第 11 章

ファイルの読み書き

ファイルは、コンピュータ上の「データを保存する入れ物」です。テキストエディタで作っている C 言語プログラムもファイルです。写真のような画像も、表計算ソフトの文書もファイルです。

一度プログラムを終了しても、ファイルにデータを保存しておけば、引き続いて動作を続けることができます。外部デバイスに保存するので、パソコンの電源を切っても消えません。駐車スペースごとに、駐車開始と終了の時刻をファイルに保存しておけば、後から統計をとって、1 日の売上高を計算することもできます。

我々の作るプログラムからもファイルが扱えます。扱い方には決まった手続きがあります。

そもそも、たいていの作業には、(1) 前処理 (2) 主処理（繰り返し）(3) 後処理、という流れがあります。ファイルを扱うのにも、それぞれの手続きがありますので、ここで学んでいきましょう。

キーワード

- ファイルポインタ（fp）
- fopen / fclose
- 文字単位・行単位
- NULL, EOF
- 標準入出力

11.1 ファイルとは

ファイルは、データを保存するものです。いくつも作れるので、名前（ファイル名）をつけて区別します。データは 8 ビット（=1 バイト）単位で保存されますので、ファイルの長さは「何バイト」と数えます。長さの上限は、通常は意識する必要はありません。

保存されるデータの形式は、一般に次の 2 通りに大別されます。

テキスト形式 (text format) そのままで画面表示ができて、文字として見ることができます。ASCII 文字のみを含むときは、ASCII 形式ともいいます。人が編集したり、アプリケーション間でデータを交換するのに向いています。（例：ソースコード、HTML、CSV など）

バイナリ形式 (binary format) そのままでは画面表示できず、何らかの変換をせねば意味のわからないものです。コンピュータ向きです。（例：実行ファイル、画像ファイル、圧縮ファイル、アプリケーションごとの独自形式など）

我々の作るソースコードは、もちろんテキスト形式です。テキスト形式には、数字やアルファベットなどの ASCII 文字だけでなく、ひらがなや漢字のようなマルチバイト文字を含めることがあります。マルチバイト文字の表現方法（文字エンコード）は日本語だけでも何通り[*1]もあって、食い違うといわゆる「文字化け」が起こります。

改行コードも OS によって異なり、`\r\n`（Windows）, `\n`（Unix 系）, `\r`（古い Mac）の 3 通りがよく使われています。ただし、C 言語の標準ライブラリが違いを吸収するので、プログラムからは気にしなくてよいことになっています[*2]。

ここでは、テキスト形式を扱いますが、マルチバイト文字には立ち入りません。

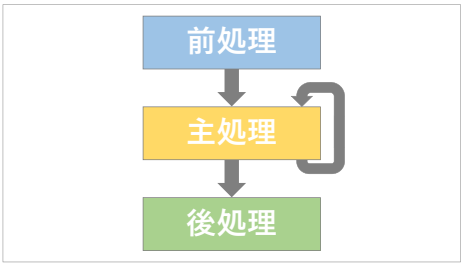
11.1.1 バッファリング

ファイルを保存する HDD や SSD といった外部デバイスは、主記憶メモリ（プログラムから見た変数）に比べると、格段に低速で、しかも一度にまとまった量のデータをやりとりする設計になっています。このため、少ない量のデータを頻繁にやり取りすると、遅さが目立ってしまいます。

そこでファイルを扱う際には、メモリ上に**バッファ** (buffer) という作業領域を作り、普段の操作はこのメモリ上で済ませておき、キリのいいところで外部デバイスと同期するという手法がとられます。この手法を**バッファリング** (buffering) といいます。

[*1] 以前は JIS（ISO-2022-JP）、Shift_JIS、EUC-JP の 3 つがよく使われていましたが、ようやく UTF-8 に統一されつつあります。

[*2] Cygwin は Unix 系の流儀で動作するため、Windows の流儀の`\r\n` が `\n` に変換されないことがあります。



11.2 プログラムからの入出力

C のプログラムでファイルを扱うには、FILE *という型[*3]が便利です。この型の変数には fp のような名前をつけるのが習慣です。**ファイルポインタ** (file pointer) の意味です。

ファイルポインタを扱う関数群があります。いずれの関数名の先頭にも f がついています。FILE の意味でしょう。

作業	使う関数
前処理	fopen()
主処理（繰り返し）	fgets(), fprintf() など
後処理	fclose()

ファイルポインタは、ファイルそのものではありません。オープンしているファイルの**状態**が保存されます。具体的に何が保存されるのかは、プログラマは知る必要はなく、この型を扱う関数群だけを使って操作すればよいことになっています。ファイルポインタの使い方は次の通りです。

```
FILE *fp;  // 構造体へのポインタ（ファイル１つにつき１個必要）

fp = fopen(ファイル名, "r");  // 前処理
....          // 主処理
fclose(fp); // 後処理
```

