付録 A

さらなる成長に向けて

ここでは、C 言語に限らず、コンピュータ言語に精通するための手法や技術、コンピュータ上の道具の使い方のヒントなどを列挙してみます。

作ったプログラムが思い通りに動作すれば、それでよいのでしょうか。確かに最初の目標としてはよいのですが、会社のチームでプログラムを共同で開発したり、あるいは個人の活動でもオープンソースソフトウェアとして公開すると、他人からの改良提案をもらったりして、一般的なコーディングスタイルに合わせていく必要が生じます。「動く」だけのプログラムではなく、「メンテナンスしやすい」「共同開発しやすい」プログラムの作法も、徐々に身につけてもらえたらよいでしょう。

後半では、C 言語のテクニカルな性質を説明します。理解できると有用なのですが、幅広い知識が必要なため、資料として参照してもらえば十分です。

コラム： クマさんに相談

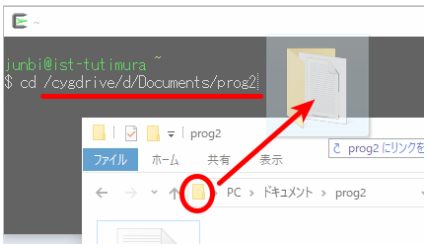
思うようにプログラムが動かず、原因がわからなくて頭を抱えているときには、誰かに相談したくなります。ある研究室では、部屋の片隅にクマのぬいぐるみが置いてあって、まずクマさんに相談することになっています。それでも解決しなければ、ようやく研究室のメンバーに相談します。

変なルールのようですが、クマさんに（？）説明を始めると、おかしなところに気づいて自己解決することが頻繁にあるのです。クマさん相手ではなくても、一人で声に出してしゃべっているうちに、自己解決した覚えのある人も多いでしょう。状況を最初から順序立てて説明すると、それだけで解決策が見えてきます。

A.1　プログラミング環境の上手な操作

以下のような操作は、作業効率を向上させます。少し意識して使い始めると、自然と使えるようになるので、試してみてください。

- ウィンドウ操作
  - ウィンドウを整然と並べる（見たいところが同時に見えるように）
  - フォーカスウィンドウの切り替え：Alt+Tab
- シェルやターミナルの操作
  - コマンド履歴の呼び出し：↑ ↓
  - 入力を補完：（入力の途中で）Tab
  - フルパス名を入力：アイコンのドラッグ



- テキストエディタの基本操作（☞??ページのコラム）
  - カット：Ctrl+X
  - コピー：Ctrl+C または Ctrl+Insert
  - ペースト：Ctrl+V または Shift+Insert
  - 行頭・行末ジャンプ：Home End
  - キーによる選択：Shift+↓ Shift+→ など
  - マウスによる選択：ダブルクリック（単語単位）、トリプルクリック（行単位）
  - 文字列検索、置換
- テキストエディタの効率的な使い方
  - 手入力を避ける（動作実績のあるコードをコピーする、補完入力する）
  - 対にして先に入力する

    (i) " . . . "　(ii) { . . . }　(iii) (. . .)　(iv) /* . . . */
  - 整形（インデント自動調整など）や予約語のハイライトなどの支援機能

A.2 効率のよいデバッグ術

- コンパイル時のエラーや警告は、最初を見る（☞**??**ページのコラム）
- コンパイル時の警告を増やしておく（☞**??**ページの**??**節）
- デバッグライトを活用する（☞**??**ページのコラム）
- ソースコードのバックアップを作って再現性を確保する
- 挙動の違いの原因を確かめるためには、ソースコードの修正を一ヶ所に留める
- `<assert.h>` の `assert()` 診断マクロを利用する **TODO:** 説明する?

