第 1 章

発展的な話題

1.1 条件演算子

if で分岐しても、いずれにしても同じ変数に代入する場面があります。このような場面では、「条件 ? 真値 : 偽値」の**条件演算子** (conditional operator) で簡潔に表現できます。

```
// if による分岐
if (cond) {     // 条件
  var = value1; // 真値
} else {
  var = value2; // 偽値
}
```

```
// 条件演算子
var = cond ? value1 : value2;

// 使用例（大きい値をとる）
max = (a > b) ? a : b;
```

3 つの項からなる演算子は、C 言語ではこれが唯一であるので、**三項演算子** (ternary operator) とも呼ばれます。var（代入される変数）の部分が長い場面で有効です。value1 と value2 の型は揃えておきましょう[*1]。

次のように、多分岐の場合も簡潔に表現できます。

```
// else if による多分岐
if (cond1) {
  var = value1;
} else if (cond2) {
  var = value2;
} else {
  var = value3;
}
```

```
// 条件演算子による多分岐
var = cond1 ? value1 :
      cond2 ? value2 :
              value3;
```

条件演算子は演算子ですので、式の途中に現れてもよいのですが、式が長くなって読みにくくなるので、変数に代入する場面くらいにとどめておくのがよいでしょう。

TODO: 順次評価演算子 (sequential-evaluation operator)?

[*1] 2 つの値の型が異なれば、複雑な格上げ規則が適用されます。

1.2 データを列挙する条件多分岐

C 言語では、else if で多分岐（☞??項）が行えますが、整数型の値で分岐する場合には、switch 文でも同じような分岐が行えます。

```
/* 文法 */
switch (整数式) {
    case 値1:
        処理1; // 省略可
        break; // 省略可
    case 値2:
        処理2; // 省略可
        break; // 省略可
    ...
    default:   // 省略可
        処理;  // 省略可
}
```

```
/* 実例 */
switch (a) {
    case 1:
       printf("aは1です\n");
       break;
    case 2: case 3: // 複数列挙
       printf("aは2か3です\n");
       // break がなければ
       // 以下を続けて実行
    default:
       printf("aは1以外です\n");
}
```

- switch 文の式の値に該当する case があれば、その処理が実行されます。
- いずれの case の値にも該当しないときには、default が実行されます。
- いずれの処理も、break も、default も省略可能です。処理がなければ何もしませんし、break がなければ次の case の処理を続けて実行します。

このように、柔軟な文法にも見えますが、逆に注意があります。

- default の綴りを間違えても、文法エラーにならず[*2]、実行されないだけです。
- break を書き忘れがちです。
- 整数型専用です。char は使えますが、浮動小数点数も、文字列（つまり char 配列）も使えません。整数であっても、case に 1〜1000 のような範囲は指定できません。

このように、あまり使い勝手がよいとは言えません[*3]。元々は、アセンブリコードに翻訳した時に、ジャンプテーブルを参照することが想定されていたのでしょう。最適化技術の進んだ昨今のコンパイラでは、else if を羅列しても効率性に大差はないでしょう。

[*2] goto 文のジャンプ先ラベルと見なされます。

[*3] Java 言語では文字列で分岐できるよう、機能拡張が行われました。オブジェクト指向言語では、型で分岐できるものもあります。

1.3 擬似乱数

擬似乱数 (pseudo-random number) を生成する関数が <stdlib.h> にあります。再現可能であるため、正確には擬似乱数ですが（☞??ページのコラム）、以下では簡単に「乱数」と表記します。

```
#include <stdlib.h>
定数
  RAND_MAX // 32767以上
関数
  int rand(void);
  void srand(unsigned int seed);
```

1.3.1 乱数の加工

rand() は、0 以上 RAND_MAX 以下の一様乱数を返します。**関数を呼び出すたびに新しい値が得られます**。RAND_MAX $\geq 2^{15} - 1$ (= 32767) であることが言語規格で保証されています。欲しい乱数の範囲や種類に合わせて、以下のように加工して使います。

0 以上 1 未満 (double)

```
double frand(void) {
    return 1.0 / ((double)RAND_MAX + 1.0) * rand();
}
```

1未満になるよう、rand() を RAND_MAX + 1 で割りたいのですが、整数演算ではオーバーフローの危険があるので、RAND_MAX を double に変換してから 1 加えています。この範囲の乱数は、さらに加工するのに都合のよいものです[*4]。

