16　文字・文字列の扱い

16–1　文字型 (char)・文字コード

C 言語には文字を扱うための変数型 char 型 が用意されている。char 型変数は (1byte 文字)1 文字を値として持つ。

文字定数を表すには、シングルクォーテーション ’ によって囲む。例えば、char 型の変数 a に対し、

```
a = 'A';
```

として代入を行うことが出来る。printf() や scanf() での変換指定には %c を用いる。

文字は実際には数値と一対一に対応させて数値として扱う。この対応表が**文字コード**であり、1byte 文字については現在通常 **ASCII コード**を用いている。char 型の値は内部的には 8 bit の整数値に他ならず、8 bit の数値として演算 (加減算) を行なうことができる。

あくまで文字なので、’ で 2 文字以上を囲んではいけない。

つまりドライに言えば、文字定数 ’x’ とは文字 x の文字コードの値のことであり、char 型とは 1 byte の短い整数型である。

sizeof 演算子は (定義により) char 型の大きさを 1 とした値を返す。

アスキーコード (ASCII code) 対応表

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
2	’ ’	’!’	’"’	’#’	’$’	’%’	’&’	’’’	’(’	’)’	’*’	’+’	’,’	’-’	’.’	’/’
3	’0’	’1’	’2’	’3’	’4’	’5’	’6’	’7’	’8’	’9’	’:’	’;’	’<’	’=’	’>’	’?’
4	’@’	’A’	’B’	’C’	’D’	’E’	’F’	’G’	’H’	’I’	’J’	’K’	’L’	’M’	’N’	’O’
5	’P’	’Q’	’R’	’S’	’T’	’U’	’V’	’W’	’X’	’Y’	’Z’	’[’	’\’	’]’	’^’	’_’
6	’‘’	’a’	’b’	’c’	’d’	’e’	’f’	’g’	’h’	’i’	’j’	’k’	’l’	’m’	’n’	’o’
7	’p’	’q’	’r’	’s’	’t’	’u’	’v’	’w’	’x’	’y’	’z’	’{’	’|’	’}’	’~’	del

例えば、’A’ の ASCII 文字コードは、十六進で 41、即ち $4 \times 16 + 1 = 65$ であり、’m’ の ASCII 文字コードは、十六進で 6D、即ち $6 \times 16 + 13 = 109$ である。一番最初の ’ ’ は空白文字、一番最後の del は削除を表す特殊文字。

十六進表示を明示するのに、0x を頭に付けて表すことがある。

0x41 = 65

表に載っていない 0〜31 の値は、「改行」「タブ」や装置制御などの特殊な意味を持つ値で、これも文字扱いだが、通常の意味での (文字としての) 表示は出来ない。又、128 以上の値は文字としての割り当てがない。

アスキーコード (ASCII code) 対応表

32=’ ’	33=’!’	34=’"’	35=’#’	36=’$’	37=’%’	38=’&’	39=’’’
40=’(’	41=’)’	42=’*’	43=’+’	44=’,’	45=’-’	46=’.’	47=’/’
48=’0’	49=’1’	50=’2’	51=’3’	52=’4’	53=’5’	54=’6’	55=’7’
56=’8’	57=’9’	58=’:’	59=’;’	60=’<’	61=’=’	62=’>’	63=’?’
64=’@’	65=’A’	66=’B’	67=’C’	68=’D’	69=’E’	70=’F’	71=’G’
72=’H’	73=’I’	74=’J’	75=’K’	76=’L’	77=’M’	78=’N’	79=’O’
80=’P’	81=’Q’	82=’R’	83=’S’	84=’T’	85=’U’	86=’V’	87=’W’
88=’X’	89=’Y’	90=’Z’	91=’[’	92=’\’	93=’]’	94=’^’	95=’_’
96=’‘’	97=’a’	98=’b’	99=’c’	100=’d’	101=’e’	102=’f’	103=’g’
104=’h’	105=’i’	106=’j’	107=’k’	108=’l’	109=’m’	110=’n’	111=’o’
112=’p’	113=’q’	114=’r’	115=’s’	116=’t’	117=’u’	118=’v’	119=’w’
120=’x’	121=’y’	122=’z’	123=’{’	124=’|’	125=’}’	126=’~’	127=(del)

```
[ 32- 63]   ! " # $ % & ’ ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
[ 64- 95] @ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
[ 96-127] ‘ a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ (del)
```

