // ←デバッグに役立つ
    }
    people_print();
}
```

他にも自由に改造してみてください。例を挙げておきます。

- bar[] のデータの範囲外を検出する
- データを 1 オリジンにする
- 人を番号ではなく、アルファベット（A, B, C, ...）で表示する

8.5　配列をランダムに操作する

これまでの例では、配列にはデータが並んでいるだけで、添字がいくつであっても、あまり意味はありませんでした。ここでは、添字の値に意味のある操作をしてみます。

8.5.1　エラトステネスのふるい

素数とは、2 以上の整数のうち、1 と自分自身以外に約数を持たないものです。

??項では、ある 1 つの整数が素数かどうかを判定しましたが、ここでは **100 までのすべての素数を列挙**してみましょう。これには、割り算も掛け算も使わず、足し算と、素数かどうかの表だけで求める**エラトステネスのふるい** (sieve of Eratosthenes) というアルゴリズムが有用です。手順の詳細は 15 ページのコラムを参照してください。

このアルゴリズムをプログラムで実現してみます。「数」を並べた表は、配列で表現して、「数」を添字に対応付けます。つまり、2 が表に残っているのなら、is_prime[2] は TRUE です。4 を消すには is_prime[4] を FALSE にします。このように、**配列の添字の値に意味を持たせる**ところが、新しい使い方です。

ソースコード 8.8 では、is_prime[] を 7 行目で確保します。表に数が並んでいる状態にするために、10 行目で TRUE を代入します。11 行目では、小さな数から素数を探すのですが、0 と 1 は除外して、2 以上をループ処理で調べます。12 行目で i が素数だとわかれば、14–16 行目で倍数を消します。i の倍数を作り出すのに、i ずつ加算しています。

is_prime[] は論理型の変数ですから、12 行目の if の**条件式では TRUE と比較しない**のが作法であることを、もう一度確認しておいてください（☞**??**項）。

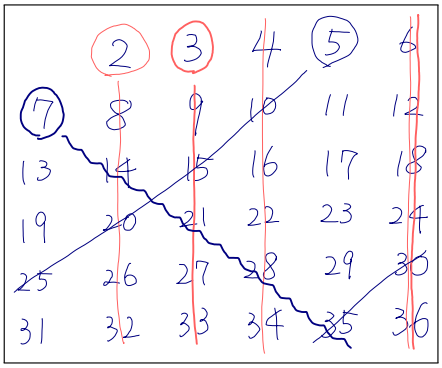
コラム：添字の型

配列の添字は、**整数型**に限られます。浮動小数点型（double や float など）では、コンパイルエラーになります。小数部分が 0 の数値であっても、型で判定されます。

```
int a[10];
double x = 1.0;
// a[x] == 5; // コンパイルエラー
```

このこともあって、C 言語では、**ループ変数は整数型**にしておくのが無難です。さらに浮動小数点型には、0.1 ずつの足し算をすると、誤差が積もってループ回数が 1 回ずれるかもしれないという問題もあります。（☞**??**項）

コラム：エラトステネスのふるいのアルゴリズム



100 までの素数を求めるには、まず表に（1 は素数ではないので）2 から 100 までの数を書き並べます。そして、小さい数から順に調べてゆき、消されてない数（＝素数）を見つけるたびに、「その素数自身は残して、倍数を消す」を繰り返します。

具体的な操作は、次のようになります。

2 は表に残っているので、素数です。4, 6, 8, ..., 100 を消します。

3 も表に残っているので、素数とわかります。6, 9, 12, ..., 99 を消します。

4 は既に消されているので、素数ではありません。何もせずに次に進みます。

この操作を最後まで続けると、素数だけが表に残ります。

ソースコード 8.8　エラトステネスのふるい

