Lesson 1
数当てゲーム

本 Lesson では《数当てゲーム》を作成します。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 Lesson 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）
- break 文
- for 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）
- 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

まず最初に、コンピュータが用意する〔当てさせる数〕と、プレーヤがキーボードから打ち込んだ値とを比較して、その結果を表示する試作版を作ります。

if 文による分岐

List 1-1 に示す試作版《数当てゲーム》を実行してみましょう。

プレーヤがキーボードから数値を打ち込むと、その数値が〔当てさせる数〕と比べて大きいか／小さいか／等しいか（正解であるか）を表示します。

List 1-1

```
/*
    数当てゲーム（その１）
*/

#include <stdio.h>

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

    printf("0〜9の整数を当ててください。\n\n");

    printf("いくつでしょう：");
    scanf("%d", &x);

    if (x > no)
        printf("\a大きいです。\n");
    else if (x < no)
        printf("\a小さいです。\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。

いくつでしょう：9
♪大きいです。
```

実行例 2

```
0〜9の整数を当ててください。

いくつでしょう：6
♪小さいです。
```

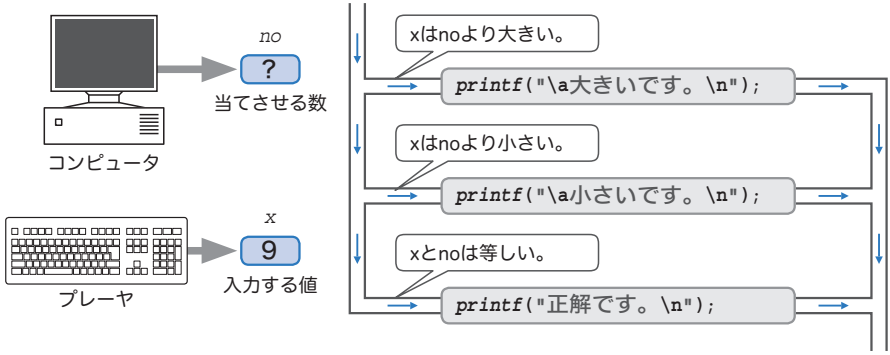
実行例 3

```
0〜9の整数を当ててください。

いくつでしょう：7
正解です。
```

本ゲームでの当てさせる数は 7 です。それを表す変数 *no* と、変数 *x* に読み込まれた値との比較を行うのが、網かけ部の **if** 文です。**Fig.1-1** に示すように、比較の結果に応じて『大きいです。』『小さいです。』『正解です。』のいずれかを表示します。

表示する文字列には、2 種類の**拡張表記**が含まれます。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〕がなるため、本書の実行例では♪記号で表します。

▶ 拡張表記は Lesson 2 で詳しく学習します。パソコンで使われる JIS コード（p.70）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

二つの変数値を比較するif文を理解しましょう。

if文は、**制御式**と呼ばれる**式**を評価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となります。

if文の構文

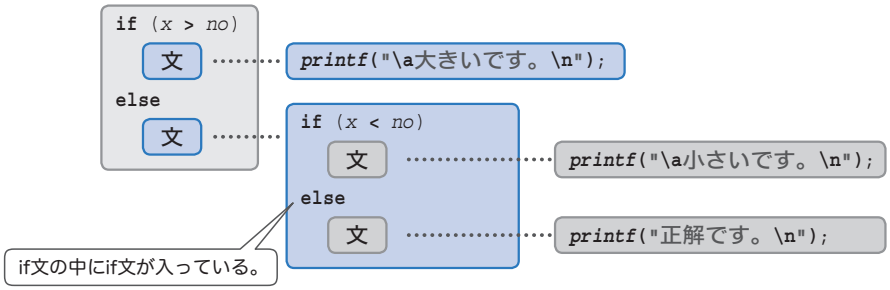
- `if（式）文`
- `if（式）文 else 文`

ところが、本プログラムのif文は、

```
if（式）文 else if（式）文 else 文
```

という形です。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

Lesson 1

多分岐の実現法

さらに奥深く if 文を理解しましょう。

本プログラムの if 文と同じ動作をするように作ったのが **List 1-2** と **List 1-3** です。

三つの if 文を比較・検討します。

List 1-1

```
if (x > no)
    printf("\a大きいです。\n");
else if (x < no)
    printf("\a小さいです。\n");
else
    printf("正解です。\n");
```