上のように、fopen() でファイルポインタ fp を手に入れます。これを使って主処理をしてから、fclose(fp) で 1 つのファイルの処理が完了です。

なお、FILE* を扱う関数の内部ではバッファリング（☞11.1.1 項）が行われているので、1 文字単位の操作を繰り返しても高速です。反面、使い方を誤ると書き込み内容を失うことがあるので、頭の隅に覚えておいてください。

[*3] この型の実体は構造体で、しかも必ずポインタで使います。今の習慣なら file_t の名前にするところですが、何か理由があって、すべて大文字の、マクロのような名前になっています。

11.2.1　オープン（前処理）

ファイルのオープンに使う fopen() のプロトタイプの意味は、次のとおりです。

```
#include <stdio.h>

FILE *fopen(const char *path, const char *mode);
```

path　どのファイルを扱いたいかを指定します。単にファイル名だけならば、実行プログラム（a.exe や a.out）と同じフォルダ[*4]のファイルになります。フォルダが異なれば、絶対パスや相対パスを使います[*5]。

mode　ファイルを読み取るのか、書き込むのかの区別をする**オープンモード** (open mode) を指定します。以下の 3 通りがよく使われます。

mode	"r"	"w"	"a"
動作	読み取り (read)	書き込み (write)	追記 (append)
ファイルの存在する必要	あり	なし	なし

戻り値　オープンに成功すると、ファイルポインタを返します。失敗すると NULL という特別な値（表 11.2）を返すので、検出してエラー処理をせねばなりません。

頻出ミス

mode を "w"（書き込み）にすると、既に同名のファイルが存在する場合は、上書きして**元のファイルは消滅**しますので、注意が必要です。

"a"（追記）は、ファイルがあれば末尾に追記するので、消滅の心配はありません。

コラム：ファイルオープン失敗

ファイルのオープンに失敗する原因は、デバイスの破損だけではありません。

読み取り　(1) 指定したファイルが存在しない (2) 存在しても読み取り権限がない

書き込み, 追記　(1) デバイスの容量不足 (2) 指定のフォルダに書き込み権限がない (3) 書き込めないファイルが既に存在している（☞9 ページの頻出ミス）

[*4] 正確には、「プログラムを実行したときのカレントディレクトリ」です。

[*5] Windows のフォルダの区切り文字は「\」と、エスケープ文字と同じですから、プログラム中に記載するなら、エスケープ文字自身を表すのに「\\」と 2 個に増やす必要があります（☞**??**節）。ただし Cygwin では、Unix 系の「/」スタイルに変換して扱うので、気にしなくて大丈夫です。

鋭意作成中

11.2.2 クローズ（後処理）

主処理が終わったら、fclose() でファイルをクローズします。書き込みファイルなら、バッファに残っているデータすべてを書き込みます。プロトタイプは次の通りです。

```
#include <stdio.h>

int fclose(FILE *fp);
```

TODO: プロトタイプ用の囲みデザインを

fopen() したら、必ず fclose() を呼び出します。しかも、同時にオープンできるファイルの数には上限があるので、こまめにオープン・クローズします[*6]。1 つのプログラムで同時にオープンするファイルは、通常なら 2〜3 個で足りるはずです。10 個も同時に必要になったら、プログラムの設計から見直したほうがよいでしょう。

コラム：NULL ポインタ

どのオブジェクトも指さない、特別なアドレスを表す NULL というポインタが用意されています。ポインタを返す関数で、エラーが起こったことを示すような場面でよく使われます（☞11.2.1 項）。値は 0 であると、言語規格で定められています。（ちなみに、ヌル文字とは何の関係もありません。）

fclose(fp) でクローズした後の fp に NULL を代入する習慣があります。2 重にクローズすることを防いだり、クローズ忘れを検査するのに都合がよいためです。

[*6] OS のプロセス保護の不十分だった時代には、不正終了したプログラムのオープンしていたファイルがクローズされずに残ることがあり、繰り返すうちに OS 全体のファイルの同時オープン数を使い果たすという事態が起こりました。今の OS なら保護が手厚くなっていて、少々のことならプログラムの終了と同時にクローズしてくれますが、作法としては、できれば早い段階で、明示的にクローズしましょう。

表 11.1　主処理に使う主な関数

対象	読み取り	書き込み・追記
1 文字 (char)	fgetc()	fputc()
1 行 (string)	fgets()	fputs()
書式つき (format)	fscanf()	fprintf()

