第1章 「まずは慣れよう」のまとめ

1.1 「まずは表示を行う」のまとめ

プログラムとコンパイル (教 p.2)

とは、ソースファイル（人間が読む／書く形式、C 言語の場合拡張子は ___）を実行ファイル（CPU が直接理解できる形式、Windows 上では拡張子は `.exe`）などに、翻訳することである。

注釈（コメント） (教 p.4)

とは、ソースファイル中の人間向けのメッセージで、コンパイラーは無視する部分である。C 言語では「___」から「___」までが注釈である。さらに新しい C 言語の仕様 (C99) では「`//`」から行末までという形も利用できる。

`printf` 関数 (教 p.6)

は、表示を行うための関数である。関数とは定義済みのプログラム部品である。 関数呼出しは処理の依頼であり、その時に渡すデータを_____という。

文 (教 p.7)

の末尾には、通常セミコロン「___」が必要である。「`{`」 と「`}`」の間に置かれた文は上から（同一行に複数文があるときは左から）順次実行される。

書式文字列と変換指定 (教 p.8) (教 p.378)

`printf` の第 1 引数のなかで、`%d` や `%f` などの「`%`」から始まる部分は変換指定と言い、第 2 引数以降の値に順に置き換えられる。整数（10 進数）を表示するための変換指定は「___」であり、浮動小数点数の表示は「`%f`」を使う。

文字列リテラル (教 p.11)

とは、一連の文字を二重引用符「"」～「"」で囲んだものであり、文字の並びを表す。一重引用符「'」～「'」は別の用途があるため、この目的には使用できない。

拡張表記 (教 p.11)

文字列中の「`\n`」は_____、「`\a`」は警報（ベル）、「`\t`」は_____を表す。このような、「`\`」を使った書き方を拡張表記という。（「`\`」は日本語環境では「`¥`」と表示されることがある。）その他の拡張表記については、教科書 p.250 を参照すること。

1.2 「変数」のまとめ

変数と宣言 (教 p.12)

とは、数値などのデータを収納するための「箱」（メモリの中の場所）である。C 言語では、変数を使うためには事前に宣言が必要である。

```c
int vx;         /* int 型の変数 vxの宣言 */
double fx;      /* double 型の変数 fxの宣言 */
int vx, vy;     /* int 型の変数 vxと vyの宣言 */
```

Q 1.2.1 C 言語の変数にはどのような名前をつけることができるか調べよ。（→教 p.108）

代入 (教 p.13)

とは、変数の値を書き換えることである。「変数 = 式」という形式で、右辺の式の値を左辺の変数に代入する。（代入される変数は必ず左辺に書く。「57 = vx」とは書けない。）

初期化 (教 p.14)

変数の生成（宣言）のときに値を入れることを初期化という。変数は次の形で初期化することができる。（初期化されないときは"不定値"が入る。）

```
型名 変数名1 = 初期化子1, … ,変数名n = 初期化子n;
```

初期化子 (initializer) は今のところ"式" (expression) と思っておいて良い。（あとで配列を紹介するときに、初期化子として式でない形が出てくる。）

1.3 「読み込みと表示」のまとめ

scanf 関数 (教 p.16)

とは、キーボードから数値などデータを読み込むための関数である。

```c
/* キーボードから整数を変数 noに読み込む */
scanf("____", &no);
```

「&」は「アンパサンド」と読む。詳しくは、ポインターのところで説明する。

(書き方に注意)

ところで scanf 関数は、セキュリティー上問題が多いことが指摘されている。そのため、 scanf の代わりに scanf_s 関数を使え、というメッセージを出すコンパイラーもある。しかし scanf_s 関数は、C 言語の標準仕様で必

須とされていないので、サポートしていない C コンパイラーも多い。そのため、本授業では `scanf_s` 関数は使用せず、`scanf` 関数を使う。

`puts` 関数 ⟪教 p.18⟫

は、文字列を出力し、最後に**改行**を行う。`printf` と異なり、書式変換は行わない。

文法のまとめ

式 (expression) とは

これまでのところ、

分類	一般形	補足説明
変数		`x`, `i` など
整数リテラル		`1`, `0`, `100`, `0xff` など
文字列リテラル	`"～"`	`"Hello\n"` など
関数呼出し	関数名（**式**，…，**式**）	`printf("Hello\n")` など

宣言 (declaration) とは

これまでのところ、

分類	一般形	補足説明
変数宣言	型 変数名 = 初期化子，…， 変数名 = 初期化子;	型は `int`, `double` など

第2章 「演算と型」のまとめ

2.1 「演算」のまとめ

演算子とオペランド (教 p.24)

とは、「+」,「-」,「*」,「/」のように 演算の働きを持った記号のことである。(教科書 p.221 に C 言語のすべての演算子の表がある。)

