第１章

数当てゲーム

本章で作成するのは《数当てゲーム》のプログラムです。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
- 等価演算子と関係演算子
- 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：試作版）*/

#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ましょう。

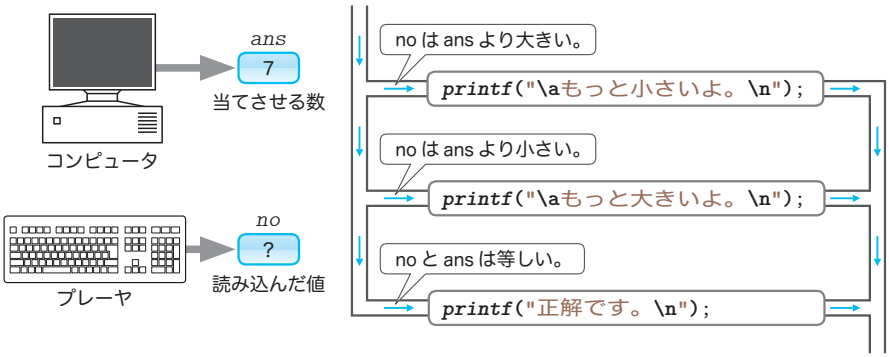


Fig.1-1 if 文によるプログラムの流れの分岐

入れ子になった if 文

二つの変数 *no* と *ans* の値を比較する **if** 文の構造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

制御式

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。

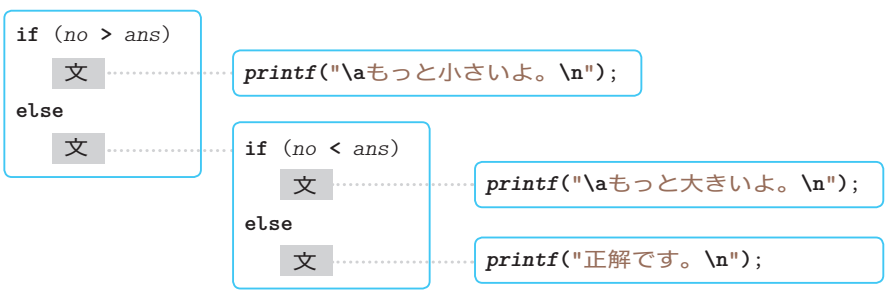


Fig.1-2 入れ子になった if 文

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**すべて**の条件判定が行われます。

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。

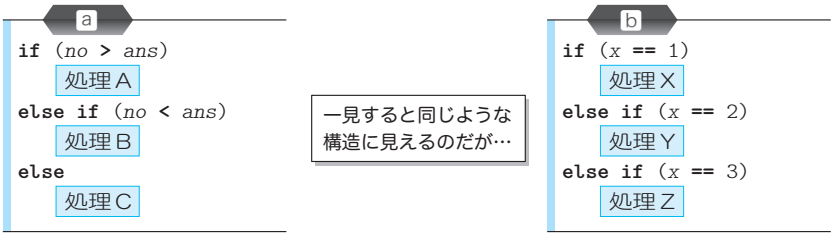
▶ no が ans より大きければ、printf("\aもっと小さいよ。\n"); の実行が完了した段階で、if 文全体の実行を終了するからです。

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

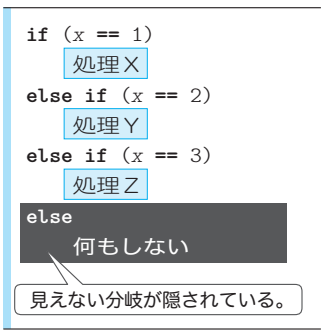
変数 x の値に応じて分岐する if 文です。

〔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。

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 式と評価

▪ 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

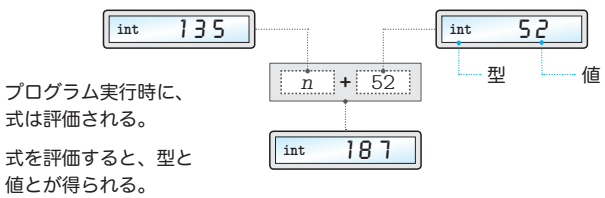
原則として、すべての式には値があります（特別な型である **`void`** 型の式だけは、例外的に値がありません）。その値は、プログラム実行時に調べられます。

