第 5 章

単純な繰り返し処理

駐車場には、何台も車が停められて、利用者が来るたびに、同じように精算処理をする必要があります。同じ処理を繰り返すことは、機械やコンピュータの得意な動作です。

プログラムで、状況によって行なうべき動作を切り替える方法は、もうすでに知っています。もう一歩、プログラム上の実行すべき場所をジャンプする方法がわかれば、繰り返し処理が実現できます。

この章では、ある条件を満たす限り同じような処理を繰り返す構文について説明します。同じメッセージを何度も表示したり、規則的に変わっていくメッセージを表示したり、ということが簡単にできるようになります。

キーワード

- for ループ・while ループ
- ループ変数
- インデント
- 個数・合計・漸化式

5.1　while 文による繰り返し

条件を満たす限り、繰り返し実行する処理は、while 文で次のように書けます。

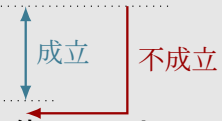
```
/* 文法 */
while (条件式) {
    文;
    ...
}
```

```
/* 実例 */
while (x < 10) {
    printf("%d\n", x);
    x++;
}
```

if とよくて似ていて、while の後に、条件式と、実行するブロック { } を書きます。if では条件式が成立したときにブロックを 1 度だけ実行しましたが、while では条件式が成立する限り、ブロックを何度でも繰り返して実行します。繰り返し実行することを「**ループ** (loop) する」といいます。

ソースコード 5.1　while で 2 の累乗を列挙

```
1  #include <stdio.h>
2
3  int main(void) {
4      int i = 1;
5      while (i < 100) {
6          printf("%d ", i);
7          i = i * 2;
8      }
9      printf("\nループ後のiの値は%dです\n", i);
10     return 0;
11 }
```



while の簡単な例（ソースコード 5.1）で 2 の累乗を表示してみましょう。

ソースコード 5.1 の実行結果

```
1 2 4 8 16 32 64
ループ後のiの値は128です
```

5 行目の while の条件が成立すれば、6–8 行目を実行した上で、また 5 行目に戻り、条件を検査します。条件が不成立ならば 9 行目にジャンプします。変数 i の値に着目してみましょう。4 行目で 1 に初期化され、7 行目を実行するたびに 2 倍になります。while の条件は 100 より小さいことですから、6–8 行目を何度か実行した後にループを抜け出して、9 行目に達するのは i が 128 になったときです。抜け出すときには、もう 6 行目の表示が行われていないことに注意してください。

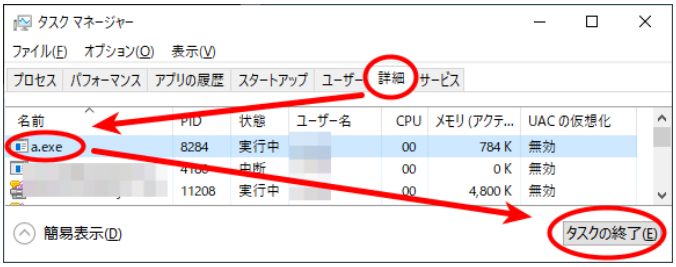


図 5.1 Windows のタスクの終了方法

次に「x を越える最小の自乗数」を求める関数を作ってみましょう。ソースコード 5.2 のようになります。min_square() は、引数 x を受け取り、$k = 1, 2, 3, \ldots$ について k^2 と x の値を比較して、x を越えたところでループを終了して、最後に k^2 を返します。

ソースコード 5.2 x を越える最小の自乗数（while 版）

```
1 int min_square(int x) {
2     int k = 1;
3     while (k*k <= x) {
4         k++;
5     }
6     return k*k;
7 }
```