＿＿＿＿＿とは、その演算の対象となる式（変数や定数など）のことである。

乗除演算子と加減演算子 (教 p.25)

```
整数 / 整数
```

という演算では、整数としての割算 (小数点以下は＿＿＿) の結果が得られる。

```
整数 % 整数
```

では、＿＿＿（余り）を求める。

Q 2.1.1 次の式の出力は?
① `printf("%d", 3 / 10)` … ＿＿　② `printf("%d", 11 % 3)` … ＿＿

printf 関数での「%」文字の表示 (教 p.25)

「`%%`」と 2 つ重ねることで「`%`」という文字そのものを出力することができる。`puts` 関数では、「`%`」はそのまま出力される。

複数の変換指定 (教 p.27)

変換指定（「`%d`」など）が複数あるときは、第 2 実引数, 第 3 実引数, ...が順に対応する。

Q 2.1.2 次の式の出力は?
① `printf("%d/%d", 10, 24)` … ＿＿＿＿
② `printf("(%d, %d, %d)", 12, 34, 5)` … ＿＿＿＿＿＿＿

代入演算子 (教 p.29)

「=」演算子は（単純）代入演算子と呼ばれる。

式と代入式 (教 p.29)

変数や定数、それらを演算子で結合したものを＿＿という。

式文 (教 p.29)

式のあとに「;」をつけて文にしたものを式文という。

2.2 「型」のまとめ

「(」～「)」 (教 p.30)

式のなかで先に演算する箇所を示すために丸括弧「(」～「)」を用いる。他の括弧（{, }, [,] など）は別の用途があるため、この目的には使用できない。

整数型と浮動小数点型 (教 p.30)

double 型とは、いわゆる実数 (正確には浮動小数点数) を扱うための型である。もちろん、実数と言っても精度には限界がある。（十進で 15 桁くらい。）

double 型の演算 (教 p.33)

実数を読み込むときは「%lf」を用いる。

```
1  /* キーボードから実数を変数 fxに読み込む */
2  scanf("___", &fx);
```

変換指定の使い分け (教 p.33)

以下の表くらいは、暗記しておくこと。

	int	double
printf	%d	注
scanf	__	____

注: _____

型と定数 (教 p. 32)

5, 10 などは整数定数、2.718 などは小数点を含むものは浮動小数点定数と呼ばれる。基本的に整数定数は int 型、浮動小数点定数は double 型である。

型と演算 (教 p.34)

```
実数型 / 実数型
```

の演算では、切捨ては行わず、通常の割算が行われる。一方 int と double が混じっている場合、

```
整数型 / 実数型
```

や

```
実数型 / 整数型
```

の場合も、整数 (int) 型のオペランドが＿＿＿＿＿＿が行われて実数 (double) 型になり、double 型の演算となる。

キャスト (cast) (教 p.36)

とは、明示的に＿＿＿＿＿＿することである。

```
( 型 ) 式
```

という形で、「式」の値を「型」としての値に変換する。例えば、

```
1  int a, b;
2  …
3  … (double)(a + b) / 2 …
```

では、double 型としての割算が行われる。 (演算子の優先順位に注意する (教 p.221)。割算よりもキャストが先に行われる。)

なお、double から int へのキャスト

```
1  double x = -2.8;
2  printf("%d", (int)x);
```

は切捨てになる。

Q 2.2.1 次の式の出力は? (%f は少数第 6 位まで出力する。)

① `printf("%f", (double)1 / 2)` … ＿＿＿＿＿＿

② `printf("%f", (double)(1 / 2))` … ＿＿＿＿＿＿

変換指定 (教 p.38)

以下のような変換指定は必要に応じて調べれば良い。

説明	例	出力
桁数を揃える	`printf("[%3d]", 1)`	`[  1]`
桁数を揃え先頭を 0 で埋める	`printf("[%03d]", 1)`	`[001]`
小数点以下の桁数を指定する	`printf("[%.3f]", 3.1415926)`	`[3.142]`
16 進数で表示する (小文字)	`printf("[%x]", 127)`	`[7f]`
16 進数で表示する (大文字)	`printf("[%X]", 127)`	`[7F]`

printf 関数が浮動小数点数を出力するとき、ある桁 (デフォルトでは小数第 6 位) まで表示して、あとは打ち切るが、この丸め方は C 言語の仕様では特に決められていないので処理系に依存する。一般的には「偶数への丸め」(bankers' rounding) を採用する処理系が多いようである。つまり、`printf("%.0f, %.0f\n", 2.5, 3.5)` は "2, 4" と出力する。

Q 2.2.2 次の printf 関数の呼出しの出力は?

