第１章

数当てゲーム

本章で作成するプログラムは《数当てゲーム》です。まず最初に、プレーヤの入力した数値と、コンピュータの用意した値とを比較するだけの試作版を作り、少しずつ機能を追加していきます。

この章で学ぶおもなこと

- if 文の構造／効率／可読性
- do 文（後判定繰返し）
- while 文（前判定繰返し）
- for 文（前判定繰返し）
- break 文
- 等価演算子・関係演算子
- 論理演算子
- 増分演算子（前置／後置）
- sizeof 演算子
- 式の評価
- ド・モルガンの法則
- 乱数の生成と種の変更
- オブジェクト形式マクロ
- 配列
- 配列の走査
- 配列要素の初期化
- 配列の要素数の設定と取得
- ⊙rand 関数
- ⊙srand 関数
- ⊙RAND_MAX

1-1 数当ての判定

本章では《数当てゲーム》のプログラムを作成します。まず最初に作るのは、プレーヤがキーボードから打ち込んだ値と、コンピュータが用意した“当てさせる数”との比較結果を表示する試作版です。

if 文による分岐

List 1-1 に示すプログラムは、試作版の《数当てゲーム》です。

まずは実行してみましょう。0 ～ 9 の範囲の数値を当てるように促されますので、キーボードから数値を打ち込みます。そうすると、打ち込んだ数値と“当てさせる数”とを比べた結果が表示されます。

List 1-1 chap01/kazuate1.c

```
/* 数当てゲーム（その１）*/

#include <stdio.h>

int main(void)
{
    int no;          /* 読み込んだ値 */
    int ans = 7;     /* 当てさせる数 */

    printf("0～9の整数を当ててください。\n\n");

    printf("いくつかな：");
    scanf("%d", &no);

    if (no > ans)
        printf("\aもっと小さいよ。\n");
    else if (no < ans)
        printf("\aもっと大きいよ。\n");
    else
        printf("正解です。\n");

    return (0);
}
```

if文

実行例 1

```
0～9の整数を当ててください。

いくつかな：9
♪もっと小さいよ。
```

実行例 2

```
0～9の整数を当ててください。

いくつかな：5
♪もっと大きいよ。
```

実行例 3

```
0～9の整数を当ててください。

いくつかな：7
正解です。
```

本ゲームでの“当てさせる数”は 7 であって、それを表すのが変数 *ans* です。また、キーボードから読み込まれた値を格納するのが変数 *no* です。

網かけ部の **if** 文では、変数 *no* と *ans* の値の大小関係を判定します。そして、その判定結果に応じて、**Fig.1-1** に示すように『もっと小さいよ。』『もっと大きいよ。』『正解です。』のいずれかを表示します。

出力する文字列には、2 種類の**拡張表記**が含まれています。おなじみの **\n** は**改行**を表し、もう一つの **\a** は**警報**を表します。警報を出力すると、ほとんどの環境では《ビープ音》が鳴るため、本書の実行例では♪記号で表します。

▶ 拡張表記は第 2 章で詳しく学習します。多くのパソコンで使われる JIS コード（p.78）では、逆斜線 \ の代わりに円記号 ¥ を使います。必要に応じて読みかえてください。

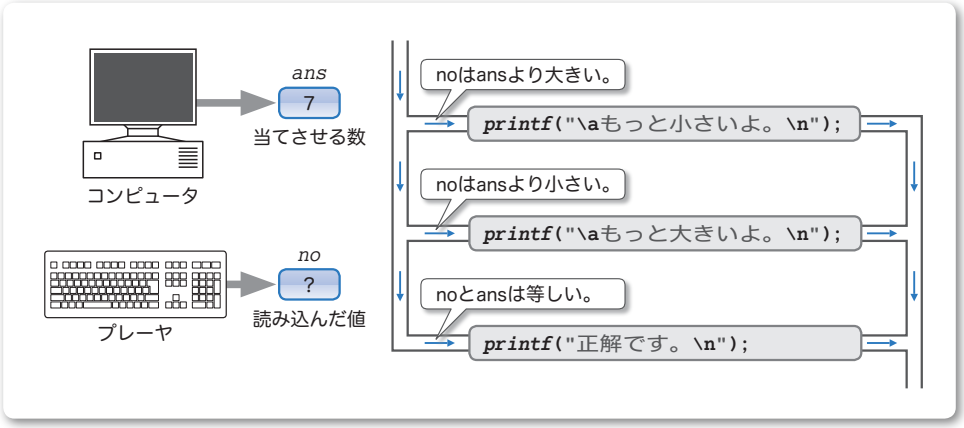


Fig.1-1 if文によるプログラムの流れの分岐

入れ子になったif文

二つの変数値 *no* と *ans* の値を比較する **if** 文を理解していきましょう。

if 文は、**制御式**と呼ばれる**式**を**評価**（**Column 1-1**：p.6）した結果によってプログラムの流れを分岐する文です。その構文は、右に示す二つの形式のいずれかです。

▶ （ ）の中に置かれた**式**が制御式です。

if文の構文

- **if**（式）文
- **if**（式）文 **else** 文

ところが、本プログラムの **if** 文は、以下の形をしています。

if（式）文 **else if**（式）文 **else** 文

もっとも、プログラムの流れを三つに分岐させるために、このような構文が特別に用意されているのでは**ありません**。その名前が示すとおり、**if** 文は**一種の文**ですから、**else** が制御する文は **if** 文でもよいわけです。**Fig.1-2** に示すように、**if** 文の中に **if** 文が入る“**入れ子**”の構造となっているのです。

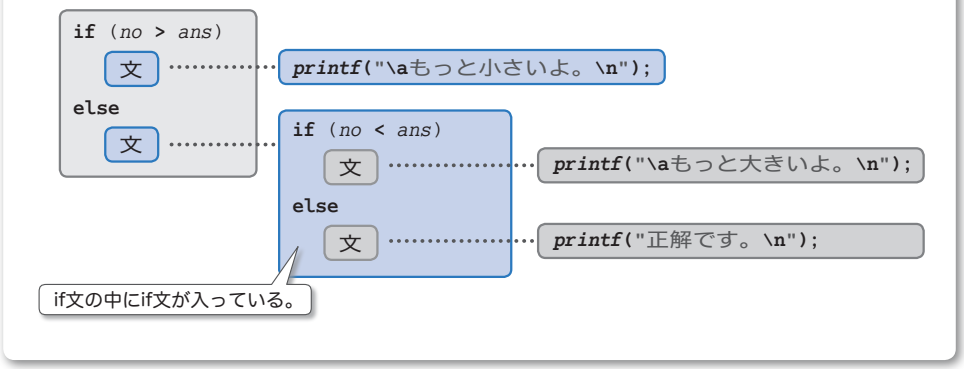


Fig.1-2 入れ子になったif文

多分岐の実現法

本プログラムの **if** 文（右の**1**）と同じ動作をするように作ったのが、**2**と**3**の **if** 文です。

これら三つを比較・検討して、さらに奥深く **if** 文を理解しましょう。

プログラム2

最後の **else** の後に網かけ部が追加されています。ここにプログラムの流れが到達するのは、二つの判定（*no* **>** *ans*）と（*no* **<** *ans*）の両方が成立しない場合、すなわち、*no* と *ans* が等しい場合のみです。

網かけ部で行われる判定は、**必ず成立する条件**ということになります。

プログラム3

if 文が三つ並んでいます。変数 *no* と *ans* の大小関係とは無関係に、三つの条件判定が**すべて**行われます。

1 List 1-1のif文

```
if (no > ans)
    printf("\aもっと小さいよ。\n");
else if (no < ans)
    printf("\aもっと大きいよ。\n");
else
    printf("正解です。\n");
```

2 最後のelseにif (no == ans)を追加

```
if (no > ans)
    printf("\aもっと小さいよ。\n");
else if (no < ans)
    printf("\aもっと大きいよ。\n");
else if (no == ans)
    printf("正解です。\n");
```

3 独立した三つのif文の並び

```
if (no > ans)
    printf("\aもっと小さいよ。\n");
if (no < ans)
    printf("\aもっと大きいよ。\n");
if (no == ans)
    printf("正解です。\n");
```

三つのプログラムにおいて、どの判定が行われる（どの制御式が評価される）のかを、変数 *no* と *ans* の大小関係別にまとめたのが、**Table 1-1** です。

Table 1-1　三つのプログラムで行われる判定

大小関係	*no* ＞*ans* のとき	*no* ＜*ans* のとき	*no* ＝*ans* のとき
1	①	① ②	① ②
2	①	① ②	① ② ③
3	① ② ③	① ② ③	① ② ③

① (*no* **>** *ans*) の判定
② (*no* **<** *ans*) の判定
③ (*no* **==** *ans*) の判定

たとえば、*no* が *ans* より大きい場合を確認してみましょう。**1**と**2**のプログラムでは、①の (*no* **>** *ans*) の判定だけが行われます。

▶ *no* が *ans* より大きければ、***printf***(**"\a**もっと小さいよ。**\n");** の実行が完了した段階で、**if** 文全体の実行を終了するからです。

一方、独立した **if** 文が三つ並んだ構造の**3**では、①の (*no* **>** *ans*) と②の (*no* **<** *ans*) と③の (*no* **==** *ans*) の判定がすべて行われます。最も効率の悪い実現法です。