コラム： 動作不良の再現性

時々受ける質問に、「自分の作っていたプログラムが動作不良を起こしていたが、何とか解消できた。しかし、なぜだったのか気になる。」というのがあります。

様子を聞くと「ここを直したらエラーが解消した」というのですが、状況的にはそれでは解消するはずがないので、さらに聞くと「別のところも修正した」と。恐らく、そちらで解消したと思えるので、実演しようとすると、「もう再現できない」。このような、歯がゆい思いをする状況が、ままあります。

原因を探ろうというのは、向上心があって良いことです。もう一歩進めて、バックアップをとるなりして、エラーを再現できると、さらに上達するでしょう。

A.3 コーディング上の良い習慣

折に触れて言及してきたことですが、あらためて列挙してみます。

- スペーシング（☞**??**ページの**??**項、**??**ページのコラム）
  (i) インデント (ii) for の ; のあとのスペース (iii) コンマの後のスペースや改行
- スコープは狭く（☞**??**ページのコラム、**??**ページの**??**項）
- 変数の初期化は使う直前（☞**??**ページの頻出ミス）
- キャストの使用は最小限に（☞**??**ページのコラム）
- 変数名・関数名の命名規則（☞**??**ページの**??**節）
- マジックナンバーを排除（☞**??**ページの**??**節）
- 見慣れた処理を組み合わせる（☞**??**ページの頻出ミス、**??**ページの**??**節）
- コメントでは見慣れない動作の理由を説明する（☞**??**ページのコラム）

MEMO: 嘘のコメントには害がある **MEMO:** 明日の自分は他人

A.4　プログラムの上達へむけて

同じ処理は 2 度書かない　同じ処理を 2 度書く（コピーする）と、修正のあった場合に、もう片方の修正を忘れがちです。

「車輪の再発明」を避ける　標準ライブラリの機能で事足りるなら、なるべくそれを使いましょう。例えば日付計算など、誰もが必要になりそうなものは、既に作られているものを探すべきです。自作して信頼性で追いつくのは至難の技です。

関数設計のヒント：モデルとビューは分離する　**??**節のように、**計算**（モデル）だけの関数と、**表示**（ビュー）だけの関数に分離しましょう。関数が再利用しやすくなります。一体にした関数だと、表示形式を変えるのに計算部分をコピーした別の関数を作ることになって、上記の「同じ処理は 2 度書かない」を破ることにもなります。

コード・リーディング　身近な人とプログラムを読み合いましょう。技術の共有に役立ちます。とても一人では思いつかないコードを身につける、あるいは避けるべきパターンを覚える、良い機会になります。

処理系依存と可搬性（ポータビリティ）　言語規格で保証された動作であるか、あるいは処理系依存でたまたまその動作になっているかは、区別せねばなりません。保証がないと、異なる環境での動作がいつおかしくなるかわかりません。目の前のコンパイラ環境の動作から言語規格は推し量れませんので、最終的には仕様書で確かめるしかありません。経験の積み重ねも重要です。

A.5　値の変化の頻度に応じた扱い

状況に応じて変化する値と、本当に固定化されて変化しない値があります。どんな値でも変化するという前提にすると、プログラムが複雑になりすぎます。逆に変化しないと決めてしまうと、ちょっとした状況変化でプログラムが役立たなくなります。

右には、変化の度合いに応じたプログラム上の実現方法を挙げました。再コンパイルすれば変えられる、実行しなおせば変えられる、実行中にも変化するなど、いくつもの手段があります。扱う値の性質を見抜いて、適切な手段を選びましょう。

↑ 変化の頻度が少ない

- ハードコーディング
- マクロ定数，const グローバル変数
- コマンドライン引数
- キー入力（標準入力）
- 関数の const 引数, const ローカル変数
- グローバル変数
- 関数の引数, ローカル変数
- ループ変数

↓ 変化の頻度が多い

A.6 分割コンパイル

A.7　開発のための技術・ツール

プログラムの規模が大きくなると、以下のような手法やツールが重要になってきます。Unix 系でよく使われるコマンド名も挙げておきます。

テスト 単体テストは**??**ページのコラムで紹介しました。本来はテストを自動化します。境界値分析（☞**??**ページのコラム）など、テストの方針も提唱されています。