① `printf("%.4f", 1.0 / 3)` … _____
② `printf("%x", 32)` … __

文法のまとめ

式 (expression)

に以下を追加、

分類	一般形	補足説明
単項演算	単項演算子 式	単項演算子は +, −, &, … など
二項演算	式 二項演算子 式	二項演算子は +, −, *, /, =, … など
かっこ	（ 式 ）	演算の順番を指定するため、 丸括弧で囲んだもの (教 p.30)
浮動小数点数 リテラル		`3.14`, `2.0`, `6.02e23`, `6.6626e-34` など (教 p.32)
キャスト	（ 型 ）式	明示的な型変換 (教 p.36)

文 (statement) とは

これまでのところ、

分類	一般形	補足説明
式文	式 _	式は通常、代入式か関数呼出し[†]（教 p.29）

[†] つまり副作用があるもの

第3章 「プログラムの流れの分岐」のまとめ

3.1 「if 文」のまとめ

if 文 (教 p.44)

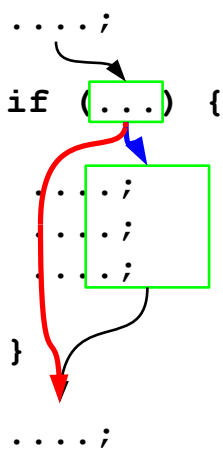
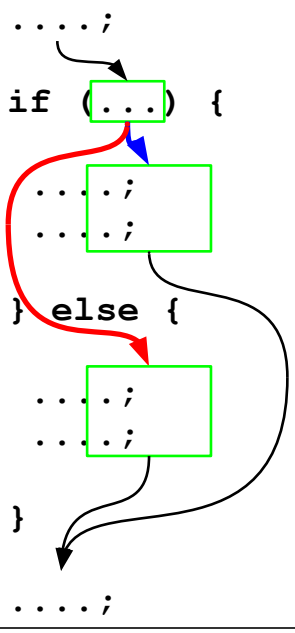
```
if ( 式1 ) 文1
```

という形のこと、式$_1$ を評価して、その値が＿＿(すなわち真) であれば、＿＿を実行する。式$_1$の値が＿＿(すなわち偽) であるときは、 何もしない。

else 付きの if 文 (教 p.46)

```
if ( 式1 ) 文1 else 文2
```

という形のこと、式$_1$ を評価して、その値が＿＿(すなわち真) であれば、＿＿を実行する。式$_1$ の値が＿＿(すなわち偽) であるときは、＿＿を実行する。

if 文	if ～ else 文
	

Q 3.1.1 次のプログラムの断片の出力は?（出力なしのときは「無」と書く。）
① `if (0) printf("X");` 答: ＿＿
② `if (0) printf("Y"); else printf("Z");` 答: ＿＿

等価演算子 (教 p.50)

「==」は両辺の値が等しければ＿＿（つまり真）を、等しくなければ＿＿（つまり偽）を返す演算子である。「==」と逆に等しくないかどうかを判定する演算子

は「　　」である。

関係演算子 (教 p.52)

以下の4つがある。

<	左辺が右辺よりも小さいとき真
>	左辺が右辺よりも大きいとき真
<=	左辺が右辺よりも小さいか等しいとき真
>=	左辺が右辺よりも大きいか等しいとき真

「≦」とか「≧」はもちろん、「=<」とか「=>」という演算子は**ない**ので注意する。

入れ子になったif文 (教 p.52)

```
1  if (no == 0)
2      puts("その数は 0です。");
3  else if (no > 0)
4      puts("その数は正です。");
5  else
6      puts("その数は負です。");
```

これは、単に else の次の文が、また if 文になっているだけのことである。

Q 3.1.2 もし、次のようになっていれば、no が 0 のときの出力はどうなるか? 改行は ↵ で表せ。

```
1  if (no == 0)
2      puts("その数は 0です。");
3  if (no > 0)
4      puts("その数は正です。");
5  else
6      puts("その数は負です。");
```

答: _____

評価 (教 p.55)

式の値を調べる（ために実行する）ことを　　する (evaluate) という。

条件演算子 (三項演算子) (教 p.58)

```
式1 ? 式2 : 式3
```

まず 式$_1$ を評価し、その値が、

非 0 (真) であれば、　　を評価して、その値を返す。　　は評価しない。
0 (偽)　であれば、　　を評価して、その値を返す。　　は評価しない。

複合文 (ブロック) (教 p.60)

文の並びを波括弧 (ブレース ― 「 」 と 「 」 ―) で囲んだものを複合文またはブロックという。 (文のほかにいくつかの宣言があってもよい。) 複合文は構文上単一の文と見なされる。 複合文中の文は上 (左) から順に一つずつ実行される。

通常、if 文の制御する文 (後述の do 文、 while 文、for 文などでも同様) は、たとえ一つの文でも (間違いを避けるため)波括弧で囲んでブロックにする。

△ 望ましくないスタイル	◎ 望ましいスタイル

`if (n1 > n2)`
`  printf("hello");`
`else`
`  printf("hi");`
	`if (n1 > n2) {`
	`  printf("hello");`
	`} else {`
	`  printf("hi");`
	`}`

教科書の例題は望ましいスタイルでないものが多いので、特に注意する。 この授業の課題の解答は「望ましいスタイル」で提出すること。 （教科書 p.61 下のほうの ▷） (教 p.61)