List 1-2

追加された網かけ部にプログラムの流れが到達するのは、それより前の判定（x > no）と（x < no）の両方が成立しない場合、すなわち、x と no が等しい場合のみです。

必ず成立する条件を、わざわざ判定することになります。

List 1-2

```
if (x > no)
    printf("\a大きいです。\n");
else if (x < no)
    printf("\a小さいです。\n");
else if (x == no)
    printf("正解です。\n");
```

List 1-3

if 文が三つ並んでいます。変数 x と no 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。

List 1-3

```
if (x > no)
    printf("\a大きいです。\n");
if (x < no)
    printf("\a小さいです。\n");
if (x == no)
    printf("正解です。\n");
```

三つの実現法において、どの判定が行われるかを、x と no の大小関係別にまとめた表を **Table 1-1** に示します。

Table 1-1　三つの実現法で行われる判定

大小関係	x ＞no のとき	x ＜no のとき	x ＝no のとき
List 1-1	①	① ②	① ②
List 1-2	①	① ②	① ② ③
List 1-3	① ② ③	① ② ③	① ② ③

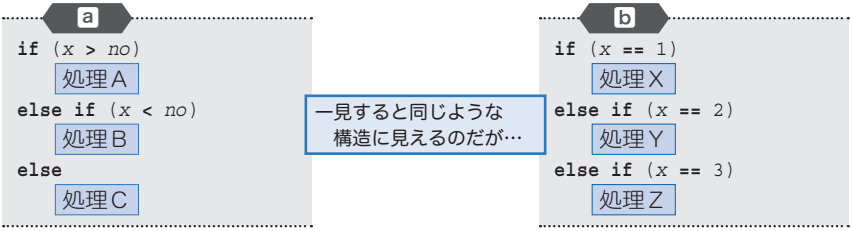
① （x > no）の判定
② （x < no）の判定
③ （x == no）の判定

たとえば x が no より大きいときは、**List 1-1** と **List 1-2** では（x > no）の判定だけが行われ、**List 1-3** では（x > no），（x < no），（x == no）のすべてが行われます。

どの条件においても判定回数が少ないのが **List 1-1** です。

＊

この if 文の優れた点は、判定回数が少ないことだけではありません。**Fig.1-3** で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

■ aのif文

List 1-1 の if 文の構造を一般化して表したものです。プログラムの流れは三つに分岐して〔処理A〕〔処理B〕〔処理C〕のいずれかの処理が実行されます。

■ bのif文

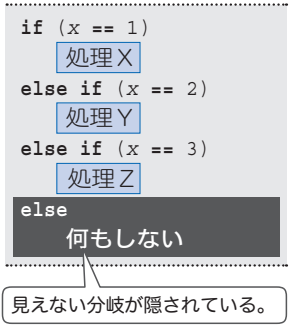
変数 x の値に応じて分岐する if 文です。

〔処理X〕〔処理Y〕〔処理Z〕のいずれか一つが行われるように見えますが、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します。すなわち、図aの if 文とは構造が**まったく異なります**。

もちろん、最後の判定 if (x == 3) を取り去ることは**できません**。

▶ もし取り去ると、x の値が 3 でなく 4 や 5 であっても〔処理Z〕が実行されてしまいます。



●**Fig.1-4**　bの解釈

図aの構造をもつ **List 1-1** の if 文は、その最後の else に if がないため、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おかれている **List 1-2** よりも、**List 1-1** のほうが優れています。

▶ プログラムの読み手に対して『x が no 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、**List 1-2** 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、実は、効率のことを気にする必要性は意外と小さいのです。

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

改良したプログラムを **List 1-4** に示します。**List 1-1** のプログラムの **if** 文の後半を削って網かけ部の **do** 文を追加しています。