```c
#include <stdio.h>

#define FALSE 0
#define TRUE  1

#define N 100
int is_prime[N + 1];  // i が表にあるなら is_prime[i]=TRUE

int main(void) {
    for (int i=2; i<=N; i++) { is_prime[i] = TRUE; } // 初期化
    for (int i=2; i<=N; i++) {       // 小さい数から調べる
        if (is_prime[i]) {           // 素数を発見
            printf("%d ", i);
            for (int j=i+i; j<=N; j+=i) {
                is_prime[j] = FALSE; // 倍数を表から消す
            }
        }
    }
    printf("\n");
}
```

ソースコード 8.8 の実行結果

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

8.5.2　度数分布（ヒストグラム）

TODO: hist → freq, data → value

与えられたデータの値ごとの出現回数（度数）を数えてみます。この統計値は**度数分布** (frequency distribution)、グラフに描くと**ヒストグラム** (histogram) といいます。

何人分かの点数のデータがあるとしましょう。もし手作業で出現回数を数えるのなら、点数ごとに「正」の字を書くところです。つまり、先頭のデータから順に調べて、その点数に応じた「正」に 1 画ずつ書き加えていきます。「正」の字を書く場所は、**点数のとりうる種類の数**だけ必要ですが、このおかげで**同じデータを何度も読み返さずにすみます**[*6]。

この作業をプログラムで実現しましょう。ソースコード 8.9 では、6–8 行目で点数データを data[i] に格納し、点数の最大値を 3 行目のマクロ定数 POINT_MAX に定義しました。

出現回数を数える配列は 10 行目の hist[] です。「正」の字を書く場所に対応します。hist[0] は 0 点の人数、hist[1] は 1 点の人数というように割り当てることにして、すべて 0 で初期化しておきます（13 行目）。

ソースコード 8.9 の実行結果

```
0点が1人います
1点が1人います
2点が1人います
3点が1人います
4点が0人います
5点が3人います
6点が0人います
7点が2人います
8点が0人います
9点が2人います
10点が1人います
```

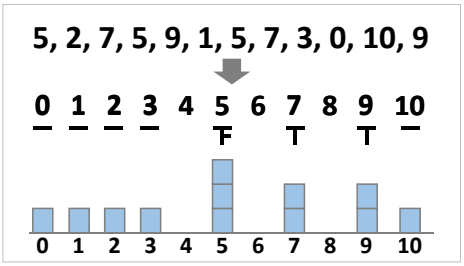
data[i] には、先頭から 5, 2, 7, ... が格納されているので、「正」の 1 画を書き加えることに対応する操作は hist[5]++; hist[2]++; hist[7]++; ... です。これをループで実現したのが 16–17 行目です。この 2 行はまとめて「hist[data[i]]++;」と短く書くこともできます。

プログラムには 2 種類の配列（データと出現回数）が現れるので、一見すると煩雑ですが、**配列ごとにいつも同じループ変数**を使っていることに着目すると、規則的に見えてきます。ここでは下の表のように、data[i] と hist[j] の組み合わせにしています。

目的	配列名	ループ変数	ループ変数の範囲
データ	data	i	$0 \leq i < num$
出現回数	hist	j	$0 \leq j <$ POINT_MAX+1

10 点満点なら 11 通りの点数があるので、POINT_MAX はプログラム上に単独で現れることはなく、常に +1 が付きます。そして 12 行目と 19 行目の j のループは、良いループ（☞**??**節）に合致します。

[*6] 逆に、「正」の字の場所（メモリ）を節約したければ、データを何度も読み直すことにして、0 点の人を見つけ終わったら、次はまた先頭から 1 点の人を探す、という方法もあります。（☞17 ページの頻出ミス）



ソースコード 8.9 ヒストグラム

```
1  #include <stdio.h>
2
3  #define POINT_MAX 10
4
5  int main(void) {
6      int data[] = {            // データ：添え字はiにする
7          5, 2, 7, 5, 9, 1, 5, 7, 3, 0, 10, 9,
8      };
9      int num = sizeof(data)/sizeof(data[0]);
10     int hist[POINT_MAX + 1]; // 出現回数：添え字はjにする
11
12     for (int j=0; j<POINT_MAX+1; j++) { // 出現回数の初期化
13         hist[j] = 0;
14     }
15     for (int i=0; i<num; i++) {         // 出現回数を数える
16         int d = data[i]; // data[i] の出現回数を1増やす
17         hist[d]++;       // 2行をまとめて hist[data[i]]++; でもよい
18     }
19     for (int j=0; j<POINT_MAX+1; j++) { // 結果表示
20         printf("%d点が%d人います\n", j, hist[j]);
21     }
22 }
```

頻出ミス

10-21 行目を、以下の 2 重ループの処理に置き換えると、配列は不要になりますが、計算時間が飛躍的に増えるので（POINT_MAX 倍）、実用的ではありません。