繰り上がりの計算 (addtime.c)

```c
#include <stdio.h>

int main(void) {
    int hour1, minute1, hour2, minute2,
        hour3, minute3;

    printf("hour1 を入力して下さい:");
    scanf("%d", &hour1);
    printf("minute1 を入力して下さい:");
    scanf("%d", &minute1);
    printf("hour2 を入力して下さい:");
    scanf("%d", &hour2);
    printf("minute2 を入力して下さい:");
    scanf("%d", &minute2);

    hour3   = hour1 + hour2;
    minute3 = minute1 + minute2;

    if (minute3 >= 60) {
        hour3   = hour3 + 1 ;
        minute3 = minute3 - 60;
    }
    printf("その和は、%d 時間 %d 分です。\n",
           hour3, minute3);
    return 0;
}
```

Q 3.1.3 次のように入力したとき、出力はどうなるか

```
hour1 を入力して下さい: 1
minute1 を入力して下さい: 40
hour2 を入力して下さい: 2
minute2 を入力して下さい: 30
```

2 つの数を大きい順に並べる (maxswap.c)

```c
#include <stdio.h>

int main(void) {
    int n1, n2, tmp;

    printf("整数 1 を入力して下さい:"); scanf("%d", &n1);
    printf("整数 2 を入力して下さい:"); scanf("%d", &n2);

    if (n2 > n1) {    /* n1 と n2 を入れ換える */
        tmp = n1;
        n1 = n2;
        n2 = tmp;
    }
    printf("大きい方は %d です。小さい方は %d です。\n",
           n1, n2);
    return 0;
}
```

Q 3.1.4 10 ～ 12 行めが、もし

```
    n2 = n1;
    n1 = n2;
```

だったら、整数 1 が 3、整数 2 が 4 のときどう出力されるか？

（発展）ぶら下がりの else (dangling else)

```c
if (h < 12) if (h < 6) printf("A"); else printf("B");
```

は、どのように文法的に解釈されるか?

解釈 1:

```c
if (h < 12) {
    if (h < 6) {
        printf("A");
    } else {
        printf("B");
    }
}
```

解釈 2:

```c
if (h < 12) {
    if (h < 6) {
        printf("A");
    }
} else {
    printf("B");
}
```

Q 3.1.5 上の解釈 1, 解釈 2 は `h` が以下の値のとき、どのように出力するか?
（出力なしのときは「無」と書く。）

	解釈 1	解釈 2
h = 3		
h = 9		
h = 15		

さらに次の文の空欄を埋めよ。

ぶら下がりの else は、解釈 ＿ と同等である。つまり、 ＿＿＿＿ の if と対応する。

論理演算子 (教 p.62)

は以下のような演算子である。左右非対称である — つまり左オペランドを評価して値が決まれば、 ＿＿＿＿＿＿ は評価しない — ことに注意する（短絡評価）。

演算子	呼び方	説明
&&	論理 AND 演算子 かつ	左オペランドを評価して、0 (偽) であれば、 1 を返す。非 0 (真) であれば、 右オペランドを評価して、0 であれば 0 を、非 0 であれば 1 を返す。
||	論理 OR 演算子 または	左オペランドを評価して、非 0 (真) であれば 1 を返す。0 (偽) であれば、 右オペランドを評価して、0 であれば 0 を、非 0 であれば 1 を返す。

Q 3.1.6 次のプログラム（の一部）は誤っている（作成者の意図に反している）可能性が高い。このままだと、どう出力するか答えよ。 (教 p.65)

1.
```
int a = 1;
if (a == 0);
    printf("hello");
```

＿＿＿＿＿＿＿＿＿＿＿＿

2.
```
int a = 0;
if (a = 0)
    printf("hello");
```

＿＿＿＿＿＿＿＿＿＿＿＿

3.
```
int a = 2, b = 2, c = 2;
if (a == b == c)
    printf("hello");
```

＿＿＿＿＿＿＿＿＿＿＿＿

4.
```
int a = 10;
if (-1 <= a <= 1)
    printf("hello");
```

＿＿＿＿＿＿＿＿＿＿＿＿

5.
```
int a = 2;
if (a & -1 <= a)
```

```
    printf("hello");
```

3.2 「switch 文」のまとめ

switch 文と break 文 (教 p.66)

ある式の値 (整数型) によって、プログラムの流れを複数に分岐するときに使う。

```
switch ( 式1 ) 文1
```

文$_1$は、通常、複合文（ブロック）である。 switch 文は式$_1$ を評価して、文$_1$ の中の _____ と「:」の間に書かれた定数と一致するところにジャンプする。 (どの case にも一致しないときは、 _____ にジャンプする。) ただし、break 文に出会うと、一気に switch 文を飛び出る。逆に break 文がなければ、そのまま次の文を実行する。

`case〜:` や `default:` のようにプログラムの飛び先を示す目印を _____ (名札) と呼ぶ。

> default の綴りを間違ってもコンパイルエラーにならないので注意すること。

Q 3.2.1 次のプログラムの断片の出力は?

```
1  int no = 2;
2
3  switch (no) {
4      case 1:  printf("A");
5      case 2:  printf("B");
6      case 3:  printf("C");
7      default: printf("D");
8  }
```

文法のまとめ

文 (statement)

