2024 前学期

情報工学基礎演習

【教科書】筧，石田，他「入門C言語」, 実教出版, 2014

学生用資料

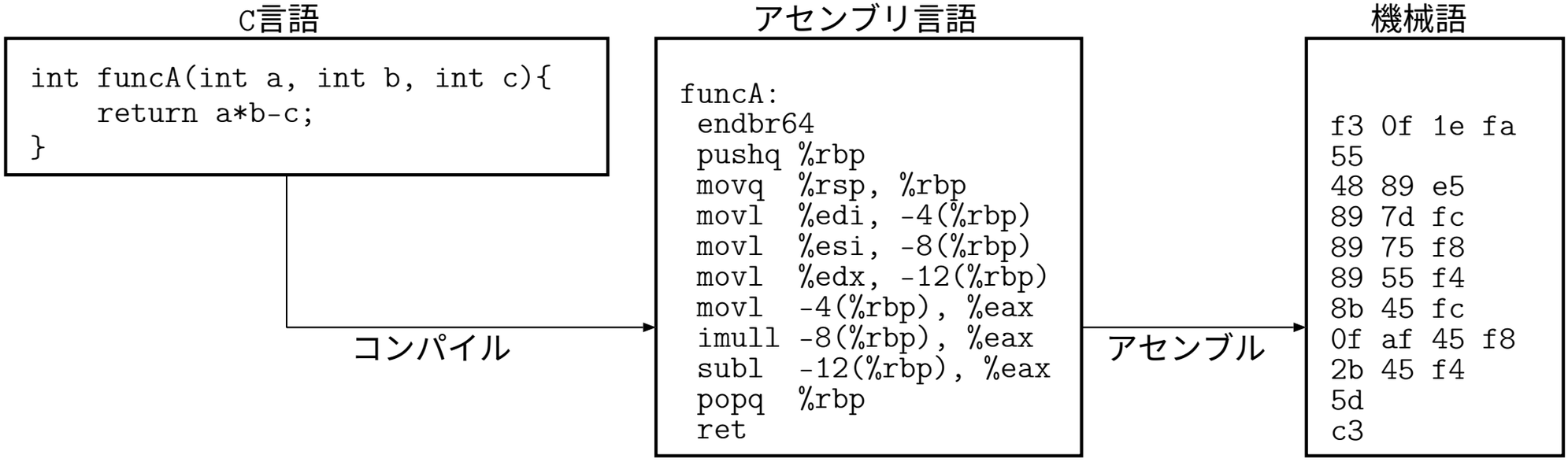
情報工学科 鷹合 大輔，田村 修

資料作成日：March 25, 2024

KIT 金沢工業大学

第0章はじめに — 0.1 C言語の来歴 —

- C言語は50年以上の歴史を持つコンパイラ言語（実行が早い）
- システム記述用言語と呼ばれ，UNIXやLinuxといった「オペレーティングシステム」の開発に使用されている．
- プログラムの拡張子は .c（小文字）が使われる．



コンパイラはコンパイル（アセンブリ言語への変換）が完了すると，アセンブラを呼び出して機械語生成まで行う．

- C言語のプログラムは main**関数** からスタートする．
- /* と */ の間は注釈（コメント）扱い．
- // から行末までは注釈扱い．
- #includeでインクルードする**ヘッダファイル**を指定する．使いたい関数にあわせて，ヘッダファイルをインクルードする必要がある．
- { と }の間を**ブロック**という．

```
1 /* はじめての
2     Cプログラム */
3 #include<stdio.h> // 標準入出力関数 (printfやscanfなど) を使う
4 int main(void){
5
6     printf("Hello World!!\n"); // 文字列の出力
7
8     return 0; // mainの戻り値
9 }
```

- **1文字**を表すときは，'A'のように**シングルクォート**で囲む．
- **文字列**を表すときは，"ABC"のように**ダブルクォート**で囲む．
- 8進数の整数を表すときは，0777のように冒頭に0をつける．
- 16進数の整数を表すときは，0xffのように冒頭に0xをつける．
- 正の整数であることを明示するときは，末尾にUをつける．
- 桁の大きな整数や，高精度の実数の末尾にはLをつける．
- 指数部付きの書き方例：2e-3 と書くと，$2 \times 10^{-3} = 0.002$を表す．

次のプログラムを実行せよ．

ex1_1.c

```c
#include <stdio.h>

int main(void)
{
    printf("%c %c %c\n", 'A','B','C')
    printf("%s\n", "Hello World!!")

    printf("%f\n", 3.14);
    printf("%f\n", 2e-3);

    printf("%d %d %d\n", 777, 0777, 0x777); /* %dは10進表記の指定 */
    printf("%o %o %o\n", 777, 0777, 0x777); /* %oは 8進表記の指定 */
    printf("%x %x %x\n", 777, 0777, 0x777); /* %xは16進表記の指定 */

    return 0;
}
```

- 変数は宣言してから使う.
- 変数を宣言すると，**変数を格納するための領域がメモリ上に確保される**．(教科書の図1.2，図1.3をいつも意識すること)
- 初期状態では変数の値は**不定**であることに注意.
- 変数名としては使えないものがあることに注意.
- 変数や関数に付けられる名前のことを**識別子**や**シンボル**ともいう.

次のプログラムを実行せよ．

ex1_2.c

```c
#include <stdio.h>

int main(void)
{
    int a,b,c;
    printf("%d\n",a);
    printf("%d\n",b);
    printf("%d\n",c);

    a++;
    b++;
    c++;

    printf("%d\n",a);
    printf("%d\n",b);
    printf("%d\n",c);
    return 0;
}
```

64bit OS（Windows/Linux）のデータ型

格納対象	符号	データ型	範囲
32bit整数	有	`int`	-2147483648 (0x8000,0000)～2147483647 (0x7FFF,FFFF)
	無	`unsigned int`	0 (0x0000,0000)～4294967295 (0xFFFF,FFFF)
16bit整数	有	`short`	-32768 (0x8000)～32767 (0x7FFF)
	無	`unsigned short`	0 (0x0000)～65535 (0xFFFF)
8bit整数/文字	有	`char`	-128 (0x80)～127 (0x7F)
	無	`unsigned char`	0 (0x00)～255 (0xFF)
64bit整数	有	`long long`	-9223372036854775808 ～ 9223372036854775807
	無	`unsigned long long`	0～18446744073709551615
64bit実数	–	`double`	$\pm 1.0 \times 10^{-307 \sim 308}$

他に`long int`型（`int`を略して`long`と書いてもよい）等がある．

`long int`型はLinuxでは64bit整数，Windowsでは32bit整数になるので注意が必要．

2進表記は，符号付きか，符号なしかで，値がかわる

0010,0011 (16進表記:0x23)	8ビット符号なし2進数とみなすと　32+2+1= 35 8ビット符号付き2進数とみなすと　32+2+1= 35
1010,0011 (16進表記:0xA3)	8ビット符号なし2進数とみなすと　128+32+2+1 = 163 8ビット符号付き2進数とみなすと　-128+32+2+1 = -93
1111,1110 (16進表記:0xF7)	8ビット符号なし2進数とみなすと　128+64+32+16+8+4+2 = 254 8ビット符号付き2進数とみなすと　-128+64+32+16+8+4+2 = -2

変数に文字（英数字・記号）を格納する際は**ASCIIコード**が使われる

（例）kit をASCIIコードで表すと 0x6B, 0x69, 0x74 となる

ASCIIコード表

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

```
char c0, c1, c2;

c0 = 'k'; // c0 = 0x6B; と同じ
c1 = 'i'; // c1 = 0x69; と同じ
c2 = 't'; // c2 = 0x74; と同じ
```

```
0x41: A
0x61: a
0x20: SP(Space)
0x0A: LF(Line Feed)
0x0D: CR(Carriage Return)
0x7F: DEL(Delete)
```

次のプログラムを実行せよ．

ex1_3.c

```c
#include <stdio.h>

int main(void)
{
    char c;

     c = 'A';
     printf("%c %d %x\n", c, c, c);
     c++;
     printf("%c %d %x\n", c, c, c);
     c++;
     printf("%c %d %x\n", c, c, c);

     return 0;
}
```

第2章入出力と演算子

- プログラムは，**文字端末（キーボードとモニタ）**に対して入出力[*1]を行うことが多い．
  - キーボードは，文字列をプログラムに入力する装置．
  - モニタは，プログラムが出力した文字列を表示する装置．
- **アドレス演算子&**は，よく使うので覚えておくこと．
- **ビット演算子（6種）**は，後学期科目「組込みシステム」で多用するので，必ず覚えること．

[*1]内部的には**ストリーム**と呼ばれる入出力専用バッファを使うことで非常に効率的な入出力処理を行っている．詳細は後学期科目「オペレーティングシステム」で説明する．

- 教科書で使われている **¥記号**（エン） は，授業スライドや Linux 環境では **\ 記 号**（バックスラッシュ） になるので適宜読み替えること[*2].
- 1文字の出力には`putchar`を使う.
- 書式を指定して文字列を出力したいときは，`printf`を使う．教科書 p.28 の書式の指定方法を一通り試しておくこと.