コラム： プログラムの強制終了

ループ処理は、ちょっとしたミスで終わらなくなることがよくあります。この手の**無限ループ**を実行してしまうと、プログラムを強制的に停止させる必要があります。どのような操作をするかは OS や環境に依存するのですが、Cygwin や Linux のターミナルでは Ctrl+C （Ctrl を押したまま C を押す）です。Visual C++ のような統合開発環境では、停止のためのボタンが GUI 上に用意されています。

それでも停止してくれないプログラム（プロセス）には、最後の手段があります。Cygwin を含む Windows なら**タスクマネージャ**（図 5.1）、Unix 系では **`kill` コマンドや `top` コマンド**の使い方を調べてみてください。

頻出ミス

Cygwin のターミナルで Ctrl+Z を押すと、プログラムは終了したように見えても休止しているだけで、再開を待っています。Windows では休止している**実行ファイルを上書きできない**ので、gcc がおかしなエラー（Device or resource busy）を出すようになります。`fg` コマンドでプログラムを再開して、Ctrl+C で停止しなおすか、図 5.1 のようにタスクマネージャで a.exe を終了しましょう。

5.1.1　ループ変数

プログラムでは、同じ処理を**10 回繰り返す**という場面がよくあります。これを while で実現すると、右のようになります。

```
int i = 0;
while (i < 10) {
    処理;
    i++;
}
```

変数 i の働きに着目してください。初期値は 0 で、**処理を 1 回するごとに 1 増える**という、規則的な動きをします。このため、i が 10 になれば、目的の処理を 10 回実行したと判断できるので、i が 10 以上でループ処理をやめる（＝ i が 10 未満でループする）ことで 10 回ループが実現できます。このように、変数 i は**ループの回数を制御する重要な役目**を担っているので、このループの**ループ変数** (loop variable) と呼ばれます。

5.2　for 文による繰り返し

「同じ処理を 10 回繰り返す」ために、もう一つの手段があります。for 文です。右のように**ループ変数が 3 回まとまって出現**して、実用上はこちらがよく使われます。

```
for (int i=0; i<10; i++) {
    処理;
}
```

for 文の文法は、以下の通りです。while で実現した場合と対比してみます。for 文の () の内側の 3 つの式は、いずれも空欄でも構いません。空欄にしても 2 個の;（セミコロン）は残したままにします。

```
for (初期化式; 条件式; 増分式) {
    文;
    ...

}
```

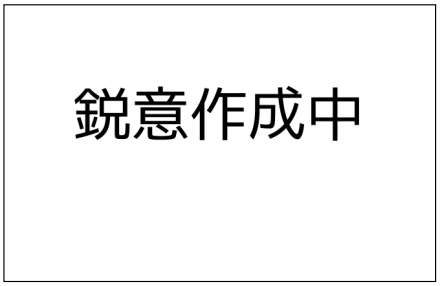
```
初期化式;
while (条件式) {
    文;
    ...
    増分式;
}
```

5.2.1　for と while の使い分け

for と while は、上のような対応関係をみると相互に書き換えができそうです。対応するものがなければ空欄にしてもよいのですから[*1]。では、なぜ同じような働きをする構文が 2 つもあるのでしょうか。

for では、**ループ変数の出現する式がまとまっている**という特徴があります。1 ヶ所を見るだけでループの動きがすぐにわかります。そして同じループ変数が 3 回出現するこ

[*1] 1 点だけ例外があって、while の条件式は空欄にはできません。このことは**??**節に影響があります。



とで、間違いのないループであることを**視覚的**にとらえることができます。逆に言えば、「`for (i=0; j<10; k++)` はおかしなループ」だと、反射的に判断できるわけです。

while には、増分式をループ処理の最後に書くことになります。処理部分が長くなると、ループ変数の登場する 3 つの式が離れてしまうので、書き忘たり変数名を間違えたりしがちです。ですから**回数の決まっているループでは for** が好まれます。

while は長い条件式を書くのに適しています。あるいは、ループ変数が明確に決まってないとか、ループ回数が事前に決まっていないような時によく使われます。

プログラマがループ処理を書く時に、最初に考えることは「**ループ変数を何にするか**」です。特別の理由がなければ `i`, `j`, `k` といった短い変数名が使われます。`l`(エル)は `1`(いち)と見間違えるので避けましょう。