に以下を追加、

分類	一般形	補足説明
if 文	`if` （式）文	(教 p.44)
else 付き if 文	`if` （式）文 `else` 文	(教 p.46)
複合文(ブロック)	{ 宣言 … 文 … }	(教 p.60)
switch 文	`switch` （式）文	(教 p.66)
ラベル付き文	`case` *整数リテラル* : 文 `default` : 文	(教 p.67)
break 文	`break` ;	(教 p.67)

式 (expression)

に以下を追加、

分類	一般形	補足説明
三項演算子	式 ? 式 : 式	(教 p.58)

第4章 「プログラムの流れの繰返し」のまとめ

4.1 「do 文」のまとめ

do 文 (教 p.74)

```
do 文1 while ( 式1 ) ;
```

まず、文$_1$（ループ本体と呼ばれる）を実行する。式$_1$ が＿＿（真）である限り、＿＿の実行を繰り返す。

注: 必ず 1 回はループ本体を実行する。

後述の while 文や for 文に比べると、 do 文の実際の**使用頻度は低い。**

プログラムの実行が止まらなくなったときは、Ctrl-c で強制終了する。

複合文（ブロック）内での宣言 (教 p.75)

（do 文に限らず）ブロックの中で宣言された変数は、そのブロックでのみ有効である。

論理否定演算子とド・モルガンの法則 (教 p.77)

単項演算子の「＿」は、真偽を逆にする演算子で、論理否定演算子とも言う。

複合代入演算子 (教 p.80)

「*=」, 「/=」, 「%=」, 「+=」, 「-=」, などのことである。例えば、 sum += t は＿＿＿＿＿＿＿＿とほぼ同じ意味になる。つまり、この代入を実行した後の sum の値は、実行する前の sum の値に t を加えた値になる。

後置増分演算子と後置減分演算子・前置増分演算子と前置減分演算子 (教 p.81) (教 p.88)

a++	a の値を一つだけ増やす	（式全体の値は、＿＿＿＿＿＿の値）
a--	a の値を一つだけ減らす	（式全体の値は、＿＿＿＿＿＿の値）
++a	a の値を一つだけ増やす	（式全体の値は、＿＿＿＿＿＿の値）
--a	a の値を一つだけ減らす	（式全体の値は、＿＿＿＿＿＿の値）

Q 4.1.1 次のプログラムの断片の出力は何か? 答＿＿＿＿＿

```
1  n = 3;
2  do {
3      printf("%d ", n);
4  } while (n-- > 0);
```

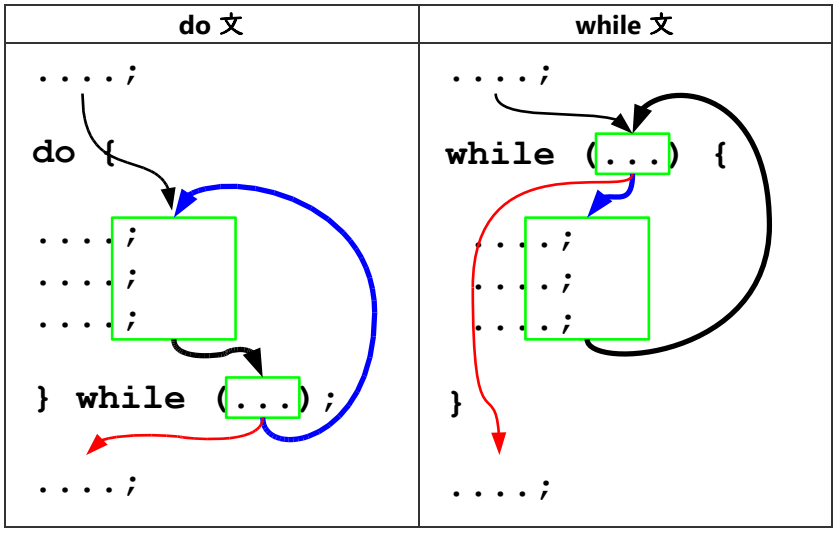
4.2 「while 文」のまとめ

while 文 (教 p.82)

```
while ( 式1 ) 文1
```

式$_1$ が＿＿（偽）でない限り、＿＿（ループ本体）の実行を繰り返す。

注: ループ本体が一度も実行されないことがある。



Q 4.2.1 次のプログラムの断片の出力は何か? 答 ＿＿＿

```c
n = 3;
while (n-- > 0) {
    printf("%d ", n);
}
```

文字定数と putchar 関数 (教 p.86)

文字定数は 1 文字を＿＿＿＿＿「'」～「'」で囲んだもののことである。
「\n」や「\t」などの拡張表記は 1 文字として扱われる。

putchar 関数は、引数として受け取った文字を標準出力に出力する。

整数値を逆順に表示

Q 4.2.2 教科書 List 4-10 に似ている、次のプログラム (reverse.c) について...

```c
#include <stdio.h>

int main(void) {
    int no = 12345;

    do {
        printf("%d", no % 10); /* ① */
        no /= 10; /* ② */
    } while (no > 0); /* ③ */

