プログラミングおよび実習II・第 8 回

2003 年 11 月 12 日

1 文字列比較による整列

第 6 回の「実力チェック」では、標準入力 (キーボード) から点数と名前の組を読み込み、それを点数の小さいものから順に出力するプログラムを作りました。今回は、同じ入力に対して、名前がアルファベット順になるように出力するプログラム prog8-1.c を作成してみましょう。このプログラムの実行例は次のようなものになります。

prog8-1.c の実行例

```
s1542h017% ./prog8-1
 1番目の点数と名前 => 35 Hiroshi
 2番目の点数と名前 => 23 Koji
 3番目の点数と名前 => 67 Satoshi
 4番目の点数と名前 => 87 Junko
 5番目の点数と名前 => 64 Ichiro
 6番目の点数と名前 => 89 Mari
 7番目の点数と名前 => 73 Daisuke
 8番目の点数と名前 => .
 73 Daisuke
 35 Hiroshi
 64 Ichiro
 87 Junko
 23 Koji
 89 Mari
 67 Satoshi
s1542h017%
```

整数や実数のデータの場合は、C の演算子 < や <= 等を使って簡単に比較することができますが、2 つの文字列の比較 (どちらがアルファベット順で先にあるべきかの判定) をする演算子は C には用意されていません。ここでは、文字コードの大小関係に基づいて 2 つの文字列の比較を行う標準ライブラリ関数 strcmp を使うことにします。

1.1 文字列の比較

C の標準ライブラリ関数の 1 つである strcmp は

```
int strcmp(const char *s1, const char *s2);
```

と宣言される標準ライブラリ関数で、前回に紹介した関数 strcpy の仲間です。strcmp を利用する際にも、やはりプログラムの冒頭部分に

```
#include <string.h>
```

の 1 行が必要です。2 つの文字列 (それぞれ先頭文字が格納されているアドレス) s1 と s2 を引数として呼び出すと、関数 strcmp は、この 2 つの文字列中の文字コードを 1 文字 (1 バイト) ずつ比較して、比較の結果を次のような int 型の返り値として返します。

辞書式の順で s1 が s2 より前の場合	···	負の値
s1 と s2 が同じ文字列の場合	···	0
辞書式の順で s1 が s2 より後の場合	···	正の値

関数 strcmp は、文字列中の文字の文字コード (char 型) を整数として比較しますから、2 つの文字のどちらが前となるかは、その 2 つの文字の文字コードのどちらが小さいのかによって決まります[1]。たとえば、 ASCII コードの文字列の場合、関数 strcmp は次のような返り値を返します。

```
strcmp("abc", "abc")  ==  0
strcmp("abc", "xyz")   <  0
 strcmp("ab", "abc")   <  0
strcmp("abc", "abd")   <  0
  strcmp("a", "ABC")   >  0
```

最後の例で "a" より "ABC" の方が前となっているのは、 ASCII コードでは a の文字コードより A の文字コードの方が小さいからです[2]。

1.2 実力チェック

前回の「付録 B」の prog6-2.c のプログラム例での関数 quicksort の定義を変更して、標準入力 (キーボード) から、点数と名前の組を次々と読み込み、読み込まれたデータの組を、名前が ASCII コードでの辞書式順になるように出力するプログラム prog8-1.c を作りなさい。ただし、解答用紙のプログラム名は「prog8-1.c」とし、冒頭に「#include <string.h>」の 1 行が追加されているものとして、解答用紙には関数 quicksort の (変更後の) 定義のみを書きなさい。

2 要素の個数によって整列に要する時間がどう変るかを調べる

この節では、前回の prog7-2.c で計測した整列に必要な時間が、整列を行う配列の長さ (要素数) に応じてどのように変化するか調べる方法について考えます。 1-542 や 1-609 実習室では 100 分の 1 秒単位でしか時間を計ることができないため、目標の精度 (たとえば、誤差 ±1% 以内) を得るためには、何度も整列を繰り返して十分な累積時間となるようにしなければなりません。このため、前回のプログラム prg7-2.c では、実際に実験を行ってみて、十分な累積時間となるように繰り返しの回数を調節する必要がありました。