```
    for (int j=0; j<POINT_MAX+1; j++) {
        int count = 0;
        for (int i=0; i<num; i++) {
            if (data[i] == j) count++;
        }
        printf("%d点が%d人います\n", j, count);
    }
```

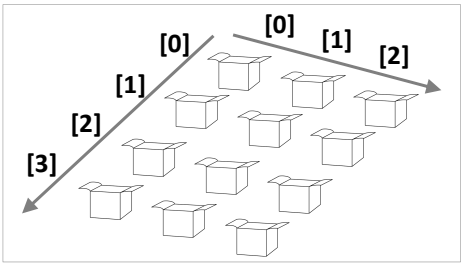
8.5.3　覆面算（発展的内容）

たとえば、$\begin{array}{r} \text{SEND} \\ +\ \text{MORE} \\ \hline \text{MONEY} \end{array}$ のようなアルファベットに数字を割り当てて $\begin{array}{r} 9567 \\ +\ 1085 \\ \hline 10652 \end{array}$ のように式を成り立たせる、という古典的なパズルがあります [?]。**覆面算** (alphametic) といいます。同じアルファベットには同じ数字を、異なるアルファベットには異なる数字を割り当てます。最上位桁（この例では S と M）は 0 を除外します。正しい割当が複数存在する問題もありますが、この例では上記が唯一の割当です。

プログラムで探索させると、ソースコード 8.10 のようになります。異なる数字を割り当てるために、数字が**使用済みかどうか**を論理型の配列 used[] で管理しています。数字 x が使用済みなら used[x] が真です。インデントは褒められたものではありませんが、8 重ループなので、いたしかたないでしょう。

ソースコード 8.10　覆面算

```
#include <stdio.h>
#define FALSE 0
#define TRUE  1

int val(int a, int b, int c, int d, int e) { // 桁ごとの数をintに変換
    return 10000*a + 1000*b + 100*c + 10*d + e;
}
int main(void) {
    int used[10];  // 数字xが使用済みなら used[x]は真
    for (int i=0; i<10; i++) { used[i] = FALSE; } // 初期化
    for (int s=1; s<10; s++) { if (!used[s]) { used[s] = TRUE;
    for (int e=0; e<10; e++) { if (!used[e]) { used[e] = TRUE;
    for (int n=0; n<10; n++) { if (!used[n]) { used[n] = TRUE;
    for (int d=0; d<10; d++) { if (!used[d]) { used[d] = TRUE;
    for (int m=1; m<10; m++) { if (!used[m]) { used[m] = TRUE;
    for (int o=0; o<10; o++) { if (!used[o]) { used[o] = TRUE;
    for (int r=0; r<10; r++) { if (!used[r]) { used[r] = TRUE;
    for (int y=0; y<10; y++) { if (!used[y]) { used[y] = TRUE;
        if (val(0,s,e,n,d) + val(0,m,o,r,e) == val(m,o,n,e,y)) {
            printf("  %d%d%d%d\n+ %d%d%d%d\n------\n %d%d%d%d%d\n",
                        s,e,n,d,    m,o,r,e,           m,o,n,e,y);
        }
    used[y] = FALSE; }}                              //   9567
    used[r] = FALSE; }}                              // + 1085
    used[o] = FALSE; }}                              // ------
    used[m] = FALSE; }}                              //  10652
    used[d] = FALSE; }}
    used[n] = FALSE; }}
    used[e] = FALSE; }}
    used[s] = FALSE; }}
}
```



8.6 2 次元配列

配列は直線上に並んだ多数の箱のようなものでした。多数の箱が、平面に 2 次元の広がりで敷き詰めることもできるように、配列も 2 次元の添字を使うことができます。変数定義でも、式中でも、変数名に続けて [ ] を 2 回繰り返します。

```
/* 1次元配列（定義と代入） */
int a[6];
a[0] = 1;  a[1] = 2;
a[2] = 3;  a[3] = 4;
a[4] = 5;  a[5] = 6;
```

```
/* 2次元配列（定義と代入） */
int a[3][2];
a[0][0] = 1;  a[0][1] = 2;
a[1][0] = 3;  a[1][1] = 4;
a[2][0] = 5;  a[2][1] = 6;
```

初期化子は、2 次元だとブロックが入れ子になります。要素数は**最初の次元だけ**が省略できます。

```
/* 1次元配列（初期化子） */
int a[6] = {
  1, 2,   ← 省略可
  3, 4,
  5, 6,
};
```

```
/* 2次元配列（初期化子） */
int a[3][2] = {
  { 1, 2 },   ← 省略可、必須
  { 3, 4 },
  { 5, 6 },
};
```

2 次元配列を、関数の引数として渡すこともできます。関数プロトタイプの仮引数では、要素数は**最初の次元だけ**が省略できます。呼び出す際の実引数には、1 次元のときと同じく、変数名のみを書いて、[ ] をつけません。