コラム： インクリメントの順序

前置の「`++x;`」と後置の「`x++;`」は、単独なら 1 加えるという、同じ働きをします。式の中に現れると、その「値」がインクリメントの後か前か、どちらになるかが違います。

```
// 増加が前
y = ++x;
  ↕
++x;
y = x;
```

```
// 増加が後
y = x++;
  ↕
y = x;
x++;
```

しかし C 言語には、評価順序に動作保証のない場面も多くあります。2 項演算子の 2 つのオペランドや、関数の引数[a]などがそうです。このような場面では処理系依存になるので、単純な使い方にとどめておくのがよいでしょう。

```
int x = 0;                          /* 処理系依存の例 */
int y = (++x) - (++x);      // × 1-2(=-1) または 2-1(=1)
printf("%d:%d", ++x, ++x); // × "3:4"    または "4:3"
```

[a] 関数の引数の区切りのコンマは、コンマ演算子（順次評価）ではありません。

5.3　ループ処理の実例

ここでは、典型的なループの使用例を見ていきましょう。

5.3.1　偶数を表示

1 から n の整数の中から、偶数のみを表示してみましょう。ソースコード 5.3 に 3 つのパターンを用意してみました。

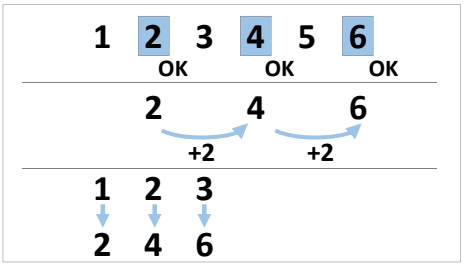
ソースコード 5.3　偶数を表示する

```
1  void print_even(int n) {          // (A)
2      for (int i=1; i<=n; i++) { // 通常のループ
3          if (i % 2 == 0) {      // 偶数を選び出す
4              printf("%d ", i);
5          }
6      }
7  }
8
9  void print_even2(int n) {         // (B)
10     for (int i=2; i<=n; i+=2) { // 2ずつ増やす変則ループ
11         printf("%d ", i);
12     }
13 }
14
15 void print_even3(int n) {          // (C)
16     for (int i=1; 2*i<=n; i++) { // 2*i で偶数を作り出す
17         printf("%d ", 2*i);      // 表示も 2*i
18     }
19 }
```

(A) print_even() では、n 回ループの中で、**条件分岐**で偶数を選び出しています。

(B) print_even2() では、少し変則的なループで実現しています。ループ変数を 2 から始めて、2 ずつ増やすことで、**偶数だけを生成**しています。n が奇数でも偶数でも、どちらでも正しく動きます。

(C) print_even3() では、**計算式で偶数**を作り出すことにしました。i が整数なら 2*i は偶数です。ですから i の代わりに 2*i と書けば偶数になります。2*i と書くところは、ループの条件と、表示の 2 ヶ所あります。
今はこの式が単純なのでよいですが、複雑な式だと、複数の場所に同じ式を書くと間違えそうになるので、多用するのは考えものです。



頻出ミス

下の print_even_bad() のような処理はすすめられません。この for ループは、1 から (n-1) までの (n-1) 回のループをしているように見えますが、ループ処理中で**ループ変数の i を変更**して偶数を作り出しています。つまりループ回数は (n-1) ではありません。

```
void print_even_bad(int n) {      // （非推奨）
    for (int i=1; i<=n-1; i++) { // 通常のループのように見えて
        if (i % 2 == 1) {         // 奇数なら
            i++;                  // 1増やす(!)
        }
        printf("%d ", i);
    }
}
```