配列の長さを変えると、 1 回の整列に必要な時間も変化し、それとともに一定の実験精度を得るために必要な繰り返しの回数も変ってきます。配列の長さを変える度に、予備実験を行って何回繰り返しを行えばよいかを決めなければならないとなると、このような実験をするのは結構な労力が必要となります。そこで、以下のような実現の方針とアルゴリズムで、必要な実験精度に必要な累積時間をプログラムで自動的に確保することにしましょう。

実現の方針 (1) 時間計測のプログラム中で、まず、つぎのような予備実験を行う。数秒間 (たとえば 3 秒間)、データの初期化と整列を繰り返してみて、データの初期化と整列を行うのに費される時間のおおよその比率を計測する。一定の時間だけデータの初期化

[1] つまり、処理系 (C のコンパイラおよびコンパイルしてできたオブジェクトプログラムが実行される計算機環境) で採用されている文字コードに依存します。

[2] 付録 A. の表 1 を参照してください。 ASCII コードの場合、大文字小文字の違いを無視して比較を行う関数 strcasecmp も存在します。日本語を含む各国語での比較 (たとえば、日本語のあいうえを順など) に、strcmp や strcasecmp を使うことはできません。

と整列を繰り返すためには、初期化と整列を 1 回行う度に関数 clock_time を呼び出して、返り値が目標の時間に達するまで、これを繰り返すようにすればよい。

(2) データの初期化と整列に費される時間のおおよその比率が分かったら、目標の実験精度を得るために必要な累積時間をそこから算出する。たとえば、 $\pm 1\%$ 以内の誤差にしたいのなら、 (データの初期化を含まない) 純粋に整列に費される時間が 10 秒程度になるような (データの初期化を含めた) 全体の時間を算出する。

(3) (2) で算出された時間が経過するまでデータの初期化と整列を繰り返すような本実験を行う。

(4) (1) と (3) では、ともに、一定の時間が経過するまでデータの初期化と整列を繰り返して、繰り返しの回数、データの初期化に費した時間、データの初期化と整列に費した時間の 3 つを計測することになるので、これを 1 つのサブルーチン (関数) として用意しておく。

アルゴリズム (1) 「実現の方針」の (1) や (3) の仕事を行うサブルーチンとして関数 checktime を定義しておく。checktime は引数で指定された個数の要素の整列を、やはり引数で指定された時間が経過するまで繰り返し、その繰り返しの回数、データの初期化に費した時間、データの初期化と整列に費した時間の 3 つを構造体型の返り値として返す。関数 checktime は次のような手順でこれを行う。

(1-1) 配列中の要素数 num (int 型)、整列を繰り返す秒数 sec (double 型) の 2 つの引数を受け取る。

(1-2) 十分な長さを持った Record 型の配列 data と、Record 型へのポインタ型の配列 dp を用意しておく。

(1-3) 配列 data の先頭から num 個の要素の name の部分 (メンバ) を、適当な文字列 (たとえば "Test") で初期化する。

(1-4) 関数 clock_reset を呼び出して、時計を 0 へ戻す。

(1-5) 関数 clock_on を呼び出して、時計をスタートさせる。

(1-6) 手順 (1-6-4) での clock_time 返り値が sec を越えるまで、次の 4 つの手順を繰り返す。この時、繰り返した回数を数えておく。

(1-6-1) 配列 data の先頭から num 個の要素の score の部分 (メンバ) を、乱数 (rand() の値) で初期化する。

(1-6-2) 配列 dp の先頭から num 個の要素を、配列 data 中の対応する要素のアドレスで初期化する。

(1-6-3) 関数 quicksort を呼び出して、配列 dp の先頭から num 個の要素を、各要素が指す構造体の score の部分が昇順となるように整列させる。