List 1-4

```
/*
    数当てゲーム（その２：当たるまで繰り返す）
*/

#include <stdio.h>

int main(void)
{
    int x;                      /* 読み込んだ値 */
    int no = 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("いくつでしょう：");
        scanf("%d", &x);

        if (x > no)
            printf("\a大きいです。\n");
        else if (x < no)
            printf("\a小さいです。\n");
    } while (x != no);          /* 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。

いくつでしょう：6↵
♪小さいです。
いくつでしょう：8↵
♪大きいです。
いくつでしょう：7↵
正解です。
```

do 文は、**後判定繰返し**で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);
```

制御式である**式**を評価した値が 0 でない限り、**文**は何度も実行されます。繰返しが終了するのは、評価した値が 0 になったときです。

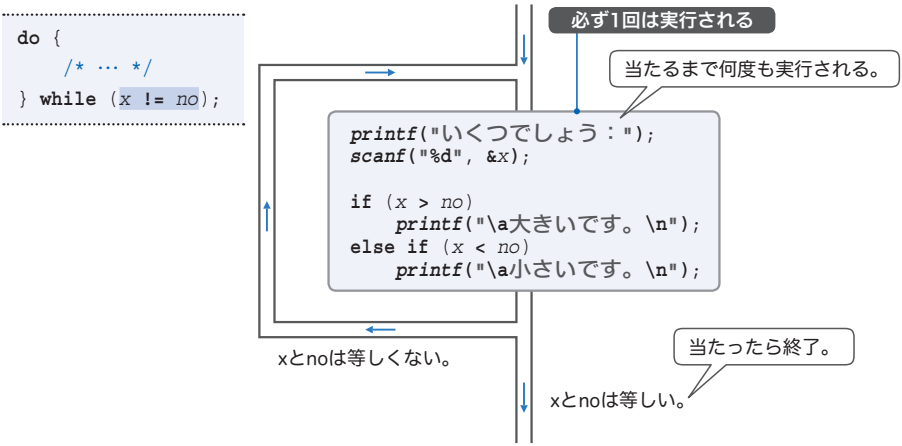
▶ 既に解説した **if** 文や、後で解説する **while** 文や **for** 文などとは異なり、末尾にセミコロン ; が付きます。

本プログラムの **do** 文による繰返しの様子を **Fig.1-5** に示しています。

まず do 文の制御式である x != no に着目しましょう。

左右のオペランドが等しいかどうかを判定する == とは対照的に、演算子 != は**等しくない**かどうかを判定します。条件が成立すれば int 型の 1 を、成立しなければ int 型の 0 を生成することは、両演算子に共通です。

読み込んだ値 x が、当てさせる数 no と等しくなければ、制御式 x != no を評価して得られる値が 1 となるため、do 文による繰返しが行われます。



● **Fig.1-5**　do文によるプログラムの流れの繰返し

当てさせる数 no と同じ値が x 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。

そうすると、画面に『正解です。』と表示して、プログラムも終了します。

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。

■ 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比較します。

Lesson 1

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。

List 1-5 に示すのは、do 文と対照的な**前判定繰返し**を行う while 文を利用して書きかえたプログラムです。

List 1-5

```
/*
    数当てゲーム（その３：当たるまで繰り返す）
*/

#include <stdio.h>

int main(void)
{
    int x;                      /* 読み込んだ値 */
    int no = 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("いくつでしょう：");
        scanf("%d", &x);

        if (x > no)
            printf("\a大きいです。\n");
        else if (x < no)
            printf("\a小さいです。\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。

いくつでしょう：6↵
♪小さいです。
いくつでしょう：8↵
♪大きいです。
いくつでしょう：7↵
正解です。
```

while 文の構文は、右のとおりです。

制御式である**式**を評価した値が 0 でない限り、**文**は何度でも実行されます。ただし、その値が 0 になったら繰返しは終了です。

while文の構文