＊

どの条件においても判定回数が少ないのが**1**の **if** 文です。

1の **if** 文が優れているのは、判定回数が少ないことだけではありません。そのことを理解するために、**Fig.1-3** を考えていきましょう。

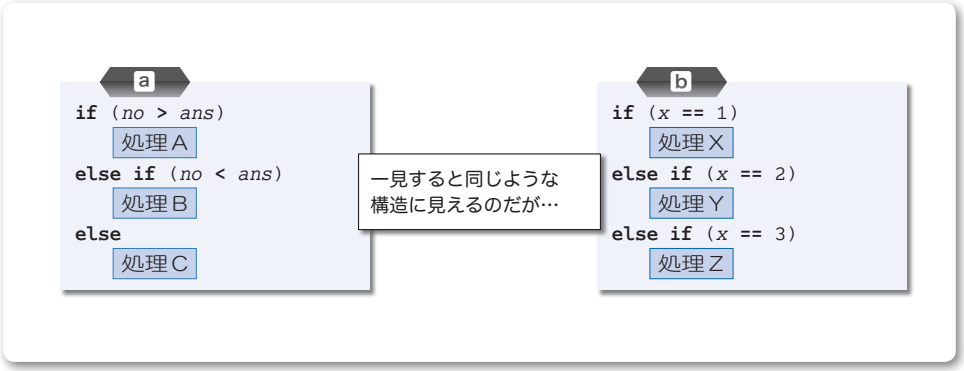


Fig.1-3 似ているようでまったく異なるif文による分岐

図aのif文

1と同じ構造のif文であり、プログラムの流れが三つに分岐します。実行されるのは、〔処理A〕〔処理B〕〔処理C〕のいずれか一つです。

▶ いずれの処理も実行されない、あるいは、二つ以上の処理が実行される、ということはありません。

図bのif文

変数xの値に応じて分岐するif文です。

〔処理X〕〔処理Y〕〔処理Z〕のどれか一つが実行されるように見えます。しかし、変数xの値が1，2，3以外であれば、**どの処理も行われません**。

Fig.1-4 に示すように、プログラムの流れは実質的に四つに分岐するからです。

図aのif文とは構造が**まったく異なります**。そのため、最後の判定if（x == 3）を省略することは**できません**。

▶ もし省略すると、xの値が3でなく4や5であっても〔処理Z〕が実行されてしまうからです。

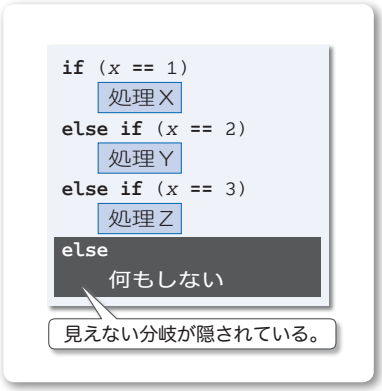


Fig.1-4 図bの解釈

＊

図aの構造をもつ1のif文は、最後のelseの後にifがありません。そのため、パッと見ただけで、**それ以上の分岐をもたない**ことが分かります。

プログラムの読みやすさという点でも、最後のelseの後に“無駄”な判定が置かれている2よりも、1のほうが優れています。

▶ プログラムの読み手に対して『noがansと等しい場合は、こんなことをやるんだよ。』と、どうしても強調したいのであれば、2のように実現しても構わないでしょう。

通常は、コンパイラの最適化技術によって、この判定は内部的に削除されるため、効率のことを気にする必要性は意外と小さいのです。

Column 1-1　式の評価

▪ 式とは

プログラミングの世界では、**式**（*expression*）という用語が頻繁に利用されます。式は、以下のものの総称です。

- ▪ 変数
- ▪ 定数
- ▪ 変数や定数を演算子で結合したもの

さて、ここで以下の式を考えます。

```
n + 52
```

変数 *n*、整数定数 52、それらを + 演算子で結んだ `n + 52` のいずれもが式です。

次に、以下の式を考えましょう。

```
x = n + 52
```

ここでは、`x`，`n`，`52`，`n + 52`，`x = n + 52` のいずれもが式です。

一般に、○○演算子によって結合された式のことを、○○式と呼びます。たとえば、代入演算子によって *x* と `n + 52` が結び付けられた式 `x = n + 52` は、**代入式**（*assignment expression*）です。

▪ 式の評価

原則として、すべての式には値があります（特別な型である **void** 型の式だけは、例外的に値がありません）。その値は、プログラム実行時に調べられます。式の値を調べることを**評価**（*evaluation*）といいます。

評価のイメージの具体例を示したのが **Fig. 1C-1** です（この図は、**int** 型の変数 *n* の値が 135 であるとしています）。

変数 *n* の値が 135 ですから、`n`，`52`，`n + 52` の各式を評価した値は 135，52，187 となります。もちろん、三つの値の型はいずれも **int** 型です。

このように、本書では、ディジタル温度計のような図で評価値を示すことにします。左側の小さな文字が《型》で、右側の大きな文字が《値》です。

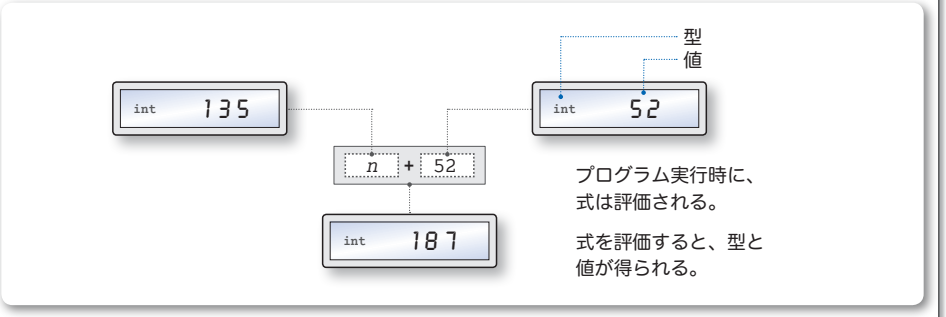


Fig.1C-1　式の評価（int型 + int型）

List 1-1 の **if** 文の最初の制御式は `no > ans` でした。もし変数 *no* に読み込まれた値が 5 であれば、この式の評価は、右ページの **Fig.1C-2** のように行われます。

関係演算子は、二つのオペランドの大小関係を判定します。この場合、判定条件が成立しませんので、式 `no > ans` を評価して得られるのは、偽を表す“**int** 型の 0”です。なお、*no* の値が 7 より大きければ、真を表す“**int** 型の 1”です（p.9）。

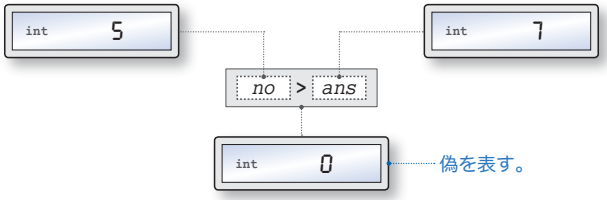


Fig.1C-2　式の評価（int型 > int型）

この例は、演算の対象となる左右のオペランドの型が **int** 型で、評価によって得られる型も **int** 型でした。関係演算子は、オペランドの型が **int** 型でなくても、**int** 型を生成します。その例を示したのが **Fig.1C-3** です。**double** 型の 7.5 と 8.4 を比較する式 `7.5 < 8.4` を評価して得られるのは、**int** 型の 1 です。

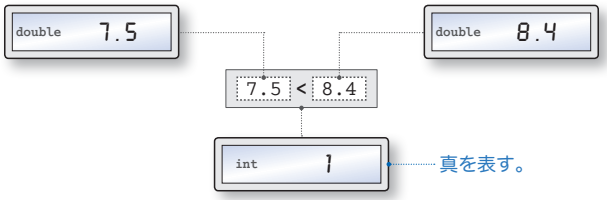


Fig.1C-3　式の評価（double型 < double型）

演算の対象となるオペランドの型は同じであるとは限りません。**int** 型の 15 あるいは **double** 型の `15.0` を、**int** 型の 2 または **double** 型の `2.0` で割る演算の例を示したのが **Fig.1C-4** です（この図では、定数である 15 と `15.0` の評価は省略しています）。少なくとも一方のオペランドが **double** 型であれば、演算結果は **double** 型となります（『入門編』：p.28）。

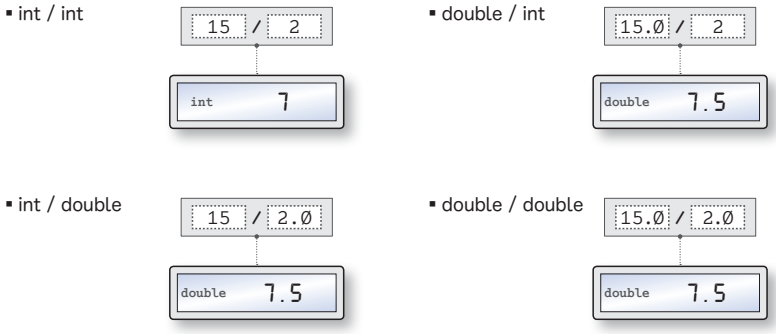


Fig.1C-4　式の評価（int型とdouble型の除算）

1-2 当たるまでの繰返し

プレーヤの数値入力が 1 回だけに限られている《数当てゲーム》は、楽しいものではありません。正解するまで繰り返し入力できるように改良しましょう。

do 文による繰返し

プレーヤの数値入力が 1 回に限られているのでは、正解するまで何度もプログラムを起動し直さなければなりません。楽しくないばかりか、手間がかかって面倒です。

正解するまで繰り返し入力できるように改良しましょう。**List 1-2** に示すのが、そのプログラムです。

List 1-2 chap01/kazuate2.c

```
/* 数当てゲーム（その２：当たるまで繰り返す：do文）*/

#include <stdio.h>

int main(void)
{
    int no;          /* 読み込んだ値 */
    int ans = 7;     /* 当てさせる数 */

    printf("0〜9の整数を当ててください。\n\n");

    do {
        printf("いくつかな：");
        scanf("%d", &no);

        if (no > ans)
            printf("\aもっと小さいよ。\n");
        else if (no < ans)
            printf("\aもっと大きいよ。\n");
    } while (no != ans);                /* 当たるまで繰り返す */

    printf("正解です。\n");

    return (0);
}
```

do文

実行例

```
0〜9の整数を当ててください。

いくつかな：6↵
♪もっと大きいよ。
いくつかな：8↵
♪もっと小さいよ。
いくつかな：7↵
正解です。
```