(1-6-4) 関数 clock_time を呼び出す。

(1-7) 関数 clock_off を呼び出して、手順 (1-6) に費した時間を記憶しておく。

(1-8) 関数 clock_reset を呼び出して、時計を 0 へ戻す。

(1-9) 関数 clock_on を呼び出して、時計をスタートさせる。

(1-10) 手順 (1-6) での繰り返しの回数と同じ回数だけ次の 2 つの手順を繰り返す。

(1-10-1) 配列 data の先頭から num 個の要素の score の部分 (メンバ) を、乱数 (rand() の値) で初期化する。

(1-10-2) 関数 clock_time を呼び出す[3]。

(1-11) 関数 clock_off を呼び出して、手順 (1-10) に費した時間を記憶しておく。

(1-12) 次のような構造体型 (ExpResult 型) のデータを、関数 checktime の返り値として返す。

```
typedef struct {
    int repeat; /* 整列を繰り返した回数 */
    double t1;  /* データの初期化に要した時間(秒) */
    double t2;  /* データの初期化と整列に要した時間(秒) */
} ExpResult;
```

ただし、repeat は手順 (1-6) での繰り返しの回数であり、t1 は、手順 (1-10) に費した時間、t2 は、手順 (1-6) に費した時間である。

(2) ある個数の要素を整列するの必要な時間を計測したい場合、その個数 num と (たとえば) 3.0 を引数として関数 checktime を呼び出し、その返り値を ExpResult 型の変数 r に格納する。

(3) つぎに、同じ num と (たとえば) 10.0*r.t2/(r.t2-r.t1) を引数として、もう 1 度関数 checktime を呼び出し、その返り値を変数 r に格納する[4]。

(4) r.repeat 回のデータの初期化と整列に要した時間が r.t2、同じ回数のデータの初期化に要した時間が t1、(ここから算出された) 整列を 1 回行うのに要した平均時間が (r.t2-r.t1)/r.repeat であったことを出力する。

3 コマンドライン引数と関数 main の引数

Unix (Linux) や Windows 等の OS 上の C プログラムの実行環境では、プログラムを起動した際にコマンドラインで与えた引数 (文字列) が何であったかを、関数 main の引数として知ることができるようになっています。整列する要素の個数など、パラメータをいろいろと変えて実験を行いたい場合は、このコマンド引数を使って指定できるようにしておくと便利です。

プログラムを起動すると、関数 main の第 1 引数には「コマンドライン引数の個数 + 1」が int 型の値として渡されます。また、main の第 2 引数には、文字列 (char * 型の値) を要素とした配列の先頭要素のアドレス (char **型) が渡されます。この配列は「コマンドライン引数の個数 + 2」個の要素からなっており、その先頭要素 (添字 0) はプログラムが起動されたときの「プログラム (コマンド) 名」、添字 1 の要素はコマンドラインでの第 1 引数、添字 2 の要素はコマンドラインでの第 2 引数、 ... となります。配列の最後の要素には (アドレスの) 0 が格納されています。

[3] この呼び出しは、手順 (1-6-4) に要した時間を再現するために行っています。clock_time の返り値は使用しません。

[4] ここでは、 ±1% 以内の誤差にすることを目標に、整列のみに要した累積時間が 10 秒となるように設定しています。

この様子は、次のようなプログラム args.c を実行してみるとよく分かります。

args.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main(int argc, char *argv[])
 5 {
 6     int i;
 7
 8     printf("argc = %d\n", argc);
 9     for (i = 0; i < argc; i++)
10         printf("argv[%d] = %s\n", i, argv[i]);
11     return EXIT_SUCCESS;
12 }
```

args.c のコンパイルと実行例

