第3章　文字と文字列の操作

インタラクティブなプログラムではユーザが入力した文字列を処理する必要がある．プログラミング第3同演習で既に習得していると思うが，本章では文字と文字列，および main() の引数である argc，argv について復習をしておく．さらに，文字や文字列処理によく使う関数について概略を述べる．文字列処理のための便利な関数もあると思うが，それらをブラックボックスとして利用するのではなく，中身についても理解して欲しい．

3.1　文字と文字列

C言語において，'a' という表記は文字定数であり，int 型を持つ．一方，"a"や"abc" という表記は文字列リテラルである．たとえば'a' と"a"は同じように見えるが，実はまったく別のものである．

図3.1の01行目は char 型変数へ文字定数を代入している例，02行目は char *型変数へ文字列リテラルを代入している例，03行目は char 型配列の初期化に文字列リテラルを使用している例である．以上の3種類の例におけるメモリオブジェクトの状況を図3.2に示す．1番目の例では，char 型変数 c に1バイトのメモリオブジェクトが割り当てられ，その内容が 0x61 (= 'a') となる．2番目の例では，0x00001000 番地に文字列リテラル (この例では'a'，'\0') が格納され，char *型変数 s には文字列リテラルの先頭番地である 0x00001000 が代入される．32ビットマシンの場合，変数 s には4バイトのメモリオブジェクトが割り当てられる．この例での 0x00001000 は番地の例であって，いつもこの番地が使われるわけではない．3番目の例では，コンパイラによって文字列リテラルの長さ (この場合は2バイト) が計算され，配列の各要素が初期化される．これらの違いをきちんと理解して欲しい．

```
01: char c = 'a';       // c に 0x61(= 'a') を代入
02: char *s = "a";      // 文字列  0x61,0x00 (= 'a', '\0') の
                        // 先頭番地を s に代入
03: char str[] = "a";   // str[0] に 0x61，str[1] に 0x00 が初期値として
                        // 代入される
```

図 3.1: 文字定数と文字列リテラルの代入

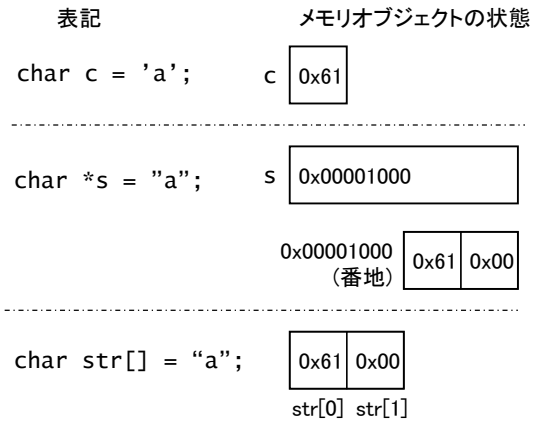

図 3.2: 文字定数・文字列リテラルの代入後のメモリオブジェクトの状況

```
01: main(int argc, char *argv[])
02: {
03:     . . .
04: }
```

図 3.3: main() の仮引数

3.2 main 関数の仮引数：argc と argv

プログラムの実行を開始するときに引数を指定すると，これらは図 3.3 のような形で main 関数に渡される．int argc はコマンド入力がいくつの文字列からなるかを示し，char *argv[] は各文字列の先頭番地を保持する配列である．32 ビットマシンの場合，int 型変数 argc には 4 バイトのメモリオブジェクトが割り当てられ，char *型配列である argc の各要素にも 4 バイトのメモリオブジェクトが割り当てられる．

たとえば，"ls -l a.c b.c"というコマンドを実行すると argc，argv は図 3.4 のようになる．"ls -l a.c b.c"は 4 つの文字列からなるため，argc は 4 となる．またそれぞれの文字列はメモリオブジェクトに保持され，'\0' で終端される．そしてそれぞれの文字列の先頭番地が argv[0]〜argv[3] にセットされる (この例では，0x00001000，0x00001010，0x00001020，0x00001030)．

3.3 よく使われる文字・文字列処理の関数

シェルで man string と入力すると文字列処理のための関数の一覧が分かる．また，たとえば man isalpha と入力すると文字のクラス分けのための関数の一覧が分かる．以下，主な関数を紹介する．詳しい使い方に関しては man コマンドで調べて欲しい．man コマンドを使いこなすこともよいプログラムを作成するために必要な技術である．解説本や Web ばかりに頼らず，man コマンドを使いこなして欲しい．