式の値を調べることを**評価**（*evaluation*）といいます。**各式が次々と評価されることによって、プログラムが実行されるのです。**

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（p.2）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値（評価結果）の大小関係を判定します（p.9）。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。

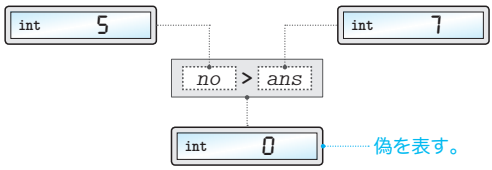


Fig.1C-2 式の評価(int型 > int型)

なお、*no* の値が 7 より大きければ、真を表す“**int** 型の 1”が得られます。

＊

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。

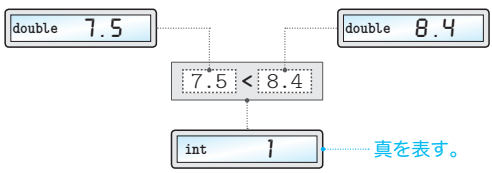


Fig.1C-3 式の評価(double型 < double型)

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。

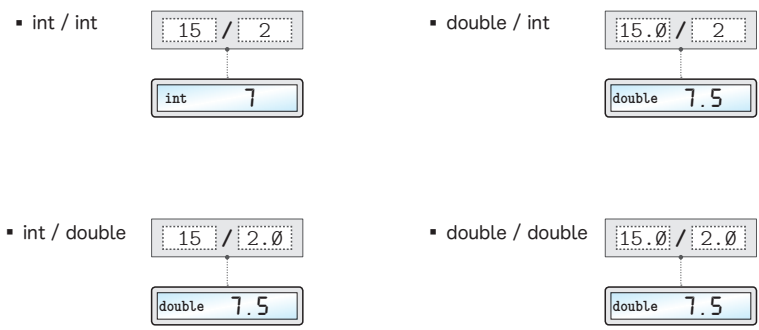


Fig.1C-4 式の評価(int型とdouble型の除算)

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文を利用）*/

#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);

    printf("正解です。\n");

    return 0;
}
```

do文

/* 当たるまで繰り返す */

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）;
```

制御式

ループ本体

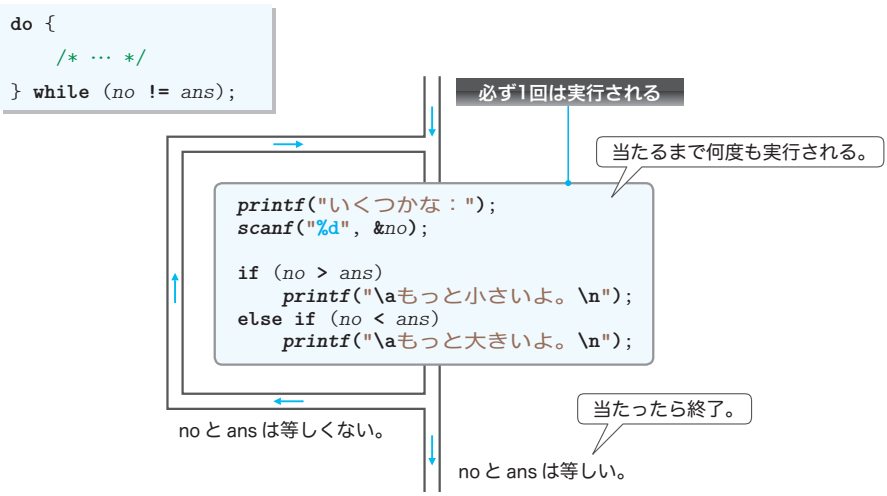
▶ 既に学習した **if** 文や、後で学習する **while** 文や **for** 文などの構文とは異なり、末尾にセミコロン ; が付きます。

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て、{} で囲まれたブロックであるループ本体が再び実行されます。