これでは、for ループの「ループ変数がまとまって現れて、どのようにループするかが一目で分かる」という特徴が台無しです。

ループ変数を変更するなどという不可解な動作は、くれぐれも避けましょう。

コラム： 手書きのメモ

プログラムの動きが複雑になってくると、いくらパソコンの画面上でソースコードと睨(にら)めっこしていても埒(らち)が明きません。

紙の上にメモを書いて、最初は具体的な値で、変数の変化の様子を追いかけたり、プログラムの制御の具合を書いてみたりして、徐々に法則性を見出しましょう。この作業はとても重要です。

5.3.2　個数を数える

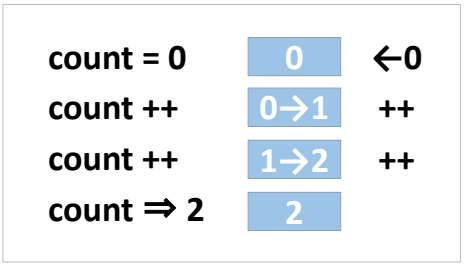
整数 n の**約数の個数**を数えてみます。n の約数とは、1 から n の間の整数のうち、n を割り切るものですから、1 から始まる n 回のループの中で、それぞれ割った余りを計算することで約数かどうかがわかります。約数を見つけるたびに 1, 2, 3, ... とカウントアップしていきましょう[*2]。ループ回数が明確に決まっているので for を用いることにして、ループ変数は i にします。そして、**個数を数えるための変数**が別に必要になります。

ソースコード 5.4　約数の個数

```
1  #include <stdio.h>
2
3  int n_divisor(int n) {
4      int count = 0;                // 個数を数える変数
5      for (int i=1; i<=n; i++) {    // 総当たりする
6          if (n % i == 0) {         // 割り切るなら
7              count = count + 1;    // 個数を1増やす（count++と同じ）
8          }
9      }
10     return count;                 // 数えた個数を返す
11 }
12
13 void print_n_divisor(int n) {
14     printf("%dの約数は%d個あります\n", n, n_divisor(n));
15 }
16
17 int main(void) {
18     print_n_divisor(7);  // 2個 (1,7)
19     print_n_divisor(15); // 4個 (1,3,5,15)
20     return 0;
21 }
```

ソースコード 5.4 では、個数を記憶するのに変数 count を用意して（4 行目）、最初はまだ約数を見つけていないので **0 で初期化**しておきます。5–9 行目のループでは、n を割り切る数値 i を**見つけるたびに**（6 行目）、count を**1 増やします**（7 行目）。ループが終わると（10 行目）、すべての約数を見つけているはずなので、その個数を返します。

[*2] 素因数分解 $n = 2^a 3^b 5^c \ldots$ から $(a+1)(b+1)(c+1)\ldots$ 個と計算する方法もありますが、ここでは愚直に総当たりで数えることにします。



ソースコード 5.4 の実行結果

```
7の約数は2個あります
15の約数は4個あります
```

実行結果は、7 は素数で、約数は 1 と 7 だけですから、正しく 2 個と求まっています。15 は 1, 3, 5, 15 が約数で、4 個です。

関数名 n_divisor() の接頭辞 **n_** は、ソースコード上では **number of** の意味でよく使われます。全体で「divisor の個数」を表していると、プログラマなら誰でもわかります[*3]。

頻出ミス

ソースコード 5.4 の 4 行目の count 変数の初期化（ここでは 0 の代入）を忘れると、とんでもないおかしな値になります。運が悪い (?) と正常に動作することもありますが、何度も関数を呼び出しているとそのうちおかしくなります。未然に防ぐために、コンパイラに警告オプションをつけるなどの工夫をしましょう。