```
while（式）文
```

▶ do 文とは異なり、構文の末尾にはセミコロンが付きません。
　また、最初に**式**を評価した値が 0 になると**文**が**一度も実行されない**点も、do 文と異なります。

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。このような繰返しを、一般に〔無限ループ〕と呼びます。

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。そこで活躍しているのが、繰返し文を強制的に抜け出すための **break** 文です。

x と *no* が等しければ **break** 文が実行されるため、**while** 文による繰返しが強制的に中断されます。

▶ **break** 文を多用したプログラムは読みにくいものです。『ある特別な条件が成立したときに、**どうしても**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 1-4** のように実現すべきです。

while 文と do 文

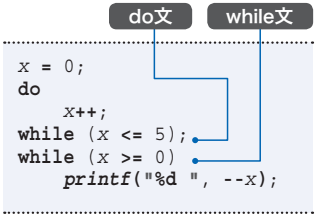
プログラム中の **while** が、**do** 文の一部であるのか、**while** 文の一部であるのかは、見分けにくいものです。

右のプログラムで考えましょう。

まず最初に変数 *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。

▶ 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、p.20 で解説します。




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

右に示すように、**do** 文が繰返しの対象としている部分を { } で囲んで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** 文の一部
} **while**	… 行の先頭が } ならば **do** 文の一部

本来は、**do** 文も **while** 文も **for** 文も、繰返しの対象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、たとえ繰返しの対象となる文が一つであっても、あえて { } を導入したほうがプログラムが読みやすくなります。

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おり、あらかじめ答えが分かるものでした。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

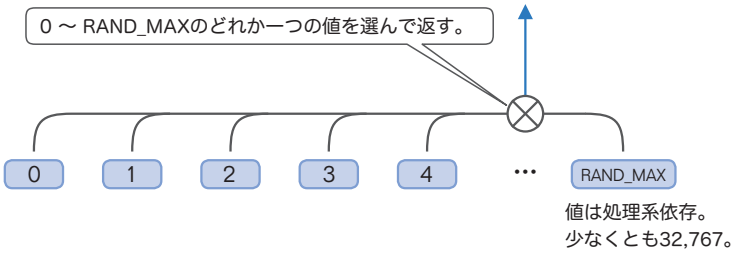
ゲームのたびに〔当てさせる数〕を変えるには、いわゆる**乱数**（らんすう）が必要です。乱数を生成するのが、次に示す ***rand*** 関数です。

	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・返却する乱数は **int** 型の整数です。その最小値は 0 で、処理系に依存する最大値は <stdlib.h> でマクロ **RAND_MAX** として定義されている値です。

その値は最低でも 32,767 と規定されているので、***rand*** 関数は **Fig.1-6** のように動作します。

この関数を呼び出せば 0 以上 **RAND_MAX** 以下の整数が得られることになります。



● **Fig.1-6**　rand関数による乱数の生成

それでは、実際に乱数を生成・表示してみましょう。**List 1-6** に示すプログラムを実行してみてください。生成できる乱数の範囲が表示された後に、実際に生成された乱数が表示されます。

List 1-6

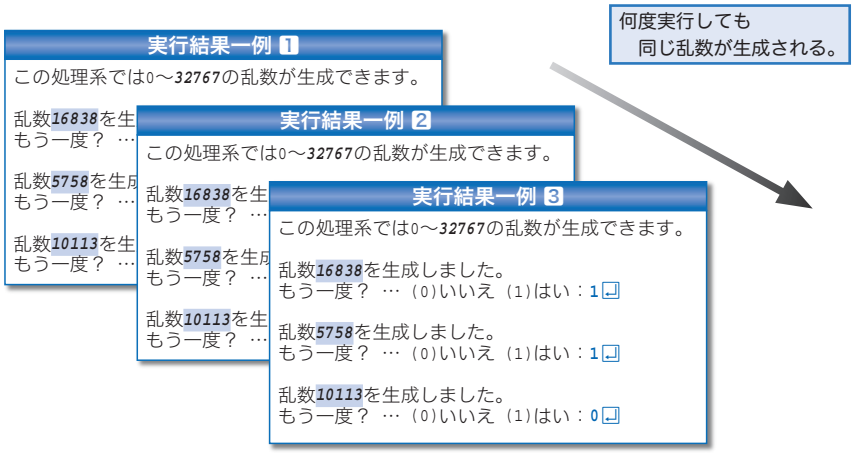
```
/*
    乱数を生成（その１）
*/