11.2.3　読み書き（主処理）

読み書きのどちらかによって、使う関数が変わります。また、文字単位か、行単位か、どちらで処理するのかによっても変わります。まとめると表 11.1 のようになります。関数名の最後の c は char（文字）、s は string（文字列）の意味でしょう。put ↔ get の対義語も使われて、（大胆に省略されてますが）規則正しい名前になっています。

```
#include <stdio.h>

int fgetc(FILE *fp);
char *fgets(const char *buff, int size, FILE *fp);
int fscanf(FILE *fp, const char *format, ...);

int fputc(int c, FILE *fp);
int fputs(const char *s, FILE *fp);
int fprintf(FILE *fp, const char *format, ...);
```

TODO: バッファリングされるときは fflash(stdout); のコラム

ファイルポインタは、ファイル上の現在の読み取り・書き込み位置を保持しています。これらの関数を呼び出すと、この位置が後ろに進んでいきます。繰り返し呼び出すことで、複数文字あるいは複数行を処理できます。

それぞれの関数の使用例は、後ほど紹介します。

コラム： フラッシュ

バッファリングは、C 言語の FILE*だけでなく、OS の内部など、いくつもの階層で行われています。普段はそのことを意識する必要はありません。

プログラムが異常終了すると、書き込み予定の内容が失われる可能性があります。（USB メモリの取り外し操作を忘れると、データに不整合が起こることがあるのと似ています。）ファイルを閉じる前に、バッファにたまったデータを強制的に書き込ませたいこともあるでしょう。この操作を、プログラミング言語では**フラッシュ** (flush) といいます。

表 11.2 頻出マクロ一覧

マクロ名	記載ヘッダ	典型的な値	意味
NULL	<stdio.h>	必ず 0	ポインタの特別な値
EOF	<stdio.h>	-1	ファイルの終端 (End Of File)
EXIT_SUCCESS	<stdlib.h>	0	正常終了
EXIT_FAILURE	<stdlib.h>	1	異常終了

11.2.4 プログラムの中断（エラー処理）

ファイル処理と直接の関係はないのですが、ファイル処理の途中でエラーが発生した場合など、プログラムを強制的に終了したくなることがあります。そのような場合に exit() が便利ですので、ここで紹介します。

```
#include <stdlib.h>

void exit(int status);
```

どの関数から exit() を呼び出してもプログラムを終了します。exit() は関数なのに、呼び出し元には戻りません。

引数の status に与える値として、正常終了を表す EXIT_SUCCESS と、異常終了を表す EXIT_FAILURE というマクロ定数が用意されています。status は main() の return で返す値と同じ働きがあります。表 11.2 のように、EXIT_SUCCESS は 0 に定義されているので、これまで main() で return 0; としてきたのは、正常終了を表しているのでした。

コラム： main 関数の return

main 関数は int 型で、いつも return 0; を書いてきましたが、この値は誰が受け取るのでしょうか?

プログラムを実行したシェル（コマンドインタプリタ）が受け取ります。シェルは環境によって違います。cmd.exe だったり bash だったりいくつもの種類があり、それぞれに受け取った後の扱い方が異なります。大枠では、0–255 の値を受け取って、**0 を正常終了、それ以外を異常終了**とみなす、というところだけは共通しています。値によって異常の内容を表すのですが、どんな値でどんな異常を表すのか、一般的な決まりはありません。

なお、C99 からは main 関数の return 0; を省略してよいことになりました。（C++ でも同じです。Java の main 関数は元々 void 型です。）

11.3 ファイルの使用例

11.3.1 書き込み

プログラムでファイルを作ってみます。表 11.1 の書き込み用の関数を使います。

- `fprintf(fp, "....", ...)` は `printf("....", ...)` とほぼ同じ動作で、画面に表示する代わりにファイルに書き込みます。
- `fputc(c, fp)` は 1 文字を書き込みます。`fprintf(fp, "%c", c)` と同じです。
- `fputs(s, fp)` は文字列を書き込みます。`fprintf(fp, "%s", s)` と同じです。

これら 3 つを取り混ぜて使っても大丈夫です[*7]。fprintf() はさまざまな型で使える万能選手です。特定の型専用の fputc() や fputs() は、効率的に動作しそうです。

実際のプログラムをみてみましょう。ソースコード 11.1 です。

前処理 5 行目で用意しておいたファイル名を使って、8 行目でファイルをオープンします。書き込みモードの `"w"` を指定します。9 行目の if でオープンに成功したかどうかを **NULL と比較**して検査します。オープンに失敗したら、もう処理を続けられません。10 行目でオープンに失敗したファイル名を表示してから、11 行目の「exit(EXIT_FAILURE);」でプログラムを中断します。

ここは main() なので、「return EXIT_FAILURE;」でも中断になりますが、通常は main() 以外の関数で行うので exit() を使ってみました。

ファイル名は、8 行目の fopen() と 10 行目の printf() の 2 ヶ所で必要です。これらが食い違うことのないよう、5 行目で変数 filename に代入してから使っています。**複数回使うものはまとめる**。プログラマの鉄則です。