[*2] 教科書 p.26 の下から 4 行目参照.

ex2_1.c

```c
#include <stdio.h>

int main(void)
{
    int x=50000;
    double f=3.14;

    printf("%d\n", x);
    printf("%12d\n", x);
    printf("\n");

    printf("%x\n", x);
    printf("%8x\n", x);
    printf("%08x\n", x);
    printf("\n");

    printf("%f\n", f);
    printf("%12f\n", f);
    printf("%12.0f\n", f);
    printf("%12.1f\n", f);
    printf("%12.2f\n", f);

    return 0;
}
```

確認 どのような出力が得られるか確かめよ.

第2章入出力と演算子　— 2.2 キーボードからの入力 —

- 1文字の入力にはgetcharを使う．
- 書式を指定して入力したいときは，scanfを使う．
- **アドレス演算子**&を使うことで，変数のメモリアドレスを知ることができる．

- 演算子を使うときは，**優先度**や**結合規則**に注意．
- 数値計算を行うときは，必要に応じて明示的に**型変換**をしないと期待する値が得られないので注意．

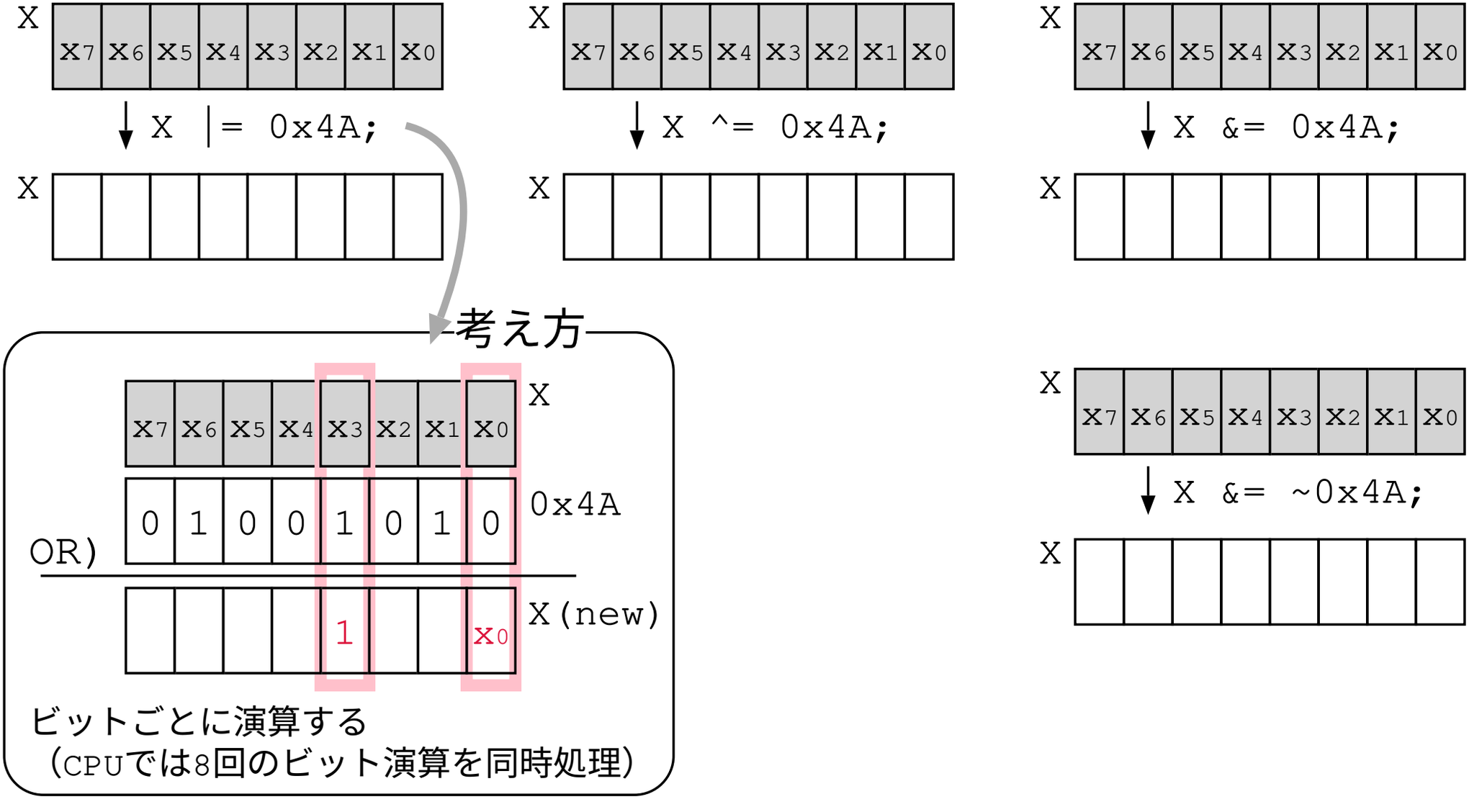
ビット演算子は6種類（後学期の「組込みシステム」で多用）

- **NOT/否定**　~
- **OR/論理和**　|　|=　（条件文で使う || と混同しないこと）
- **AND/論理積**　&　&=　（条件文で使う && と混同しないこと）
- **XOR/排他的論理和**　^　^=
- **左シフト**　<<　<<=
- **右シフト**　>>　>>=

プログラムでの使用例

```
unsigned char x=0xAA, y=0x18, z=0x0F;
x ^= 0xF0;  //     1010,1010 xor 1111,0000          →  0101,1010(=0x5A) となる

y &= ~0x08; //     0001,1000 and (not 0000,1000)
            //  → 0001,1000 and      1111,0111     →  0001,0000(=0x10) となる

z <<= 2;    //     0000,1111 を 2ビット左シフト      →  0011,1100(=0x3C) となる
```

8ビット変数Xに対して次のビット演算を行え（空白を0,1,X_n,$\overline{X_n}$で埋める）.



【論理演算の復習】

① 真理値表を完成させよ．

A	B	$\overline{A}$	$\overline{B}$	$A+B$	$A\cdot B$	$A\oplus B$
0	0					
0	1					
1	0					
1	1					

② 論理式の右辺を答えよ．

$$A+0 = \qquad A+A =$$

$$A+1 = \qquad A+\overline{A} =$$

$$A\cdot 0 = \qquad A\cdot A =$$

$$A\cdot 1 = \qquad A\cdot \overline{A} =$$

$$A\oplus 0 = \qquad A\oplus A =$$

$$A\oplus 1 = \qquad A\oplus \overline{A} =$$

$A \oplus 1 = ?$ の考え方

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}?\\?\end{bmatrix}$$

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}0\\?\end{bmatrix}$$

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}$$

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix}$$

$$A \oplus 1 = \overline{A}$$

A に対して 1 とのXOR演算を適用すると反転する

$A \oplus 0 = ?$ の考え方

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}0\\0\end{bmatrix} = \begin{bmatrix}?\\?\end{bmatrix}$$

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}0\\0\end{bmatrix} = \begin{bmatrix}1\\?\end{bmatrix}$$

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}0\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}$$

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}0\\0\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}$$

$$A \oplus 0 = A$$

A に対して 0 とのXOR演算を適用しても変わらない

$A \oplus A = ?$ の考え方

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}?\\?\end{bmatrix}$$

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\?\end{bmatrix}$$

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix}$$

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix}$$

$$A \oplus A = 0$$

A に対して A とのXOR演算を適用すると0になる

$A \oplus \overline{A} = ?$ の考え方

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}?\\?\end{bmatrix}$$

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1\\?\end{bmatrix}$$

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1\\1\end{bmatrix}$$

$$\begin{bmatrix}1\\0\end{bmatrix} \oplus \begin{bmatrix}0\\1\end{bmatrix} = \begin{bmatrix}1\\1\end{bmatrix}$$

