プログラミング及び実習2

第 4 回 文字型、書式付き入出力、繰り返し処理、define マクロ、
標準ライブラリ

佐野 彰

全角文字に注意！

```
foo.c:4:1: error: stray ‘\343’ in program
foo.c:4:2: error: stray ‘\200’ in program
```

- `\343` や `\200` のような英数字で表せないコードが出るエラーの多くは、**全角の日本語文字**（全角の空白・セミコロン・クォーテーションなど）が原因です
- macOS（clang）では `warning: treating Unicode character as whitespace` と、より分かりやすい警告が出ます

本日の内容

1. 文字型 `char`
2. 書式付き入出力関数 `printf()`, `scanf()`
3. 繰り返し処理（`while`, `do〜while`, `for`, `break`, `continue`）
4. `#define` 定数マクロ定義
5. `stdlib.h` と `math.h`

1. 文字型 char

char 文字型（character type）

`char` は 1 バイト（8 ビット）の**整数型**です。文字型という名前ですが、中身は小さな整数です。

計算機は文字を画像のまま扱わず、文字ごとに番号を決めた対応表（**文字コード**）を使います。`char` 型の変数には、文字そのものではなく、その文字の番号が入ります。

- ASCII（アスキー）：7 ビットの整数 0〜127 に、英数字・記号・制御文字を対応付けた文字コード。C 言語のプログラムで扱う英数字は、ふつう ASCII で表します
- 日本語には UTF-8, Shift-JIS, EUC-JP などの文字コードを使います。UTF-8 の日本語 1 文字は 3 バイトなので、`char` 1 つには入りません

```c
int a1 = 65;    // int 型に数値 65 を代入
int a2 = 'A';   // int 型に文字 'A' を代入（a1 と同じ 65 が入る）
char c1 = 66;   // char 型に数値 66 を代入（文字としては 'B'）
char c2 = 'B';  // 文字は ' ' で囲む（" " で囲むと文字列）
c2 = c2 + 1;    // char は整数型なので演算できる！ → 67 = 'C'
```

int と char は同じ整数型

同じ値 65 を `int` 型と `char` 型の変数に入れ、それぞれ `%d` と `%c` で出力してみます。

```
int  i = 65;
char c = 65;
printf("%d %c\n", i, i);
printf("%d %c\n", c, c);
printf("%zu %zu\n", sizeof(i), sizeof(c));
i = i + 1;
c = c + 1;
printf("%d %c\n", i, i);
printf("%d %c\n", c, c);
```

```
65 A
65 A
4 1
66 B
66 B
```

- `%d` は値を整数として、`%c` は値を文字コードとみなして文字として表示する。`int` でも `char` でも結果は同じ
- 違いは大きさ（`sizeof` の値。多くの環境で `int` は 4 バイト、`char` は 1 バイト）と、それによって決まる値の範囲だけ。どちらも値を整数として持つ

ASCII の印字可能文字（32〜126）

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15
32	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

文字コード = 行の値 + 列の値です（例: `A` は 64 + 1 = 65、`a` は 96 + 1 = 97）。`SP` は空白文字です。表の最後の 127（`DEL`）は印字可能文字ではなく、表示されない制御文字です。

文字コード表の読み方

数字・大文字・小文字は、それぞれ連続した番号に並んでいます。

文字	文字コード	備考
`'0'` 〜 `'9'`	48〜57	文字 `'0'` は整数 0 ではなく 48
`'A'` 〜 `'Z'`	65〜90	
`'a'` 〜 `'z'`	97〜122	小文字は大文字より 32 大きい
`' '`（空白）	32	表示される文字は 32〜126

`'A'` と書いても `65` と書いても、`char` 型の変数には同じ値が入ります。番号を覚える必要はなく、プログラムでは `'A'` のように文字で書くのが読みやすい書き方です。

制御文字とエスケープシーケンス

0〜31 と 127 は制御文字で、画面に文字として表示されません。よく使う制御文字と、そのままでは書けない一部の文字は、`\` に続けて書くエスケープシーケンスで表します。

書き方	文字コード	意味
`'\0'`	0	ヌル文字（文字列の終わりの印）
`'\t'`	9	タブ
`'\n'`	10	改行
`'\\'`	92	バックスラッシュ（`\` 1 つだと次の文字と組になるため）
`'\''`	39	シングルクォート（`'` で囲む文字定数の中に書くため）

```
printf("%d %d %d\n", '\n', '\t', '\0');   // 10 9 0
```