注. printf などで改行するときに用いてきた \0 は、「改行」という制御文字を表している。文字定数として改行文字を表すには ’\0’ とすればよく、傍注にあるように、ASCII コードではその値は 0x0A (=10) である。

しばしば現れる特殊文字の文字コード:
0x08: 後退 (BS=BackSpace)
0x09: 水平タブ (HT=HorizontalTab)
0x0A: 改行・行送り (LF=LineFeed)
0x0D: 行頭へ復帰 (CR=CarrigeReturn)

1 文字は 1 byte = 8 bit であるが、実際には 0〜127 なので 7 bit しか使っていない。それ故、既存のプログラムには、文字の先頭 bit は 0 であると仮定しているものがある。
一方、これでは文字数が足りない日本では、この空いた 1 bit を使って、文字コード 128 以上の値を用いてカタカナなどを表そうという動きが生じた。これがいわゆる「半角カタカナ」である。
文字の先頭 bit が 0 だと仮定しているプログラムが、先頭 bit が 1 の半角カタカナの入力を受けると、当然ながら誤動作を起こす危険がある。特にネットワークを介したやりとりでは、途中で多くの機械を経由するので、その何処かで誤動作が発生すると正常なやりとりが出来なくなるばかりか、受け取った相手の環境によっては思わぬ事態 (ハングアップとか) を引き起こす場合がある。従って、特別に許された環境以外では、いわゆる半角カタカナは使ってはいけ**ない**。

実習 16.1. 入力した 1 文字が十進数字 ('0'〜'9') なら数値に変換して表示し、そうでなければその旨を表示する。

「数字」と「数値」との違いを認識せよ。

文字型 char に対する printf・scanf の変換指定は %c である。

```
/*   ctoi.c   2010-07-05   */
#include <stdio.h>

int main(int argc, char** argv)
{
    char c;
    int i;

    printf("decimal digit? ");
    scanf("%c", &c);

    if( c >= '0' && c <= '9' )
    {
        i = c - '0';
        printf("'%c' is a decimal digit with value %d.\n",c,i);
    }
    else
    {
        printf("'%c' is not a decimal digit.\n", c);
    }
}
```

注 . 上では、十進数字 '0'〜'9' が順に並んでいることを仮定している。従って文字コードに依存するプログラムだが、通常は ASCII コードなので、この仮定は成立している。この程度は仮定しても良いだろう。

考察 16.1.1. 上の仮定の下に、('0' の実際の値は知らなくても)

```
i = c - '0';
```

で所要の計算が出来る理由は？

16–2　文字列

C 言語には固有の「文字列型」は用意されておらず、文字型 char の配列として扱う。文字列の終わりを示す目印として null 文字 '\0' が用いられ、配列の先頭から見ていって '\0' までが一つの文字列として扱われる。先頭から '\0' の手前までの文字数を文字列の長さという。

文字列 = 文字の配列
こう書くとそのまま当たり前。

文字列定数はダブルクォーテーション " によって囲んで表す。 この場合、'\0' が終端に自動的に付加されてメモリ上に格納される。

null 文字 '\0' は文字コード 0 の文字だが、そんなことは知らなくても、原理的にはよろしい。

文字列変数は通常の配列と同様に、

```
char str[100];
```

のようにすれば、str が 100 個の要素を持つ char 型の配列として宣言される。宣言時に

```
char str[100] = "This is a pen.";
```

のように初期値を代入することが出来るが、自動的に付加される '\0' を含めた要素数 (文字列の長さ +1) を確保しておく必要がある。(逆に言うと、格納できる文字列の最大文字数は**宣言した配列の要素数** −1 となるということである。) 上の例のように余分に大きめに確保するのは構わない。確保した領域の途中に '\0' が現れれば、先頭からそこまでが一つの文字列となる。また、

1 文字でも " で囲めば、「1 文字から成る文字列」であって、' で囲んだ文字定数とは異なる。

"" は空文字列 (長さ 0 の文字列) を意味する。先頭にいきなり '\0' が来た形。

```
char str[] = "This is a pen.";
```

のように配列の大きさを指定せずに初期化することも出来て、この場合は、(文字列の長さ +1) つまり '\0' を含めた領域 (この例では $14 + 1 = 15$ 文字分) が確保される。従って、