#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);
}
```

この処理系で生成可能な乱数の最大値

乱数を生成

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、繰り返し乱数を生成・表示できます。

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？**



※ここに示すのは一例です。生成される値は処理系に依存します。

● **Fig.1-7　List 1-6**の実行例

srand 関数：乱数生成のための種の設定

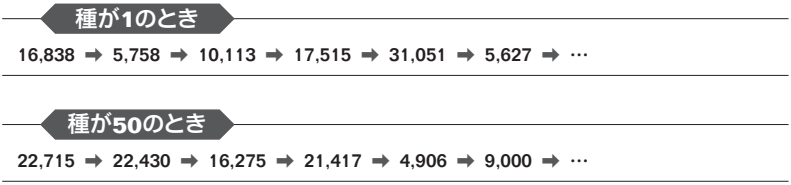
乱数は、〔種（たね）〕と呼ばれる基準値に演算を施して作られます。プログラム実行のたびに同じ乱数の系列が生成されるのは、**定数 1 という種が *rand* 関数に埋め込まれているからです。異なる系列の乱数を生成するには、種の値を変えなければなりません。

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) と呼び出すと、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します。

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。



● **Fig.1-8**　種とrand関数が生成する乱数系列の一例

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。

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ります。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。

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。それを利用したプログラムを **List 1-7** に示します。

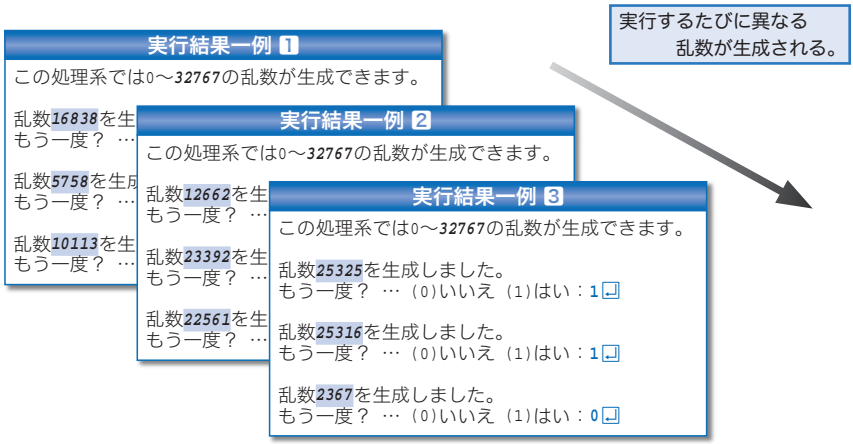
List 1-7

```
/*
    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);
}
```

現在の時刻を乱数の種にする

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 Lesson 6 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〔公式〕として利用しましょう。



※ここに示すのは一例です。生成される値は処理系に依存します。

● **Fig.1-9　List 1-7**の実行例

Lesson 1

当てさせる数をランダムにする

rand 関数が生成する範囲は 0 ～ **RAND_MAX** です。とはいえ、そのような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。

＊

List 1-8 に示すプログラムは、〔当てさせる数〕を 0 以上 999 以下の乱数とした数当てゲームです。

網かけ部では、生成した乱数を 1000 で割った剰余を変数 *no* に代入しています。

List 1-8

```
/*
    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 x;                          /* 読み込んだ値 */
    int no;                         /* 当てさせる数 */

    srand(time(NULL));                              /* 乱数の種を初期化 */
    no = 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("いくつでしょう：");
        scanf("%d", &x);

        if (x > no)
            printf("\a大きいです。\n");
        else if (x < no)
            printf("\a小さいです。\n");
    } while (x != no);                              /* 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。

いくつでしょう：500↵
♪小さいです。
いくつでしょう：750↵
♪大きいです。
いくつでしょう：620↵
正解です。
```

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。まずは何度も実行して楽しんでみましょう。

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。