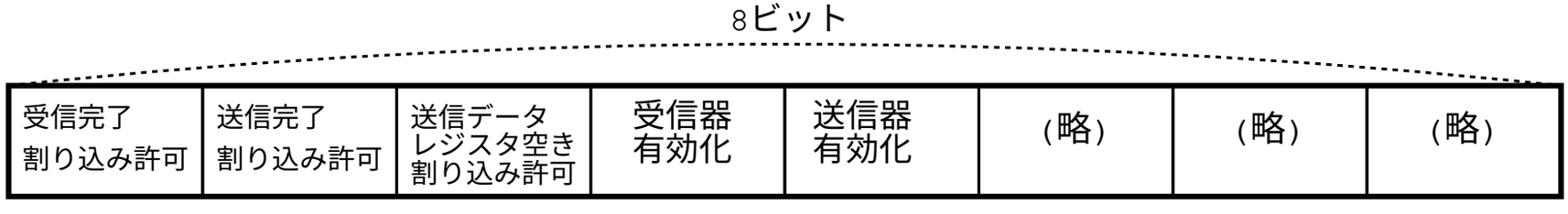
$$A \oplus \overline{A} = 1$$

A に対して $\overline{A}$ とのXOR演算を適用すると1になる

ネットワーク，オペレーティングシステム，組込みシステムに関するプログラミングではビット演算が必要な場面も多い．

- ネットワークでは，IPアドレスから，ネットワークとホストを分離するためにネットマスクを使った論理演算を行う．
- マイコンの内蔵機能（タイマ，通信）を使用するためには，**特殊機能レジスタ**[*3]に正しいビットパターンをセットしなければならない．