主処理 ファイルに書き込んでいます。fprintf(fp, ...) は（引数が 1 つ多いですが）画面表示する printf() と使い方が同じです。fputs() は`"AAA"`の文字列リテラル、fputc() は `'A'` の文字定数を出力するのに便利です。

後処理 24 行目でファイルをクローズして、25 行目で正常終了です。これまで main() では「return 0;」と書いてきましたが、EXIT_FAILURE との対比で EXIT_SUCCESS を使ってみました。

プログラムを実行すると、同じフォルダに`"output.txt"`というファルができるので、内容を確認しましょう。Cygwin や Unix 系のターミナルなら`"cat output.txt"`、Windows のコマンドプロンプトなら`"TYPE output.txt"` というコマンドで画面に表示されます。

[*7] 混ぜてはいけない関数群もあります。例えば、`FILE*` によらず、バッファリングせずにバイト列を入出力する低水準関数があります。

あるいはテキストエディタで開いても構いませんし、ファイルマネージャでダブルクリックすると、関連付けられたアプリケーションで閲覧できるでしょう。

ソースコード 11.1 九九の表をファイルに出力

```c
#include <stdio.h>
#include <stdlib.h> // exit()

int main(void) {
    char filename[] = "output.txt";  // 書き込むファイル名

    /* 前処理 */
    FILE *fp = fopen(filename, "w");
    if (fp == NULL) {
        printf("ファイルオープンに失敗しました '%s'\n", filename);
        exit(EXIT_FAILURE);  // プログラムを中断する（異常終了）
    }

    /* 主処理 */
    fputs("九九の表\n", fp);
    for (int i=1; i<=9; i++) {
        for (int j=1; j<=9; j++) {
            fprintf(fp, "%2d ", i*j);
        }
        fputc('\n', fp);
    }

    /* 後処理 */
    fclose(fp);
    return EXIT_SUCCESS; // 正常終了
}
```

頻出ミス

Windows のテキストエディタで、プログラムの出力したファイルを閲覧するときの注意事項です。（Unix 系や Mac は、ファイルロックの仕組みが違うので、**常に後者**になります。）

テキストエディタの中には、閲覧中のファイルを**書き込み禁止**にするものがあります。この場合、閲覧中はプログラムがファイル出力に失敗するので、再実行する前に**ファイルを閉じる**必要があります。

逆に**書き込みのできる**場合は、プログラムをいつでも実行しなおせますが、閲覧内容に即座に反映されるわけではないので、**ファイルを開き直す**必要があるでしょう。（書き換えを検知して、開き直しを提案してくれるエディタもあります。）

11.3.2　文字単位の読み取り

既存のファイルから 1 文字を読み取るには fgetc() を使います。関数の戻り値が、char ではなく int であることに注意してください。

```
#include <stdio.h>
int fgetc(FILE *fp);
```

- 正常に 1 文字を読み取ると、その文字コードを unsigned char の値として返します。
- ファイルの最後まできて（あるいはエラーが起こって）、もう読み取れなくなると EOF という値を返します。EOF はファイルの終端（End Of File）を表すマクロ定数で、文字コードと区別のつく値（例えば –1）に定義されています（表 11.2）。

ファイルの内容をそのまま画面表示するプログラムでの使用例を、ソースコード 11.2 で見てみましょう。下線部はソースコード 11.1 と違うところです。前処理と後処理は、fopen() の**モードが "r"** になっただけでほぼ同じですが、主処理はすべて変わっています。

15 行 読み取った 1 文字を記憶するための変数 c を用意します。

16 行 while の条件式は複雑です。「fgetc() で読み取った文字を c に代入」と「ファイルの最後まできたらループ終了」の **2 つの動作を複合**して行っています。分解して、無限ループで書き直すと右下のようになります。これでは長くなりますし、あまりにもよく使う処理なので、慣れた人は左下の書き方を**慣用句**として覚えています。我々も覚えてしまいましょう。

```
/* 慣用句 */
while ((c = fgetc(fp)) != EOF) {


    ...;
```

```
/* 分解 */
for (;;) { // 無限ループ
    c = fgetc(fp);
    if (c == EOF) break;
    ...;
```

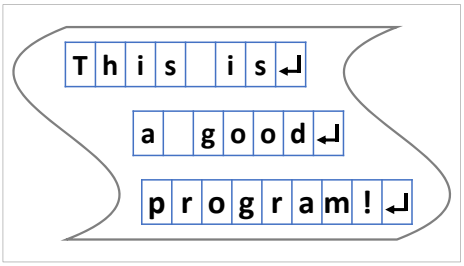
17 行 c を画面に表示[*8]します。これを繰り返して、ファイル全体を表示します。