ビルドツール 複数のソースファイルを、並列にコンパイルしたり、再コンパイルに不要なものを見分けたりして、短時間で実行ファイルを生成するツールです。例: make

デバッガ プログラムを、変数の値などを確認しながら、逐次実行するツールです。他のツールと組み合わせることもある、開発基盤です。例: gdb

メモリリーク検出 配列あふれなど、メモリの不正アクセスを検出するツールです。方式がいくつもあり、万能のものは存在しません。例: Electric Fence, valgrind

プロファイラ プログラムの実行時に、関数の呼び出された回数などを記録しておいて、後から集計するツールです。プログラムの高速化に役立ちます。例: gprof

静的解析 文法チェッカなど、ソースコードを検査するツールがあります。コンパイラに詳細な警告を出力させるのも、その第一歩です。例: lint

テキスト差分 2 つのテキストファイルの違う部分のみを抽出するツールがあります。ソースコードはもちろん、実行結果の違いを検出するのにも有用です。例: diff

バージョン管理 ソースコード（に限らず、テキストファイル）の世代管理を行なうツールがあります。差分を表示したり、複数人で共同開発する際の集中管理的な役目もあります。プログラマに必須です。例: git, subversion, cvs

ドキュメント生成 ソースコード中に、特定形式で書いておいたコメントから、HTML などの形式でプログラム全体のドキュメントを生成するツールがあります。Java 言語なら標準の javadoc、C 言語にはオープンソースの doxygen が有名です。

A.8 浮動小数点型の性質

浮動小数点型は複雑な性質を持っています。内部表現の指数部と仮数(かすう)部が固定長です。double には、図 A.1 のように 2 進数表記で、符号に 1 ビット、指数部に 11 ビット、仮数部に 52 ビットを割り当てるのが典型的です。

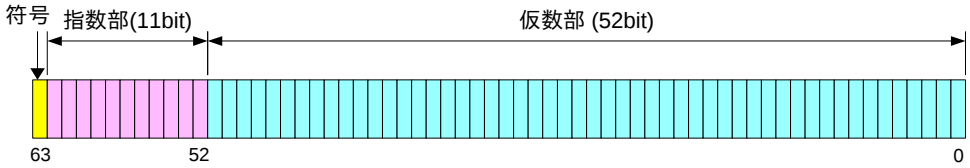


図 A.1 IEEE 754 による double のビット割り当て

このように精度が限られるため、数値誤差が常に付きまといます。誤差のない数学モデルからは想像もつかない動きの一端を紹介します。

A.8.1 定数の誤差・丸め誤差

浮動小数点が 2 進数で表記されていると、1/2 (=0.5) や 1/8 (=0.125) は正確に表せても、我々になじみのある **0.1 には定数としての誤差**が含まれます。循環小数を途中で打ち切っているためです。そして、例えばソースコード A.1 は 0.1 を 10 回足していますが、1.0 になりません。演算ごとに**最下位ビットの丸め誤差**も加わるためです。

ソースコード A.1 0.1 を 10 回足した結果を表示する

```
1  #include <stdio.h>
2
3  int main(void) {
4      double x = 0.1;
5      x = x + 0.1;  x = x + 0.1;  x = x + 0.1;
6      x = x + 0.1;  x = x + 0.1;  x = x + 0.1;
7      x = x + 0.1;  x = x + 0.1;  x = x + 0.1;
8      printf("%.17f =(?) %.17f\n", 1.0, x);
9      return 0;
10 }
```

ソースコード A.1 の実行例（GCC Intel 64 ビットの場合）

```
1.00000000000000000 =(?) 0.99999999999999989
```

このため、比較の際には「微小な誤差を等しいとみなす」処理を追加する必要があります。どの程度の誤差を許容すればよいかは、計算内容によって大きく左右されるため、統一的な手段は確立されていません。

ソースコード A.2 では、等しいとみなす相対誤差をマクロで定義して「`if (x <= y)`」の代わりに「`if (fcmp(x,y) <= 0)`」で大小比較できるようにしてみました。