UCSR0B
通信設定用のレジスタ



C言語を使ったUCSR0Bレジスタの設定例

受信器と送信器の両方を有効化するときは

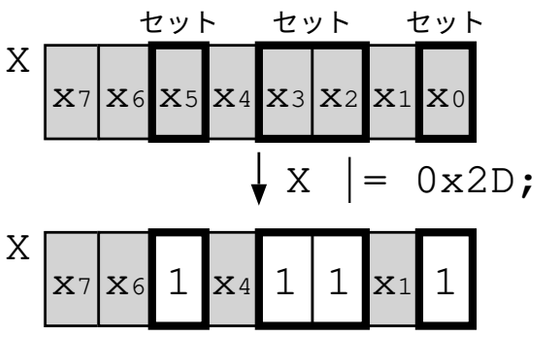
```
UCSR0B  =  0x18;
```

送信器だけを停止するには

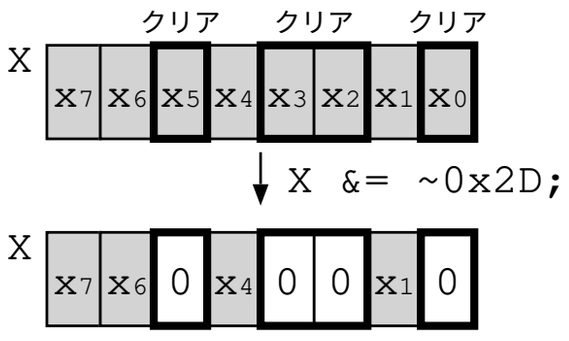
```
UCSR0B &= ~0x08;
```

[*3]数値計算用のレジスタではない．レジスタを構成する各ビットごとに機能が割り当てられている．

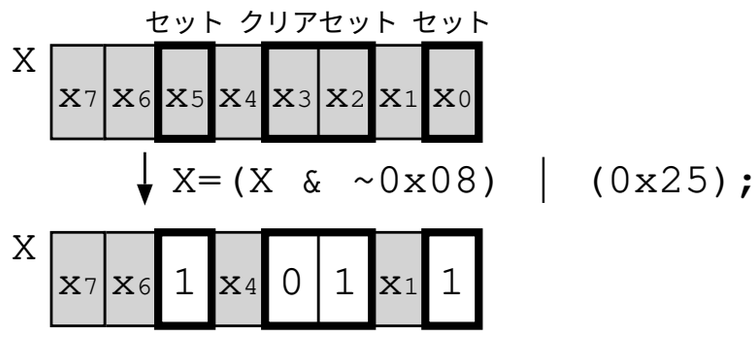
指定ビットをセットする



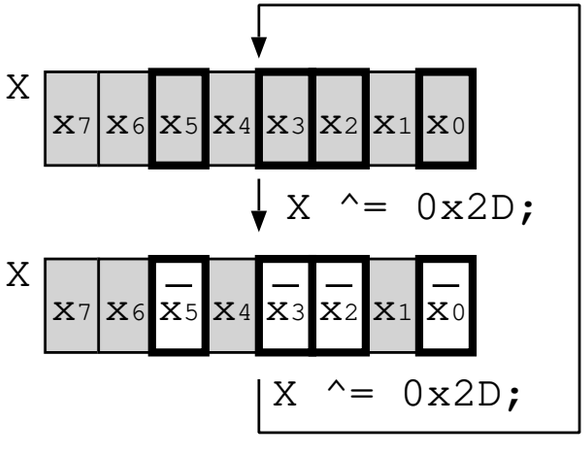
指定ビットをクリアする



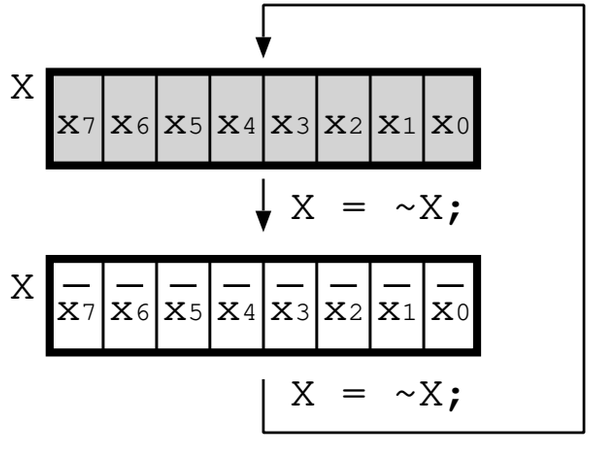
クリアとセットを同時に実施



指定ビット反転



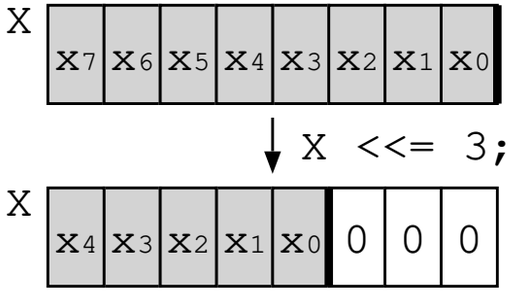
全ビット反転



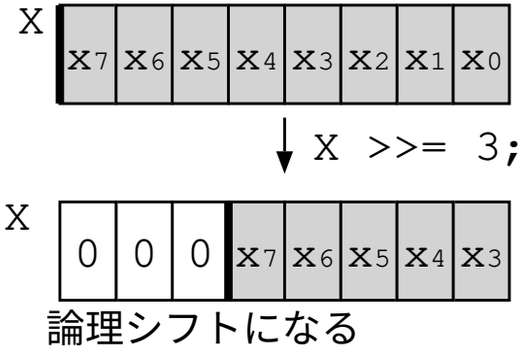
char型変数 X に対するビット操作例

シフト演算は，パターンをずらすときなどに使われる．

左シフト

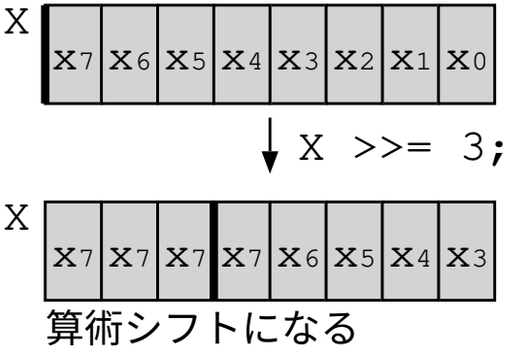


右シフト（Xがunsigned charの場合）



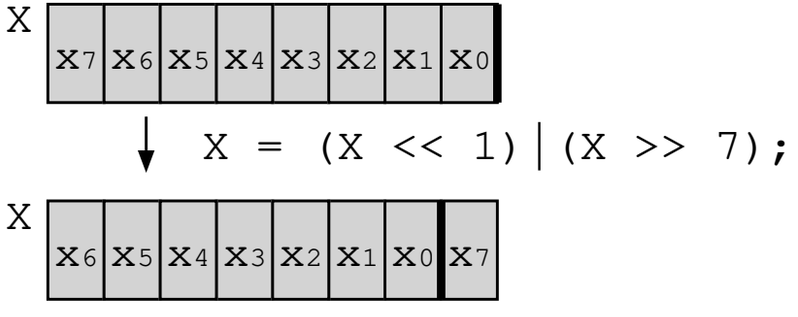
論理シフトになる

右シフト（Xがsigned charの場合）

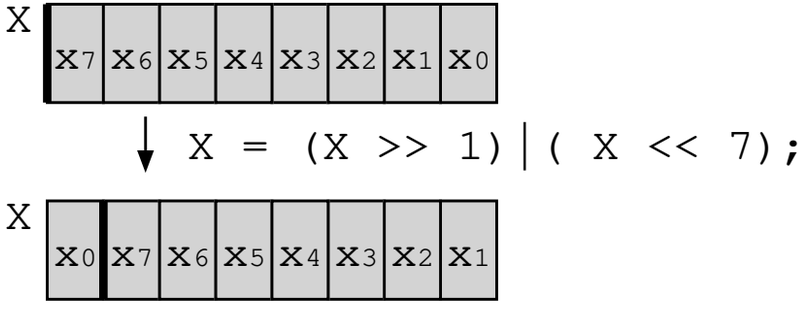


算術シフトになる

循環左シフト（Xがunsigned charの場合）



循環右シフト（Xがunsigned charの場合）

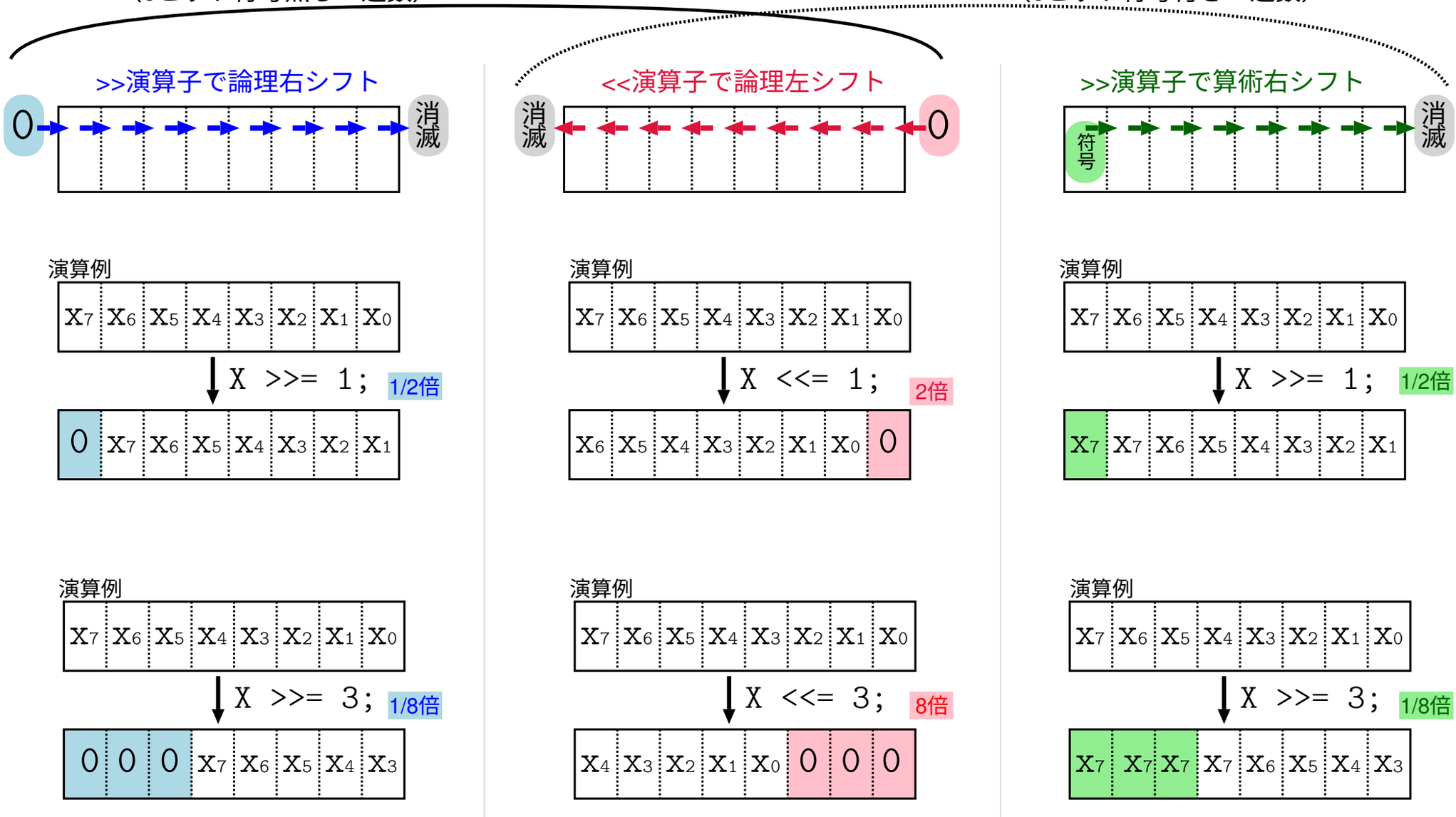


char型変数 X に対するビット操作例（シフト）

シフト演算

X が unsigned char型のときのシフト演算
（8ビット符号無し二進数）

X が signed char型のときのシフト演算
（8ビット符号付き二進数）



ex2_2.c

```c
#include <stdio.h>

int main(void)
{
    int x=-1; // unsigned int にすると...

    printf("%10d\t%08x\n" , x, x);
    x <<= 1;
    printf("%10d\t%08x\n" , x, x);
    x <<= 1;
    printf("%10d\t%08x\n" , x, x);
    x <<= 1;
    printf("%10d\t%08x\n" , x, x);

    x >>= 1;
    printf("%10d\t%08x\n" , x, x);
    x >>= 1;
    printf("%10d\t%08x\n" , x, x);
    x >>= 1;
    printf("%10d\t%08x\n" , x, x);
    return 0;
}
```

確認 5行目のコメントのとおりに書き換えて実行してみよ.

C言語で使われる制御文

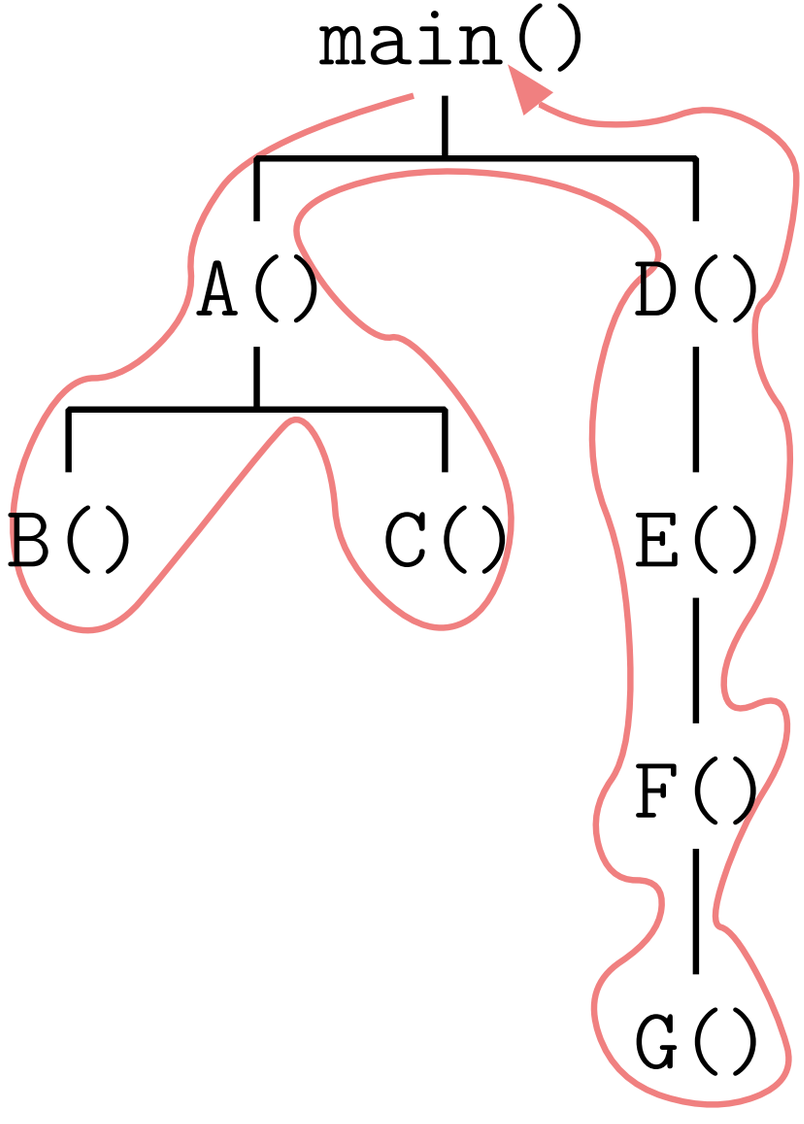
```
if(式) 文                    // 式が0でない場合だけ，文を実行.

if(式) 文1 else 文2          // 式が0でない場合は文1を実行，0のときは文2を実行.

switch(式){                  // 式の値に応じて，処理を分ける.
    ⋮
  case 定数:
    文並び
    break;
    ⋮
  default:
    文並び
    break;
}

for(式1;式2;式3) 文          // 式1は初期処理，式2は繰り返し条件，式3は文の実行後の処理

while(式) 文                 // 式が0でないなら，文を実行の繰り返し

do 文 while(式);             // 文を実行．式が0でないなら繰り返す.
```