このプログラムを実行するには、**事前に**ファイル"input.txt"を作っておきます。実行すると、このファイルの内容がそのまま表示されるはずです。

プログラムを少し改造してみましょう。先頭に「#include <ctype.h> 」を書き加え、17 行目を以下のように変更すると、大文字変換して表示するようになります[*9]。

```
        printf("%c", toupper(c));  // 17行目の差し替え
```

[*8] (f)printf() は %c で表示する文字を int で受け取るので、変数 c に (unsigned char) のキャストは不要です。
[*9] Shift_JIS などではマルチバイト文字が化けるかもしれません。UTF-8 は大丈夫です。



ソースコード 11.2 ファイルの読み取り（文字単位）

```c
#include <stdio.h>
#include <stdlib.h> // exit()

int main(void) {
    char filename[] = "input.txt";  // 読み取るファイル名

    /* 前処理 */
    FILE *fp = fopen(filename, "r");
    if (fp == NULL) {
        printf("ファイルオープンに失敗しました '%s'\n", filename);
        exit(EXIT_FAILURE);  // プログラムを中断する（異常終了）
    }

    /* 主処理 */
    int c;                  // 読み取る1文字を保存する（char ではない）
    while ((c = fgetc(fp)) != EOF) { // EOF はファイルの終端の目印
        printf("%c", c); // cの1文字を画面表示
    }

    /* 後処理 */
    fclose(fp);
    return EXIT_SUCCESS; // 正常終了
}
```

頻出ミス

fgetc() は、1 文字読み取ると、ファイルの読み取り位置を進めてしまいます。下の例のように、fgetc() を、EOF かどうかの比較と、1 文字の読み取りとの、2 回に分けて呼び出すとおかしなことになるので注意してください。

ループ中の fgetc() は 1 ヶ所だけにします。そのために、代入と同時に比較する、あの複雑な条件式の慣用句があるのです。（feof() も使えません。☞11.3.6 項）

```c
while (fgetc(fp) != EOF) { // (NG) 終端チェックで読み捨てる
    c = fgetc(fp);         // 1文字おきに読み取る
    ...;
```

11.3.3　行単位の読み取り

改行を区切としてデータを並べることがよくあります。このようなファイルは行単位で処理をしたほうが便利ですので、1 行を読み取る fgets() を使います。

```
#include <stdio.h>

char *fgets(char *buff, int size, FILE *fp);
```

buff　ファイルから読み取った 1 行を保存します。1 行というのは、改行で区切られたところまでという意味で、buff の末尾にはたいてい'\n' が付きます[*10]。

size　buff の大きさをバイト数で指定します。1 行がこれより長いと、いっぱいになったところで[*11]読み取りを打ち切ります。（残りは次の fgets() で読み取ります。）

戻り値　正常に読み取ったら buff[*12]を、ファイルの終端やエラーなどで読み取れなかったら NULL を返します。NULL かどうかは、必ず検査しましょう。

使用例はソースコード 11.3 です。下線部はソースコード 11.2 と違うところです。

読み取る文字列を格納するために、**行バッファ** (line buffer) と呼ばれる作業領域が必要になります。バッファの長さは事前（できたらコンパイル時）に決めておきたいので、1 行の長さの最大のものを想定しておきます。とは言っても、わからないことも多いので、たいていは 100 文字とか 1024 文字のような適当な値にしておき、後からでも変更できるようにマクロで定義します。

4 行目で行バッファのサイズをマクロ定義し、8 行目で行バッファに使う文字列 buff を確保します。18 行目の while ループの条件式は、以前よりも簡単になりました。fgets() が NULL でなければ buff に読み取った文字列が入っているので、19 行目の printf() の %s で buff を表示します。これで 1 文字読み取りのときの動作と同等です。

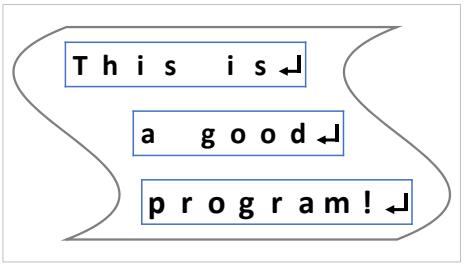
行単位で読み取るようになったので、各行に整数値の書いてあるファイルを読み取って、合計と平均を計算してみましょう。文字列から int への変換は <stdlib.h> の atoi() という関数[*13]が便利です。ソースコード 11.3 の主処理（17–20 行）を以下で差し替えます。

[*10] ファイルの末尾に'\n' がないときや、1 行が size バイトに収まりきらなかったときを除きます。また putc() と同様、OS による改行コードの違いを吸収して、'\n' に揃えてくれることになっています。