ソースコード A.2　浮動小数点の誤差を見込んだ比較

```
#include <math.h>  // fabs(), fmax()
#include <float.h> // DBL_EPSILON

/* 等しいとみなす相対誤差 */
#define RELATIVE_ERROR (DBL_EPSILON * 8)

/* xとyがほぼ等しければ0，x>yなら正，x<yなら負を返す */
int fcmp(double x, double y) {
    double diff = x - y;
    double max  = fmax(fabs(x), fabs(y));        // 絶対値の大きい方
    if (fabs(diff) < max * RELATIVE_ERROR) return 0; // 許容誤差以下
    if (diff > 0) { return +1; } else { return -1; }
}
```

A.8.2　情報落ち・桁落ち

有効精度が 10 進数 3 桁の状況を考えてみましょう。精度の悪くなる演算があります。

情報落ち 絶対値が極端に違う値の和や差からは、小さい方の値の情報が抜け落ちます。

例: $1.00 + 0.01\underline{23} \rightarrow 1.01$

桁落ち 値の近い数の差は、有効精度が少なくなります。

例: $1.00 - 0.999 \rightarrow 0.001$ （有効精度 1 桁）

演算の順序を工夫することで、精度の悪化を防げる場合もあります。

A.8.3　無限大・非数

浮動小数点数には、特別な値があります。

非数 (NaN, Not a Number) `0.0 / 0.0`（数学的に不定）や、負数の平方根（実数では計算できない）などが非数になります。printf() で「nan」のように表示されます。この値が紛れ込むと通常の四則演算ができなくなり、演算結果にも非数が伝搬します。

無限大 (infinity) 0 の割り算などで無限大になります。符号があって、`5.0 / 0.0` だと「`inf`」、`-5.0 / 0.0` だと「`-inf`」のように表示されます。

マイナスゼロ ゼロにも符号があります。`1.0 / inf` だと「`0.00`」、`1.0 / -inf` だと「`-0.00`」のように使い分けます。

A.8.4 定数

浮動小数点型には、??ページの表 ??以外にも多数の定数があります。その一部を紹介します。

機械エプシロン (machine epsilon) 「1 より大きい最小の数」と 1 との差です。仮数部の精度が反映されます。double の定数は DBL_EPSILON です。

正の最小の正規化数 (normalized number) 浮動小数点数は、仮数部が 1 以上かつ基数未満になるよう、指数部で調節して正規化しますが、限界もあります。DBL_MIN は double の限界値を表す定数です。これより絶対値が小さいと正規化できず、仮数部が 1 未満の**非正規化数** (subnormal number) になって、精度が落ちます。

丸め方向 浮動小数点型に共通の FLT_ROUNDS[*1] が現在の丸め方向を、次の値で示します。

(0) 0 方向 (1) **偶数丸め** (to nearest, ties to even) (2) 切り上げ (3) 切り捨て
(4) **四捨五入** (to nearest, ties away from zero) (-1) 不確定

(1) と (4) は、どちらも「最も近い値」に丸めるのは同じです。等距離の値があるときに、(1) 偶数を選ぶか、(4) 原点から遠い方を選ぶか、が違います。C23 より前は、どちらも「最も近い値」として (1) に分類されていました[*2]。

コラム： 浮動小数点演算の速度

浮動小数点演算の速度は、CPU の持つ演算回路によって大きく左右されます。

1990 年代より前、浮動小数点演算は CPU に専用の回路がなく、整数演算よりも数十〜数百倍以上も遅いものでした。しかも double は float の何倍も時間がかりました。整数演算でソフトウェアエミュレーションしていたからです。

2000 年より少し前から double の演算回路が当たり前になりました。整数演算との速度差が数倍にまで縮まり、しかも float より double のほうが少し早いという**逆転現象**まで起こりました。まだ float 用の回路はなく、double に変換して計算していたので、変換のオーバーヘッドがあったのでした。