```

```
11      return 0;
12  }
```

各行を実行後の no の値はどうなるか？

	1 回目	2 回目	3 回目	4 回目	5 回目
7 行め					
9 行め					

また no の初期値が 12345 以外のときを考える。 6 ～ 9 行めの do ～ while 文の代わりに、教科書 List 4-10 と同じ次のような while 文に変更すると、 no がいくつのときに、振舞いがどう変わるか？

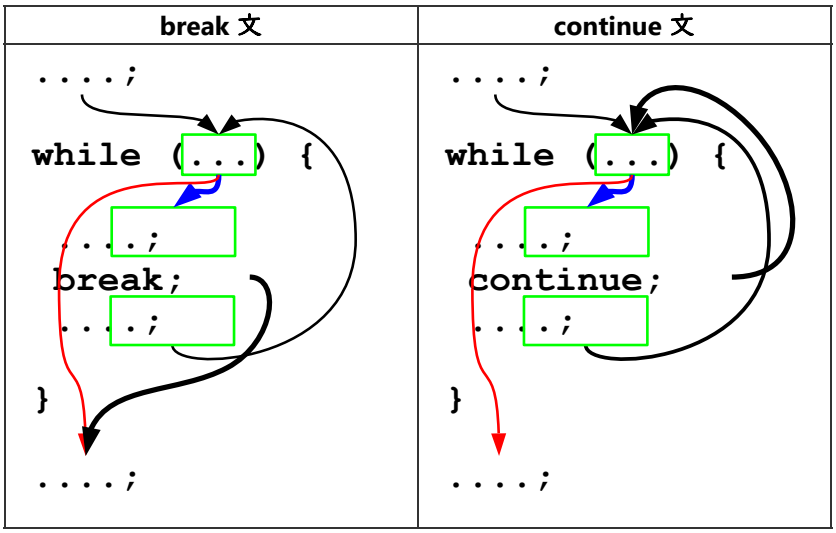
```
1       while (no > 0) {
2           printf("%d", no % 10);
3           no /= 10;
4       }
```

no が＿＿＿のときに、 変更前は、＿＿＿＿＿＿＿＿＿＿が、変更後は、＿＿＿＿＿＿＿＿。

break 文と continue 文 (教 p.92)

break 文は（もっとも内側の）繰返し文（do 文, while 文, for 文）を＿＿＿＿。（外側の繰返し文を一気に抜け出ることはできない。）

continue 文は（もっとも内側の）繰返し文のはじめ（do 文, while 文の場合は条件式、 for 文の場合は第 3 式）にもどる。



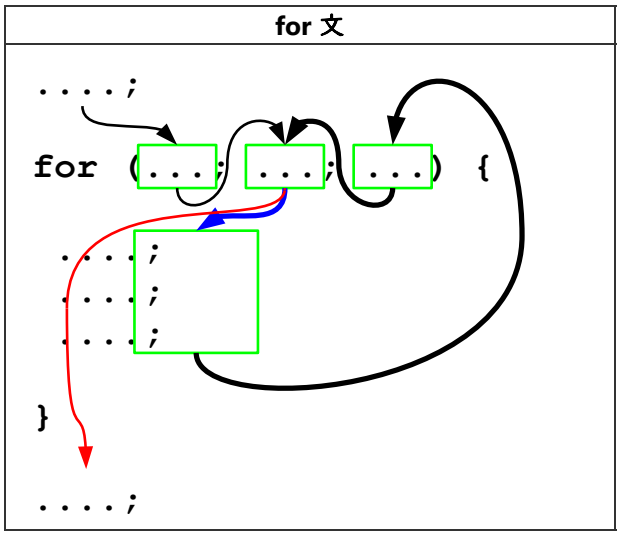
4.3 「for 文」のまとめ

for 文 (教 p.94)

```
for ( 式1; 式2; 式3 ) 文1
for ( 宣言1 式2; 式3 ) 文1
```

ループに入る前にまず＿＿（または ＿＿）を実行する。
＿＿が非0（真）である間、＿＿（ループ本体）と ＿＿を繰返し実行する。

詳細: 式$_1$～式$_3$は省略可能である。 式$_2$を省略したときは、1 (真) と書くのと同じ意味になる。



左のような for 文は右に示す while 文と（ほぼ）等価である。

for 文	（ほぼ）等価な while 文

`for (A ; B ; C) {`
　ループ本体
`}`
	＿＿＿＿;
	`while (` ＿＿＿ `) {`
	　ループ本体
	　＿＿＿＿＿＿＿＿;
	`}`

では、なぜ左の書き方が好まれるか？ — 繰り返しを制御する変数に対する処理が、一箇所にまとまっていて、一目でどのような繰り返しか理解しやすいからである。

for 文による一定回数の繰返し ((教 p.97))

for 文には、良く使う決まり文句的な形がある。

① `for (i = 0; i < n; i++) ...`　　i が ＿＿＿＿＿＿ n 回繰り返す
② `for (i = 1; i <= n; i++) ...`　　i が ＿＿＿＿＿＿ n 回繰り返す
③ `for (i = n; i > 0; i--) ...`　　i が ＿＿＿＿＿＿ n 回繰り返す
④ `for (i = n - 1; i >= 0; i--) ...` i が ＿＿＿＿＿＿ n 回繰り返す