[*11] buff に、文字列の終端の'\0' は必ず付きます。size はこれも含んだ大きさなので、文字列の最大の長さは (size-1) です。

[*12] 戻り値の buff の値を利用する例は、見たことがありません。NULL ではない値の代表値なのでしょう。

[*13] long への変換なら atol()、double なら atof() です。☞??節



ソースコード 11.3 ファイルの読み取り（行単位）

```c
#include <stdio.h>
#include <stdlib.h> // exit()

#define BUFF_SIZE 1024

int main(void) {
    char filename[] = "input.txt";  // 読み取るファイル名
    char buff[BUFF_SIZE];  // 読み取る1行を保存する行バッファ

    /* 前処理 */
    FILE *fp = fopen(filename, "r");
    if (fp == NULL) {
        printf("ファイルオープンに失敗しました '%s'\n", filename);
        exit(EXIT_FAILURE);  // プログラムを中断する（異常終了）
    }

    /* 主処理 */
    while (fgets(buff, BUFF_SIZE, fp) != NULL) {
        printf("%s", buff);            // NULL はファイルの終端の目印
    }

    /* 後処理 */
    fclose(fp);
    return EXIT_SUCCESS; // 正常終了
}
```

```c
    /* 主処理（17〜20行の差し替え） */
    int count = 0, sum = 0;
    while (fgets(buff, BUFF_SIZE, fp) != NULL) {
        printf("'%s'を読み取りました\n", buff); // 読み取った1行を表示
        int val = atoi(buff);
        sum += val;
        count++;
    }
    printf("個数=%d, 合計=%d, 平均=%f\n",
           count, sum, (double)sum/(double)count);
```

平均を知るためには、データの個数も数える必要があります。ループ 1 回につき、count に 1 を加え、sum にデータの値を加えています。ファイルを最後まで読み取ったらループが終了するので、結果を出力します。

頻出ミス

ファイルオープン 1 回につき、読み取りループは 1 つです。ファイルの終端に達すると、それ以上の読み取りはできません。上のように、データの「個数」と「合計」を同時に求めれば大丈夫です。

「個数」と「合計」を別のループで求めるなら、2 つめのループの前に、ファイルを先頭から読み直すためにクローズして再びオープンする必要があります。配列とは状況が違うわけです。世の中には、再びオープンできないファイルもあるので、ひとまずはループ 1 つですませる努力をしてみましょう。

入力ファイルの例

```
1
3
5
7
9
2
4
6
8
10
```

出力結果の例

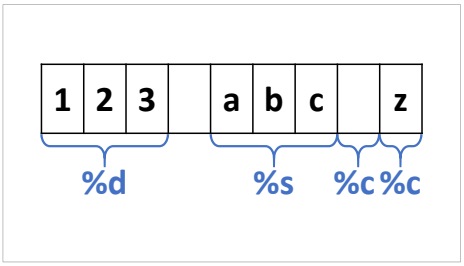
```
'1
'を読み取りました
'3
'を読み取りました

～(中略)～

'を読み取りました
'10
'を読み取りました
個数=10, 合計=55, 平均=5.500000
```

読み取った 1 行をシングルコーテーション (') で囲って表示したので、1 行の末尾に改行文字'\n' のついていることがわかります。int 変換には影響はありませんが、状況によっては除去する必要があります。

頻出ミス

この処理ではデータ 1 個につき改行は 1 個です。input.txt の**ファイルの末尾で改行**を繰り返すと、改行だけでも 1 つのデータ（数値なら 0）があるとして扱われます。

テキストエディタには、たいてい改行文字を視覚的に表示する機能があります。ファイルの末尾に「EOF」のようなマークを表示する場合もあります。設定をよく見直して、活用しましょう。



11.3.4 書式つきの読み取り

表 11.1 を見ると、読み取りには書式つきの関数 fscanf() がまだ紹介せずに残っていますが、この関数を現実に使う場面は、それほどありません。なぜなら、scanf() 系の関数は、書式の整ったものを読み取るために作られていて、想定してない文字列がやってくると、エラー処理ができないからです。実際のプログラミングの現場では、**fgets() と sscanf() を組み合わせて**使いますので、sscanf() を紹介します。

```
#include <stdio.h>

int sscanf(const char *str, const char *format, ...);
```

sscanf() は、与えられた文字列 str から format に従って値を読み取り、... に列挙された変数にその値を保存します。fgets() で読み取った 1 行の中から、整数や小数や文字列が混在していても、1 回の sscanf() で複数の値を読み取れます。scanf() はキーボードから値を読み取りますが、それの文字列版といったところです。**??**項も参照してください。