マルチコアが当たり前の今では float 用の回路もできていて、逆転現象こそ解消しているものの、float と double の速度差はそれほどありません。ちなみに整数の short と int も同じような関係で、short に速度メリットはほとんどありません。

[*1] C89 までは定数でした。C99 からは fesetround() で変更できる変数に変わったはずですが、以前の実装のままになっている環境も見受けられます。
[*2] 最も近い値として、偶数丸めの実装が一般的であったので、影響は少ないでしょう。

A.9　入出力インタフェース

プログラムが利用者からデータを受け取る手段は、いくつかあります。もちろんファイルをオープンすれば、いくらでもデータを受け取れますが、そこまでしなくても、簡単にすます方法があります。ファイルの出力にも、簡便な方法があります。

A.9.1　コマンドライン引数

プログラムを実行するときに、ターミナルでコマンド名（a.exe など）に続けて書いた文字列を**コマンドライン引数** (command line argument) といいます。スペース文字が区切りとなって、第 1 引数, 第 2 引数, ... と分解されます。プログラムで受け取るためには、`main()` 関数の引数を void とせず、次のようにします。

```
int main(int argc, char *argv[]) { ... }
```

プログラム開始時には、コマンドライン引数（に加えてコマンド名）が argv[] に格納され、その個数が argc でわかります。

コマンドライン引数は文字列で得られるので、int で扱うためには `<stdlib.h>` の関数を利用して atoi(argv[1]) と変換します。double なら atof(argv[1]) です。

```
./a.exe 123 4.56
```

↓

```
argc = 3;
argv[0] = "./a.exe"; // コマンド名
argv[1] = "123";     // 第1引数
argv[2] = "4.56";    // 第2引数
```

コマンドライン引数を付け忘れたときの対策として、argc が必要数ではない時に**使用法を表示してプログラムを中断**するのが習慣です。配列の添字溢れを防ぐためです。

ソースコード A.3　コマンドライン引数を受け取る

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[]) {
5      if (argc != 3) {        // 引数が2個以外なら
6          printf("使用法: %s 整数 小数\n", argv[0]); // コマンド名
7          exit(EXIT_FAILURE); // 中断
8      }
9      int    a = atoi(argv[1]); // 第1引数
10     double b = atof(argv[2]); // 第2引数
11     printf("a = %d, b = %f\n", a, b);
12 }
```

A.9.2 キーボード

キーボードからの入力は scanf() で受け取るのが簡便ですが、**問題も多く**あります。右の例では、変数の a と b にキーボードから入力した数値が代入されます。

scanf() は、printf() と似た書式文字列に従って、キーボードから受け取った数値や文字列を、引数で指定されたアドレスに書き込みます。

```
printf("整数を入力してください> ");
int a;
scanf("%d", &a);

printf("小数を入力してください> ");
double b;
scanf("%lf", &b);