```
/* 1次元配列（引数渡し） */
int func(int a[6]);
                  ← 省略可
int a[6];
func(a); // 実引数は変数名のみ
```

```
/* 2次元配列（引数渡し） */
int func2(int a[3][2]);
                  ← 必須、省略可
int a[3][2];
func2(a); // 実引数は変数名のみ
```

TODO: メモリイメージ

コラム： 配列の配列

C 言語の 2 次元配列は、内部では添字を 1 次元相当に変換しています。M×N 要素の配列だと、以下のように、[i][j] は [i*N+j] のような動作をします。

```
/* 1次元配列 */
int a[M * N];
a[i * N + j] = 0;
```

←

```
/* 2次元配列 */
int a[M][N];
a[i][j] = 0;
```

[i*N+j] の計算式に N が必要なので、関数プロトタイプの仮引数で、2 次元目の要素数が省略できないことが理解できるでしょう。

メモリ上では、N 要素の int 配列が M 個並んでいるようなものですから、このタイプの 2 次元配列を「配列の配列」、また長さが揃ってることから「長方形配列」ということもあります。

なお、C 言語の 2 次元配列には「ポインタの配列」もあって、長さの揃わない配列を作ることもできます（☞**??**項）。

8.6.1　多次元配列

2 次元配列を作った方法を応用して、[ ] を繰り返すと 3 次元、4 次元のような**多次元配列** (multi-dimensional array) になります。初期化子や、関数との受け渡しの方法も、これまでの配列と同様です。省略できる要素数も、**最初の 1 次元だけ**です。

もっとも、多次元になってくると、変数の占める領域が大きくなるので、ローカル変数としては確保できなくなって[*7]、`static` をつけたり、グローバル変数にしてしまうこともよくあります[*8]。関数に渡すにしても、次元数を合わせる必要があるので、ほぼ専用の関数になって、引数で渡す意味が薄くなりがちです。本格的に利用するには、ポインタを駆使します（☞**??**項）。

[*7] ローカル変数を割り当てるメモリ（スタックメモリ）は、通常は数 MB 程度の領域しか確保されていません。さらに、領域をあふれた場合に、それを検出する機能は C 言語規格にはありません。つまり、コンパルエラーにも、実行時エラーにもならず、動作がおかしくなるだけの場合があります。

[*8] 割り当てられるメモリの種類が異なるため、数 MB のようなサイズの制限を受けませんが、それでも無尽蔵ではないので、最終的にはポインタ配列にして、メモリを動的に確保することになります。

8.6.2 2 次元配列の縦横合計

3 人の 5 科目の点数データから、人ごとの合計と、科目ごとの合計を求めましょう。

人と科目の 2 種類のループが入り乱れるので、煩雑になりますが、ソースコード 8.11 では、ループ変数を、**人には p、科目には s** と決めたので、少しは整理されているでしょう。添字の変数がいつでも同じになって、data なら [p][s]、人ごとの合計 sum_p なら [p]、科目ごとの合計 sum_s なら [s] です。

ソースコード 8.11 2 次元配列の縦横合計

```
#include <stdio.h>

#define PERSON 3
#define SUBJECT 5
int data[PERSON][SUBJECT] = {
        { 80,  70,  40,  60,  80 }, // sum_p[0]
        { 50,  90,  60,  40,  30 }, // sum_p[1]
        { 70,  40,  70,  60,  50 }, // sum_p[2]
};// sum_s[0]  [1]  [2]  [3]  [4]

void calc_sum(int data[PERSON][SUBJECT], int sum_p[], int sum_s[]) {
    for (int p=0; p<PERSON;  p++) { sum_p[p] = 0; } // 初期化(人)
    for (int s=0; s<SUBJECT; s++) { sum_s[s] = 0; } // 初期化(科目)
    for (int p=0; p<PERSON;  p++) {
        for (int s=0; s<SUBJECT; s++) {
            sum_p[p] += data[p][s]; // 加算(人)
            sum_s[s] += data[p][s]; // 加算(科目)
        }
    }
}