0 以上 6 未満 (int)

```
int a1 = rand() % 6;            // 下位ビット重視
int a2 = (int)(frand() * 6);    // 上位ビット重視
```

a1 のように、6 の剰余をとると、目標の範囲に入ります。ただし、かつての乱数は、特に下位ビットのランダム性が悪く、偶数と奇数が交互に発生するものまであったため、上位ビットを生かす方法が良いとされていました。そのため a2 では、まず frand() を 6 倍して、0 以上 6 未満の浮動小数点数を得ます。そして int にキャストして小数部分を切り捨てます。これで目的の範囲の整数が得られます[*5]。

1 以上 6 以下 (int)

```
int b1 = 1 + (rand() % 6);          // x + (rand() % (y-x+1))
int b2 = 1 + (int)(frand() * 6);    // x + (int)(frand() * (y-x+1))
```

a1 や a2 にを 1 加えます。これで**サイコロの目**に相当するものが作れました。範囲を x 以上 y 以下にするなら、1 を「x」，6 を「$y - x + 1$」と読み替えます。

[*4] Java なら Math.random() が同じ範囲の乱数を（より高精度に）生成します。
[*5] もし frand() の範囲に 1 が含まれていれば、非常に低い確率で a2 が 6 になることに注意してください。

1.3.2　乱数系列の指定

乱数を生成する際に、内部的に用いる値を**乱数の種** (seed) と呼びます。次の乱数のための種を指定するのが srand() です。同じ種を srand() で指定すれば、その後の rand() で得られる乱数系列（生成される一連の値）が同じになります。右の例は GCC 7.5 (glibc-2.27) の場合です。srand() で指定しないときは、種は 1 が使われます。

```
srand(314159); // 314159を種にする
int a = rand(); // → 414777680
int b = rand(); // → 2009630532
int c = rand(); // → 102799611
srand(314159); // 同じ種
int d = rand(); // → 414777680
int e = rand(); // → 2009630532
int f = rand(); // → 102799611
// rand()の具体的な値は処理系依存
```

- srand() による種の指定は、通常は **main() で 1 度だけ**行なえば十分です。
- <time.h> の time(NULL) を種にすると、典型的には 1 秒ごと[*6]に異なる乱数系列になります。（☞**??**ページのソースコード**??**）
- 乱数の生成方法は言語規格では規定されていないので、発生する値は処理系に依存します。実行環境によらず同じ値が必要なら、自前で生成ルーチン[*7]を用意します。

[*6] time() で得られる現在時刻が、1 秒単位であることが多いためです。（言語規格上は処理系依存です。）
[*7] 松本 眞氏の Mersenne Twister がおすすめです。

1.4 時刻

<time.h> に時刻を扱う関数があります。現在時刻と、プログラムの消費したプロセッサ時間の、2 通りの時刻を扱います。

プロセッサ時間（CPU タイム）は、入出力待ち（例えばキー入力待ち）の間は止まります。逆にマルチスレッドで動作すれば、実際の経過時間よりも早く進みます。

```
#include <time.h>

定数
  CLOCKS_PER_SEC
  TIME_UTC
```

```
型
  clock_t
  time_t
  struct tm
  struct timespec
```

```
関数
  clock_t clock(void);
  time_t time(time_t *timer);
  int timespec_get(struct timespec *ts, int base);

  double difftime(time_t time1, time_t time0);

  time_t mktime(struct tm *timeptr);
  struct tm *gmtime(const time_t *timer);
  struct tm *localtime(const time_t *timer);
```

clock() プログラムの消費したプロセッサ時間を返します。戻り値は clock_t 型で、CLOCKS_PER_SEC で割ると秒単位の値になります。プログラムの実行開始後に **0 から始まるとは限らない**ので、プログラムの 2 ヶ所での差を取る必要があります。現実の環境では精度が 0.01 秒ぐらいと（コンピュータの動作速度からすれば）粗いことも多く、0 秒と測定されることもよくあります。

```
    clock_t start = clock();
    routine_taking_long_long_time(); // 時間のかかる処理
    clock_t end = clock();
    printf("所要時間は %g 秒です。\n",
           (double)(end - start) / CLOCKS_PER_SEC);
```