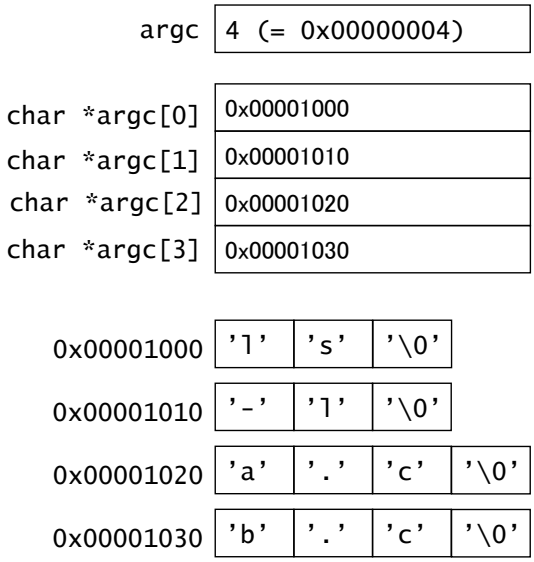

図 3.4: "ls -l a.c b.c"を実行したときの argc，argv の例

3.3.1 文字の入出力

文字の入力については以下のような関数が用意されている．

- #include <stdio.h>
- int getchar();
- int getc(FILE *stream);
- int ungetc(int c, FILE *stream);

これらの関数を利用するには stdio.h ファイルを include しておく必要がある．getchar() は標準入力から 1 文字を読み込む．getc() は stream で指定した入力ストリームから 1 文字を読み込む．FILE 型やストリームという用語はプログラミング第 3 同演習で学習したことと思う．

ungetc() は getc() などで読み込んだ文字を，stream で指定した入力ストリームにプッシュバックする関数である．プッシュバックされたデータは次の getc() などで読み込むことができる．なぜこのような関数が必要なのか不思議に思うかもしれない．以下に一例を挙げる．ungetc() は 1 文字ずつ読み込みながら構文解析をする場合に必要になる．たとえば，i=1; という文を 1 文字ずつ読み込みながら構文解析することを考えてみよう．ここでは，'i'，'='，'1' の間に空白が入っていない．このような場合，'=' を読み込んではじめて変数 i を認識することができる．その際，次の要素の構文解析のため'=' を入力ストリームにプッシュバックしておく必要がある．

文字の出力については以下のような関数が用意されている．

- #include <stdio.h>
- int putchar(int c);

- `int putc(int c, FILE *stream);`

これらの関数を利用するには stdio.h ファイルを include しておく必要がある．putchar() は文字 c を標準出力に出力する関数である．putc() は文字 c を stream に出力する関数である．これらの関数の返り値は出力したデータである．

プログラムの実行が開始されたとき，標準入力，標準出力，標準エラー出力という 3 つのストリームがあらかじめ利用可能であることは学習していると思う．これらのストリームを stream に指定するときは，それぞれ stdin，stdout，stderr という予約語を用いる．

3.3.2 書式指定の文字列入出力

書式指定の文字列入力については以下のような関数が用意されている．

- `#include <stdio.h>`
- `int scanf(const char * restrict format, ...);`
- `int fsanf(FILE * restrict stream, const char * restrict format, ...);`
- `int ssanf(const char * restrict str, const char * restrict format, ...);`

これらの関数を利用するには stdio.h ファイルを include しておく必要がある．どのような書式が指定可能かは man コマンドに譲り，ここでは説明を省略する．fcanf() は stream によってデータを読み込むストリームを指定することができる．scanf() は str で指定したメモリオブジェクトからデータを読み込む．これらの関数の返り値は，指定した書式どおりに正しく入力できたデータの個数である．

書式指定の文字列出力については以下のような関数が用意されている．

- `#include <stdio.h>`
- `int printf(const char * restrict format, ...);`
- `int fprintf(FILE * restrict stream, const char * restrict format, ...);`
- `int snprintf(char * restrict str, size_t size,`
  `const char * restrict format, ...);`

これらの関数を利用するには stdio.h ファイルを include しておく必要がある．fprintf() は stream により出力するストリームを指定する．sprintf() は str で指定したメモリオブジェクトに文字列を書き込む．また，size によりメモリオブジェクトのサイズを指定する．これらの関数の返り値は実際に出力した文字数である．

これらの中で fprintf() は今後よく使うようになるはずである．これは以下のような理由による．プログラムの実行中にエラーが発生することはよくある．またインタラクティブなプログラムの場合はユーザが間違った入力をすることもある．原因はともあれ，プログラム実行中にエラーが発生したらその旨を表示する必要がある．このとき使用する