- 条件式では**論理演算子**（`&&` `||` `!`）がよく使われるので慣れておくこと．
- 繰り返し文を途中で止める`break`や`continue`の使い方に慣れること．

- for文では，**カンマ演算子**を使うことも多いので覚えておくこと．
- 繰り返し文を途中で止めるbreakやcontinueの使い方に慣れること．

- 一連の処理を関数化しておくことで保守性が高まる.
- C言語側で用意してある関数を**標準関数**という.
- プログラマが自作する関数を**ユーザ関数**という.
- C言語では`main`という名前のユーザ関数を書かないと，プログラムとして実行できない.

■ 教科書 図4.3（プログラムを複数のCファイルに分割して作る）



x.c

```c
#include<stdio.h>

void A(void){
    printf("(A_A) start\n");
    B();
    C();
    printf("(A_A) end\n");
}

void B(void){
    printf("(B_B)\n");
}

void C(void){
    printf("(C_C)\n");
}
```

z.c

```c
#include<stdio.h>

void F(void){
    printf("(F_F) start\n");
    G();
    printf("(F_F) end\n");
}

void G(void){
    printf("(G_G)\n");
}
```

y.c

```c
#include<stdio.h>

void D(void){
    printf("(D_D) start\n");
    E();
    printf("(D_D) end\n");
}

void E(void){
    printf("(E_E) start\n");
    F();
    printf("(E_E) end\n");
}

void main(void){
    printf("main start\n");
    A();
    D();
    printf("main end\n");
}
```

```
$ gcc x.c y.c z.c -w↵
 (-wは警告を表示させないコンパイラオプション)
$ ./a.out
```

- 関数は名前，引数の形式，戻り値の型を宣言してから使うことが殆ど．これを**プロトタイプ宣言**という．
- プロトタイプ宣言をしていない関数を呼び出すと警告，場合によってはビルドエラーになる．

教科書のプログラムのビルド，実行方法

①2値の最大値

```
$ gcc list4_1.c list4_2.c ↵
$ ./a.out

$ gcc list4_3.c ↵
$ ./a.out
```

②2数の最小公倍数

```
$ gcc list4_4.c list4_5.c ↵
$ ./a.out

$ gcc list4_6.c ↵
$ ./a.out
```

確認 リスト4.3で使われている**条件演算子（三項演算子）**は，よく使われるので覚えておくこと．

リスト4.3を条件演算子を使わずに書くと，以下のように書き直せる．

```
int max(int x, int y)
{
      if( x > y)
         return x;
      else
         return y;
}
```

- C言語では，関数に変数を引数として渡すときは値が渡される．このため，元の変数が変更されることはない．

- 関数の中で，その関数自身を呼び出すことを**再帰呼び出し**という．
- 再帰呼出しを使うと，階乗計算などのプログラムが簡単に書ける．
- 再帰呼出しを行う場合は，終了条件をよく確認する必要がある．

教科書のプログラムのビルド，実行方法

①再帰による階乗計算

```
$ gcc list4_8.c list4_9.c ↵
$ ./a.out

$ gcc list4_10.c ↵
$ ./a.out
```

②2反復による階乗計算

```
$ gcc list4_8.c list4_11.c ↵
$ ./a.out
```

確認 教科書のリスト4.10において，関数factの中身を条件演算子で書き直してみよ．

第5章記憶ｸﾗｽと通用範囲 — 5.1 記憶ｸﾗｽと通用範囲 —

- **通用範囲（スコープ）**：プログラムのどこから変数が使えるか．
  - **ローカル変数**：関数の中で宣言されている変数のこと．他の関数からは読み書き不能．**ブロック**を使うと，更に通用範囲を限定できる（変数のブロック化）．
  - **グローバル変数**：関数外で宣言されている変数のこと．複数の関数から読み書き可能．
- **記憶クラス**：変数用の記憶領域をどこに確保するか．
  - メモリの**スタック領域**（関数呼び出し時に消費，復帰時に開放）[*4]
  - メモリの**スタティック領域**（プログラム実行中は固定）
  - メモリの**割り付け記憶域**（専用の関数を使って動的に確保・開放）
  - レジスタ

[*4]スタック領域は，プログラムの進行に応じて，増えたり，減ったりする領域．スタック領域が不足すると，プログラムが正常に実行できなくなる．配布したLinux環境では，1プログラムあたり8MB程度．

- 関数を宣言するときに static をつけると，他のCファイルからは呼び出せなくなる．
- 他のCファイルで定義されている関数を呼び出すときは，extern をつけたプロトタイプ宣言をすればよい．

- 多数の似たようなデータを扱いたい場合は**配列**を使うとよい．
- 配列は変数の集合体といえる．配列のどこを読み書きするかは**添字（インデックス）**と呼ばれる0から始まる番号を使う．
- 繰り返し文と相性がよく，効率的にプログラムを記述できる．

```
double h[6];  // 配列hの宣言（ h[0]からh[5] までの6個の double型データ の集まり）

int x[6];     // 配列xの宣言（ x[0]からx[5] までの6個の int型データ の集まり）

char s[6];    // 配列sの宣言（ x[0]からx[5] までの6個の char型データ の集まり）
```

■ 配列の格納アドレスを確認するプログラム

ex6_1.c

```c
#include <stdio.h>

int main(void)
{
    int x[6]={50,100}; // intをcharや，doubleにかえてみよ.
    int n;

    for(n=0;n<6;n++){
        printf("%p\t%d\n" , &x[n], x[n]);
    }
    return 0;
}
```

```
$ gcc ex6_1.c↵
$ ./a.out↵
0x7fffeb26b390  50
0x7fffeb26b394  100
0x7fffeb26b398  0
0x7fffeb26b39c  0
0x7fffeb26b3a0  0
0x7fffeb26b3a4  0
```

- 2次元配列は，要素数が等しい1次元配列が複数並んだものと考えればよい．行列計算などで使う．
- 6.3.3 は省略
- 6.3.4 は省略

- 関数に配列を渡すことができる．
- 配列を受け取る関数は，配列だけでなく，**配列の要素数**も引数として受け取るようになっていることが多い．

```
int func(int x[], int n);  // 配列とその要素数を受け取る関数の宣言例
```

- 6章では触れられていないが，実はC言語では，関数に渡されるのは**「配列のコピーではなく，配列の先頭アドレス」**になる．
  - 配列をコピーしない分だけ，関数呼び出しが早い．
  - 呼び出された関数側で配列を書き換える処理は，**「呼び出し元の関数が持っている配列の書き換え」**となることに注意する．

ex_2.c

```c
#include <stdio.h>

void func(int x[], int n);

int main(void)
{
    int x[6]={50,100};
    int i;

    for(i=0;i<6;i++){
        printf("%d: %d\n",i, x[i]);
    }
    printf("-----\n");
    func(x,6);
    for(i=0;i<6;i++){
        printf("%d: %d\n",i, x[i]);
    }

    return 0;
}

void func(int x[], int n)
{
    int i;
    for(i=0;i<n;i++){
        x[i]=i*i;
    }
}
```

確認 main内の配列xが，funcによって書き換えられることを確認せよ．

- C言語は**メモリ**を読み書きしやすい言語（原始的な高級言語）.
  - **アドレス**を指定した読み書きが簡単にできる.
- アドレスを格納するための変数を**ポインタ変数**という.
- ポインタ変数を宣言するときは，何の**データ型**（int型や，double型）のアドレスを格納するのかを指定する.
- ポインタ変数を使って，ポイント先のデータを読み書きが可能.
- **ポインタ変数は加減算できる（ポイント先をずらせる）.** 例えば+1で隣のデータのアドレス，+2で更にその隣のデータをポイントする[*5].
- 一般的なPCでは，**ポインタ変数の大きさは8バイト**.