なぜ直接 fscanf() で読み取ると不都合があるかを説明します。例えば、`fscanf(fp, "%d,%d", &a, &b);` で 2 つの値を読み取るとします。すると fscanf() は、ファイルから 1 つめの値を読み取った後に、`","`が出てくるまで、それ以外の文字を読み飛ばします。もし`","`のないファイルを読み込んでいると、スペースと改行を同一視して、ファイルの末尾までどんどん読み飛ばします。これが 1 回の fscanf() で起こるので、プログラムでは途中で止められません。つまり、`","`がないというエラーを報告することができません。

fgets() と sscanf() を組み合わせると、少なくとも 1 行で止まるので、ファイルの末尾までは影響を受けなくてすみます。

また scanf() 系の関数で文字列を`"%99s"`などで受け取る際には、このように**長さの上限を設定**する必要がありますが（☞**??**節）、sscanf() なら fgets() の段階でも制限されるので、直接 fscanf() で受け取るよりも安全です。

11.3.5　2 つのファイルを同時に読み書き

読み取りと書き込みを同時に行ってみましょう。**ファイル 1 つにつきファイルポインタ 1 つ**が必要なので、ファイルポインタ変数を 2 つ用意するところに注意してください。変数の名前は fp に限らず何でもよいので、2 つの変数名を考えます。

ソースコード 11.4 では変数名を fpin と fpout としてみました。前処理で 2 つのファイルをオープンして、後処理では 2 つとも閉じるのを忘れないようにします。

ソースコード 11.4　ファイルのコピー

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

#define BUFF_SIZE 1024

int main(void) {
    char infile[]  = "data.txt";
    char outfile[] = "output.txt";
    char buff[BUFF_SIZE];

    /* 前処理 */
    FILE *fpin = fopen(infile, "r");
    if (fpin == NULL) {
        printf("ファイルオープンに失敗しました '%s'\n", infile);
        exit(EXIT_FAILURE);
    }
    FILE *fpout = fopen(outfile, "w");
    if (fpout == NULL) {
        printf("ファイルオープンに失敗しました '%s'\n", outfile);
        exit(EXIT_FAILURE);
    }

    /* 主処理 */
    while (fgets(buff, BUFF_SIZE, fpin) != NULL) {
        int val = atoi(buff);
        fprintf(fpout, "%dのデータがありました\n", val);
    }

    /* 後処理 */
    fclose(fpin);
    fclose(fpout);
    return EXIT_SUCCESS; // 正常終了
}
```

鋭意作成中

11.3.6 終端検査、エラー検査

MEMO: fflush() に差し替えるか、ugetc() もあり

あまり使わない関数ですが、使い方を間違えそうなので紹介しておきます。

ファイルの読み取りに失敗した場合、ファイルの終端なのか、エラーが起こったのかを見分ける手段が用意されています。

```
#include <stdio.h>

int feof(FILE *stream);
int ferror(FILE *stream);
```

feof() はファイルの終端で 0 以外の値を返します。ferror() はエラー発生で 0 以外の値を返します。いずれも、fgetc() などで読み取りに失敗した後に有効な値を返します。つまり、どちらも 0 であったとしても、**次の読み取りが成功する保証はありません**。あくまでも読み取りに失敗した後に、その理由を調べるためのものです。

頻出ミス

feof() と ferror() を while の条件にするには無理があります。fgetc() などでの終端検出は依然として必要なので、これなら無限ループにしても同じです。

```
while (!feof(fp) && !ferror(fp)) {
    int c = fgetc(fp);
    if (c == EOF) break; // この検査は省略できません
    ...
}
```

11.4 オープンするファイルを実行時に決める

これまでのプログラムでは、取り扱うファイル名をプログラム中に埋め込んでいました。（このような手法を**ハードコーディング** (hard coding) といいます。）これでは対象のファイルを変更するのに再コンパイルが必要なので、とても不便です。もちろん、キーボードから対話的にファイル名を受け取るように改造することも簡単です。しかし、それでも使い勝手はよくありません。なぜなら、プログラムの作成中には、繰り返し実行して動作を確かめたいので、そのたびにファイル名を手入力する必要があるからです。

取り扱うファイルをプログラムに指示することは、**多くのプログラムに共通**する操作なので、いろいろ工夫されています。

まずコマンド（a.exe や a.out）の後ろに書いた文字（**コマンドライン引数** (command line argument) といいます）をプログラムで受け取りましょう。`main()` の引数を void ではなく、以下のようにします。

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("使い方: %s filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    char *file = argv[1];
    ...
```

このプログラムを"`./a.exe input.txt`"として実行すると、main() 関数が `argc=2`, `argv[0]="./a.exe"`, `argv[1]="input.txt"` となった状態で呼び出されて、上のような処理で `file` にファイル名が手に入ります。input.txt をつけずに実行すると `argc=1` となって、最初の if が成立して、使い方のメッセージを表示します。argv[0] には常にコマンド名が入り、argc がコマンドライン引数の個数 +1 になるというわけです。