```
no = 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。

```
no = 100 + rand() % 900;        /* 100～999の乱数を生成 */
```

まとめ

●乱数が必要であれば、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 回だけでよく何度も呼び出す必要はない）。

● *rand* 関数を呼び出せば 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                  /* 0 以上   a   未満の乱数 */
b + rand() % a                  /* b 以上 b + a 未満の乱数 */
```

※処理系によっては、生成される乱数に偏りが生じることもある。その場合は、以下の方法を試してみよう。

```
rand() / (RAND_MAX / a + 1)         /* 0 以上   a   未満の乱数 */
rand() / (RAND_MAX / a + 1) + b     /* b 以上 b + a 未満の乱数 */
```

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**プログラムを **List 1-9** に示します。

List 1-9

```
/*
    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 x;                      /* 読み込んだ値 */
    int no;                     /* 当てさせる数 */
    int max_stage = 10;         /* 最大入力回数 */
    int cnt = max_stage;        /* 残り何回入力できるか？ */

    srand(time(NULL));                          /* 乱数の種を初期化 */
    no = 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。いくつでしょう：", cnt);
        scanf("%d", &x);
        cnt--;                                  /* 残り回数が一つ減る */

        if (x > no)
            printf("\a大きいです。\n");
        else if (x < no)
            printf("\a小さいです。\n");
    } while (x != no  &&  cnt > 0);

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

    return (0);
}
```

入力制限回数のチェック

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

もう一つの新しい変数 *cnt* 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、*cnt* の値は 10，9，8，… とデクリメントされます。

この値が 0 になるとゲームは終了です。そのため、do 文の判定として、式 *x* != *no*

だけでなく、網かけ部の *cnt* > 0 が追加されています。

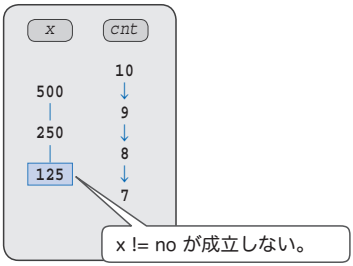
二つの式を結ぶ論理 AND 演算子 **&&** は、両方のオペランドがともに非 0 である場合にのみ **int** 型の 1 を生成し、そうでなければ 0 を生成します。

そのため、図**a**に示すように当たった場合だけでなく、図**b**に示すように 10 回入力しても当たらず *cnt* が 0 になった場合も、ちゃんと繰返しは終了します。

なお、何回目の入力で当たったのかは、*max_stage* から *cnt* 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 *cnt* の値は 7 となっています。したがって、*max_stage* - *cnt* によって 3 が得られます。

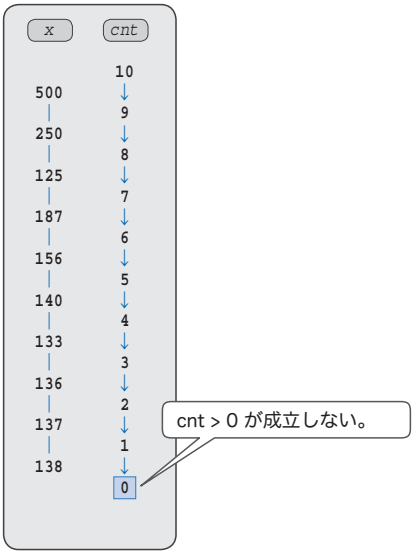
a 125を当てる（3回目で正解）

```
実行例
0～999の整数を当ててください。

残り10回。いくつでしょう：500
♪大きいです。
残り9回。いくつでしょう：250
♪大きいです。
残り8回。いくつでしょう：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。いくつでしょう：500
♪大きいです。
残り9回。いくつでしょう：250
♪大きいです。
残り8回。いくつでしょう：125
♪小さいです。
残り7回。いくつでしょう：187
♪大きいです。
残り6回。いくつでしょう：156
♪大きいです。
残り5回。いくつでしょう：140
♪大きいです。
残り4回。いくつでしょう：133
♪小さいです。
残り3回。いくつでしょう：136
♪小さいです。
残り2回。いくつでしょう：137
♪小さいです。
残り1回。いくつでしょう：138
♪小さいです。