[*5] +1したからといって，1番地先にセットされるわけではない．int型データをポインタしているポインタ変数を+1すると，格納されている値は4番地先にセットされることになる.

- int型データが置かれているアドレスを格納させたいときは，`int *p;` のように宣言する．
- int型変数aが用意されている場合，`p=&a;` と書けば，変数aのアドレスをポインタ変数pに格納できる．
- `*p=10;` と書けば，ポイント先に10が格納される．
  （`a=10;` と書かなくても，変数aの値を10にできる[*6]）

[*6]変数aのアドレスを調べてポインタ変数pに格納しておけば，aを直接読み書きする代わりにポインタ変数pを使って間接的に変数aを読み書きできる．

ex7_1.c

```c
#include <stdio.h>

int main(void)
{
    int *p;
    int a,b;

    printf("%p %p\n",&a, &b);
    printf("(a,b)=(%d,%d)\n",a, b);

    p = &a;   printf("p: %p, *p : %d\n",p, *p);
    *p = 10;  printf("p: %p, *p : %d\n",p, *p);

    printf("(a,b)=(%d,%d)\n",a, b);

    p = &b;       printf("p: %p, *p : %d\n",p, *p);
    *p = 12345;  printf("p: %p, *p : %d\n",p, *p);

    printf("(a,b)=(%d,%d)\n",a, b);
    return 0;
}
```

```
0x7fff4f4936b8 0x7fff4f4936bc
(a,b)=(-3211,99)
p: 0x7fff4f4936b8, *p : -3211
p: 0x7fff4f4936b8, *p : 10
(a,b)=(10,99)
p: 0x7fff4f4936bc, *p : 99
p: 0x7fff4f4936bc, *p : 12345
(a,b)=(10,12345)
```

ex7_2.c

```c
#include <stdio.h>

int main(void)
{
    void *p;

    int x=100;
    char c='A';

    printf("%d %c\n", x, c);

    p = &x;
    *((int*)p)=1000;

    p = &c;
    *((char*)p)='B';

    printf("%d %c\n", x, c);
    return 0;
}
```

```
100 A
1000 B
```

- 7.2.4 は省略
- 7.2.5 は省略
- 7.2.8 において，リスト7.23以降は省略

ex7_3.c

```c
#include <stdio.h>

int main(void)
{
    int x[5]={10,20,30,40,50}; // 配列
    int *p;                    // ポインタ変数

    p = x; // ここは p=&x[0]; と書いても同じ
    printf("%p %d\n", p, *p);
    printf("\n");

    printf("%p %d\n", p+0, *(p+0));
    printf("%p %d\n", p+1, *(p+1));
    printf("%p %d\n", p+2, *(p+2));
    printf("%p %d\n", p+3, *(p+3));
    printf("%p %d\n", p+4, *(p+4));

    return 0;
}
```

```
0x7fff186827a0 10

0x7fff186827a0 10
0x7fff186827a4 20
0x7fff186827a8 30
0x7fff186827ac 40
0x7fff186827b0 50
```

ex7_4.c

```c
#include <stdio.h>

int main(void)
{
    int x[5]={10,20,30,40,50}; // 配列
    int *p;                    // ポインタ変数

    p = x; // ここは p=&x[0]; と書いても同じ
    printf("%p %d\n", p, *p);

    p++;
    printf("%p %d\n", p, *p);

    p++;
    printf("%p %d\n", p, *p);

    p--;
    printf("%p %d\n", p, *p);

    p--;
    printf("%p %d\n", p, *p);

    return 0;
}
```

```
0x7fffd5ac7500 10
0x7fffd5ac7504 20
0x7fffd5ac7508 30
0x7fffd5ac7504 20
0x7fffd5ac7500 10
```

```c
#include <stdio.h>

int main(void)
{
    int x[5]={10,20,30,40,50}; // 配列
    int *p;                    // ポインタ変数
    p = a; // ここは p=&x[0]; と書いても同じ

    printf("%d\n", x[0]);
    printf("%d\n", x[1]);
    printf("%d\n", x[2]);

    printf("%d\n", *x);  // x は先頭要素のアドレス（*をつけて実体にアクセス）
    printf("%d\n", *(x+1));
    printf("%d\n", *(x+2));

    // ポインタは配列風にインデックス指定でアクセスできる
    printf("%d\n", p[0]);
    printf("%d\n", p[1]);
    printf("%d\n", p[2]);

    printf("%d\n", *p);
    printf("%d\n", *(p+1));
    printf("%d\n", *(p+2));

    // ポインタはポイント先の更新が可能（配列できないので，x++などはエラー）
    printf("%d\n", *(p++));
    printf("%d\n", *(p++));
    printf("%d\n", *(p++));
    return 0;
}
```

ex7_6.c

```c
#include <stdio.h>

int main(void)
{
    char s[]="Kanazawa Institute of Technology";
    char *p;                        // ポインタ変数
    p = s;

    while(*p!='\0'){
        printf("%p %c\n",p, *p);
        p++;
    }
    return 0;
}
```

確認 繰り返し文の継続条件についてよく確認せよ.

ex7_7a.c

```c
#include <stdio.h>

int main(void)
{
    char *p[3];

    p[0]="Kanazawa Institute of Technology";
    p[1]="ISHIKAWA";
    p[2]="JAPAN";

    printf("\n%s\n%s\n%s\n", p[0], p[1], p[2]);

    p[0]="Hello World!!";
    p[1]="TOKYO";

    printf("\n%s\n%s\n%s\n", p[0], p[1], p[2]);
    return 0;
}
```

ex7_7b.c

```c
#include <stdio.h>

int main(void)
{
    char *p[3]={"Kanazawa Institute of Technology", "ISHIKAWA", "JAPAN"};

    printf("\n%s\n%s\n%s\n", p[0], p[1], p[2]);

    p[0]="Hello World!!";
    p[1]="TOKYO";

    printf("\n%s\n%s\n%s\n", p[0], p[1], p[2]);
    return 0;
}
```

ex7_8a.c

```c
#include <stdio.h>

int main(void)
{
    char a[10], b[5], c[8];

    scanf("%s", a);
    scanf("%s", b);
    scanf("%s", c);

    printf("%s\n%s\n%s\n", a, b,c);
    return 0;
}
```

ex7_8b.c

```c
#include <stdio.h>

int main(void)
{
    char a[10], b[5], c[8];
    char *p[3];

    p[0]=a;
    p[1]=b;
    p[2]=c;

    scanf("%s", p[0]);
    scanf("%s", p[1]);
    scanf("%s", p[2]);

    printf("%s\n%s\n%s\n", a, b,c);
    return 0;
}
```

ex7_8c.c

```c
#include <stdio.h>

int main(void)
{
    char a[10], b[5], c[8];
    char *p[3]={a,b,c};

    scanf("%s", p[0]);
    scanf("%s", p[1]);
    scanf("%s", p[2]);

    printf("%s\n%s\n%s\n", a, b,c);
    return 0;
}
```