time() 現在時刻を time_t 型で返します。ポインタ型の引数にも（同じ値を）書き込みます。ただし NULL を指定すれば書き込みません。戻り値か引数か、どちらか片方を使用すれば十分です。

```
time_t t = time(NULL); // 戻り値
  または
time_t t;  time(&t);   // 引数
```

time_t は、典型的に long（32 bit または 64 bit）が割り当てられ、Unix 系の時刻と同じく、協定世界時（UTC）の 1970 年 1 月 1 日 0 時 0 分 0 秒からの経過秒数を格納することが多いですが、言語規格上は、型も起点も精度も、何も規定されていません。現実には、1 秒単位であることを前提にしたプログラムもよく見かけます。

difftime() 引数で受け取った 2 つの時刻の差を、double 型の秒数で返します。時刻は time_t 型なので、この関数の内部で double 型への変換が行われることになります。

```
time_t start = time(NULL);
routine_taking_long_long_time(); // 時間のかかる処理
time_t end = time(NULL);
printf("経過時間は %g 秒です。\n", difftime(end, start));
```

mktime() 構造体 struct tm のポインタを引数で受け取り、格納されている時刻を time_t 型に変換して返します。struct tm には右のメンバーが規定されていますが、ここでは tm_wday と tm_yday は利用されません。
次の localtime() 関数と、ちょうど逆の動作をします。

```
// struct tm のメンバ
int tm_sec;  // 秒 [0, 60]
int tm_min;  // 分 [0, 59]
int tm_hour; // 時 [0, 23]
int tm_mday; // 日 [1, 31]
int tm_mon;  // 月 [0, 11]
int tm_year; // 西暦年-1900
int tm_wday; // 曜日 [0, 6]
int tm_yday; // 通算日数 [0,365]
int tm_isdst;// 夏時間フラグ
```

gmtime(), localtime() time_t 型の時刻をポインタ引数で受け取り、構造体 struct tm に変換して、ポインタで返します。struct tm の実体は、固定的な領域に割り当てることが許されています。つまり、繰り返し呼び出すと上書きされる可能性があるので、すぐにローカル変数にコピーしてから用います。gmtime() は協定世界時、localtime() は現地時刻に変換します。

```
time_t t1 = time(NULL);
struct tm s1 = *localtime(&t1);  // ポインタの指す実体をコピー
printf("今日は%d年%d月%d日です。\n",
       1900 + s1.tm_year, 1 + s1.tm_mon, s1.tm_mday);
```

tm_year は、西暦年数から 1900 を引いた値です。
tm_mon は 0 オリジンで数えます。（☞**??**項）
tm_wday は、0 が日曜日を示します。1 が月曜日です。
tm_yday は、1 月 1 日を 0 として、経過日数を数えたものです。
tm_sec は、通常は 59 までです。60 は閏秒のために用意されています。

timespec_get() C11 で追加された、time() の高精度版です。新しい関数だけあって、関数名が「引数の構造体名 + 動詞」と、現代的な命名規則（☞??節）に則っています。

```
struct timespec ts;
timespec_get(&ts, TIME_UTC)
double sec = ts.tv_nsec * 1e-9;
```

第 1 引数の timespec 構造体の変数には 2 つのメンバが規定されています。time_t 型の tv_sec には time() に相当[*8]するものが、long 型の tv_nsec には秒未満がナノ秒単位で格納されます。tv_nsec に 10^{-9} を掛けると秒単位の小数部分になります。

第 2 引数には、定数 TIME_UTC[*9]を指定します。

この関数をサポートしている環境はまだ少なく、また（言語規定による）精度の保証もありませんが、今後は利用価値が高まるものと思われます。

1.4.1 実行時間の測定

プログラム全体の実行時間を測るのであれば、Unix 系の time コマンドも有用で簡便です。`time ./a.out` とすれば、a.out の実行にかかったプロセッサ時間などが表示されます。time コマンドは、シェルの種類によっては内部コマンドにもあるので、外部コマンドを明示するには `/usr/bin/time ./a.out` とフルパスで指定します。外部コマンドのほうが、表示内容が詳細です。