♪残念。正解は139でした。
```



● **Fig.1-10　List 1-9**の実行例と変数の値の変化

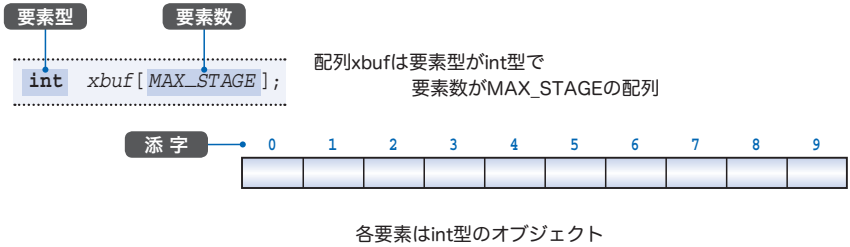
1-4 入力履歴の保存

プレーヤが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（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たプログラムが **List 1-10** です（実行例は p.22 に示します）。

入力した値の格納先である *xbuf* は、**Fig.1-11** に示すように、要素型が **int** 型で、要素数が *MAX_STAGE* すなわち 10 の**配列**（*array*）です。



● **Fig.1-11**　配列

配列の要素数は定数式で与える必要があり、次の宣言は許されません。

```
int max_stage = 10;
int xbuf[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* の代わりに**オブジェクト形式マクロ**（*object-like macro*）である *MAX_STAGE* を導入しています。

▶ コンパイル時にマクロ *MAX_STAGE* が 10 に置換されます。宣言 **int** *xbuf*[*MAX_STAGE*]; は **int** *xbuf*[10]; と解釈されるため、エラーにはなりません。

配列の宣言において [ ] 内に与える値が**要素数**であるのに対し、個々の要素をアクセスするために [ ] 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。配列 *xbuf* の要素は、先頭から順に *xbuf*[0], *xbuf*[1], …, *xbuf*[9] です。末尾要素の添字は、要素数から 1 を引いた値となるため *xbuf*[10] という要素は存在しません。

List 1-10

```
/*
    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 */

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

    srand(time(NULL));                          /* 乱数の種を初期化 */
    no = 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。いくつでしょう：", MAX_STAGE - stage);
        scanf("%d", &x);
        xbuf[stage++] = x;          /* 読み込んだ値を配列に格納 */

        if (x > no)
            printf("\a大きいです。\n");
        else if (x < no)
            printf("\a小さいです。\n");
    } while (x != no  &&  stage < MAX_STAGE);

    if (x != no)
        printf("\a残念。正解は%dでした。\n", no);
    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, xbuf[i], xbuf[i] - no);

    return (0);
}
```

変数max_stageに代わるマクロ

入力履歴を保存

入力履歴を表示

配列 *xbuf* 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。

入力履歴の配列への格納

stage は、残り入力回数を表す cnt の代わりに導入した変数です。

最初は 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 になるとゲームは終了です。

stage をインクリメントする箇所は、三つの演算子 [], ++, = が絡(から)みあっています。

```
xbuf[stage++] = x;
```

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。これらの違いをきちんと理解しましょう。

■ 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 3 が b に代入され、それからインクリメントが行われて、a の値が 4 となります。最終的に、a は 4 に、b は 3 になります。

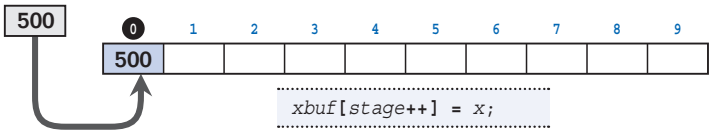
▶　前置および後置に関しては、デクリメントを行う減分演算子 -- についてもまったく同様です（**List 1-9** では、いずれの形式でも同じ結果が得られます）。