複雑な処理になると、変数を**使う直前に初期化**することも工夫のひとつです。忘れていないことを簡単に確認できるようにするためです。

コラム： インデント（字下げ）

ソースコード 5.4 のような入り組んだ処理になってくると、インデントの重要性が増してきます。5 行目の for のブロックの終わりが 9 行目だということがすぐにわかるように、6–8 行目は行の先頭に 4 文字ほどスペースを多めに入れて字下げしておきます。6 行目の if のブロックの終わりも 8 行目ですから、その間はさらに 4 文字余分に空けます。このような整形は機械的に行えるので、テキストエディタの編集支援機能を使いこなしましょう。

[*3] 英文法的には number of + 名詞の複数形とすべきところですが、プログラム上は n_+ 単数形もよく見かけます。

5.3.3　合計を計算する

次に、個数を数える代わりに、**約数の合計**を計算してみます。

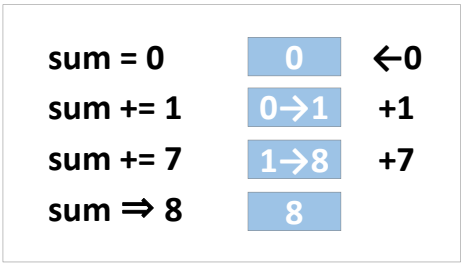
ソースコード 5.5　約数の和

```
1  #include <stdio.h>
2
3  int sum_divisor(int n) {
4      int sum = 0;
5      for (int i=1; i<=n; i++) {
6          if (n % i == 0) {
7              sum = sum + i;
8              printf("i=%d, sum=%d\n", i, sum); // デバッグライト
9          }
10     }
11     return sum;
12 }
13
14 void print_sum_divisor(int n) {
15     printf("%dの約数の和は%dです\n", n, sum_divisor(n));
16 }
17
18 int main(void) {
19     print_sum_divisor(7);  // 8  (1,7)
20     print_sum_divisor(15); // 24 (1,3,5,15)
21     return 0;
22 }
```

ソースコード 5.5 では、ソースコード 5.4 とループの構造は同じですが、count の代わりに、**合計値を累積**するための変数 sum を用意します（4 行目）。約数を見つける前なので、やはり **0 で初期化**します。約数がみつかると、sum を**約数の値だけ増やします**（7 行目）。

ソースコード 5.5 の実行結果

```
i=1, sum=1
i=7, sum=8
7の約数の和は8です
i=1, sum=1
i=3, sum=4
i=5, sum=9
i=15, sum=24
15の約数の和は24です
```



実行結果は、「和は 24 です」だけでは正しいかどうかがわかりにくいので、8 行目で途中の変数の値も表示させてみました。約数が見つかるたびに、i が**見つけた約数**、sum が**それまでの和**として表示されます。これを見ると、sum は突然得られるわけではなく、ループ処理の中で徐々に計算が進むことがわかります。この変化の過程をよく理解しておきましょう。

コラム： デバッグライト

ソースコード 5.5 の 8 行目のような変数表示のコードは**デバッグライト** (debug write) と呼ばれます。条件が成立したり、変数の変化するタイミングなどで表示させて、プログラムが思い通りに動作していることを確かめる（あるいは、どちらかというと、思い通りに動作しないときに原因を探る[a]）ためのものです。

プログラムが完成してしまえば無用にも思えるのですが、次に改造することがあれば役立ちますので、消してしまわずに、**コメントにして残して**おきます。

表示形式には「**関数名: 変数名=値, 変数名=値**」のようなパターンが好まれます。行頭の関数名は、通常の出力と違うことの目印でもあります。いちいち関数名を書いていると間違えそうですが、C99 からは、現在の関数名を表す `__func__` という特殊な変数が用意されたので、以下のように簡便に書けるようになりました。

```
printf("%s: i=%d, j=%d\n", __func__, i, j);
```