List 1-1 の **if** 文の後半を削った上で、網かけ部の **do** 文が追加されています。

do 文は、**後判定繰返し**（p.11）によって処理を繰り返す文であり、その構文は右のとおりです。

do文の構文

```
do 文 while（式）;
```

▶ 既に学習した **if** 文や、後で学習する **while** 文や **for** 文などの構文とは異なり、末尾にセミコロン ; が付きます。

do と **while** とで囲まれた文は**ループ本体**と呼ばれます。（ ）中に置かれた式である**制御式**を評価した値が 0 でない限り、ループ本体は何度も繰り返し実行されます。繰返しが終了するのは、制御式を評価した値が 0 になったときです。

本プログラムの do 文による繰返しの様子を、**Fig.1-5** を見ながら理解しましょう。

do 文の制御式は、*no* != *ans* となっています。

演算子 != が行う判定は、左右のオペランドの値が**等しくない**かどうかの条件です。その条件が成立すれば int 型の 1 を、成立しなければ int 型の 0 を生成します。

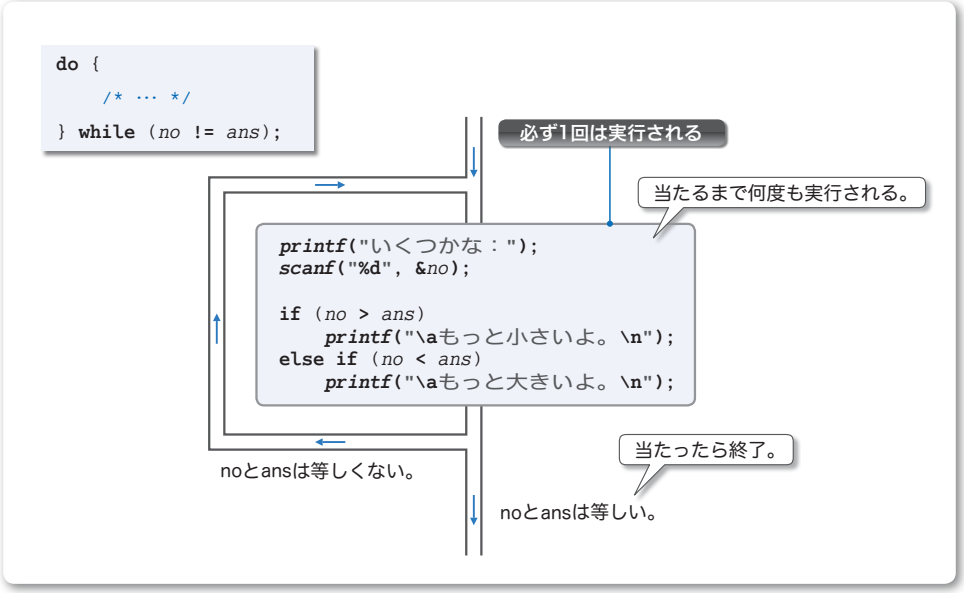


Fig.1-5 do文によるプログラムの流れの繰返し

読み込んだ値 *no* が、当てさせる数 *ans* と等しくなければ、制御式 *no* != *ans* を評価して得られる値が 1 となります。そのため、do 文による繰返しが行われ、{ } で囲まれたブロックであるループ本体が再び実行されることになります。

当てさせる数 *ans* と同じ値が *no* に読み込まれると、制御式を評価した値が 0 となるため、繰返しは終了です。画面に『正解です。』と表示して、プログラムは終了します。

等価演算子と関係演算子

等価演算子（*equality operator*）と**関係演算子**（*relational operator*）は、判定条件が成立すれば int 型の 1 を、成立しなければ int 型の 0 を生成します。

▶ int 型の 1 は“真”を表して、int 型の 0 は“偽”を表します（p.20）。

等価演算子　==　!=

二つのオペランドが等しいか／等しくないかを判断します。

関係演算子　<　>　<=　>=

二つのオペランドの大小関係を判断します。

while 文による繰返し

C言語の＜繰返し文＞には、do 文の他に while 文と for 文があります。

do 文と対照的な**前判定繰返し**を行う while 文を利用して前のプログラムを書きかえてみましょう。そのプログラムを **List 1-3** に示します。

List 1-3 chap01/kazuate3.c

```
/* 数当てゲーム（その３：当たるまで繰り返す：while文）*/

#include <stdio.h>

int main(void)
{
    int no;          /* 読み込んだ値 */
    int ans = 7;     /* 当てさせる数 */

    printf("0〜9の整数を当ててください。\n\n");

    while (1) {
        printf("いくつかな：");
        scanf("%d", &no);

        if (no > ans)
            printf("\aもっと小さいよ。\n");
        else if (no < ans)
            printf("\aもっと大きいよ。\n");
        else
            break;
    }

    printf("正解です。\n");

    return (0);
}
```

while文

break文

実行例

```
0〜9の整数を当ててください。

いくつかな：6
♪もっと大きいよ。
いくつかな：8
♪もっと小さいよ。
いくつかな：7
正解です。
```

while 文の構文は、右のとおりです。

while文の構文

```
while（式）文
```

制御式である**式**を評価した値が 0 でない限り、ループ本体である**文**が何度も実行されます。ただし、評価した値が 0 になったら繰返しは終了です。

本プログラムの while 文の制御式は 1 ですから、**繰返しは永遠に行われる**ことになります。このような繰返しは、一般に＜無限ループ＞と呼ばれます。

break 文

ただ繰り返すばかりでは、いつまでもプログラムが終わりません。本プログラムでは、繰返し文を強制的に抜け出すために、break 文を利用しています。

no と *ans* が等しいときに break 文が実行されますので、while 文による繰返しが強制的に中断されることになります。

▶ break 文を用いたプログラムは、読みにくく理解しにくくなる傾向があります。『ある特別な条件が成立したときに、何らかの事情によって繰返し文を強制的に終了したい。』といった状況でのみ利用すべきです。ここで取り上げている《数当てゲーム》の繰返しは単純な構造ですから、break 文など使わず **List 1-2** のように実現すべきです。

while 文と do 文

プログラム中の **while** が、**do** 文の一部であるのか、**while** 文の一部であるのかは、見分けにくいものです。そのことを、右に示すプログラムで考えましょう。

まず変数 *x* に 0 が代入されます。その後、**do** 文によって *x* が 5 になるまで値がインクリメントされます。

続く **while** 文では、*x* の値をデクリメントしながら、その値を表示します。

do文のwhile

```
x = 0;
do
    x++;
while (x <= 5);
while (x >= 0)
    printf("%d ", --x);
```

while文のwhile

▶ 増分（インクリメント）演算子 ++ および減分（デクリメント）演算子 -- については、1–4 節で学習します。

右に示すように、**do** 文のループ本体を { } で囲んだブロックにしてみましょう。

そうすると、**行の先頭を見ただけで見分けがつく**ようになります。

```
x = 0;
do {
    x++;
} while (x <= 5);
while (x >= 0)
    printf("%d ", --x);
```

while	…	行の先頭が **while** ならば **while** 文の先頭部分。
} **while**	…	行の先頭が } ならば **do** 文の末尾部分。

本来は、**do** 文も **while** 文も **for** 文も、ループ本体が単一の文であれば、わざわざブロックを導入する必要はありません。

とはいえ、**do** 文に限っては、ループ本体がたとえ単一の文であっても、あえてブロックにしたほうがプログラムが読みやすくなります。

前判定繰返しと後判定繰返し

繰返しは、処理を続けるかどうかの判断のタイミングによって、２種類に分類されます。

前判定繰返し（while 文・for 文）

処理を行う前に、処理を続けるかどうかの判定を行います。ループ本体が 1 回も実行されないことがあります。

後判定繰返し（do 文）

処理を行った後に、処理を続けるかどうかの判定を行います。ループ本体は、少なくとも 1 回は実行されます。

1-3 当てさせる数をランダムに

ここまでの《数当てゲーム》は“当てさせる数”がプログラム中に埋め込まれていて、あらかじめ答えが分かっていました。この値が自動的に変わるようにして、ゲームとしての楽しさをアップさせましょう。

rand 関数：乱数の生成

ゲームのたびに“当てさせる数”を変えるには、いわゆる**乱数**（らんすう）が必要です。乱数を生成するのが、次に示す ***rand*** 関数です。

	rand
ヘッダ	`#include <stdlib.h>`
形　式	`int rand(void);`
機　能	0 以上 **RAND_MAX** 以下の範囲の擬似乱数整数列を計算する。 なお、他のライブラリ関数は、本関数を呼び出さないかのように動作する。
返却値	生成した擬似乱数整数を返す。