のが fprintf(stderr, ...) である．すなわち，標準エラー出力にエラーメッセージを出力する．エラーメッセージを出力する際に printf() を使用して標準出力に出力してはいけない．標準出力も標準エラー出力も通常は画面に設定されているため，どちらに出力しても同じではないかと思うかもしれない．しかし，リダイレクトを指定したときに違いが出る．“command > file” のように指定することにより，コマンドの標準出力をファイルにセーブすることができる．このとき，もしエラーメッセージも標準出力に出力するようにプログラムされていると，エラーメッセージもファイルにセーブされてしまい，画面を見ていてもエラーに気付かない．また，ファイルにも実行結果とは関係のないエラーメッセージが混じってしまい，あとでデータを解析するときなどに都合が悪い．

以上のように，正常な実行結果は標準出力に出力し，エラーメッセージは標準エラー出力に出力するようにプログラムすることを習慣づけるべきである．

3.3.3 行単位の入力

行単位の入力の関数には以下のものがある．

- #include <stdio.h>
- char *fgets(char * restrict str, int size, FILE * restrict stream);

この関数を利用するには stdio.h ファイルを include しておく必要がある．fgets() は stream が示すストリームから最大 size - 1 文字を，改行文字または EOF まで読み込み，str が指すメモリオブジェクトに格納する．文字列の最後には’\0’ が付加される．

man fgets を実行すると，fgets() とほぼ同様の機能を持ち，標準入力から行単位で入力を行う char *gets(char *str) という関数もあることがわかる．しかし gets() にはセキュリティ上の脆弱性があるため，**絶対に使用してはいけない**．なぜ脆弱性があるかというと，入力文字列を格納するメモリオブジェクトの長さの上限が指定できないからである．入力文字列の長さの上限はあらかじめ予想することはできない．しかし，引数 char *str で指定するメモリオブジェクトのサイズは有限である．もし用意しておいたメモリオブジェクトのサイズよりも長い文字列が入力されると，入力用のメモリオブジェクトを超えたメモリ領域が入力文字列によって破壊されてしまう．このような脆弱性は，不正な実行コードを送りつけて不正な動作をさせるという攻撃に利用されてしまう．

どの関数にセキュリティ上の脆弱性があるかは，man を調べればわかる．ぜひ man コマンドを活用して欲しい．

3.3.4 文字列単位の出力

文字列単位の出力のためには以下の関数がある．

- #include <stdio.h>
- int puts(const char *str);
- int fputs(const char *str, FILE *stream);

これらの関数を利用するには stdio.h ファイルを include しておく必要がある．puts() は s が指す文字列を標準出力に出力する．fputs() は stream によって出力するストリームを指定する．

3.3.5 文字列の長さ

文字列の長さを調べるには以下の関数を使う．

- `#include <string.h>`
- `size_t strlen(const char *s);`

この関数を利用するには string.h ファイルを include しておく必要がある．strlen() は s が指す文字列の長さを返す．文字列の長さとは，NULL 文字までの文字数である (NULL 文字は含まない)．

3.3.6 文字列の比較

2 つの文字列が一致するかを調べるには以下の関数を使う．

- `#include <string.h>`
- `int strcmp(const char *s1, const char *s2);`
- `int strncmp(const char *s1, const char *s2, size_t len);`
- `int strcasecmp(const char *s1, const char *s2);`
- `int strncasecmp(const char *s1, const char *s2, size_t len);`

これらの関数を利用するには string.h ファイルを include しておく必要がある．strcmp() は s1，s2 が指す文字列を比較し，s1 が s2 より大きいか，等しいか，小さいかを返す．strncmp() は先頭から最大 len 文字のみを比較する．strcasecmp() は大文字，小文字の区別をせずに文字列を比較する．strncasecmp() は大文字，小文字の区別をせずに先頭から最大 len 文字のみを比較する．

3.3.7 文字列の連結

2 つの文字列を連結するには以下の関数を使う．

- `#include <string.h>`
- `char *strcat(char * restrict s, const char * restrict append);`
- `char *strncat(char * restrict s, const char * restrict append, size_t count);`

これらの関数を利用するには string.h ファイルを include しておく必要がある．strcat() は文字列 s に文字列 append を連結する．strncat() は文字列 s に文字列 append のうち最大 count 文字を連結する．これらの関数の返り値は s の値である．