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/kazuate2while.c

```
/* 数当てゲーム（その２[別解]：当たるまで繰り返す＝while文を利用）*/

#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

```
while（式）文
```

（ループ本体）

（制御式）

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、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 文の制御式が 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。繰返し文を強制的に抜け出すために本プログラムで利用しているのが、break 文です。

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 do 文によって（break 文を使わずに）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 節で詳しく学習します。

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、あらかじめ正解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、その定義の一例です。

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。

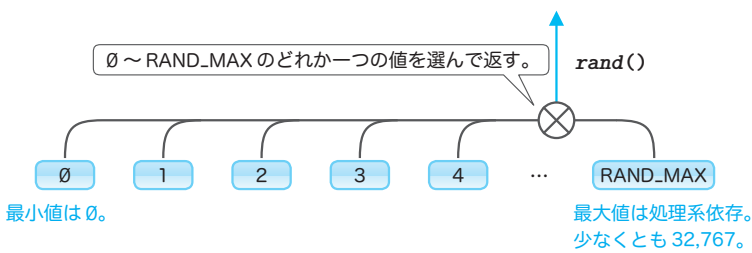


Fig.1-6 rand 関数による乱数の生成

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。

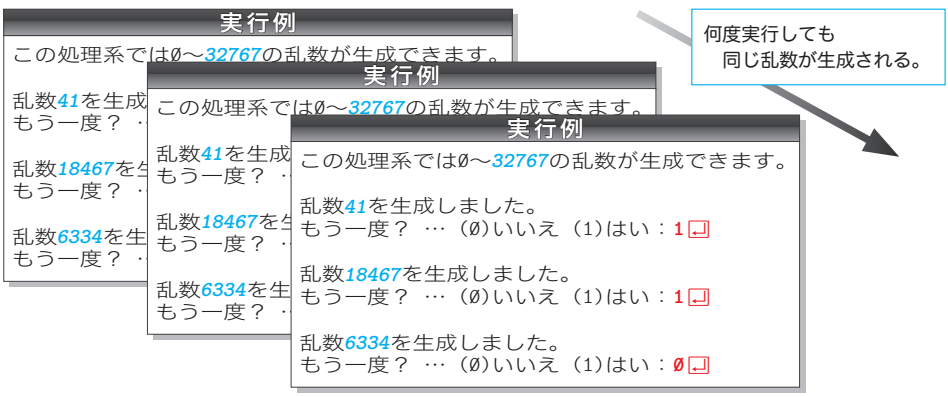
0 〜 RAND_MAXの乱数を生成して返却。

まず最初に、生成できる乱数の“範囲”が表示されます。最小値は 0 であり、最大値は **RAND_MAX** の値（処理系依存の値）です。

その後、***rand()*** が返却した乱数値が表示されます。当然、その値は 0 以上 **RAND_MAX** 以下の値です。

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？**



※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 1-4 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、という仕組みです。

Fig.1-8 に示すのは、ある処理系で生成される乱数系列の例です。

種が 1 のときは、最初の ***rand*** 関数の呼出しでは 41 が生成されて、次の呼出しでは 18,467、その次は 6,334、… と乱数が生成されます。

また、種が 50 であれば、201、20,851、6,334、… が順に生成されます。

種が 1 のとき

41 ⇨ 18,467 ⇨ 6,334 ⇨ 26,500 ⇨ 19,169 ⇨ 15,724 ⇨ …

種が 50 のとき

201 ⇨ 20,851 ⇨ 6,334 ⇨ 29,710 ⇨ 25,954 ⇨ 296 ⇨ …

※ここに示すのは一例であり、生成される値は処理系に依存します。

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。

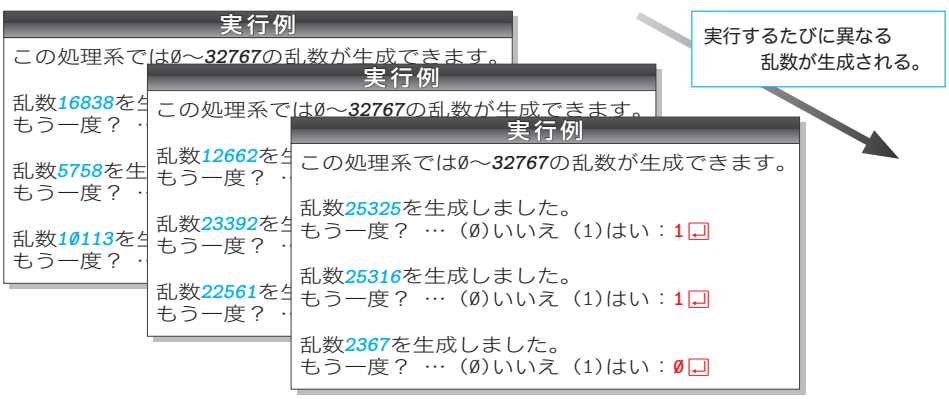
一般的に使われるのが、**“プログラム実行時の時刻を種にする”**という手法です。その手法を利用したプログラムを **List 1-5** に示します。

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設定）*/

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

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

    srand(time(NULL));      /* 現在の時刻に基づいて乱数の種を設定 */

    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（なお、<time.h> ヘッダのインクルードもあわせて必要です）。



※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 1-5 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（あまり）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。

▶ while 文版の **List 1-3**（p.10）ではなく、do 文版の **List 1-2**（p.8）をもとに、わずかな修正と追加を行うだけです。

List 1-6 chap01/kazuate3.c