この関数が生成する乱数は **int** 型の整数です。その最小値が 0 であることは、全処理系で共通です。しかし、最大値は処理系に依存するため、<stdlib.h> ヘッダで **RAND_MAX** という名前の**オブジェクト形式マクロ**（*object-like macro*）として定義されることになっています。以下に示すのが定義の一例です。

RAND_MAX

```
#define RAND_MAX  32767      /* 定義の一例：値は処理系によって異なる */
```

なお、**RAND_MAX** の値は最低でも 32,767 であると規定されています。そのため、***rand*** 関数は **Fig.1-6** のように動作します。

それでは、実際に乱数を生成・表示してみましょう。**List 1-4** に示すプログラムを実行してみてください。

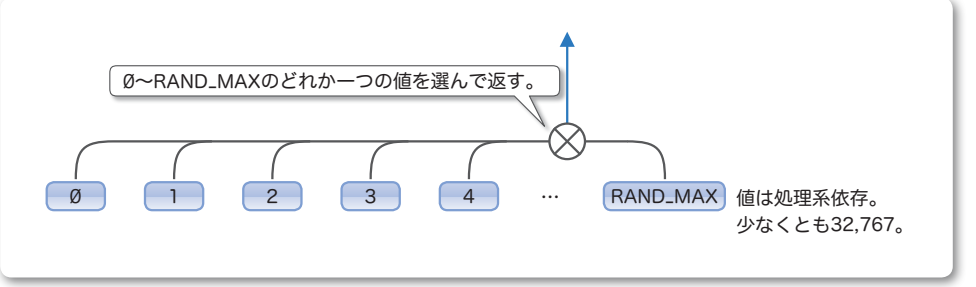


Fig.1-6　rand関数による乱数の生成

List 1-4 chap01/random1.c

```
/* 乱数を生成（その１）*/

#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int retry;          /* もう一度？ */

    printf("この処理系では0～%dの乱数が生成できます。\n", RAND_MAX);

    do {
        printf("\n乱数%dを生成しました。\n", rand());

        printf("もう一度？ … (0)いいえ (1)はい：");
        scanf("%d", &retry);

    } while (retry == 1);

    return (0);
}
```

rand関数が生成する乱数の最大値

乱数を生成して返却

まず最初に生成できる乱数の“範囲”が表示され、それから実際に生成された乱数が表示されます。

なお、もう一度行うかどうかの問いかけに対して〔はい〕を選択すれば、乱数を繰り返し生成・表示できるようになっています。

プログラムを何度か実行してみてください。そうすると、**Fig.1-7** に示すように、**いつも同じ乱数の系列が生成されます**。これはおかしいですね。はたして ***rand*** 関数が生成する値は、**本当にランダムなのでしょうか？**

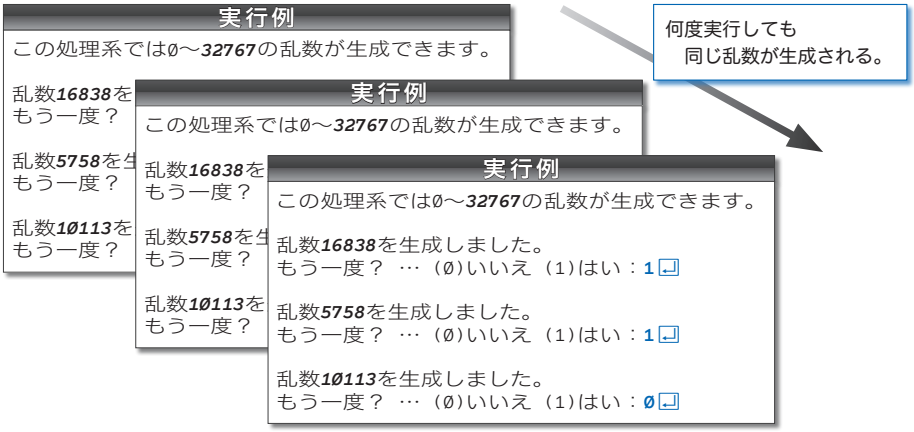


Fig.1-7 List 1–4の実行例

srand 関数：乱数生成のための種の設定

rand 関数は、“種（たね）”と呼ばれる基準値に演算を施して乱数を作ります。プログラム実行のたびに同じ乱数の系列が生成されるのは、**_rand_ 関数中に定数値 1 が種として埋め込まれているから**です。異なる系列の乱数を生成するには、種の値を変えなければなりません。

それを行うのが、以下に示す *srand* 関数です。

	srand
ヘッダ	`#include <stdlib.h>`
形　式	`void srand(unsigned seed);`
機　能	後続する *rand* 関数の呼出しで返す新しい擬似乱数列の種を *seed* に設定する。本関数を同じ種の値で呼び出すと、同じ擬似乱数列が生成される。本関数より前に *rand* 関数を呼び出した場合、本関数が最初に種の値を 1 として呼び出されたときと同じ列が生成される。なお、他のライブラリ関数は、本関数を呼び出さないかのように動作する。
返却値	なし。

たとえば、*srand*(50) と呼び出したとします。そうすると、その後に呼び出される *rand* 関数は、設定された新しい種の値 50 を利用して乱数を生成する仕組みとなっています。

Fig.1-8 に示すのは、ある処理系で生成される乱数系列の具体例です。

種が 1 のときは、最初の *rand* 関数の呼出しでは 16,838 が生成されて、次の呼出しでは 5,758、その次は 10,113、 … と乱数が生成されます。

また、種が 50 であれば、22,715、22,430、16,275、… が順に生成されます。

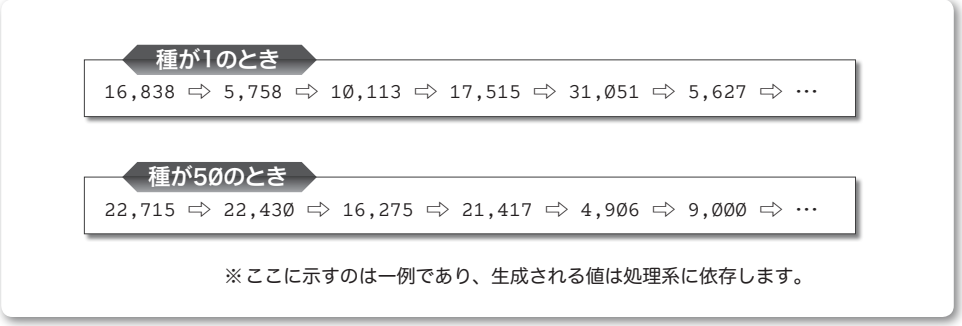


Fig.1-8　種とrand関数が生成する乱数系列の一例

この図が示すように、いったん種の値が決まると、それ以降に生成される乱数の系列も決まってしまいます。したがって、プログラム実行のたびに異なる系列の乱数を生成するには、**種の値そのものを、定数ではなくランダムにしなければなりません**。

しかし、乱数生成の準備のために乱数が必要というのも、おかしな話です。

▶ *rand* 関数が生成するのは、擬似乱数と呼ばれる乱数です。擬似乱数は、乱数のように見えますが、ある一定の規則に基づいて生成されます。擬似乱数と呼ばれるのは、次に生成される数値の予測がつくからです。本当の乱数は、次に生成される数値の予測がつきません。

一般的に使われるのが、**“プログラム実行時の時刻を種にする”**という手法です。その手法を利用したプログラムを **List 1-5** に示します。

List 1-5 chap01/random2.c

```
/* 乱数を生成（その２：現在の時刻に基づいて乱数の種を初期化）*/

#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int retry;              /* もう一度？ */

    srand(time(NULL));      /* 乱数の種を初期化 */

    printf("この処理系では0～%dの乱数が生成できます。\n", RAND_MAX);

    do {
        printf("\n乱数%dを生成しました。\n", rand());

        printf("もう一度？ … (0)いいえ (1)はい：");
        scanf("%d", &retry);

    } while (retry == 1);

    return (0);
}
```

プログラムを実行してみてください。**Fig.1-9** に示すように、起動するたびに異なる乱数の系列が生成されます。

▶ 現在の時刻を取得する *time* 関数の詳細は、第６章で詳しく学習します。それまでは、プログラムの網かけ部を“決まり文句”として覚えておくとよいでしょう。

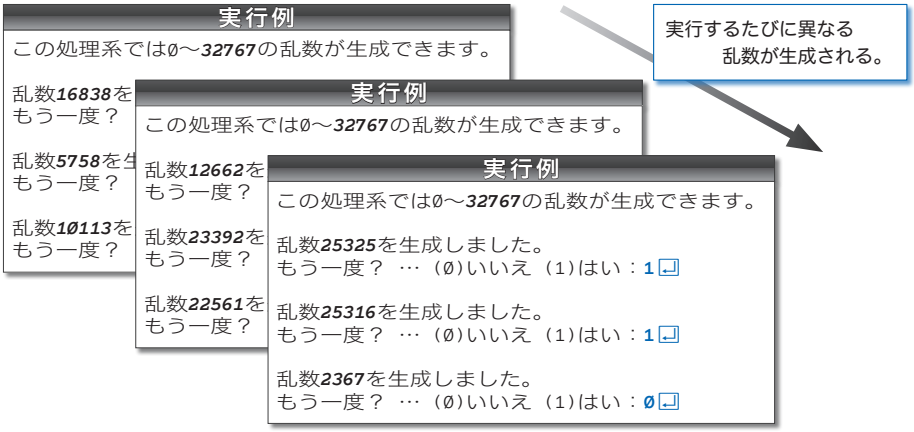


Fig.1-9 List 1–5の実行例

当てさせる数をランダムにする

rand 関数が生成する値の範囲は 0 ～ **RAND_MAX** です。とはいえ、コンピュータに都合のよいように決められた範囲の乱数が必要となることは、まずないでしょう。