ここまで来れば、あとはコマンド実行時にファイル名をうまく入力できれば楽になります。同じファイル名でよければ、シェルの**履歴** (history) 機能で十分です。カーソルキーの ↑ を押すと、1 つ前に入力したコマンドが、引数を含めて表示されるはずです。初めて入力するときでも、**補完入力**（ファイル名の途中まで入力して Tab を何度か押す）をすれば、存在するファイル名を間違いなく入力できます。

ファイルマネージャとターミナルの連携機能で、ファイルの**アイコンをターミナルにドロップ**すると、そのファイルのフルパス名が入力されるようになっていることが多いです。これは強力な機能です。（☞**??**節）

eclipse や Visual C++ のような統合開発環境でも、コマンドライン引数を指定できますが、残念ながら使い勝手はあまりよくありません。

鋭意作成中

11.5 現実的なファイルオープン

ここまで、ファイルオープン時の煩雑なエラー処理を何度も行ってきました。世の中のプログラマが、みんなこの処理を書いているかというと、現実は少し違います。

エラーチェック付きオープン オープン失敗時に単純にプログラムを中断させてよいなら、以下のようなエラーチェック付きの関数がよく使われます。これを fopen() の代わりに呼び出すだけで、エラー処理が省けます。

```
/* ファイルオープン（NULL は 返さず exit() する）*/
FILE *xfopen(char *filename, char *mode) {
    FILE *fp = fopen(filename, mode);
    if (fp == NULL) {
        fprintf(stderr, "ファイルオープン失敗 '%s'\n", filename);
        exit(EXIT_FAILURE);
    }
    return fp;
}

int main(void) {
    FILE *fpin  = xfopen("input.txt", "r");  // エラー処理不要
    FILE *fpout = xfopen("output.txt", "w"); // エラー処理不要
    ...
```

標準入出力 stdin と stdout, stderr（☞??項）は、プログラム開始時点でオープンされているので、エラー処理は不要です。オープン時のエラーにはシェルが対処します。（プログラムが起動されないことでしょう。）

出荷される製品 本格的なプログラムなら、利用者に正しいファイル名の再入力を促すなど、もっと手間のかかる処理を書くことになるでしょう[*14]。

[*14] 銀行のオンラインシステムや原子力プラントの制御プログラムのように、クリティカルな場面で使われるものだと、プログラムの大半がエラー（例外）処理になってしまうことも珍しくありません。

11.6 練習問題

1. **[複数の手段の使い分け（☞11.3.1 項）]**
   （a）`fputc(c, fp)` は `fprintf(fp, "%c", c)` で代用できる。
   標準ライブラリに、この単機能の関数がなぜ用意されているのか、考察せよ。
   （b）数学関数の `sqrt(x)` は `pow(x, 0.5)` で代用できる。
   標準ライブラリに、この単機能の関数がなぜ用意されているのか、考察せよ。
2. **[文字単位の読み取り（☞11.3.2 項、11 ページと 14 ページ上の頻出ミス）]**
   11 ページのソースコード 11.2 を参考に、読み取ったファイルに含まれている文字数を数えてみよ。また、数字（0 から 9 の文字）がいくつあったかを数えてみよ。同様にアルファベットの大文字や小文字も数えてみよ。（☞**??**項）
3. **[行単位の読み取り（☞11.3.3 項）]**
   13 ページのソースコード 11.3 を参考に、1 行に 1 つの整数の記述されたファイルを読み取り、読み取った数値の**最大値**と**最小値**を表示せよ。
4. **[ファイル出力の使い道]**
   画面（標準出力）に表示される文字は、リダイレクトの機能（☞**??**項）を使えば、簡単にファイルに保存できる。では、プログラムで `fopen(outfile,"w")` などを用いてファイルを保存することが有益なのは、どのような場面か、例を挙げよ。**ヒント**：プログラム中で扱うファイルがいくつかに着目せよ。

コラム： 廃止された gets()

標準入力から 1 行を読み取る `gets()` は、C11 で廃止されました。fgets(..., stdin) と似た動作をしていたのですが、以下の点で動作が異なります。

- 読み取る文字列の長さの上限を設定できないため、バッファオーバーフローを防げません。（これが廃止の理由です。）
- 末尾の改行文字（`'\n'`）を削除します。

しかし、gets() の代替品を標準ライブラリ関数から探しても、末尾の`'\n'` を削除するものが見当たらず、fgets() の後に次のような処理を加えるしかなさそうです。

```
while (fgets(buff, sizeof(buff), stdin) != NULL) {
    int len = strlen(buff);
    if (len > 0 && buff[len-1] == '\n') { len--; buff[len]='\0'; }
    ...
}
```

なお、新設された gets_s() はオプション扱いのため、例えば GCC では使えません。

索引