printf("a=%d, b=%f\n", a, b);
```

- 例えば %d に対応する引数には、int の値を書き込みます。書式の詳細は☞**??**節
- % ではない、普通の文字に対しては、その文字がキーボードから入力されるまで待ちます。（つまり、それ以外の文字を読み飛ばします。）
- スペース文字や改行文字（Enter）は、どれも同じ（空白文字）と扱われます。

printf() との混同 引数をアドレスで渡すために & が必要です。さらに double の書式には l（エル）を追加して %lf にします。printf() とは、似ているようで違います。

プロンプト表示 プログラムを実行すると、scanf() の部分で入力待ちになって、停止したようにも見えるので、右上の例のように、プロンプト（入力を促すメッセージ）を表示するのがよいでしょう[*3]。

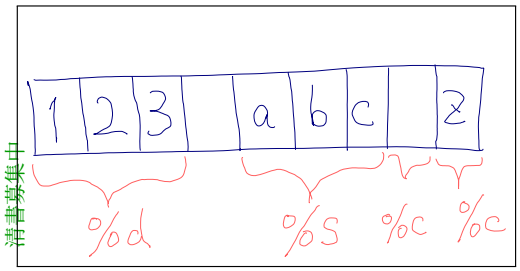
書式違い printf() と scanf() の f は formatted（**書式の定まった**、整形済みの）の意味です。printf() は書式を定めて出力するのでいいのですが、scanf() は**書式の定まった文字列を受け取る**ように作られています。それを人間が手入力する場面に流用しているので、書式が少し違うだけで処理できなくなります。例えば数値を受け取るはずのところでアルファベットが入力されると、何もせずに次に進みます。（しかも**??**節のような再入力処理をしていると無限ループになります。）

書式違いの対策には、まず fgets() でキーボードから Enter を区切りとした文字列を受け取って、その後に atoi() や sscanf() で解析します。これなら受け取りに失敗しても、行単位で再入力を促せます。ソースコード A.4 のように、処理を関数に独立させておくと、後からの差し替えにも好都合です。

[*3] 対話的なのでわかり易い動きと思えるのですが、何度も繰り返すと、メッセージが煩わしくもなります。ファイルにリダイレクトすることが前提なら、プロンプトは標準エラー出力にして、ファイルに残らない工夫をします。定型処理を繰り返すのであれば、コマンドライン引数のほうが向いています。

```
printf("整数を入力してください> ");
int a;
scanf("%d", &a); // "123"→123

printf("文字を入力してください> ");
char c;
scanf("%c", &c); // [Enter]→'\n'
```



受け取り位置の継続 2 つの scanf() で「%d の**数値**」と「%c の**文字**」を連続して受け取るとします。キーボードから123 Enter を入力したとします。

- 最初の %d では、数値部分の123 のみを受けとって、Enter (='\n') を受け取らずに次の scanf() に送ります。
- 続く %c では**新たにキーボード入力を受け付けることなく**、残された'\n' を受け取ってしまいます。しかも悪いことに、この文字を表示しても何も表示されない（改行されるだけ）ので、文字を受け取れてないとの誤解まで招きます。

このような場面でも、Enter 区切りで受け取る fgets() が役立ちます。

ソースコード A.4　キーボードから数値を受け取る

```
1  #include <stdio.h>
2  #include <stdlib.h> // atoi(), atof(), BUFSIZ
3
4  int input_int(void) {
5      char buff[BUFSIZ];
6      if (fgets(buff, BUFSIZ, stdin) == NULL) exit(EXIT_FAILURE);
7      return atoi(buff); // 文字列→int変換
8  }
9
10 double input_double(void) {
11     char buff[BUFSIZ];
12     if (fgets(buff, BUFSIZ, stdin) == NULL) exit(EXIT_FAILURE);
13     return atof(buff); // 文字列→double変換
14 }
15
16 int main(void) {
17     printf("整数を入力してください> ");
18     int a = input_int();
19
20     printf("小数を入力してください> ");
21     double b = input_double();
22
23     printf("a=%d, b=%f\n", a, b);
24 }
```

A.9.3 標準入出力・リダイレクト・フィルタ

UNIX には、名前に uni-（単一の）の接頭辞がついている通り、物事を統一的に扱おうという思想があります。この影響を受けてか、シェルから起動するプログラムには、たいてい**キーボード**と**コンソール画面**を、**ファイル**として統一的に扱う仕組みがあります。

シェル（コマンドインタプリタ）は、プログラム起動時に、表 A.1 の 3 つのファイルを開きます。通常はキーボードやコンソール画面に接続されていますが、本物のファイルに切り替えることもできます。この操作を**リダイレクト** (redirect) といいます。

標準入力 (standard input) ファイルポインタ名は `stdin` です。ですから `scanf("...")` と `fscanf(stdin, "...")` が同じです。普段はキーボードにつながっていますが、あらかじめ入力する内容を`"input.txt"`のようなファイルに記録しておけば、プログラム起動時の不等号でリダイレクトするだけで、キー入力したことになります。