通常は、ある**特定の範囲の乱数**が必要です。もし“0 以上 10 以下”の乱数が必要であれば、以下のように求められます。

```
rand() % 11                          /* 0以上10以下の乱数を生成 */
```

非負の整数値を 11 で割った剰余が 0, 1, …, 10 となることを利用します。

▶ 誤って 10 で割らないように気をつけましょう。10 で割った剰余は 0，1，…，9 となるため、10 が生成されなくなります。

＊

乱数を生成する方法が理解できました。数当てゲームの“当てさせる数”を 0 以上 999 以下の乱数にしましょう。そのプログラムを **List 1-6** に示します。

List 1-6 chap01/kazuate4.c

```
/* 数当てゲーム（その４：当てさせる数は0～999の乱数）*/

#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int no;      /* 読み込んだ値 */
    int ans;     /* 当てさせる数 */

    srand(time(NULL));       /* 乱数の種を初期化 */
    ans = rand() % 1000;     /* 0～999の乱数を生成 */

    printf("0～999の整数を当ててください。\n\n");

    do {
        printf("いくつかな：");
        scanf("%d", &no);

        if (no > ans)
            printf("\aもっと小さいよ。\n");
        else if (no < ans)
            printf("\aもっと大きいよ。\n");
    } while (no != ans);                    /* 当たるまで繰り返す */

    printf("正解です。\n");

    return (0);
}
```

実行例

```
0～999の整数を当ててください。

いくつかな：500
♪もっと大きいよ。
いくつかな：750
♪もっと小さいよ。
いくつかな：625
正解です。
```

網かけ部では、生成した乱数を 1000 で割った剰余を変数 *ans* に代入しています。

当てさせる数がランダムになるだけで、数当てゲームは飛躍的に面白くなります。何度も実行して楽しみましょう。

ところで、平均的に**最短で当てる方法**は分かりますか。最初に 500 を入力し、それより大きいか／小さいかによって 750 あるいは 250 を入力する、といった具合で、**半分ずつに絞り込んでいきます。**

*

当てさせる数の範囲の変更は容易です。具体例を二つ示します。

当てさせる数を 1 ～ 999 にする

プログラム網かけ部を、次のように書きかえます。

```
ans = 1 + rand() % 999;          /* 1～999の乱数を生成 */
```

当てさせる数を 3 桁の整数（100 ～ 999）にする

プログラム網かけ部を、次のように書きかえます。

```
ans = 100 + rand() % 900;        /* 100～999の乱数を生成 */
```

まとめ

乱数生成の準備

乱数を生成する前に、現在の時刻に基づいて“種”の値を設定する。

```
#include <time.h>
#include <stdlib.h>
/* … */
srand(time(NULL));                    /* 乱数の種を初期化 */
```

srand 関数の呼出しは、*rand* 関数を最初に呼び出す時点よりも前に行う（1 回だけでよく何回も呼び出す必要はない）。

乱数の生成

rand 関数を呼び出すと、0 以上 RAND_MAX 以下の乱数が得られる。なお、特定の範囲の乱数を得るには、以下のようにする。

```
    rand() % (a + 1)                  /* 0 以上   a   以下の乱数 */
b + rand() % (a + 1)                  /* b 以上 b + a 以下の乱数 */
```

入力回数に制限を設ける

何度も入力していれば、いつかは当たります。入力できる回数を最大10回に制限して、**プレーヤに緊張感を与える**ように変更したプログラムが**List 1-7**です。

List 1-7 chap01/kazuate5.c

```
/* 数当てゲーム（その５：入力回数に制限を設ける）*/

#include <time.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int no;                 /* 読み込んだ値 */
    int ans;                /* 当てさせる数 */
    int max_stage = 10;     /* 最大入力回数 */
    int remain = max_stage; /* 残り何回入力できるか？ */

    srand(time(NULL));      /* 乱数の種を初期化 */
    ans = rand() % 1000;    /* 0〜999の乱数を生成 */

    printf("0〜999の整数を当ててください。\n\n");

    do {
        printf("残り%d回。いくつかな：", remain);
        scanf("%d", &no);
        remain--;           /* 残り回数をデクリメント */

        if (no > ans)
            printf("\aもっと小さいよ。\n");
        else if (no < ans)
            printf("\aもっと大きいよ。\n");
    } while (no != ans && remain > 0);

    if (no != ans)
        printf("\a残念。正解は%dでした。\n", ans);
    else {
        printf("正解です。\n");
        printf("%d回で当たりましたね。\n", max_stage - remain);
    }

    return (0);
}
```

プレーヤが入力できる最大回数である10を表すのが、変数*max_stage*です。

もう一つの新しい変数*remain*は、**残り何回入力できるか**を表します。もちろん、その初期値は*max_stage*すなわち10です。**Fig.1-10**に示すように、プレーヤが値を入力するたびに、*remain*の値を10，9，8，… とデクリメントします（値を1だけ減らします）。

この値が0になるとゲームは終了です。そのため、**do**文の判定には、式*no* **!=** *ans*だけでなく、網かけ部の*remain* **>** 0が追加されています。

二つの式を結ぶ論理AND演算子**&&**は、両方のオペランドがともに非0である場合にのみ**int**型の1を生成し、そうでなければ0を生成します。

そのため、当たった場合（図a）だけでなく、10回入力しても当たらずremainが0になった場合（図b）も、ちゃんと繰返しは終了します。

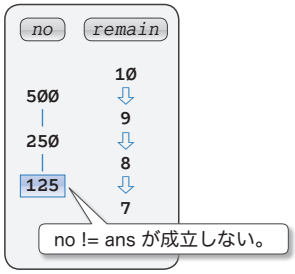
▶ 繰返しの終了条件と && 演算子については、**Column 1-2**（次ページ）で学習します。

なお、何回目の入力で当たったのかは、*max_stage* から *remain* を引くことによって得られます。たとえば、図aに示す例では、ゲーム終了時の *remain* の値は 7 です。そのため、*max_stage* - *remain* すなわち 10 - 7 によって 3 が得られます。

a 125を当てる（3回目で正解）

```
実行例
0～999の整数を当ててください。

残り10回。いくつかな：500
♪もっと小さいよ。
残り9回。いくつかな：250
♪もっと小さいよ。
残り8回。いくつかな：125
正解です。
3回で当たりましたね。
```



b 139を当てる（10回やっても不正解）

```
実行例
0～999の整数を当ててください。

残り10回。いくつかな：500
♪もっと小さいよ。
残り9回。いくつかな：250
♪もっと小さいよ。
残り8回。いくつかな：125
♪もっと大きいよ。
残り7回。いくつかな：187
♪もっと小さいよ。
残り6回。いくつかな：156
♪もっと小さいよ。
残り5回。いくつかな：140
♪もっと小さいよ。
残り4回。いくつかな：133
♪もっと大きいよ。
残り3回。いくつかな：136
♪もっと大きいよ。
残り2回。いくつかな：137
♪もっと大きいよ。
残り1回。いくつかな：138
♪もっと大きいよ。
♪残念。正解は139でした。
```

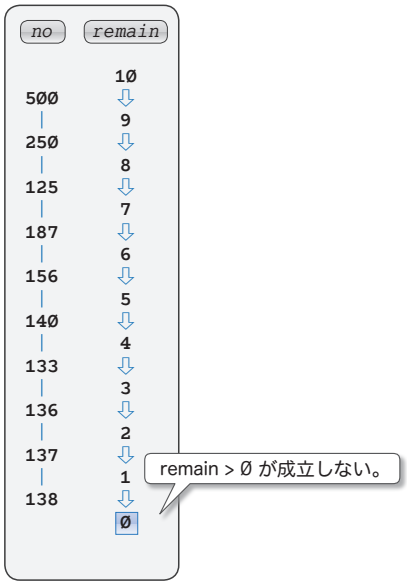


Fig.1-10 List 1–7の実行例と変数の値の変化

Column 1-2 論理演算とド・モルガンの法則

プレーヤの入力回数を制限する《数当てゲーム》のプログラムを作成しました。入力回数を制限するための do 文は、**Fig.1C-5** a のようになっています。この do 文は、図 b のように実現しても、同じように動作します。

a List 1–7のdo文

```
do {
    /*… 中略 …*/
} while (no != ans && remain > 0);
```

『正解していない』
かつ
『まだ残り回数がある』

b 同じ動作をするdo文

```
do {
    /*… 中略 …*/
} while (!(no == ans || remain <= 0));
```

『正解した』
または
『残り回数がなくなった』
の否定

Fig.1C-5 do文の制御式

図 a では論理積を求める論理AND演算子 && を利用し、図 b では論理和を求める論理OR演算子 || を利用しています。

これらの演算子の働きをまとめたのが、**Fig.1C-6** です。

■ 論理積　両方とも真であれば真

x	y	x && y
非 0	非 0	1
非 0	0	0
0	非 0	0
0	0	0

■ 論理和　一方でも真であれば真

x	y	x || y
非 0	非 0	1
非 0	0	1
0	非 0	1
0	0	0

Fig.1C-6 論理積を求める&&演算子と論理和を求める||演算子

C言語では、0 以外の値を真とみなし、0 を偽とみなします。論理AND 演算子 && は、両方のオペランドが真（0 以外の値）であれば 1 を生成し、そうでなければ 0 を生成します。また、論理OR 演算子 || は、オペランドの一方でも真（0 以外の値）であれば 1 を生成し、そうでなければ 0 を生成します。