文字と整数の変換

文字コードが連続していることを使うと、文字の変換を足し算・引き算で書けます。

```c
char c = 'A';
printf("%c %d\n", c, c);          // A 65   (同じ値を文字と整数で表示)
printf("%c\n", c + 1);            // B      (次の文字)
printf("%d\n", 'a' - 'A');        // 32     (大文字と小文字の差)
printf("%c\n", 'g' - 'a' + 'A');  // G      (小文字 → 大文字)

char d = '7';
printf("%d\n", d - '0');          // 7      (数字の文字 → 整数)
printf("%c\n", '0' + 5);          // 5      (整数 → 数字の文字)
```

`'g' - 'a'` は「`g` がアルファベットの何文字目か（0 から数える）」を表します。それに `'A'` を足すと、同じ位置の大文字になります。

文字の入出力（%c）

`printf()` の `%c` は値を文字として、`%d` は整数として表示します。同じ変数を両方で表示できます。

`scanf()` の `%c` は、**空白や改行も 1 文字として読み込みます**。数値を読んだ後に `%c` を使うと、入力の最後の改行（10）が読み込まれてしまいます。

```
int n; char c;
scanf("%d", &n);      // 入力 12⏎ → n = 12（改行は入力に残る）
scanf("%c", &c);      // 残っていた改行を読む → c = 10（'\n'）
scanf(" %c", &c);     // 書式の先頭の空白が、空白・改行を読み飛ばす → 次の文字
```

%c の前に空白を入れる: `scanf(" %c", &c);` と書くと、空白・改行を読み飛ばしてから 1 文字を読みます。

char の範囲

型	大きさ	範囲
`char`	1 バイト	多くの環境で -128〜127（`signed char` と同じ）
`signed char`	1 バイト	-128〜127
`unsigned char`	1 バイト	0〜255

範囲を超える値を代入すると、`int` と同じように桁あふれして値が変わります。

```
char e = 127;
e = e + 1;              // e は −128 になる（127 の次の値は入らない）
unsigned char u = 255;
u = u + 1;              // u は 0 になる
```

`char` が符号付きかどうかは、C 言語の規格では処理系が決めます。文字を扱うだけなら、ASCII の範囲（0〜127）はどちらでも同じです。

2. printf() と scanf()

printf() 関数

`printf()` は第 1 引数（出力書式文字列）を出力し、その中の「変換指定」に従って
第 2 引数以降の値を出力します。返り値（`int`）は出力した文字数です（失敗時は負の数）。

```c
int a = 1;
printf("%dは整数、%fは実数です。", a, 2.3);
// ⇒ 1は整数、2.300000は実数です。
```

指定子	意味		指定子	意味
`%d`	整数 (int)		`%f`	浮動小数点数
`%ld` / `%lld`	整数 (long / long long)		`%e`	指数形式
`%u`	整数 (unsigned int)		`%c`	文字 (char)
`%o` / `%x`	8 進数 / 16 進数表記			

変換指定の表示幅と精度指定

`%f` などの変換指定では、`%w.pf` の形式で「表示幅 w」と「精度 p」を整数で指定できます（w, p はどちらも省略できます）。`%f` の精度は小数点以下の桁数です。

```
printf("*%f*", 123.4567);      // *123.456700*    (指定なし 6 桁)
printf("*%12f*", 123.4567);    // *  123.456700* (右寄せ！)
printf("*%.2f*", 123.4567);    // *123.46*        (四捨五入！)
printf("*%12.2f*", 123.4567);  // *      123.46*
```

scanf() 関数

`scanf()` は標準入力（キーボード入力）を書式に従って読み込み、第 2 引数以降の変数に代入します。返り値（`int`）は代入に成功した項目の数です。

```
int a; double x;
scanf("%d%lf", &a, &x);  // 変数名の前に & （変数のメモリアドレス）
```

- 入力の**空白・改行・タブを区切り**として項目を分け、それぞれの変数に代入する
- 書式中の空白は 0 個以上の空白・改行・タブに一致する（`scanf("%d %lf", &a, &x);` は `"%d%lf"` と同じ動作）
- `scanf("%d%lf\n", &a, &x);` は、改行の後に空白以外の文字が入力されるまで終了しない

scanf() では区別が必要！ `%f` は `float`、`%lf` は `double` です（`printf()` ではどちらも `%f`）。

3. 繰り返し処理

do while 文

```c
int a = 10;
while (a < 10) {
    a++;                // 一度も実行されない
}
printf("%d\n", a);      // 10
```

```c
int a = 10;
do {
    a++;                // 一度だけ実行される
} while (a < 10);
printf("%d\n", a);      // 11
```