```
s1542h017% cc -o args args.c
s1542h017% ./args
argc = 1
argv[0] = ./args
s1542h017% ./args -hello
argc = 2
argv[0] = ./args
argv[1] = -hello
s1542h017% ./args This is a test.
argc = 5
argv[0] = ./args
argv[1] = This
argv[2] = is
argv[3] = a
argv[4] = test.
s1542h017%
```

もう 1 つ、main の引数を利用したプログラムをみてみましょう。このプログラムは、コマンドライン引数として与えられた (日本語風の) 四則演算を計算して、その結果を表示するものです。

calc.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <string.h>
 4
 5 int main(int argc, char *argv[])
 6 {
 7     int a, b;
 8
 9     if (argc < 4) {
10         printf("引数の数が足りません\n");
11         return EXIT_FAILURE;
12     }
13     if (sscanf(argv[1], "%d", &a) != 1) {
14         printf("第1引数が整数ではありません\n");
15         return EXIT_FAILURE;
16     }
17     if (sscanf(argv[3], "%d", &b) != 1) {
18         printf("第3引数が整数ではありません\n");
19         return EXIT_FAILURE;
```

```
20      }
21
22      if (strcmp(argv[2], "足す") == 0)
23          printf("%d + %d = %d\n", a, b, a+b);
24      else if (strcmp(argv[2], "引く") == 0)
25          printf("%d - %d = %d\n", a, b, a-b);
26      else if (strcmp(argv[2], "掛ける") == 0)
27          printf("%d * %d = %d\n", a, b, a*b);
28      else if (strcmp(argv[2], "割る") == 0)
29          printf("%d / %d = %d ... %d\n", a, b, a/b, a%b);
30      else {
31          printf("演算の種類が理解できません\n");
32          return EXIT_FAILURE;
33      }
34
35      return EXIT_SUCCESS;
36  }
```

calc.c の実行例

```
s1542h017% ./calc 23 足す 5
23 + 5 = 28
s1542h017% ./calc 23 掛ける 5
23 * 5 = 115
s1542h017% ./calc 23 割る 5
23 / 5 = 4 ... 3
s1542h017% ./calc 23 引く 5
23 - 5 = 18
s1542h017% ./calc 23引く5
引数が足りません
s1542h017%
```

コマンドライン引数はスペースやタブ文字で区切られますから、「足す」や「割る」の両側にはこれらの区切り記号が必要となります。

4 演習問題

以下の 2 つの演習問題に取り組みなさい。プログラムが完成したら、これまで通り mprog2 コマンドを実行して、自分が作成したプログラムを提出してください。今回の演習問題については、いずれも教員や TA のチェックは必要ありません。

4.1 prog8-1.c

「1.2 実力チェック」で作成した関数 quicksort を用いて、「1. 文字列比較による整列」の節のプログラム prog8-1.c を完成しなさい。

4.2 prog8-2.c

まず、前回の演習問題で作成したプログラム prog7-2.c を prog8-2.c にコピーしなさい。つぎに、この prog8-2.c を、コマンドライン引数で指定された個数 (最大 100000 個) の要素を整列するのに要した (平均) 時間を、次の実行例のような形式で表示するプログラムに変更しなさい。ただし、整列 1 回の平均所要時間 (秒) は小数点以下 10 桁まで表示しなさい。また、「2. 要素の個数によって整列に要する

時間がどう変るかを調べる」の節の「実現の方針」や「アルゴリズム」に基づいて、測定誤差が ±1% 以内になるように (データの初期化を含まない) 整列に費された累積時間が自動的に 10 秒程度になるようなプログラムにしなさい。

prog8-2.c の実行例

```
s1542h017% ./prog8-2
要素数が指定されていません。
s1542h017%  ./prog8-2 0
要素数の指定が正しくありません。
s1542h017% ./prog8-2 10
10個の要素の整列に要した時間（15.550 - 5.770）/ 8356128 = 0.0000011704 秒
s1542h017% ./prog8-2 100
100個の要素の整列に要した時間（11.910 - 1.810）/ 601463 = 0.0000167924 秒
s1542h017% ./prog8-2 1000
1000個の要素の整列に要した時間（11.240 - 1.280）/ 47456 = 0.0002098786 秒
s1542h017% ./prog8-2 10000
10000個の要素の整列に要した時間（11.370 - 1.400）/ 2458 = 0.0040561432 秒
s1542h017% ./prog8-2 100000
100000個の要素の整列に要した時間（11.530 - 1.570）/ 119 = 0.0836974790 秒
s1542h017% ./prog8-2 1000000
要素数の指定が正しくありません。
s1542h017%
```