[*8] 言語仕様上は、time() と型は同じですが、値まで同一であるとは定められていません。起点や精度が異なる可能性を考慮したものと思われます。

[*9] C11 では、これ以外の定数は用意されていません。C23 では種類が増えて、時間の測り方を指示できるようになる見込みです。

1.5 再帰呼び出し

二項係数とは、$(x+1)^n$ を展開した式の x^k の係数のことで、${}_nC_k$ あるいは $\binom{n}{k}$ と表記します。これは、n 個のものから k 個を選び出す、組み合せの数でもあります。${}_nC_k = \frac{n!}{k!(n-k)!}$ という、数学的には美しい公式が有名ですが、階乗を用いるため、通常の整数演算には不向きです[*10]。

ソースコード**??**の実行結果

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
    1   6  15  20  15   6   1
```

二項係数は、平面に並べるとパスカルの三角形になります。この図からわかるように、次の漸化式が成り立ちます。初期条件は ${}_nC_0 = {}_nC_n = 1 \quad (n \geq 0)$ です。

$$ {}_nC_k = {}_{n-1}C_{k-1} + {}_{n-1}C_k \quad (1 \leq k \leq n-1) $$

式の両辺に二項係数が現れていて、**自己再帰的**に定義されているとも解釈できます。これをプログラムにしてみます。

ソースコード 1.1　再帰呼び出しによる二項係数

```
1 int choice(int n, int k) {
2     if (k == 0 || k == n) return 1;           // 末端処理（初期条件）
3     if (k < 0 || k > n  || n < 0) return 0;   // エラー処理
4     return choice(n-1, k-1) + choice(n-1, k); // 再帰呼出し
5 }
```

このように、ある関数の処理中に、同じ関数（自分自身）を呼び出すことを**再帰呼び出し**といいます。漸化式の定義通りの式を、素直に計算しているだけに見えますが、言語や条件によっては再帰呼出しができないため（☞**??**ページのコラム）、通常の（つまり自分以外の）関数を呼び出すこととは区別します。

再帰呼び出しは、プログラミング言語上は必要不可欠な機能ではなく、**ループ処理に置き換えが可能**だとの証明がありますが、表記が簡潔になる利便性もあって、状況によっては好んで使われます。ただし次のように、扱いの難しい面もあります。

- 再帰の末端の場合分け処理を、確実に行う必要があります。漸化式で言えば、初期条件を反映する部分です。これ以上再帰呼び出しをせずに、呼び出し元に戻るための処理ですから、この処理がないと**無限ループと同じ**ことになり、更にスタック領域（☞**??**項）を食いつぶして、原因のわかりにくい動作不良（強制終了）を起こします。

[*10] 階乗計算を正直に行うと、すぐにオーバーフローします。64 bit 整数でも $n = 20$ 程度が限界です。公式通りではなく、筆算で行う ${}_{25}C_2 = \frac{25 \cdot 24}{1 \cdot 2} = 300$ の計算のように、先に約分する工夫が有用です。

- 呼び出しの深さが、深くなりすぎない場面で使います。
- 変数によって、呼び出しごとに異なるものにするのか、引き継いだ共通なものにするのかを見極めます。前者なら関数内のローカル変数にします。後者ならグローバル変数や、関数に渡す場合でも参照渡しにします。

例えば、1 から n までの和を求める場面で、$a_0 = 0,\ a_n = n + a_{n-1}\quad (n = 1, 2, \ldots)$ の数列を考えて再帰呼び出しをするのは考えものです。関数の呼び出しと、終了して呼び出し元に戻るのを繰り返すのは問題ありませんが、続けて呼び出すとスタック領域のメモリが開放されません。たとえ無限ループでなくても、スタック領域を使い尽くすとプログラムが強制終了させられます。

ソースコード 1.2　再帰呼び出しによる和

```
1  #include <stdio.h>
2
3  int sum(int n) {
4      int sum = 0;
5      for (int i=1; i<=n; i++) { sum += i; }
6      return sum;
7  }
8
9  int sum_recursive(int n) {
10     if (n == 0) { return 0; }
11     return n + sum_recursive(n - 1);
12 }
13
14 int main(void) {
15     for (int i=10; i<=1000000; i*=10) {
16         printf("sum(%d) = %d\n", i, sum(i));
17         printf("sum_recursive(%d) = %d\n", i, sum_recursive(i));
18     }
19 }
```

索引