3.3.8 文字列の複製

文字列の複製を作成するには以下の関数を使う．

- `#include <string.h>`
- `char *strcpy(char * restrict dst, const char * restrict src);`
- `char *strncpy(char * restrict dst, const char * restrict src, size_t len);`

これらの関数を利用するには string.h ファイルを include しておく必要がある．strcpy() は文字列 src を dst が示すメモリオブジェクトにコピーする (終端を示す'\0' も含む)．strncpy() は文字列 src を dst が示すメモリオブジェクトに最大 len 文字分だけコピーする．これらの関数の返り値は dst の値である．

3.3.9 文字列中の文字の検索

文字列中にある文字が含まれているかを調べるには以下の関数を使う．

- `#include <string.h>`
- `char *strchr(const char *s, int c);`
- `char *strrchr(const char *s, int c);`

これらの関数を利用するには string.h ファイルを include しておく必要がある．strchr() は文字列 s の先頭から文字 c を検索し，最初に見つかったところを指すポインタを返す．strrchr() は文字列 s の末尾から文字 c を検索し，最初に見つかったところを指すポインタを返す．見つからなかった場合は両関数とも NULL を返す．

3.3.10 文字列中の文字列の検索

文字列中にある文字列が含まれているかを調べるには以下の関数を使う．

- `#include <string.h>`
- `char *strstr(const char *big, const char *little);`
- `char *strcasestr(const char *big, const char *little);`
- `char *strnstr(const char *big, const char *little, size_t len);`

これらの関数を利用するには string.h ファイルを include しておく必要がある．strstr() は文字列 big の先頭から文字列 little が含まれるかを検索し，見つかったところを指すポインタを返す．strcasestr() は文字列 big の先頭から文字列 little が含まれるかを，大文字・小文字を区別せずに検索し，見つかったところを指すポインタを返す．strstr() は文字列 big の末尾から文字列 little が含まれるかを検索し，見つかったところを指すポインタを返す．見つからなかった場合はこれらの関数は NULL を返す．

3.3.11 文字列数値の整数への変換

文字列で示されている数値を整数へ変換するには以下の関数を使う．

- #include <stdlib.h>
- #include <limits.h>
- long strtol(const char * restrit nptr, char ** restrit endptr, int base);

この関数を利用するには stdlib.h ファイルと limits.h ファイルを include しておく必要がある．strtol は文字列 nptr が示す文字列数値を base 進法で解釈して整数に変換し，その値を返す．endstr の値が NULL でない場合は，文字列数値表記に該当しない最初の文字を指すポインタが*endstr に設定される．文字列 nptr の先頭には任意個の空白文字があってもよく，そのあとに 1 つの'+'，'-' 符号があってもよい．

3.3.12 文字のクラス分け・大文字と小文字の変換

主な文字のクラス分け関数と，大文字・小文字変換の関数を挙げる．クラス分け関数 (e.g., is...()) は，真の場合は非ゼロの値を返し，偽の場合はゼロを返す．なお，これらの関数はシングルバイト文字コードのみに有効である．マルチバイト文字コードの場合には isalnum() などの代わりに iswalnaum() などを用いなければならない．

- #include <ctype.h>
- int isalnum(int c); - c が数字かアルファベットであるか．
- int isalpha(int c); - c がアルファベットであるか．
- int isblank(int c); - c が空白あるいはタブか．
- int isdigit(int c); - c が数字であるか．
- int islower(int c); - c が小文字のアルファベットであるか．
- int isprint(int c); - c が表示可能であるか．
- int isspace(int c); - c が空白文字 ('\t', '\n', '\v', '\f', '\r, スペース) であるか．

```
01:     char *src = "src string";
02:     char *dst;
03:
04:     strcpy(dst, src);
```

図 3.5: 間違いやすい例

```
01:     char *src = "src string";
02:     char dst[80];
03:
04:     strcpy(dst, src);
```

図 3.6: 図 3.5 の正しい記述例

- int isupper(int c); - c が大文字のアルファベットであるか.
- int tolower(int c); - c を小文字に変換.
- int toupper(int c); - c を大文字に変換.