コマンドライン引数の取り扱いについては「3. コマンドライン引数と関数 main の引数」の節を参考にしてください。余裕のある受講者は、構造体そのものを整列するプログラム prog7-3.c についても、同様の方法で時間測定を行うプログラム prog8-3.c を作成して、整列時間を prog8-2.c の場合と比較してみましょう。

プログラミングおよび実習Ⅱ・第 8 回・終り

付録 A. 文字と文字コード

コンピュータの中で文字が扱われるときには、異なる文字に異なる整数を割り当てておき、文字の違いを整数の違いとして扱います。それぞれの文字に割り当てられた整数を、その文字の**文字コード**と呼びます。それぞれの文字に、ある割り当て方に基づいて整数を割り当てることを文字の**符号化**と呼びます。たとえば、 ASCII コードと呼ばれる文字コード (符号化) では、英字や数字等の文字に 0 から 127 までの整数を表 1(次ページ) のように割り当てています。

ASCII コードは、コンピュータで文字情報が扱われるときの最も基本的な文字コード (の割り当て方) の規格です。 ASCII コードは、米国で決められた規格ですので、英字や数字、限られた記号などしか表現することができませんが、日本を含め他の地域で使われる文字の文字コードも、多くの場合、この ASCII コードを拡張あるいは若干変更する形で規格が定められています。

ASCII コードでの文字コードが 0 から 31 までと 127 の文字は**制御文字**と呼ばれ、目に見える文字を表すのではなく、通信や表示 (印刷) の書式等を制御するために用いられます。これに対して、英字や

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
SP	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

(右肩の整数が文字コード)

表 1 ASCII コード

数字などの目に見える文字を**図形文字**と呼びます。文字コード 32 の SP はいわゆる「間隔[5] (Space)」を表すもので、画面には何も表示されずにカーソルが 1 文字分進みます。SP は制御文字に分類されることも図形文字に分類されることもあります。

33 から 126 までの文字コード (と図形文字の対応関係) は「ISO 646-US」と呼ばる標準規格となっています。日本では、この「ISO 646-US」の代わりに、「ISO 646-JP」あるいは「JIS X 0201 Roman」と呼ばれる規格の文字コードが用いられることも多くあります。92 と 126 の 2 つの文字コードを除けば、ISO 646-US (ASCII コード) と ISO 646-JP (JIS X 0201 Roman) との 2 つの規格の間に違いはありません。ISO 646-JP (JIS X 0201 Roman) では、92 は「¥」(円の通貨記号) に、126 は「‾」(上線) になっています。使用しているコンピュータの環境によって、\ (バックスラッシュ) が ¥ で表示されたり、~ (チルド、波線) が ‾ で表示されたりするのはこのためです。

これらの図形文字の他に、C によるプログラミングでは、次のような制御文字がよく用いられます。

略号	英名	和名	C での表記[6]	意味や用途
NUL	Null	ナル[7]	\0	C 言語で文字列の終端を表すために使用されます。
BEL	Bell	ベル	\a	印刷装置や表示装置のベルを鳴らすなどして、注意を喚起します。
BS	Backspace	後退	\b	カーソルを (同一行内で)1 文字分戻します。
HT	Horizontal Tabulation	水平タブ	\t	カーソルを同一行内で次の (あらかじめ決められた) タブ位置 (通常 8 文字間隔) まで進めます。
LF	Line feed[8]	改行	\n	C 言語ではカーソルを次の行の行頭へ移動します。一般には、単にカーソルの 1 行分下への移動を表すこともあります。
VT	Vertical Tabulation	垂直タブ	\v	カーソルを次の (あらかじめ決められた) タブ行まで下に進めます。
FF	Form Feed	改ページ[9]	\f	印刷位置を次のページに進めたり、現在の画面を消去してカーソルを初期位置に戻したりします。
CR	Carriage Return	復帰	\r	カーソルを行頭に移動します。