[a] このようなデバッグ手法は**プリントデバッグ** (print debug) と呼ばれます。

MEMO: `__FILE__` `__LINE__`はマクロ

5.3.4　平方根で 3 乗根を求める（漸化式）

電卓の中にはルート（平方根・2 乗根・自乗根）の計算のできるものがあります。しかし 3 乗根まで計算できるものは、まず見つかりません。そのような電卓でも 3 乗根の近似値を計算する方法がありますので、プログラムで実現してみましょう。

正の実数 x に対して、次の漸化式で与えられる数列 $\{a_n\}$ を考えてみます。

$$a_1 = 1, \quad a_n = \sqrt{\sqrt{x \cdot a_{n-1}}} \qquad (n = 2, 3, 4, \ldots)$$

このとき、a_∞ は $\sqrt[3]{x}$ に収束して、しかも収束が早いので a_4 くらいでも良い近似値になります。a_4 の計算は、電卓にルートとメモリ機能があれば簡単な操作で行えます。

この計算を C 言語に翻訳してみます。ソースコード 5.6 に 3 通りの関数を作りました。ルート演算には <math.h> の sqrt() を使います。

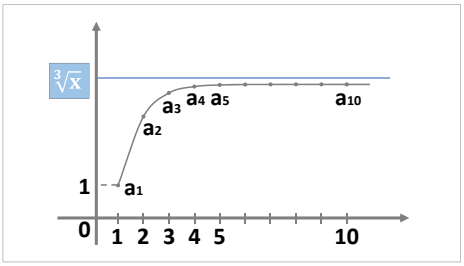
(A) cbrt4() では、電卓での a_4 の計算をそっくり再現しました。
ところで、変数 a3 や a4 の役目をよくよく考えると、a4 を求めるのに必要なのは 1 つ前の a3 の項のみで、それより前は不要です。それならば、異なる変数を用意する必要はなく、すべて同じ 1 つの変数を使いまわしできそうです。

(B) cbrt10() では、数列に同一の変数を使いまわせることに気づいたので、ループを使いました。数列の次の項を求めるところが、ループ処理の 1 回分に対応します。繰り返しはプログラムの得意とするところですから、a_4 と言わずに a_{10} を求めてみましょう。このように、**回数の決まったループには for** が適しています。

(C) cbrt_eps() では、ループのもう一つの考え方を採用しました。今、3 乗根を求めようとしているのですから、理想的な値が手に入れば、その値の 3 乗が元の x と等しくなります。始めのうちは誤差がありますが、計算を続ければより正しい 3 乗根に近づきます[*4]。それならば、許容できる x との誤差をあらかじめ設定しておき、誤差がそれ以上の間は計算を繰り返す、という作戦も考えられます。この場合のような、**条件によるループには while** が適しています[*5]。23 行目の while の条件式の fabs() は絶対値を返す数学関数で、やはり <math.h> で定義されています。20 行目で許容誤差をあらわす EPS というマクロを定義しています。暫定解 a の 3 乗と x の差の絶対値が許容誤差を超える間、つまり $|a^3 - x| > \mathrm{EPS}$ の間にループします。

実行結果には、<math.h> の本物の 3 乗根関数 cbrt() の値も表示させてみました。8.0 の 3 乗根を求めているので、正確には 2.0 ですが、いずれも良い近似値が得られていることがわかります。

[*4] この漸化式はうまく収束してくれますが、どんな漸化式でもある値に収束するわけではありません。
[*5] 許容誤差を小さくしすぎると、その精度が達成できずに無限ループになる可能性があるので、実用的にはループ回数に上限を設けます。