`while` はブロックの**先頭**で、`do〜while` はブロックの**末尾**で繰り返しの条件を調べます。
そのため `do〜while` では、ブロックが必ず 1 回は実行されます。

for 文

```
int i;
for (i = 0; i < 10; i++) {
    printf("%d\n", i);
}
```

- `i = 0` — 初期化（最初に一度だけ実行）
- `i < 10` — この条件を満たしている間、繰り返す
- `i++` — ブロックを 1 回実行するたびに、その最後に実行

`for` 文と `while` 文は互いに書き換えられます。上のコードは次のコードと同じ動作です。

```
int i; i = 0;
while (i < 10) { printf("%d\n", i); i++; }
```

break 文と continue 文

```c
int a;
while (1) {              // continue はここへ（無限ループ）
    scanf("%d", &a);
    if (a < 0) break;    // break 文は繰り返しループを完全に抜ける
    if (a == 0) continue; // continue 文は中断して次の繰り返し処理へ
    printf("%d\n", a);
}
// break はここへ
```

`==` や `<=` などの比較演算子は、比較結果が真なら `1`、偽なら `0` の整数値を返します。
`while (1)` は条件が常に成立する無限ループです。

4. #define 定数マクロ定義

#define 定数マクロ定義

意味や役割が分かりにくい数値などの定数を、好きな名前で表せます。

```c
#include <stdio.h>
#define PI 3.14159265358979323846

int main() {
    double rad = x/180 * PI;   // → x/180 * 3.14159265358979323846 に置き換わる
```

- ソースコード中の `PI` は、**コンパイル前にすべて定義した文字列に置き換わる**
- あとで変更するかもしれない定数を `#define` しておけば、ソース中に散らばった値を 1 か所の修正でまとめて変更できる
- 定数の名前は、変数と区別しやすいように大文字で書く習慣がある
- `#define` の行の末尾にはセミコロン `;` を付けない（付けると `;` まで置き換わる）

#define の落とし穴

```
#define PI 1+2.14159265358979323846

double x = 90;
double rad = x/180 * PI;
// → x/180 * 1+2.14159265358979323846  に単純置換される
//   0.5 * 1 + 2.14… = 2.641593（意図した 1.570796 にならない！）
```

`#define` は**単純な文字列置換**なので、演算子を含む値は括弧で囲む必要があります。

```
#define PI (1+2.14159265358979323846)   // これなら 1.570796
```

5. stdlib.h と math.h

stdlib.h ユーティリティ標準関数ライブラリ

```c
#include <stdio.h>
#include <stdlib.h>   // 必須。忘れると Error か Warning

int main() {
    int a;
    scanf("%d", &a);
    if (abs(a) < 9) exit(1);  // |a| が9未満なら、終了して1を返す
    ...
```

よく使う関数: 文字列→数値変換 `atof()`, `atoi()` ／終了処理 `exit()`, `abort()` ／整数絶対値 `abs()` ／疑似乱数 `rand()`, `srand()` （定数 `RAND_MAX`）

math.h 数学標準関数ライブラリ

```c
#include <stdio.h>
#include <math.h>     // 必須。忘れると Error か Warning
#define PI 3.14159265358979323846   // 円周率は自分で定義する

int main() {
    double rad = 45.0 / 180 * PI;  // 度 → ラジアン
    printf("%f\n", sin(rad));      // 0.707107
    printf("%f\n", sqrt(2.0));     // 1.414214
    printf("%f\n", pow(2, 10));    // 1024.000000
```

三角関数 `sin()`, `cos()`, `tan()` ／逆三角関数 `asin()`, `acos()`, `atan()` ／指数・対数 `exp()`, `log()`, `log10()` ／実数絶対値 `fabs()` ／べき乗 `pow()`, `sqrt()`, `cbrt()`

- 引数と返り値はすべて `double` 型。三角関数の引数の単位は**弧度法（ラジアン）**
- 処理系によってはコンパイル時に `-lm` オプションが必要（`cc prog.c -o prog -lm`）

円周率は #define PI で定義する

C 言語の規格が定める `math.h` には、円周率の定数がありません。**次のように自分で定義して使います。**

```
#define PI 3.14159265358979323846
```

`M_PI` は使わない: 多くの環境の `math.h` には `M_PI` という円周率の定数がありますが、C 言語の規格には含まれません。規格どおりにコンパイルする環境（aiJudge の採点環境は `-std=c11`）では定義されず、**コンパイルエラーになります。**

次回予告と今日やること

次回（10/22）：配列変数

今日の 5 講時（実習）

1. 第 3 回課題の解答例を演習ページで確認する（`if` / `while` の組み合わせ、素因数分解など）
2. `char` 型と ASCII コード、`printf()` / `scanf()` の変換指定、`#define` を確認する
3. 演習課題 ex04 に取り組み、100 点以上を取る

質問は **Teams「★ 質問用チャネル」** か、教室の TA へ。