表 2 よく使われる制御文字

[5]空白と呼ばれることもよくあります

printf による 1 byte 文字の入力

ASCII コードのように、1 byte (= 8 bit) の整数で表現できる文字を関数 printf で出力する場合は、printf の出力書式文字列の中に %c という**変換仕様**を置き、その位置に埋め込みたい文字の文字コード (整数) を、対応する実引数として printf に渡します。たとえば、次のプログラム ascii.c は、32 から 126 までの文字コードと文字の対応関係を 1 行に 8 文字ずつ出力するものです。

ascii.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main()
 5 {
 6     int c;
 7
 8     for (c = 32; c <= 126; c++) {
 9         printf("%3d=%c", c, c);
10         if (c%8 == 7 || c == 126)
11             printf("\n");
12         else
13             printf(" ");
14     }
15
16     return EXIT_SUCCESS;
17 }
```

プログラム 8 行目の printf の呼び出しでは、%3d と %c のどちらの変換仕様も、変数 c の値に基づいた出力をさせていますが、%3d では c の値の十進数表記 (いくつかの数字の並び) が、%c では c の値を文字コードとする 1 つの文字が出力されることに注意してください。このプログラムをコンパイルして実行すると次のようになります。

ascii.c の実行例

```
s1542h017% cc ascii.c -o ascii
s1542h017% ./ascii
 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=~
s1542h017%
```

---

[2] 8 – 10ページの 4.2 節「文字定数」を参照してください

[3]「空白」と呼ばれることもありますが、SP (スペース) と混同しやすいので「ナル」と呼びましょう

[4]「New Line」とも呼ばれます

[5]「書式送り」とも呼ばれます

文字定数

C のプログラム中で、特定の文字の文字コードを利用したい場合、その文字を「'」(一重引用符) で囲むことによって、囲まれた文字の文字コード (int 型の整数値) を表すことができます。たとえば、C のプログラム中で 'A' と書くと、英文字 A の文字コード (ASCII コードが使用されている場合は 65) を表します。このような表記によるプログラム中の定数を**文字定数**と呼びます。このとき、8 – 8ページの表 2 にあるような制御文字は、表中の「C での表記」を「'」で囲み、たとえば、改行文字の文字コードは '\n' のように書きます。「'」という文字 (一重引用符) 自体の文字コードは '\'' で、「\」(バックスラッシュ) の文字コードは '\\' で表すことができます。

ASCII コードが使われている場合なら、プログラム ascii.c は次のように書いても同じように動作します。

ascii2.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main()
 5 {
 6     int c;
 7
 8     for (c = ' '; c <= '~'; c++)
 9         printf("%3d=%c%c", c, c, (c%8 == 7 || c == '~') ? '\n' : ' ');
10     return EXIT_SUCCESS;
11 }
```

char 型

ASCII コードで符号化された文字は 0 から 127 の整数で表現されますが、C には、この範囲の整数を扱うのに都合のよい char 型というデータ型が用意されています。char 型のデータは 1 byte (= 8 bit) で表現できる符号付きあるいは符号なし[10]の整数です。符号付きの場合は −128 から 127 まで[11]、符号なしの場合は 0 から 255 までの範囲の整数を表現することができ、整数に関する四則演算等を char 型のデータに対しても行うことができます。ただし、式の中に現れた char 型のデータは、まず int 型に変換されてから計算が行われます。

char 型の変数

char 型のデータを格納する変数は、int 型の変数を宣言するのと同様に、C の予約語 char を使ってつぎのように宣言します。

```
char ch;
```

1 つの char 型の変数は、1 つの char 型のデータ (整数) を記憶することができます。記憶できる数値の範囲が狭いことを除けば、int 型の変数と全く同じように代入したり参照したりすることが可能です。

たとえば、先のプログラム ascii.c や ascii2.c の 5 行目の「int c;」は「char c;」としても全く同じように動きます。これは、変数 c で扱う範囲の整数 (文字コード) が char 型で十分表現可能で