一般に配列変数も宣言時に初期化できる。

```
char str[15] = "This is a pen.";
```

と等価。

実の所、純粋にユーザとしてなら、テキスト処理は sed, awk, perl などの言語を用いる方が手軽で便利なので、C 言語で文字列の複雑な処理をする機会は少ないであろう。

注． char 型配列に 1 文字ずつ文字を代入していって、それを文字列として扱いたいという場合には、最後に自分で '\0' を付ければ良いが、文字列を扱うための標準的に用意されている関数を使うと、それを自動で行なってくれるので、その方が安全で良い。

printf() や scanf() での変換指定には %s を用いる。

```
printf("%s", str);
scanf("%s", str);
```

のように、値を表示・代入するものとして文字列名 (char 型の配列名) を指定すれば良いのだが、これはつまり char 型へのポインタということ。一般に変換指定 %s に対応する型は char 型へのポインタであって、printf() ではそこから読んでいって '\0' の直前までを表示し、scanf() ではそこから書き込んでいって最後に自動で '\0' を付ける。

注 . 例えば、

```
char str[100];
```

として、str が char 型の配列として宣言されているとき、(初期化時以外の) プログラム中で

```
str = "This is a pen.";
```

というように代入演算子 = によって文字列を代入することは**出来ない**。

一般に配列名に値を代入することは出来ない。

では、プログラム中での代入などの操作はどうするかと言うと、strcpy() など用意された一連の関数を用いる。これらの関数はヘッダファイル string.h を読込んで用いる。いろいろあるがマニュアル等を参照されたい。

文字列の長さ: strlen()
文字列コピー: strcpy()
文字列の結合: strcat()
文字列の比較: strcmp()
など。

課題 12 (✓切 7/11(日)). 数字列を数値 (int 型) に変換する関数 myatoi() を作成し、動作例のプログラムを書いて動作を確認せよ。(符号対応・十六進版など拡張版も出来れば作ってみよ。)

注 . 次頁の注にあるように、標準入力からの文字列の取得には scanf() でなく fgets() を用いよ。

```
fgets(buf,LEN,stdin);
```

は、標準入力 stdin を読込んで、文字列終端文字 '\0' も入れて指定文字数 LEN になるまで、又はその前に改行文字 '\n' が現れるまで、文字列変数 buf に書込む。改行文字 '\n' を読んだ場合は'\n' も書込まれる (つまり '\n' '\0' で終わる)。従って、文字列の終了判定は「'\0' または '\n'」とすると良い。

尚、fgets() は、標準入力 stdin から以外に、データファイルから入力を取ることも出来る (後述)。

文字数限定版 (最初の n 文字のみ) の strncpy(), strncat(), strncmp() なども安全性などから有用。

stdin: 標準入力 (STanDard INput)

注． キーボード等から実行時に利用者の入力を取るプログラムは、予期せぬ (又は悪意の) 入力があり得るので不用意に書くと危険なことがある。実は `scanf()` もその一つで、書込む配列の大きさのチェックをしないので、(不用意に又は故意に) 長過ぎる文字列の入力を受けた時に、配列として確保した領域を越えて書込んでしまい、予期せぬ動作を引起こすことがある。これを意図的に狙うのが **buffer overflow** (**buffer overrun**) と呼ばれる攻撃である。

このようなセキュリティホールを避けるには、文字数限定の文字列入力関数として `fgets()` を用いるなどの方法がある。

```
/* bof.c : buffer overflow sample */
#include <stdio.h>

#define MAXLEN 10

int main(int argc, char** argv)
{
    unsigned int x=1;
    char str[MAXLEN]="sample...";

    printf("x: %p, %d, %x\n", &x, x, x);
    printf("s: %p, %s\n", str, str);

    scanf("%s",str);

    printf("x: %p, %d, %x\n", &x, x, x);
    printf("s: %p, %s\n", str, str);
}
```

```
/* bofsafe.c : buffer overflow safety */
#include <stdio.h>

#define MAXLEN 10

int main(int argc, char** argv)
{
    unsigned int x=1;
    char str[MAXLEN]="sample...";

    printf("x: %p, %d, %x\n", &x, x, x);
    printf("s: %p, %s\n", str, str);

    fgets(str, MAXLEN, stdin);

    printf("x: %p, %d, %x\n", &x, x, x);
    printf("s: %p, %s\n", str, str);
}
```

メモリ上での変数の配置に注意しつつ、長過ぎる入力に対する動作を比較せよ。