```
$ ./a.exe < input.txt // input.txtにキー入力する内容を保存してから
```

標準出力 (standard output) ファイルポインタ名は `stdout` です。通常は画面に表示されますが、ファイルに保存したければ、やはり（逆向きの）不等号でリダイレクトします。

```
$ ./a.exe > output.txt // 画面出力を output.txt に上書き保存
```

この操作では、同名のファイルが存在していると、上書きして元の内容は消えてしまいます。既存ファイルに追記するなら >> と不等号を二重にします。

標準入力と標準出力の**両方を同時にリダイレクト**できます。

```
$ ./a.exe < input.txt > output.txt // 入出力同時にリダイレクト
```

標準エラー出力 (standard error output) 画面出力のうち、エラーメッセージや、時間のかかる処理の途中経過など、ファイルにリダイレクトされたくないものに使います。

2 つのプログラムで、標準入力と標準出力を直接つなげることもできます。**パイプ** (pipeline) でつなげる、といいます。中間ファイルを生成する必要がなくなります。

```
# 中間ファイルを生成
$ ./a.exe > tmpfile.txt
$ ./b.exe < tmpfile.txt
```

```
# パイプで連結して一括処理
$ ./a.exe | ./b.exe
```

標準入力から受け取ったデータを加工して、標準出力に結果を流すだけの単純なプログラムを**フィルタ** (filter) といいます。パイプはフィルタを組み合わせるのに役立ちます。このような、単機能のプログラムを組み合わせて、複雑な動作を実現することは、UNIX の思想に合致します。**TODO:** 1 入力 1 出力が限界です。

表 A.1　標準入出力の種類

名称	既定の接続先	FILE*	リダイレクト操作
標準入力	キーボード	`stdin`	`command < file`
標準出力	コンソール画面	`stdout`	`command > file`（上書き） `command >> file`（追記）
標準エラー出力	コンソール画面	`stderr`	`command 2> file` など

コラム：キーボードの EOF

キーボードがファイルである以上、終端 (EOF) を示すことができるはずです。ターミナルエミュレータによって、この操作が違います。

Cygwin や Unix 系では、Ctrl+D です。状況によっては Enter の直後に入力する必要があります。

Windows のコマンドプロンプトでは Ctrl+Z です。**TODO:** Unix 系での注意

A.10　関数の引数の可変長

printf() や scanf() 関数では、状況によって**引数の個数や型**が変わります。このような引数を**可変長引数** (variable length arguments) といいます。本書の守備範囲を越えますが、**TODO:** 書くスペースができた自作プログラムでも `<stdarg.h>` の `va_arg()` といったマクロで実現できます。

可変長引数をとる関数では、プロトタイプ宣言による引数の型チェックは望めませんので、呼び出しには細心の注意が必要です。ただし、標準ライブラリ関数の printf() や scanf() については、コンパイラが特別扱いをして、書式文字列を解析して、対応する引数の型チェックを行なう場合があります[*4]。

なお、（単純化のためか）可変長引数に現れる **float は double に格上げされる**ことに決まっているので、printf() では float と double の区別がありません（☞**??**節）。しかし、scanf() はポインタ型で扱うため、float* と double* には大きな違いがあります。

[*4] 伝統的な慣習（標準ライブラリ関数を特別扱いしない）からすると異例ですが、現実的な対処です。

A.11　メモリの割付

プログラム実行開始時には、プログラムがメモリに読み込まれ、作業領域としてスタック領域とよばれるメモリを確保します。この領域の大きさは、シェルの設定によって調節できますが、プログラムの実行中には変化できませんので、使い尽くさないように配慮が必要です。

プログラムの実行時に関数呼び出しを行うと、スタック領域と呼ばれるメモリが消費されます。実引数や、呼び出し元アドレスが保存され、関数内で使うローカル変数が確保されたりします。そして関数から抜け出すときに、確保したメモリが開放されます。

これ以外に、ヒープ領域というメモリもあります。OS に要求して、動的に確保と開放のできる領域です。C 言語からは malloc() や free() といった関数で操作しますが、本書では割愛します。

索引