変数 *no* に読み込んだ値が正解でないとします。このとき、式 *no* != *ans* を評価した値は、偽を示す int 型の 0 です。そのため、右オペランドの *remain* > 0 をわざわざ判定しなくても、制御式 *no* != *ans* && *remain* > 0 も偽すなわち 0 となることが分かります。左オペランド *x* と右オペランド *y* の一方でも 0 であれば、論理式 *x* && *y* 全体が偽すなわち 0 となるからです。

このように、&& 演算子の左オペランドを評価した値が 0 であれば、右オペランドの**評価は行われない**ことになっています。

同様に、|| 演算子の場合は、左オペランドを評価した値が 1 であれば、右オペランドの評価は行われないことになっています。もし一方でも真（0 以外の値）であれば、式全体が真すなわち 1 となることが明確になるからです。

論理演算の式全体の評価結果が、左オペランドの評価の結果のみで明確になる場合に、右オペランドの評価が行われないことを**短絡評価**（*short circuit evaluation*）と呼びます。

＊

Fig.1C-5 のプログラムに戻りましょう。図 b の制御式では、論理否定演算子 ! が使われています。論理否定演算子の働きは、**Fig.1C-7** のとおりです（!*x* が生成する値は、式（*x* == 0）が生成する値と同じです）。

■論理否定　偽であれば真

x	!*x*
非 0	0
0	1

Fig.1C-7　論理否定を求める!演算子

『各条件の否定をとって、論理積・論理和を入れかえた式』の否定が、もとの条件と同じになることを、**ド・モルガンの法則**と呼びます。この法則を一般的に示すと、以下のようになります。

① *x* && *y* と !(!*x* || !*y*) は等しい。
② *x* || *y* と !(!*x* && !*y*) は等しい。

図 a の制御式 *no* != *ans* && *remain* > 0 が、繰返しを続けるための《継続条件》であるのに対し、図 b の式 !(*no* == *ans* || *remain* <= 0) は、繰返しを終了するための《終了条件》の否定です。

すなわち、**Fig.1C-8** に示すイメージです。

```
do {
    /* … */
} while (継続条件);
```

```
do {
    /* … */
} while (!終了条件);
```

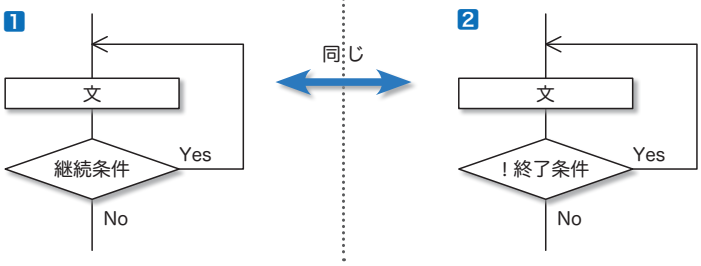


Fig.1C-8　ド・モルガンの法則

1-4 入力履歴の保存

プレーヤの入力した値を保存しておけば、当てさせる数にどのように近づいていったのか（または離れていったのか）を、ゲーム終了時に確認できるようになります。

配列

プレーヤが入力した値を保存しておき、ゲーム終了時にその値を表示するように改良しましょう。そのプログラムを **List 1-8** に示します（実行例はp.26 に示します）。

List 1-8 chap01/kazuate6.c

```c
/* 数当てゲーム（その６：入力履歴を表示）*/

#include <time.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_STAGE   10      /* 最大入力回数 */ ―― 1

int main(void)
{
    int i;
    int stage;              /* 入力した回数 */
    int no;                 /* 読み込んだ値 */
    int ans;                /* 当てさせる数 */
    int num[MAX_STAGE];     /* 読み込んだ値の履歴 */ ―― 2

    srand(time(NULL));      /* 乱数の種を初期化 */
    ans = rand() % 1000;    /* 0〜999の乱数を生成 */

    printf("0〜999の整数を当ててください。\n\n");

    stage = 0;
    do {
        printf("残り%d回。いくつかな：", MAX_STAGE - stage);
        scanf("%d", &no);
        num[stage++] = no;          /* 読み込んだ値を配列に格納 */

        if (no > ans)
            printf("\aもっと小さいよ。\n");
        else if (no < ans)
            printf("\aもっと大きいよ。\n");
    } while (no != ans && stage < MAX_STAGE);

    if (no != ans)
        printf("\a残念。正解は%dでした。\n", ans);
    else {
        printf("正解です。\n");
        printf("%d回で当たりましたね。\n", stage);
    }

    puts("\n--- 入力履歴 ---");
    for (i = 0; i < stage; i++)
        printf(" %2d : %4d %+4d\n", i + 1, num[i], num[i] - ans);

    return (0);
}
```

10に置換される

本プログラムでは、入力された値の格納先に**配列**（*array*）を利用しています。配列は、同一型の変数を直線的に並べたデータ構造です。配列内の個々の変数が要素です。

配列の宣言時は、要素数を定数式として与えなければなりません。すなわち、以下のような宣言はコンパイルエラーとなります。

```
int max_stage = 10;
int num[max_stage];      /* エラー：max_stageは定数式ではない */
```

そのため、本プログラムでは、変数 *max_stage* の代わりとなる *MAX_STAGE* をオブジェクト形式マクロとして1で宣言しています。

▶ コンパイルの前段階で、マクロ *MAX_STAGE*（プログラムの網かけ部3箇所）が 10 に置換されます。

入力した値の格納先である配列 *num* の宣言が2です。**Fig.1-11** に示すように、要素型が **int** 型で、要素数が 10 である配列です。

▶ この宣言 **int** *num*[*MAX_STAGE*]; は、**int** *num*[10]; に置換されるため、エラーにはなりません。

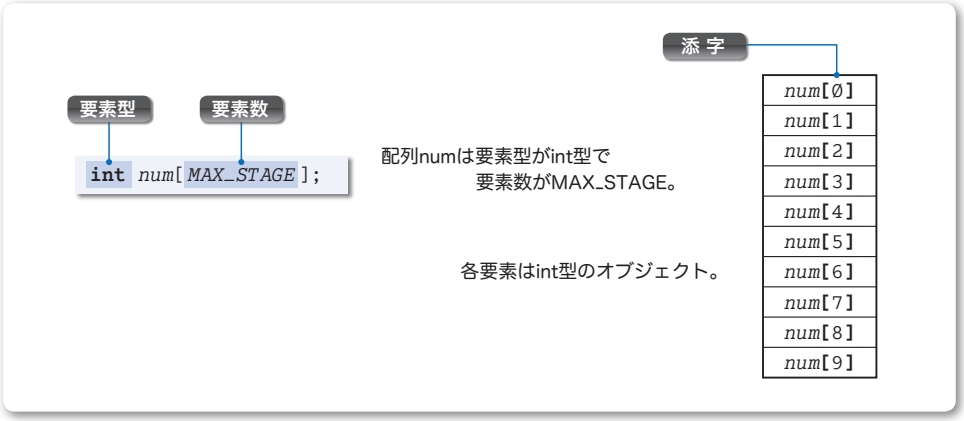


Fig.1-11 配列

配列の宣言において [ ] 内に与える値が**要素数**です。一方、個々の要素をアクセス（読み書き）するために **[ ]** 内に与える値が**添字**（*subscript*）です。

先頭要素の添字は 0 で、それ以降の添字は一つずつ増えていきます。そのため、配列 *num* の要素は、先頭から順に *num*[0], *num*[1], …, *num*[9] という式でアクセスできます。末尾要素の添字は、要素数から 1 を引いた値となるため、*num*[10] という要素は存在しません。

配列 *num* の個々の要素は、通常の（配列でない単独の）**int** 型オブジェクトと同じ性質であり、値を代入したり取り出したりできます。

▶ 宣言 **int** a[10]; での [ ] は、宣言のための記号（区切子）であるのに対し、要素をアクセスする a[3] における **[ ]** は、**添字演算子**（*subscript operator*）です。
本書では、前者を細字 [ ] で表記し、後者を**太字 [ ]** で表記しています。

入力された値の配列への格納

プレーヤが打ち込んだ値を配列の要素に格納していく部分を、**Fig.1-12** を見ながら理解していきましょう。

本プログラムで新しく導入された変数が *stage* です。この変数は、**List 1-7** での残り入力回数を表す変数 *remain* の代わりとして働きます。ゲーム開始時は 0 であり、プレーヤがキーボードから値を入力するたびにインクリメントしていきます。この値が *MAX_STAGE* すなわち 10 と等しくなるとゲームは終了です。