3.4 ポインタに関する間違いやすい例

上述の関数を利用する際に気をつけなければならないことは，char *型の変数がメモリオブジェクトを指すようにしなければならないことである．たとえば図 3.5 のような間違いをすることがあるのではないだろうか．この例では，char*型変数 dst はメモリオブジェクトを指していない．すなわち，strcpy() が dst の指すメモリオブジェクトまで割り当て，その番地を dst に設定してくれるという思い違いである．このような場合，正しくは図 3.6 のように dst がメモリオブジェクトを指すようにしておかなくてはならない．

しかし図 3.6 はオーバフローを考慮していない．この例では src が指す文字列の長さは配列 dst[] のサイズを超えないことは明らかであるが，一般的にこのことは成り立たない．もし dst[] のサイズを超えるような文字列が src に指定されると，dst[] を超えたメモリ領域にまでコピーが行われてしまう．dst[] の次には別の変数のメモリ領域が取られているかもしれない．そのような場合，他の変数の値を書き換えてしまうことになる．このような誤りをオーバフローと呼ぶ．オーバフローを考慮すると，strcpy() ではなく strncpy() を使用すべきである (図 3.7 参照)．この例では strncpy() の第 3 引数で dst[] のサイズを指定しているため，この領域を超えてデータがコピーされることはない．

次に図 3.7 において src の指す文字列の長さが 80 バイトより長い場合を考えてみよう．この場合，dst[] は空文字 ('\0') で終端されなくなる．したがって，その後 dst[] を文字列として扱うと，dst[] の本来の領域を超えた部分も文字列の一部として扱われてしまうことになる．たとえば strlen(dst) を実行した場合を考えてみよう．そこでさらに注意

```
01:    char *src = "src string";
02:    char dst[80];
03:
04:    strncpy(dst, src, sizeof dst);
```

図 3.7: オーバフローへの対処の例 (1)

```
01:    char *src = "src string";
02:    char dst[80];
03:
04:    memset(dst, 0, sizeof dst);
05:    strncpy(dst, src, sizeof dst - 1);
```

図 3.8: オーバフローへの対処の例 (2)

深くプログラミングするには図 3.8 のようにする．04 行目では memset() により dst[] のすべての要素を 0 で初期化する．05 行目では strncpy() の第 3 引数に sizeof dst - 1 を指定することにより，src が指す文字列が 80 文字以上の場合も dst[] に格納される文字列は必ず'\0' で終端される．

3.5 エラー処理

ライブラリ関数やシステムコールはいつも正常に実行されるとは限らない．たとえば fgets() には入力するストリームを指定するが，読み込むことのできないストリーム (たとえば stdout) を指定すると fgets() の実行はエラーになり，返り値は NULL になる．fgets() のあとに入力データを利用する処理がある場合，fgets() でエラーが発生したらそれ以降の処理は意味のないものになってしまう．したがって，ライブラリ関数を利用する場合には man コマンドで返り値を調べ，エラーが発生する可能性のあるものについてはエラー処理をきちんと行うことが重要である．

上述した fgets() の場合には図 3.9 のようになる．fgets() や getc() はエラーが発生したり EOF になった場合には NULL を返すが，返り値からはどちらが発生したのか分からない (07 行目)．そこで ferror() や feof() を呼び出してどちらが発生したのかを調べなければならない (09 行目)．ferror() はエラーの場合には非ゼロの値を返す．したがって 09 行目は本来は図 3.10 のように記述すべきだと思うかもしれない．ここで C 言語における真偽値は非ゼロが真，ゼロが偽であることを思い出して欲しい．そこで ferror() のような真偽値を返す関数の返り値を条件式として使用する場合には，08 行目のように記述してもよい．

```
01: int
02: func()
03: {
04:     FILE *st;
05:     char lbuf[80];
06:     ...
07:     if (fgets(lbuf, sizeof(lbuf), st) == NULL) {
08:         /* エラー or EOF */
09:         if (ferror(st)) {
10:             /* エラー処理 */
11:             fprintf(stderr, "input error in fgets()\n");
12:             return -1;
13:         } else
14:             return 0;    /* EOF */
15:     }
16:     /* 正常な場合の処理の続き */
17:     ...
18: }
```

図 3.9: fgets() のエラー処理

```
09:         if (ferror(st) != 0) {
```

図 3.10: if 文の例

3.6 課題2 (bufcache) におけるコマンド処理

課題 2(bufcache) のプログラムは，ユーザが入力したコマンドを判別しそれに従った処理をしなければならない．一番簡単な方法は，入力された文字列を定義されているコマンド名と 1 つずつ比較していくものである (図 3.11 参照)．図 3.11 では，char 型の配列である cmd にすでに入力された文字列が代入されているとする．しかし，一見してこの方法は洗練されたものではないことが分かるであろう．すべての定義されたコマンド名について同じような if 文を延々と記述していかなければならない．コマンド数がほんの数個ならこの方法でもよいが，コマンド数が多かったり，新たにコマンドを加えるときなどは，この方法では面倒である．