ソースコード 5.6　3 乗根を近似的に求める

```
#include <stdio.h>
#include <math.h>
                    // 以下は電卓での操作（[MR]はメモリ呼び出し）
double cbrt4(double x) { // (A)      // xをメモリに記憶
    double a1 = 1.0;                // [1]
    double a2 = sqrt(sqrt(x * a1)); // [×][MR][=][√][√]
    double a3 = sqrt(sqrt(x * a2)); // [×][MR][=][√][√]
    double a4 = sqrt(sqrt(x * a3)); // [×][MR][=][√][√]
    return a4;
} // よく考えるとa1,a2,a3,a4は同じ変数でもよい

double cbrt10(double x) { // (B)     // xをメモリに記憶
    double a = 1.0; // 初項a1       // [1]
    for (int i=2; i<=10; i++) {
        a = sqrt(sqrt(x * a));      // [×][MR][=][√][√]
    }
    return a;
}

#define EPS 0.001   // 許容誤差
double cbrt_eps(double x) { // (C)
    double a = 1.0;
    while (fabs(a*a*a - x) > EPS) {  // aの3乗とxの差の絶対値を比較
        a = sqrt(sqrt(x * a));
    }
    return a;
}

int main(void) {
    double x = 8.0;
    printf("cbrt=%f, cbrt4=%f, cbrt10=%f, cbrt_eps=%f\n",
            cbrt(x), cbrt4(x), cbrt10(x), cbrt_eps(x));
    return 0;
}
```

ソースコード 5.6 の実行結果

```
cbrt=2.000000, cbrt4=1.978456, cbrt10=1.999995, cbrt_eps=1.999979
```

コラム： ノイマン型アーキテクチャあるいはプログラム内蔵方式

現在のようなコンピュータ言語の登場する前には、歯車による機械式計算機や、真空管やリレーによる計算機がありました。一種の電卓のようなもので、ボタンやレバーうまく操作することで望みの計算を実現していました。このコンピュータ（＋人間）の動作は、ソースコード 5.6 の `cbrt4()` と似ているように思えます。

その後、ノイマン型アーキテクチャとよばれる計算機が登場しました。プログラム内蔵方式とも呼ばれ、人間の行っていた操作をデータのようにメモリに記憶しておきます。そして状況次第で動作を切り替えるという、条件分岐に相当する機能も実現されました。これが大きな進歩で、`cbrt_eps()` のような動作ができるようになりました。

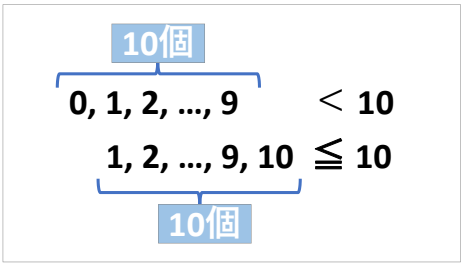
現在のほとんどのプログラミング言語は、この系譜に則っています。「プログラム上に現在実行中のところがあって、だんだんと進んでいく」という考え方は、ここからきています。

コラム： 0 オリジン vs. 1 オリジン

0 から数え始めることを、日本人のプログラマはよく **0 オリジン**といいますが、英語圏では zero origin よりも zero-based のほうが通じやすいようです。1 始まりの **1 オリジン** (one-based) も同様です。

日常生活では、1 番目 2 番目と、1 オリジンで数えることが多いですが、C 言語では配列（☞**??**章）を 0 から数え始めるため、0 オリジンがよく出てきますし、「何番目」かを計算するにはこちらが向いています。

0 オリジンに慣れてしまえば、まったく難しくないのですが、日常生活の 1 オリジンとギャップがあるため、この違いを埋め合わせるのに苦労するのは、プログラマの宿命でしょう。



5.4 良いループと悪いループ

10 回ループの書き方は何通りもあるのですが、良いものと悪いものがあります。

```
for (i=0; i<10;  i++) { /* 10回 */ } // ○
for (i=0; i<=9;  i++) { /* 10回 */ } // ×
for (i=1; i<=10; i++) { /* 10回 */ } // ○
for (i=1; i<11;  i++) { /* 10回 */ } // ×
```

×のついているものがなぜ悪いのか、一目瞭然ですね。確かに 10 回のループではありますが、ソースコード上には**「10」という数字が現れていません**から、これでは読み手に誤解を与えます。