本プログラムで利用しているのは、**後置形式**の増分演算子です。

プレーヤが入力した値をどのように配列の要素に保存していくのかを、**Fig.1-12** に示す例で考えましょう。

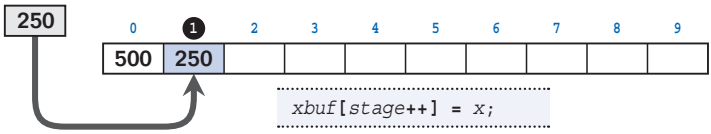
▶　黒い丸記号●の中に書かれている値が stage です。

a プレーヤが 1 回目に入力した値の格納（stageは0）



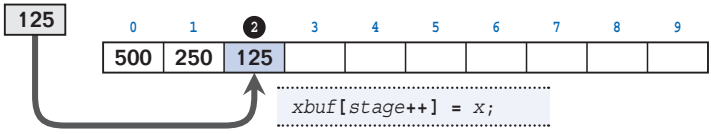
xbuf[0]に500を代入した直後にstageをインクリメントして1にする。

b プレーヤが2回目に入力した値の格納（stageは1）



xbuf[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）



xbuf[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。

…（以降省略）…

● **Fig.1-12**　入力履歴の配列への格納

a　プレーヤが 500 を入力します。変数 *stage* の値は 0 であるため、*xbuf*[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* の値は 1 であるため、*xbuf*[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。

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, xbuf[i], xbuf[i] - no);
```

この **for** 文の働きを日本語に翻訳（？）すると、次のような感じになります。

> まず *i* の値を 0 にして、*i* の値が *stage* より小さい間、*i* の値を一つずつ増やしながら繰り返す。

たとえば 5 回目の入力で正解していれば *stage* の値は 5 となっています。したがって、この **for** 文では五つの要素 *xbuf*[0]，*xbuf*[1]，…，*xbuf*[4] を表示することになります。

ちなみに、**for** 文ではなく **while** 文を用いると、次のように煩雑になります。

```
i = 0;
while (i < stage) {
    printf(" %2d : %4d %+4d\n", i + 1, xbuf[i], xbuf[i] - no);
    i++;
}
```

プレーヤが入力した値 *xbuf*[*i*] に続けて表示するのが、正解との差 *xbuf*[*i*] - *no* です。

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ています。

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*** 関数の詳細は、Lesson 2 で詳しく学習します。

実行例

```
0～999の整数を当ててください。

残り10回。いくつでしょう：500↵
♪大きいです。
残り9回。いくつでしょう：250↵
♪大きいです。
…（中略）…

残り1回。いくつでしょう：122↵
正解です。
10回で当たりましたね。

--- 入力履歴 ---
 1 :  500 +378
 2 :  250 +128
 3 :  125   +3
 4 :   62  -60
 5 :   93  -29
 6 :  108  -14
 7 :  116   -6
 8 :  121   -1
 9 :  123   +1
10 :  122   +0
```

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。

したがって、すべての要素を 0 で初期化する宣言は、次のようになります。

```
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。

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 */
```

▶ 静的記憶域期間をもつ配列（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。

なお、初期化子である {1, 2, 3} を代入で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 */
```

配列の要素数の取得

配列 a の要素数は、要素型によらず **sizeof**(a) / **sizeof**(a[0]) で求められます。

要素数をマクロで定義しないほうが都合のよい場合は、まず配列を宣言し、その後で要素数を求めるといいでしょう。プログラム例を **List 1-11** に示します。

List 1-11

```
/*
    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
```

変数 na は、配列 a の要素数である 5 で初期化されます。もし配列 a の宣言を、

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

と変更すると、変数 na は 6 で初期化されます。

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。**

まとめ

配列の宣言時は、要素数を定数式で与えなければならない。通常、宣言以外の箇所でも要素数を知る必要があるので、以下のように宣言すればよい。

(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身につけてください！

■ 課題 1-1

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 1-2

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。

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 1-3

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、当てさせる数の範囲に基づいて、プレーヤが入力できる最大の回数を計算して設定すること。

■ 課題 1-4

List 1-10 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` には符号を付けないように変更せよ。

■ 課題 1-5

List 1-10 のプログラムの `do` 文を `for` 文を用いて書き直せ。

■ 課題 1-6

おみくじプログラムを作成せよ。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。