読み込んだ値 *x* を配列に格納するのが、図内の 1 の箇所です。ここは、三つの演算子 `[]`, `++`, `=` が絡(から)みあっています。

*

インクリメント演算子とも呼ばれる増分演算子 `++` には、`++a` という形式の前置形式と、`a++` という形式の後置形式の 2 種類があります。まずは、これらの違いを理解しましょう。

前置増分演算子 ++a

前置形式の `++a` では、式全体の評価が行われる**前**に、オペランドの値がインクリメントされます。したがって、*a* の値が 3 であるときに、

```
b = ++a;            /* aをインクリメントしてからbに代入 */
```

を実行すると、まず *a* がインクリメントされて値が 4 となり、それから式 `++a` を評価した値である 4 が *b* に代入されます。最終的に、*a* と *b* は 4 になります。

後置増分演算子 a++

後置形式の `a++` では、式全体の評価が行われた**後**に、オペランドの値がインクリメントされます。したがって、*a* の値が 3 であるときに、

```
b = a++;            /* bに代入してからaをインクリメント */
```

を実行すると、まず式 `a++` を評価した値である 3 が *b* に代入され、それから *a* がインクリメントされて値が 4 となります。最終的に、*a* は 4 に、*b* は 3 になります。

▶ 前置と後置の評価のタイミングに関しては、デクリメントを行う減分演算子 `--` についてもまったく同様です。

本プログラムの 1 では、**後置形式**の増分演算子を利用しています。プレーヤが入力した値を配列の要素に保存していく様子を理解していきましょう。

▶ 前ページの **Fig.1-11** では、配列の各要素を縦に並べて、枠の中には個々の要素をアクセスする“式”を書いていました。本図（**Fig.1-12**）では、各要素を横に並べて、枠の中には各要素の“値”を書いています。各要素の添字は、枠の上の小さい数値です（このように表記する図での数字のゼロは、0 ではなく、中にスラッシュのない 0 で表記します）。

なお、黒丸記号●の中に書かれている添字の値は、変数 *stage* の値と一致します。

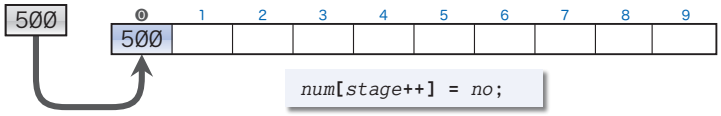
a　プレーヤが500を入力します。変数stageの値が0であるため、num[0]に500が代入され、それからstageの値がインクリメントされて1になります。

b　プレーヤが250を入力します。変数stageの値が1であるため、num[1]に250が代入され、それからstageの値がインクリメントされて2になります。

以上の処理を繰り返して、入力された値を配列に先頭から順に格納していきます。

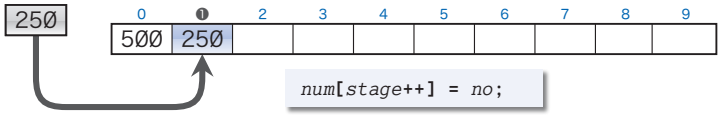
```
stage = 0;
do {
    printf("残り%d回。いくつかな：", MAX_STAGE - stage);
    scanf("%d", &no);
    num[stage++] = no;        /* 読み込んだ値を配列に格納 */ ―1
    /*… 中略 …*/
} while (no != ans && stage < MAX_STAGE);
```

a プレーヤが1回目に入力した値の格納（stageは0）



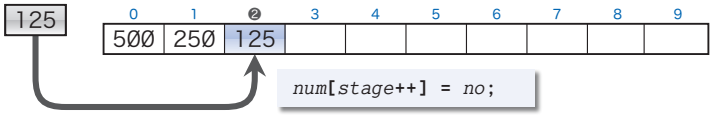
num[0]に500を代入した直後にstageをインクリメントして1にする。

b プレーヤが2回目に入力した値の格納（stageは1）



num[1]に250を代入した直後にstageをインクリメントして2にする。

c プレーヤが3回目に入力した値の格納（stageは2）



num[2]に125を代入した直後にstageをインクリメントして3にする。

… 以下省略 …

Fig.1-12　入力履歴の配列への格納

for 文による入力履歴の表示

ゲームが終了すると、プレーヤが入力した値の履歴を表示します。それを行うのが、以下に示す **for** 文です。

```
for (i = 0; i < stage; i++)
    printf(" %2d : %4d %+4d\n", i + 1, num[i], num[i] - ans);
```

この **for** 文が行う繰返しを日本語で表現すると、次のようになります。

> まず *i* の値を 0 にして、*i* の値が *stage* より小さいあいだ、*i* の値を一つずつ増やすことによって、ループ本体を *stage* 回実行する。

数当てゲームの本体である **do** 文が終了した時点での変数 *stage* の値は、プレーヤが数値を入力した回数です。もし 7 回目の入力で正解していれば *stage* の値は 7 となっています。その場合、この **for** 文は、繰返しを 7 回行うことになります。

Fig.1-13 に示すように、各繰返しでは、配列 *num* 内の添字が *i* である要素 *num*[*i*] に着目します。黒丸記号●内の添字が、変数 *i* の値と一致します。

ループ本体内では、***printf*** 関数によって、3 個の値を表示します。

① 何回目の入力であるのか	*i* + 1
② プレーヤが入力した値	*num*[*i*]
③ 入力した値と正解との差	*num*[*i*] - *ans*

実行例

```
0～999の整数を当ててください。

残り10回。いくつかな：500↵
♪もっと小さいよ。
残り9回。いくつかな：250↵
♪もっと小さいよ。
… 中略 …
残り4回。いくつかな：116↵
正解です。
7回で当たりましたね。

--- 入力履歴 ---
  1 :  500 +384
  2 :  250 +134
  3 :  125   +9
  4 :   62  -54
  5 :   93  -23
  6 :  108   -8
  7 :  116   +0
```

①で表示するのは、変数 *i* に 1 を加えた値です。添字が 0 から始まるのに対し、私たち人間が数える数値は 1 から始まります。1 を加えているのは、変数の値と表示する値の差を補正するためです。

▶ たとえば、図**c**では、変数 *i* の値である 2 に 1 を加えた 3 を表示します。

②では、プレーヤが入力した値である *num*[*i*] をそのまま表示します。

▶ たとえば、図**c**では、*num*[*i*] すなわち *num*[2] の値である 125 を表示します。

③では、入力した値と正解との差を表示します。入力した値のほうが大きければ + 符号を付け、入力した値のほうが小さければ - 符号を付けて表示します。

▶ たとえば、図**c**では、*num*[*i*] の値 125 から正解 116 を引いた値 9 を『+9』として表示します。

書式文字列 **"%d"** によって **int** 型の値を表示する際は、値が負のときにのみ - 符号が付くことは（おそらく経験からも）知っていますね。

書式文字列を **"%+d"** とすると、**値が正や 0 であっても符号が表示されます。**

▶ ***printf*** 関数や書式文字列の詳細は、第２章で詳しく学習します。

```
for (i = 0; i < stage; i++)
    printf(" %2d : %4d %+4d\n", i + 1, num[i], num[i] - ans);
```

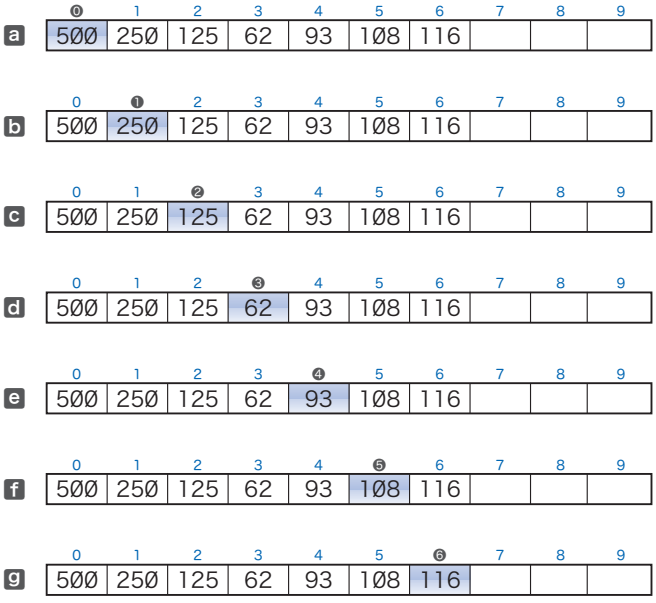


Fig.1-13 配列numの走査による入力履歴の表示

配列内の各要素をなぞりながら1個ずつ順に着目していくことを**走査**（*traverse*）と呼びます。基本的な用語ですから、必ず覚えなければなりません。

＊

さて、**for**文は、変数*i*の値が*stage*未満のあいだ繰り返します。そのため、**for**文終了時の変数*i*の値は、*stage* - 1ではなく*stage*となることに注意しましょう。

▶ 本プログラムの**for**文を、**while**文で書きかえると以下のようになります。

```
i = 0;
while (i < stage) {
    printf(" %2d : %4d %+4d\n", i + 1, num[i], num[i] - ans);
    i++;
}
```

ループ本体が実行されるのは、変数*i*の値が0, 1, …, *stage* - 1の*stage*回です。最後に***printf***関数が呼び出されるときの変数*i*の値は*stage* - 1です。その値がインクリメントされ、*stage*と等しくなったときに、制御式*i* < *stage*が成立しなくなって繰返しが終了します。