```
/* 数当てゲーム（その３：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));       /* 乱数の種を設定 */
    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：499
♪もっと大きいよ。
いくつかな：749
♪もっと小さいよ。
いくつかな：624
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 499 を入力し、それより大きいか／小さいかによって 749 あるいは 249 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 */
```

＊

試作版の kazuate1.c に始まって、kazuate2.c、kazuate3.c と、２～３行程度のわずかな追加と修正を繰り返すだけで、数当てゲームが完成しました。

まとめ

✲ 乱数生成の準備（種の設定）

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));                  /* 乱数の種を設定 */
```

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（少なくとも 1 回行えばよく、何度も行わなくともよい）。

なお、この準備を行わない場合、種の値は 1 となり、同じ系列の乱数が生成される。

✲ 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。RAN_MAX の値は処理系に依存する 32,767 以上の値である。

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/kazuate4.c

```
/* 数当てゲーム（その４：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 */
    const 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));          /* 乱数の種を設定 */
    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ⓐ）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。

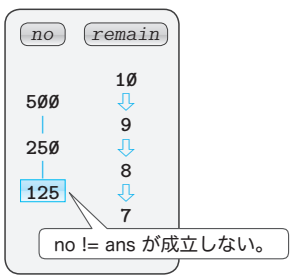
なお、何回目の入力で当たったのかは、*max_stage* から *remain* を引くことによって得られます。たとえば、図aに示す例では、ゲーム終了時の *remain* の値は 7 です。そのため、*max_stage* **-** *remain* すなわち 10 **-** 7 によって 3 が得られます。

▶ *max_stage* の宣言には **const** が指定されていますので、*max_stage* の値は変更不能となります。そのため、仮に *remain*-- とすべきところを *max_stage*-- と書き間違えたとしても、コンパイルエラーが発生します（ミスを防げるわけです）。

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。
```

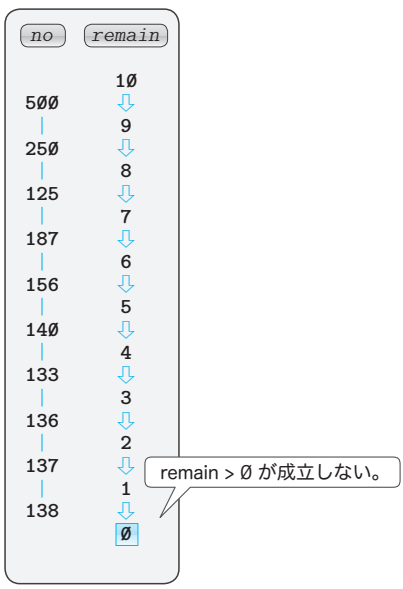


Fig.1-10 List 1-7 の実行例