[10]符号付きか符号なしかは、C の処理系によって違っています

[11]C の規格上は −127 から 127 までとなっている可能性もあります

あり、また、関数 printf の第 2 引数以降に char 型のデータが現れた場合、ちょうど式の計算の中で char 型のデータが int 型へ変換されてから計算が行われるのと同じように、まず int 型に変換されてから関数 printf へ渡されるからです。

scanf による 1 byte 文字の入力

関数 scanf を使って、1 byte の整数で表現された文字を入力することもできます。入力書式文字列の中には %c という変換仕様を書きます。ちょうどその変換仕様の位置に現れた (入力された) 文字の文字コードが、対応する (scanf の) 実引数で指定された変数等に格納されます。変換仕様 %c に対応する実引数は char 型へのポインタ (char 型のデータの記憶領域のアドレス) でなければなりません。

次は、入力された行中の文字の文字コードをすべて表示するプログラム chars.c とその実行例です。プログラムの 9 行目では、関数 scanf に char 型のデータの格納場所 (アドレス) を渡さなければなりませんので、5 行目の「char c;」を「int c;」とすることはできません。

chars.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int main()
 5 {
 6     char c;
 7
 8     printf("文字列を入力してください = ");
 9     do {
10         scanf("%c", &c);
11         printf("%3d=%c\n", c, c);
12     } while (c != '\n');
13     return EXIT_SUCCESS;
14 }
```

chars.c の実行例

```
s1542h017% ./chars
文字列を入力してください = This is a test.
 84=T
104=h
105=i
115=s
 32=
105=i
115=s
 32=
97=a
 32=
116=t
101=e
115=s
116=t
 46=.
 10=

s1542h017%
```

プログラム chars.c の実行例で、キーボードからの入力がすべて完了してからプログラムの出力が始まっているのは、改行 (Enter) キーが押されるまでは、キーボードから入力された文字列は実行中のプ

ログラムに渡されないからです。chars の実行が開始されると、7 行目の printf の呼び出しで「文字列を入力してください =」という出力を行った後、改行 (Enter) キーが押されるまでは、9 行目で始めて関数 scanf が呼び出された状態で停止しています。ユーザーが「This is a test.」に続いて、改行 (Enter) キーを押したときに始めて、T から改行文字までの 16 文字をプログラムが読み取ることが可能となり、scanf は先頭の文字 T の文字コードを変数 c に格納して、関数 main に復帰します。それ以降の scanf の呼び出しはプログラムを停止させる事なく直ちに完了し、入力された文字が順に処理されていきます。

関数 scanf の変換仕様として、%d、%f、%lf など、数値を読み取るものを使用すると、 入力された数値に先行する空白類 (スペース、水平タブ、改行等) は読み飛ばされ (無視され) ますが、%c の場合は空白類も 1 つの文字として、その文字コードが変数等に格納されることに注意してください。

空白類を無視して文字を入力したい場合は、%c の前に空白を置いて、入力書式文字列を " %c" のようにします。入力書式文字列に空白が現れている場合、scanfは、その位置に現れた (入力された) 空白類をすべて読み飛ばします。

付録 B. 前回の演習問題のプログラム例

prog7-1.c の関数 quicksort の定義

```
 1 int quicksort(Record *d[], int h, int t)
 2 {
 3     Record *tmp;
 4     int r;
 5     int i, j, count = 0;
 6
 7     if (h >= t)
 8         return count;
 9
10     r = d[(h+t)/2]->score;
11     i = h;
12     j = t;
13     while (1) {
14         count++;
15         while (d[i]->score < r) {
16             i++;
17             count++;
18         }
19         count++;
20         while (r < d[j]->score) {
21             j--;
22             count++;
23         }
24         if (j <= i)
25             break;
26         tmp = d[i];
27         d[i++] = d[j];
28         d[j--] = tmp;
29     }
30     count += quicksort(d, h, i-1);
31     count += quicksort(d, j+1, t);
32     return count;
33 }
```