int main(void) {
    int sum_p[PERSON], sum_s[SUBJECT];
    calc_sum(data, sum_p, sum_s);        // 合計を計算する
    for (int p=0; p<PERSON; p++) {
        for (int s=0; s<SUBJECT; s++) {
            printf(" %2d ", data[p][s]); // データ表示
        }
        printf("| %d\n", sum_p[p]);      // 合計表示(人)
    }
    for (int s=0; s<SUBJECT; s++) {
        printf("%d ", sum_s[s]);         // 合計表示(科目)
    }
    printf("\n");
}
```

8.7　練習問題

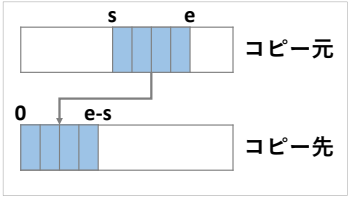
1. **[添字の範囲（☞3 ページの頻出ミス）]**
   右のプログラムで、配列の添字の不正なものをすべて指摘し、不正なものの確実な検出方法が存在するかを述べよ。

```
int main(void) {
    int a[10];
    a[-1] = -1;    a[ 9] =  9;
    a[ 0] =  0;    a[10] = 10;
    a[ 1] =  1;    a[11] = 11;
}
```

2. **[最大値（☞8.4.1 項）]**
   (i) 9 ページのソースコード 8.2 の 7 行目の暫定最大値との比較を <= に置き換えると、どのように動作が変化するのか説明せよ。**(ii) 最小値**を求める関数を作れ。
3. **[最大値の添字（☞8.4.1 項）]**
   (i) 9 ページのソースコード 8.2 を参考に、次の関数を作れ。
   - `int max_index(int n, int x[])` は x[] の最大値の要素の**添字**を返す。

   最大値をとる要素が複数あった場合には、**どれか 1 つ**の添字を返せばよいが、どれを選んだかを**コメント中で説明**せよ。**(ii)** 暫定最大値との比較に等号を含めるかどうかで、どのように動作が変化するのか、2.(i) との**違いがわかるように**説明せよ。
4. **[配列の一部をコピーする（☞8.4.3 項）]**
   11 ページのソースコード 8.5 を参考に、次の関数を作れ。
   - `void slice(int dst[], int src[], int start, int end)` は、
     src[start] から src[end-1] までの (end-start) 個の配列要素を、
     dst[0] から dst[end-start-1] までにコピーする。

   簡単のため 0 ≤ start < end を前提としてよい。src[end] はコピーしないことに注意し、調整項のない**良いループ**（☞**??**節）を実現せよ。



5. **[配列の要素をスワップ（☞8.4.4 項）]**
   13 ページのソースコード 8.7 を完成せよ。
6. **[度数分布（☞8.5.2 項）]**
   17 ページのソースコード 8.9 を参考に、100 点満点の成績データを作り、10 点刻みの度数分布を調べ、「0–9 点?人, 10–19 点?人, ...」のように表示せよ。（ヒント：100 点はどの区分に入るのかをよく検討せよ。）
7. **[2 次元配列（☞8.6.2 項）]**
   21 ページのソースコード 8.11 を改造して、3 人 5 科目の点数の総計（合計の合計）も求めて、出力される**表の右下**部分に追加して表示せよ。calc_sum() 関数の型を void から int に変更し、ループ処理で総計を求め、戻り値とせよ。main() からは calc_sum() を **1 度だけ呼び出せ**ば十分である。

索引

■ 記号・数字
1 オリジン 10

■ D
dummy **10**

■ N
n_ 4

■ S
sizeof 6
static 20

■ い
インデックス **2**
インデント 18

■ え
エラトステネスのふるい . . **14**

■ か
可変長配列 **5**
慣用句 6, 8, 9, 11

■ く
グローバル変数 20

■ こ
合計 3, 21

■ さ
最大値 3
参照渡し 8, 11

■ し
実行時エラー 8
初期化 2
初期化子 **2**, 6, 19

■ す
スタックメモリ 20
スワップ 8

■ せ
セキュリティ 5

■ そ
添字 **2**
素数 14

■ た
多次元配列 **20**

■ ち
長方形配列 . . . . → 配列の配列

■ て
定数式 **4**

■ と
度数分布 **16**

■ は
ハードコーディング 12
配列 **2**
配列の配列 20
番兵 **7**

■ ひ
ヒストグラム **16**

■ ふ
覆面算 **18**
プログラミング用語 10

■ ほ
ポインタ 8
ポインタの配列 20

■ ま
マクロ定数 4, 16
マジックナンバー **4**

■ よ
要素 **2**, 9–11
要素数 **2**, 4, 6, 7, 19, 20

■ ろ
ローカル変数 5, 20
論理型 18