配列の要素の初期化

配列について少し詳しく学習しましょう。まずは、初期化のための宣言です。

要素を初期化するには、個々の要素に対する初期化子を先頭から順にコンマ , で区切って並べ、それを { } で囲んだものを与えます。たとえば、

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

と宣言すると、要素 a[0], a[1], a[2], a[3], a[4] が順に 1, 2, 3, 4, 5 で初期化されます。

以下に示すのは、すべての要素を 0 で初期化する宣言です。

```
int a[5] = {0, 0, 0, 0, 0};      /* すべての要素を0で初期化 */
```

ただし、{ } 形式の初期化子を与える配列の宣言では、**初期化子が与えられていない要素は 0 で初期化される**ことになっています。したがって、

```
int a[5] = {0};                  /* すべての要素を0で初期化 */
```

と宣言すると、初期化子が与えられていない a[1] 以降のすべての要素も 0 で初期化されます。こちらのほうが簡潔です。

▶ 静的記憶域期間（p.153）をもつ配列（関数の外で定義された配列と、関数の中で **static** 付きで定義された配列）は、初期化子を与えなくても、すべての要素が 0 で初期化されます。

配列の宣言時には、**要素数を省略する**こともできます。

```
int a[] = {1, 2, 5};             /* 要素数を省略 */
```

この場合、初期化子の個数に基づいて、配列 a の要素数は 3 とみなされます。すなわち、以下の宣言と同じです。

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

なお、初期化子の個数が、配列の要素数を超えるとエラーになります。

```
int a[3] = {1, 2, 3, 5};         /* エラー：初期化子が多すぎる */
```

なお、初期化子である {1, 2, 3} を、代入の右辺の式として用いることはできません。したがって、以下の代入はエラーとなります。

```
int a[3];
a = {1, 2, 3};                   /* エラー：このような代入はできない */
```

▶ 初期化子については、p.68 や p.252 などでも学習します。

配列の要素数の取得

List 1-8 では、配列を宣言するのに先だって、その要素数をマクロとして定義しました。

要素数を事前にマクロで定義しないほうが都合のよい場合などは、まず配列を宣言しておき、その後で要素数を求めることになります。

配列の要素数を求めるための定石は、**sizeof** 演算子を用いて計算する方法です。その方法を利用したプログラム例を **List 1-9** に示します。

List 1-9 chap01/array.c

```
/* 配列の要素数と各要素の値を表示 */

#include <stdio.h>

int main(void)
{
    int i;
    int a[] = {1, 2, 3, 4, 5};
    int na = sizeof(a) / sizeof(a[0]);     /* 要素数 */

    printf("配列aの要素数は%dです。\n", na);

    for (i = 0; i < na; i++)
        printf("a[%d] = %d\n", i, a[i]);

    return (0);
}
```

実行結果

```
配列aの要素数は5です。
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 5
```

Fig.1-14 に示すように、配列の大きさは **sizeof**(*a*) によって求められ、要素の大きさは **sizeof**(*a*[0]) によって求められます。

int 型の大きさは処理系によって異なりますが、**sizeof**(*a*) / **sizeof**(*a*[0]) によって求められる値は、**int** 型の大きさとは無関係に配列の要素数となります。たとえば、**int** 型が2バイトであれば **sizeof**(*a*) は 10、**sizeof**(*a*[0]) は 2 ですから、10 / 2 の演算によって、要素数 5 が求められます。また、**int** 型が4バイトであれば、20 / 4 の演算によって、やはり要素数 5 が求められます。

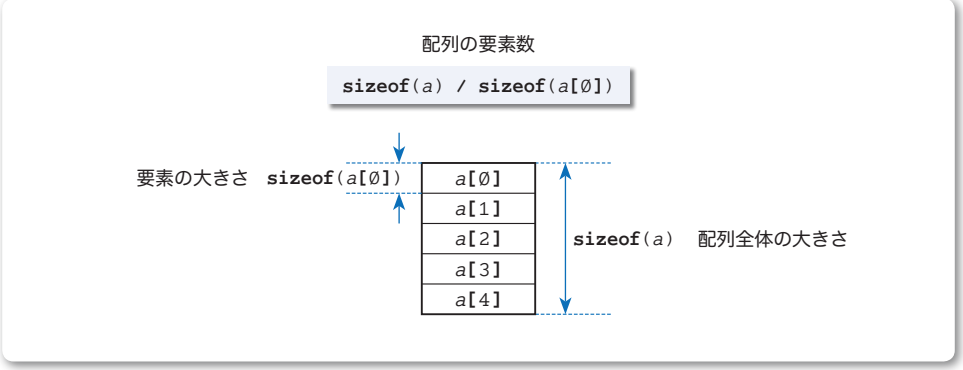


Fig.1-14 配列の要素数を求める式

変数 *na* が、配列 *a* の要素数である 5 で初期化されることが分かりました。もし配列 *a* の宣言を、

```
int a[] = {1, 3, 5, 7, 9, 11};
```

```
配列aの要素数は6です。
a[0] = 1
a[1] = 3
a[2] = 5
a[3] = 7
a[4] = 9
a[5] = 11
```

に変更すると、変数 *na* は 6 で初期化されます。もちろん、右に示すように、期待どおりの実行結果が得られます。

初期化子の増減に伴って**プログラムの他の箇所を修正する必要はありません。**

▶ 配列の要素数を **sizeof**(*a*) / **sizeof**(*a*[0]) ではなく、**sizeof**(*a*) / **sizeof**(**int**) によって求める方法を解説しているテキストがあります。しかし、これは、よい方法ではありません。

何らかの理由によって要素型を変更するとしたらどうなるかを考えてみましょう。たとえば『配列の要素に格納すべき値が **int** 型では収まらなくなったので要素型を **long** 型に変更する』とします。その場合、要素を求める式 **sizeof**(*a*) / **sizeof**(**int**) を **sizeof**(*a*) / **sizeof**(**long**) に変更しなければならなくなります。

式 **sizeof**(*a*) / **sizeof**(*a*[0]) であれば、要素型に依存しません。

まとめ

増分演算子と減分演算子

オペランドの値をインクリメントする増分演算子 **++** と、デクリメントする減分演算子 **--** には前置形式と後置形式とがある。前置形式では、式が評価される前にインクリメント／デクリメントが行われ、後置形式では、式が評価された後にインクリメント／デクリメントが行われる。

配列

配列は、同一型の要素が直線状に並んだデータ構造である。宣言する際は、要素型と要素数を与える。その際、要素数は定数式として与えなければならない。各要素をアクセスするには、添字演算子 **[]** を用いる。先頭要素の添字は 0 である。

配列の初期化子は、個々の要素に対する初期化子を先頭から順にコンマ , で区切って並べ、それを { } で囲んだ形式である。

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

配列の要素数

通常、宣言以外の箇所でも配列の要素数を知る必要がある。以下のように宣言するとよい。

① オブジェクト形式マクロで要素数を事前に定義する。

```
#define NA  7                          /* 配列aの要素数を先に定義 */
int a[NA];
```

② 配列を宣言した後に要素数を取得する。

```
int a[7];
int na = sizeof(a) / sizeof(a[0]);     /* 配列aの要素数を後で取得 */
```

自由課題

本文に示したプログラムを読んで理解するだけでなく、ここに示す問題を解いたり、自分でプログラムを設計・開発したりして、プログラミング力を磨いてください。

※ 自由課題ですから、解答はありません。

課題 1–1

《おみくじ》のプログラムを作成せよ。0 ～ 6 の乱数を生成し、その値に応じて、〔大吉〕〔中吉〕〔小吉〕〔吉〕〔末吉〕〔凶〕〔大凶〕を表示すること。

課題 1–2

前問で作成したプログラムを、出る運勢が均等とならないように改良したプログラムを作成せよ（たとえば、〔末吉〕〔凶〕〔大凶〕を出にくくするとよい）。

課題 1–3

当てさせる数を –999 以上 999 以下の整数とした《数当てゲーム》を作成せよ。

プレーヤが入力できる最大の回数が、どのくらいであれば適当であるのかも考察すること。

課題 1–4

当てさせる数を 3 以上 999 以下の 3 の倍数（3, 6, 9, …, 999）とした《数当てゲーム》を作成せよ。3 の倍数でない値が入力された場合に、ただちにゲームを終了するものと、比較結果を表示せず再入力させる（入力回数もカウントしない）ものの二つを作ること。

プレーヤが入力できる最大の回数が、どのくらいであれば適当であるのかも考察すること。

課題 1–5

当てさせる数の範囲を事前に決定するのではなく、プログラム実行時に乱数で決定する《数当てゲーム》を作成せよ。たとえば、生成して得られた二つの乱数が 23 と 8,124 であれば、23 以上 8,124 以下の数を当てさせるようにする。

なお、プレーヤが入力できる最大の回数は、当てさせる数の範囲に応じて適切な値に自動的に（プログラム内での計算によって）設定すること。

課題 1–6

開始時にプレーヤにレベルを選択させる《数当てゲーム》を作成せよ。たとえば、以下のように表示して選択させること。

```
レベルを選んでください（1)1 ～ 9　(2)1 ～ 99　(3)1 ～ 999　(4)1 ～ 9999：
```

課題 1–7

List 1-8（p.22）のプログラムでの入力履歴表示では、正解との差が 0 であっても符号を付けて表示するため、少々みっともない。0 に対しては符号を付けないように変更せよ。

課題 1–8

List 1-8 の `do` 文を `for` 文を用いて書き直したプログラムを作成せよ。