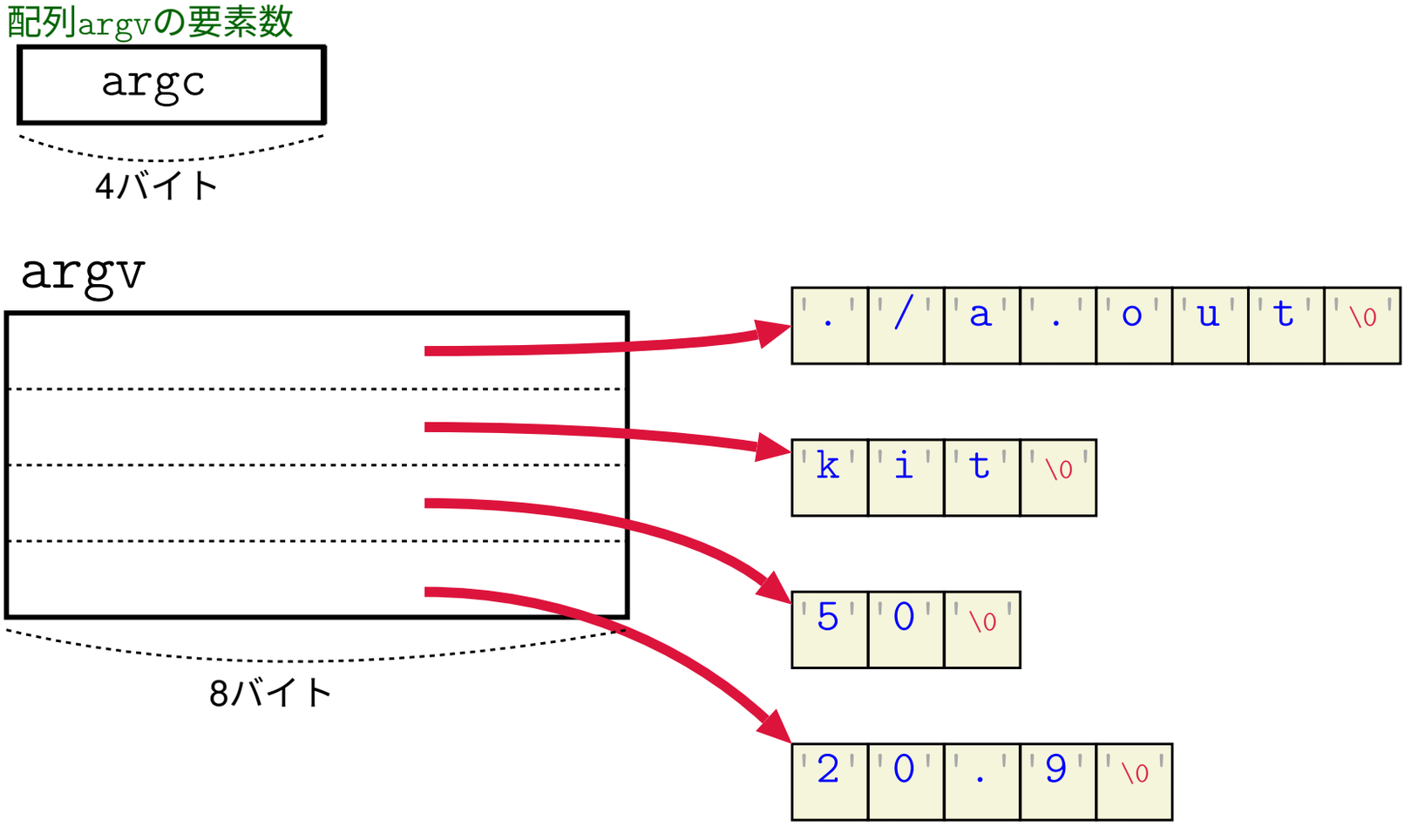
```
Technology↵
UNIX↵
JAPAN↵
Technology
UNIX
JAPAN
```

ソースコード

```
int main( int argc, char *argv[] ){
    :
}
```

端末

```
$ ./a.out kit 50 20.9
```



- **構造体**は，複数のデータ型をまとめるためのデータ型（ユーザが定義）
  - 構造体の**配列**も生成可能．
  - 構造体への**ポインタ変数**も使用可能．
- 構造体の**初期化**の仕方を覚えること．
- 構造体変数名のメンバを読み書きするときは ピリオド `.` を使う．
- ポインタ変数でメンバを読み書きするときは アロー演算子 `->` を使う．
- 構造体は，関数の引数にすることができる（構造体のコピーが関数に引き渡される）．
- 実は，構造体を関数の引数にすることはあまりなく，**構造体のアドレスを引数にすることの方が多い**．

ex8_1.c

```c
#include <stdio.h>

struct MY_DATE
{
    char name[128];
    int yy;
    int mm;
    int dd;
};

int main(void)
{
    struct MY_DATE x = { "阪神・淡路大震災", 1995, 1, 17};
    struct MY_DATE y = { "東日本大震災", 2011, 3, 11};
    struct MY_DATE z = { "能登半島地震", 2024, 1, 1};

    printf("%s %d/%d/%d\n", x.name, x.yy, x.mm, x.dd);
    printf("%s %d/%d/%d\n", y.name, y.yy, y.mm, y.dd);
    printf("%s %d/%d/%d\n", z.name, z.yy, z.mm, z.dd);

    return 0;
}
```

ex8_1.c

```c
#include <stdio.h>

struct MY_DATE
{
    char name[128];
    int yy;
    int mm;
    int dd;
};

int main(void)
{
    int n;

    struct MY_DATE date[] =
    {
        { "阪神・淡路大震災", 1995, 1, 17},    // 0
        { "東日本大震災", 2011, 3, 11},        // 1
        { "能登半島地震", 2024, 1, 1}          // 2
    };

    for( n = 0; n < 3 ; n++){
        printf("%s %d/%d/%d\n",
            date[n].name, date[n].yy, date[n].mm, date[n].dd);
    }

    return 0;
}
```

【重要】文字列を配列に格納する方法

（教科書リスト8.7を読む前に）

ex8_3a.c

```c
#include <stdio.h>

int main2(void)
{
    char x[16] = "Hello World!!"; // 配列の宣言時なら，文字列を配置可能
    x[4]='@';                     // 配列内の文字の変更も可能
    printf("%s\n",x);

    return 0;
}
```

ex8_3b.c

```c
#include <stdio.h>

int main(void)  // ビルドエラー
{
    char x[16] = "Hello World!!";
    x = "JAPAN"; // 宣言後は「代入文」で文字列を配置することは認められていない
    printf("%s\n",x);

    return 0;
}
```

ex8_3c.c

```c
#include <stdio.h>  // printf(), scanf()
#include <string.h> // strcpy()

int main(void)
{
    char x[16]="Hello World!!";

    printf("%s\n", x);

    strcpy(x, "(*_*)"); // strcpy()で配列上の文字列を書き換える
    printf("%s\n", x);

    scanf("%s",x);      // scanf()で配列上の文字列を書き換える
    printf("%s\n", x);

    return 0;
}
```

ex8_3d.c

```c
#include <stdio.h>

int main(void) // 実行エラー
{
    char *x = "Hello World!!"; //「変更禁止領域上の文字列」へのポインタ

    printf("%s\n",x);
    x[4]='@'; // ここで実行エラー（この文をコメントアウトすると動く）
    printf("%s\n",x);
    return 0;
}
```

ex8_4a.c

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

struct MY_DATE
{
    char name[128];
    int yy;
    int mm;
    int dd;
};

int main(void)
{
    struct MY_DATE *p;
    struct MY_DATE x = { "阪神・淡路大震災", 1995, 1, 17};

    p = &x;
    printf("%s %d/%d/%d\n", p->name, p->yy, p->mm, p->dd);

    strcpy(p->name, "恐怖の大王");
    p->yy = 1999;
    p->mm = 7;

    printf("%s %d/%d/%d\n", p->name, p->yy, p->mm, p->dd);

    return 0;
}
```

ex8_4b.c

```c
#include <stdio.h>

struct MY_DATE
{
    char name[128];
    int yy;
    int mm;
    int dd;
};

int main(void)
{
    struct MY_DATE *p;
    struct MY_DATE date[] =
    {
        { "阪神・淡路大震災", 1995, 1, 17},    // 0
        { "東日本大震災", 2011, 3, 11},        // 1
        { "能登半島地震", 2024, 1, 1}          // 2
    };

    p = &date[0];
    printf("%s %d/%d/%d\n", p->name, p->yy, p->mm, p->dd);
    p++;
    printf("%s %d/%d/%d\n", p->name, p->yy, p->mm, p->dd);
    p++;
    printf("%s %d/%d/%d\n", p->name, p->yy, p->mm, p->dd);

    return 0;
}
```

ex8_5a.c

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

struct MY_DATE
{
    char name[128];
    int yy;
    int mm;
    int dd;
};

void func(struct MY_DATE a)
{
    strcpy(a.name, "恐怖の大王");
    a.yy = 1999;
    a.mm = 7;
}

int main(void)
{
    struct MY_DATE x = { "能登半島地震", 2024, 1, 1};

    printf("%s %d/%d/%d\n", x.name, x.yy, x.mm, x.dd);
    func(x);
    printf("%s %d/%d/%d\n", x.name, x.yy, x.mm, x.dd);

    return 0;
}
```

ex8_5b.c

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

struct MY_DATE
{
    char name[128];
    int yy;
    int mm;
    int dd;
};

void func(struct MY_DATE *a)
{
    strcpy(a->name, "恐怖の大王");
    a->yy = 1999;
    a->mm = 7;
}

int main(void)
{
    struct MY_DATE x = { "能登半島地震", 2024, 1, 1};

    printf("%s %d/%d/%d\n", x.name, x.yy, x.mm, x.dd);
    func(&x);
    printf("%s %d/%d/%d\n", x.name, x.yy, x.mm, x.dd);

    return 0;
}
```

ex8_5c.c

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

typedef struct MY_DATE
{
    char name[128];
    int yy;
    int mm;
    int dd;
} MY_DATE_t;


void func(MY_DATE_t *a)
{
    strcpy(a->name, "恐怖の大王");
    a->yy = 1999;
    a->mm = 7;
}

int main(void)
{
    MY_DATE_t x = { "能登半島地震", 2024, 1, 1};

    printf("%s %d/%d/%d\n", x.name, x.yy, x.mm, x.dd);
    func(&x);
    printf("%s %d/%d/%d\n", x.name, x.yy, x.mm, x.dd);

    return 0;
}
```