そこで次のように考える．定義されたコマンド名の文字列とそのコマンドを処理するための関数へのポインタをメンバーとする構造体を定義し，この構造体の配列を用意する (図 3.12 参照)．struct command_table のメンバーとして，コマンド名の文字列へのポインタである char *cmd と，その処理関数へのポインタである void (*func)() を定義する (05，06 行目)．そして struct command_table 型の配列として cmd_tbl[] を定義する

```
01:     char cmd[16];
02:     ...
03:     ... /* cmd[] に入力されたコマンド名が設定される */
04:     ...
05:     if (strcmp(cmd, "help") == 0) {
06:         /* help コマンドの処理 */
07:         ...
08:     } else if (strcmp(cmd, "init") == 0) {
09:         /* init コマンドの処理 */
10:         ...
11:     } else if (....) {
```

図 3.11: 入力されたコマンド名の判別

(07 行目)．08 行目以降は cmd_tbl[] の初期化である．cmd_tbl[] の終端を示すため，12 行目で両メンバーに NULL を設定している．

このように定義しておくと，入力されたコマンド名を判別してそれにしたがった処理を行う部分は図 3.13 のようになる．この例では，ユーザが入力した文字列を解析し，main() の引数と同じように int ac と char *av[] を設定しておくものとする．そして 07 行目の for 文で入力されたコマンド名の文字列とコマンドテーブルに登録されているコマンド名を比較し (08 行目)，一致した場合は対応する関数を呼び出している (09 行目)．入力されたコマンド名が不正の場合にはエラーメッセージを表示している (13 行目)．

```
01: void help_proc(int, char *[]), init_proc(int, char *[]), ...;
02: ...    /* 関数のプロトタイプ宣言 */
03: ...
04: struct command_table {
05:     char *cmd;                    /* コマンド名へのポインタ */
06:     void (*func)(int, char *[]);  /* 処理関数へのポインタ */
07: } cmd_tbl[] = {
08:     {"help", help_proc},          /* help コマンド */
09:     {"init", init_proc},          /* init コマンド */
10:     {"buf",  buf_proc},           /* buf コマンド */
11:     ...
12:     {NULL,   NULL}                /* cmd_tbl の終端 */
13: };
```

図 3.12: コマンドテーブル

```
01:     struct command_table *p;
02:     int ac;
03:     char *av[16];
04:     ...
05:     ... /* ユーザの入力を解析し，ac，av を設定 */
06:     ...
07:     for (p = cmd_tbl; p->cmd; p++)
08:         if (strcmp(av[0], p->cmd) == 0) { /* 文字列を比較 */
09:             (*p->func)(ac, av);           /* 処理関数の呼出し */
10:             break;
11:         }
12:     if (p->cmd == NULL)
13:         fprintf(stderr, "unknown command: %s\n", av[0]);
14:     ...
```

図 3.13: 入力されたコマンドの処理

練習問題

1. 図 3.1 のように変数 c，s，str を定義し，それぞれの変数の値を表示するプログラムを作成しなさい．このことから何が分かるか説明しなさい．

2. プログラム実行時の引数 (argc，argv) の値，および argv[] が指す文字列を表示するプログラムを作成しなさい．

3. 標準入力から 1 行の文字列を読み込み，これを解析して int argc，char *argv[] を設定する関数 void getargs(int *argc，char *argv[]) を作成しなさい．この 2 つの引数に加えて他の引数を利用してもよい．このとき，各文字列前後の余分な空白文字は省きなさい．

4. 文字列を読み込み，この文字列の長さを表示するプログラムを作成しなさい．このとき，文字列前後の空白文字は省くものとする．また，EOF が入力されるまでこの動作を繰り返すものとする．

5. 文字列と文字を読み込み，この文字列にこの文字が含まれているかを表示するプログラムを作成しなさい．ただし，EOF が入力されるまでこの動作を繰り返すものとする．

6. 10 進数を表す文字列を読み込み，この文字列を整数型に変換して表示するプログラムを作成しなさい．ただし，10 進数以外の文字列が入力されるまでこの動作を繰り返すものとする．

7. 図 3.12 と図 3.13 を参考にし，課題 1 のコマンドを受け付け，それぞれのコマンドの引数が何個必要であるかを表示するプログラムを作成しなさい．ただし，quit という文字列が入力されるまでこの動作を繰返すものとする．