Q 4.3.1 上記の形で、`n` が 0 や負の数だった場合はどうなるか？

＿＿＿＿＿＿＿＿

Q 4.3.2 上記のそれぞれの形でループを抜けたあとの `i` の値は何か? （ただし、`n` は非負とする。）

① ＿＿ ② ＿＿ ③ ＿＿ ④ ＿＿

Q 4.3.3 つぎのような階乗の計算をするプログラム fact.c について…

```c
#include <stdio.h>

int main(void)  {
    int i, n, fact = 1;

    printf("正の数を入力してください。 ");
    scanf("%d", &n);
    for (i = 1 /* ① */;  /* 通常 1 行を 3 行にわけた。*/
                i <= n  /* ② */;
                        i++ /* ③ */) {
        fact = fact * i;
    }
    printf("あなたの入力した数の階乗は %dです。\n", fact);

    return 0;
}
```

変数 i, fact の値の変化（n が 4 のとき）はどうなるか？

行	1 回め		2 回め		3 回め		4 回め	
	i	fact	i	fact	i	fact	i	fact
9								
11		→		→		→		→
10	→		→		→		→	

次のプログラム (polygon.c) は正 n 角形の座標を順に出力する。

```c
#include <stdio.h>
#include <math.h> /* sin, cos のために必要 – 教 p.217*/

int main(void) {
    int n, i;

    printf("nを入力して下さい: "); scanf("%d", &n);
    for(i = 0; i < n; i++) {
        double theta1 = 2 * 3.1416 *  i / n;
        double theta2 = 2 * 3.1416 *  (i + 1) / n;
        printf("%.3f %.3f %.3f %.3f\n",
               100 * cos(theta1), 100 * sin(theta1),
               100 * cos(theta2), 100 * sin(theta2));
    }

    return 0;
}
```

式文と空文 (教 p.101)

文 (statement) に以下を追加する。

分類	一般形	補足説明
空文	;	“何もしない”文、{} と書いても意味は同じ。

4.4 「多重ループ」のまとめ

2 重ループ (教 p.102)

for 文や while 文などのループ本体が、また for 文や while 文などの繰返し文を含んでいることを二重ループという。二重・三重・… ループをまとめて、多重ループという。特別な文法や実行規則があるわけではない。

Q 4.4.1 次の二重ループのプログラム (triangle.c) は、数字の三角形を出力する。

```
#include <stdio.h>

int main(void) {
    int i, j, n;
    printf("n を入力して下さい: "); scanf("%d", &n);
    for (i = 1/*①*/; i <= n/*②*/; i++/*③*/) {
        for (j = 1/*④*/; j <= i/*⑤*/; j++/*⑥*/) {
            printf("%d", j % 10);/*⑦*/
        }
        printf("\n");/*⑧*/
    }
    return 0;
}
```

```
n を入力して下さい: 4↲
1
12
123
1234
```

n = 3 のとき、ループ内の式、文はどの順で実行されるか？

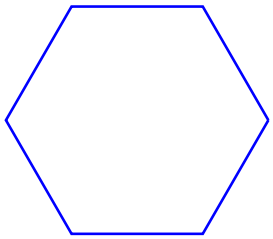
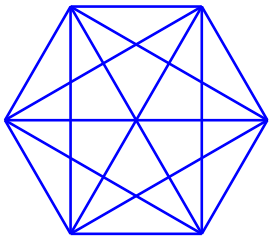
①②____

____

Q 4.4.2 次の二重ループを使ったプログラム (diamond.c) は、ダイアモンド図形という図形の座標を順に出力する。

```
#include <stdio.h>
#include <math.h> /* sin, cos のために必要 — 教 p.217 */

int main(void) {
    int n, i, j;

    printf("nを入力して下さい: "); scanf("%d", &n);
    for (i = 0/*①*/; i < n - 1/*②*/; i++/*③*/) {
        double th1 = 2 * 3.1416 *  i / n/*④*/;
        for (j = i + 1/*⑤*/; j < n/*⑥*/; j++/*⑦*/) {
            double th2 = 2 * 3.1416 *  j / n/*⑧*/;
            printf("%.3f %.3f %.3f %.3f\n",
                   100 * cos(th1),
                   100 * sin(th1),
                   100 * cos(th2),
                   100 * sin(th2));/*⑨*/
        }
    }

    return 0;
```

```
21 }
```

正 n 角形（n = 6）	ダイアモンド図形（n = 6）
	

`n = 4` のとき、ループ内の式、文はどの順で実行されるか？

①② _____

コンマ演算子 (教 p.233)

```
式1, 式2
```

という**式**は、$式_1$、$式_2$をこの順に評価し、 ____ の値を**捨て**、 ____ の値（と型）を持つ。

注: 関数を呼び出すときに引数を区切るコンマ（例： `printf("%d %d", i, j)`）はコンマ演算子**ではない**。

コンマ演算子の例 (comma.c)

```c
#include <stdio.h>

int main(void) {
    int i, j;
    for (i = 0, j = 6; i < j; i++, j--) {
        printf("i = %d, j = %d, ", i, j);
    }
    return 0;
}
```

Q 4.4.3 このプログラムの出力はどうなるか？

_____

また `i = 0, j = 5` の場合はどうか?