（省略）

- **ファイル**は，**補助記憶装置**における情報の保存単位

  ①**テキストファイル**　文字/改行コードだけで出来ているファイル．
  `*.c` `*.py` `*.txt`などテキストエディタで表示できる．

  ②**バイナリファイル**　テキストファイル以外のファイル．
  `*.mp4` `*.docx` `*.pdf` `a.out`など

- 記憶メディア（ハードディスク，DVD，SSD）の種類に関係なく，ファイルの基本操作は**オープン，リード，ライト，クローズ**の4つ[*7]．

- C言語ではファイルをオープンすると，自動的に**ファイル構造体**が確保されてそのメモリアドレス（**ファイルポインタ**）が返される．その後はファイルポインタを使って目的のファイルをリード/ライトを行い，最後にクローズする流れとなる．

[*7]あくまで一般ユーザや，アプリケーションプログラム開発者から見たときの話であることに注意．実際にはOSによって記憶装置の構造にあわせた読み書き処理が行われる．

補助記憶装置は動作が遅いので，ファイル読み書き回数が多いのは困る（アプリケーションプログラムの待ち時間が長くなる）．

- C 言語では主記憶装置の一部をファイル用**バッファ**[*8]として使用する．バッファを使うことで補助記憶装置を読み書き回数を削減し，ファイル入出力の高速化を図っている（**高水準入出力**という[*9]）．
- バッファの内容を（補助記憶装置などに）吐き出させることを**フラッシュ**という．

[*8]ファイル入出力用のバッファのことを**ストリーム**ともいう．高水準入出力のことを**ストリーム入出力**ともいい，高水準入出力を使って「ファイルをオープン」することを「ストリームをオープンする」という言い方もする．

[*9]バッファ（ストリーム）を使わない方法も可能で，それを**低水準入出力**という．低水準入出力の関数は後学期の「オペレーティングシステム」で扱う．

- 高水準入力用の関数名はfで始まることが多い(fopen, fread, fwrite, fclose, fflush, fprintf, fscanf, fgets等)[*10].
  - man 3 fopen↵などで使い方を調べるとよい.
- FILE は**ファイル構造体の型**，FILE * は**ファイル構造体を指す型**
  - 構造体と言ってもメンバを知る必要はない.
- EOF は End Of File（**ファイル末尾**）の意味で，値としては-1
  - fcloseやfscanfなどでエラーが起きたときに返される値.
- NULL は**ヌル（ヌルポインタ）**といい，値としては0
  - fopenやfgetsなどポインタを返す関数でエラーが起きたときに返される値.
  - ポインタ変数を初期化するときに使うことも多い.

[*10]教科書では簡単のため，fread/fwriteなどは省略している.

ex9_1.c

```c
#include <stdio.h>

int main(void)
{
    FILE *fp;

    fp = fopen("os.txt","w"); // 書き込み
    if( fp == NULL )
    {
        printf("ファイルオープンエラー\n");
        return 0;
    }

    fprintf(fp, "1991 Linux\n");
    fprintf(fp, "1985 Windows\n");
    fprintf(fp, "2001 macOS\n");

    fclose(fp);

    return 0;
}
```

ex9_2.c

```c
#include <stdio.h>

int main(void)
{
    FILE *fp;
    int x;
    char s[10];

    fp = fopen("os.txt","r"); // 読み出し
    if( fp == NULL )
    {
        printf("ファイルオープンエラー\n");
        return 0;
    }

    while( fscanf(fp, "%d%s", &x, &s[0]) != EOF )
    {
        printf("%d --> %s\n", x, s);
    }
    fclose(fp);

    return 0;
}
```

ex9_3.c

```c
#include <stdio.h>

int main(void)
{
    FILE *fp;
    int n=0;
    char s[256];

    fp = fopen("/etc/os-release","r"); // 読み出し
    if( fp == NULL )
    {
        printf("ファイルオープンエラー\n");
        return 0;
    }

    while( fgets(s, 256, fp) != NULL )
    {
        printf("%3d: %s", ++n, s);
    }
    fclose(fp);

    return 0;
}
```

ex9_4.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
    FILE *fp;
    int n=0;
    char s[256];

    if( argc!=2 )
    {
        printf("ファイル名を指定してください\n");
        return 0;
    }

    fp = fopen(argv[1],"r"); // 読み出し
    if( fp == NULL )
    {
        printf("ファイルオープンエラー\n");
        return 0;
    }

    while( fgets(s, 256, fp) != NULL )
    {
        printf("%3d: %s", ++n, s);
    }
    fclose(fp);

    return 0;
}
```

■ 標準入出力

通常，Cプログラムは標準入力 stdin，標準出力 stdout，標準エラー出力 stderr という3つの特別なファイル(ストリーム)が開かれた状態でスタートする（教科書p.241）．一部の入出力関数はそれらのファイルに対して入出力操作を行っている．

- `fprintf(stdout, "Hello\n");` と `printf("Hello\n");` は同じ．
- `fscanf(stdin, "%s", msg);` と `scanf("%s", msg);` は同じ．

ex9_5.c

```c
#include <stdio.h>

int main()
{
    printf("Hello\n");
    fclose(stdout);
    printf("World!!\n");

    return 0;
}
```

確認 このプログラムを動かすとどうなるか．

- 標準関数には①入出力関数，②文字列操作関数，③データ変換関数，④メモリ関数，⑤数学関数，⑥割り込み関数，などがある[*11].
- 使用にあたっては**マニュアルを読むこと**が重要.
  - **ヘッダーファイル**（例：`math.h`）
  - **関数の引数，返り値**（例：`double cos(double x);`）
  - **コンパイラオプション**（数学関数なら `-lm`）
- Linuxでは端末でマニュアルを参照可能

```
$ man 3 cos ↵
        ↑ 標準関数を見たいときは"3"を指定

        (manコマンドの終了は Qキー )
```

[*11] よく使う関数名をある程度覚えておくとよい.

特に次の関数は使えるようにしておくとよい．

- `sprintf` ..... 書式付き文字列を文字配列上に並べる
- `strlen` ..... 文字列の長さを取得する
- `strcmp` ..... 2つの文字列の比較（返り値に注意）
- `atoi` ..... 数を表す文字列をint型整数にする．例）"123"→123
- `rand, srand` ..... 乱数の取り出し，乱数生成器の初期化
- `malloc, free` ..... メモリの動的確保と開放
- `abort, exit` ..... プログラムの中断

演習

確認 コマンドラインで指定された回数だけ"Hello!!"と表示するプログラムを作成してみなさい.

確認 1〜6の値をランダムで10個出力するプログラムを作成せよ．その際，乱数生成器の初期化に使うシード値はコマンドラインで指定された数を使うように改造せよ.

確認 文字を2000万個おける領域を`malloc`で確保し，その領域を`'@'`で埋める．続いて標準出力に"bye..."と表示し，確保した領域を開放する.

第11章 プリプロセッサと分割コンパイル
— 11.1 プリプロセッサ —

- C言語ではコンパイル前に**プリプロセス**という処理が行われる.
- プリプロセッサというプログラムにより #で始まる文が処理される.
  - #include<システムフォルダにあるヘッダファイル>
    記述例）#include<stdio.h>
  - #include"自分で用意したヘッダファイル"
    記述例）#include"../mylib/mydefs.h"
  - #define マクロ名 処理や値
    記述例）#define N 100
  - #undef マクロ名
  - #if 〜 #else 〜 #endif
    コンパイルする領域を選択するために使う

第11章 プリプロセッサと分割コンパイル
― 11.2 分割コンパイル ―

プログラムの規模が大きくなってくると，機能ごとにソースファイルを分けてプログラムを開発する方がやりやすい（**分割コンパイル**）.