では、「10」という数字をソースコード上に素直に書くためには、どのようなループを書けばよいでしょうか。2 通りのパターンがあります。

- (0 ~ 9)　ループ変数を 0 から始めて、条件は <（イコールなし）
- (1 ~ 10) ループ変数を 1 から始めて、条件は <=

いま仮に、本当に (n+1) 回のループをさせたい場面があったとしましょう。それならばソースコードには (n+1) と表記すべきであって、n に = をつけたり取ったりして調節すると誤解を招くということです。(n+1) 回ループの例を挙げます。

```
for (i=0; i<n+1;  i++) { /* (n+1)回 */ } // ○
for (i=0; i<=n;   i++) { /* (n+1)回 */ } // ×
for (i=1; i<=n+1; i++) { /* (n+1)回 */ } // ○
```

0 から始める（0（ゼロ）オリジン）と、1 から始める（1（いち）オリジン）のどちらにするかは、場面によって判断します。このあとに出てくる配列を扱うなら 0 オリジンが向いていますし、日常生活の数を扱うなら 1 オリジンが簡単です。

5.5 練習問題

1. **[ループの実行順序（☞5.1 節）]**
   2 ページのソースコード 5.1 の 6 行目と 7 行目を入れ替えると、実行結果はどのように変化するか。**TODO:** インデントのみを直す課題
2. **[条件を満たすものを表示する（☞5.3.1 項）]**
   376 は特別な数である。2 乗した値 $376^2 = 141376$ の下 3 桁が、元の数に一致する。このような 3 桁の整数を、総当たりですべて求めて表示せよ。
3. **[条件を満たすものを表示する（☞5.3.1 項）]**
   1 から 50 までの整数のうち、3 の倍数と、一の位あるいは十の位に 3 の現れるものを、昇順に「3 6 9 12 13 15 ... 30 31 ...」のように表示せよ。（☞**??**章の練習問題 **??**.）同じ数が何度も表示されないよう、条件を工夫せよ。（☞**??**節）
4. **[合計を計算する（☞5.3.3 項）]**
   次の関数を作れ。いずれもループで積算せよ。
   - `int sum(int n)` は n 以下の正の整数の合計を返す。
   - `int sum_odd(int n)` は n 以下の正の奇数の合計を返す。
5. **[デバッグライト（☞11 ページのコラム、5.3.4 項）]**
   13 ページのソースコード 5.6 の cbrt10() と cbrt_eps() にデバッグライトを追加して、変数の変化を観察してみよ。cbrt_eps() のループ回数も確かめてみよ。
6. **[ループ処理中で自作関数を呼び出す]**
   約数の合計と、その数自身が一致するものを完全数という。ただし、約数の合計からはその数自身を除くものとする。10 ページのソースコード 5.5 の sum_divisor() を利用（または改造）して、10000 以下の完全数（4 個ある）をすべて求めよ。

   ——— 完全数の最初の 2 個は 6, 28 である。5 個目を求めるには、かなりの工夫が必要である。現在のところ 50 個ほどだけが知られている。———
7. **[数列の和・double へのキャスト]**
   S_n $(n = 0, 1, 2, \ldots)$ を以下で定義する。

$$S_n = \sum_{k=0}^{n} \frac{(-1)^k}{2k+1} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots$$

   $n = 0, 1, 2, \ldots, 100$ について、$4 \cdot S_n$ を表示せよ。なお、ライプニッツの公式によって $4 \cdot S_\infty$ は π（円周率）に収束すると知られている。

   ——— $(-1)^k$ の計算には、累乗関数を用いるまでもない。k の偶数奇数で $+1$ と -1 を使い分けてもよい。符号を記憶する変数 `sign = 1` を用意して、ループするごとに `sign = -sign;` と符号を反転させてもよい。———

索引