_____

4.5 「プログラムの要素と書式」のまとめ

キーワード (教 p.108)

`if` や `else` など C 言語にとって特別な意味のある単語を _____(keyword) と呼ぶ。変数名などに使用することはできない。(ただし、変数名などの一部に使用するのは構わない。)

自由形式 《教 p.110》

C 言語では、原則としてレイアウト（空白の数や改行）はプログラム の意味に影響を及ぼさない。空白がいくつ連続しても空白 1 文字と同じであり、改行も空白と同じである。

注意すべきところ:

- 前処理指令 (`#include` ..., `#define` ...) などの途中では、改行できない。
- 文字列リテラル、文字定数の途中でも、改行できない。

文法のまとめ

文 (statement)

に以下を追加する。

分類	一般形	補足説明
do 文	`do` 文 `while ( 式 ) ;`	（教 p.74）
while 文	`while ( 式 )` 文	（教 p.82）
continue 文	`continue ;`	（教 p.93）
for 文	`for ( 式 ; 式 ; 式 )` 文	（教 p.94）
for 文	`for ( 宣言 式 ; 式 )` 文	（教 p.94）

式 (expression)

に以下を追加する。

分類	一般形	補足説明
後置演算	式 後置演算子	C の後置演算子は ++, -- のみ（教 p.81）
コンマ演算子	式 , 式	（教 p.233）

第5章 「配列」のまとめ

5.1 「配列」のまとめ

配列 (教 p.116)

同一の型のデータを集めて、番号（____、そえじ）でアクセスできるようにしたもの。 C 言語の配列の添字は____から始まる。

```
/* 初期化しないとき */
int va[5];
/* 配列の初期化は、式をコンマで区切って { } で囲む。*/
int vb[5] = { 15, 20, 30 };
/*（残りの要素は （i） で初期化される。） */

/* 初期化子を代入することはできない。— 教 p.121 */
vb = { 15, 20, 30, 0, 0 };
/* 配列同士の代入はできない。 — 教 p.130*/
vb = va;
```

(i) ____

Q 5.1.1 次のプログラムの断片の出力は?

```
int a[] = { 3, 7, 5 };
printf("%d", a[2]);
```

____

配列の走査 (教 p.118)

配列は for 文と相性が良い。5 個の要素を持つ配列の各要素に対して同じ操作を行なうときには次のような for 文を使う。

```
for ( (i) ; (ii) ; (iii) ) {
  a[i] = …;
}
```

(i) ______ (ii) ______ (iii) ____

配列の全要素の並びを反転する (教 p.123)

2 つの変数 x, y を入れ替えるのに、

```
x = y; y = x;
```

と書いてもダメでなぜ?、別の変数（例えば temp）を一つ用意して、

```
______________________________
```

と書く必要がある。

なお、List 5-8 は真ん中のループを、以下のようにコンマ演算子を使うかたちにすることも可能である。

```
  …
  for (i = 0, j = 6; i < j; i++, j--) {
    int temp = x[i];
    x[i]     = x[j];
    x[j]     = temp;
  }
  …
```

オブジェクト形式マクロ（定数マクロ）《教 p.124》

プログラム中で繰り返し使う定数は名前をつける。

```
#define NUMBER 5
```

この指令は NUMBER というオブジェクト形式＿＿＿＿を定義する。マクロは他のコンパイル処理に先だって、一括して置換される。マクロを定義すると、次のような利点がある。

- 値の変更が容易になる。
- 定数の意味がわかり易くなる。秘密の数値（マジックナンバー）を直接プログラムに埋め込まないこと！

マクロが使われるのは、次のような箇所である。

- 配列の要素数など、文法上定数が要求されるところ
- 円周率などの数学定数・物理定数など絶対に変らない定数

（これら以外の箇所では、通常の変数を使うのが普通である。）

C99 規格では、配列の要素数に変数を使って、次のような書き方も一応可能になった。

```
  int n = 3;
  int a[n];  /* C99 では許容だが */

  for (i = 0; i < n; i++) {
    a[i] = 0;
  }
```

しかし、すべての処理系がこれをサポートしなくても良いことになっているので非推奨とする。（演習の解答では使ってはいけない。）

マクロ名は通常すべての文字を＿＿＿＿とする慣習がある。小文字を使ってもコンパイルエラーになるわけではないが、強く非推奨とする。（逆に変数名は必ず小文字を混ぜること。）

代入式の評価《教 p.126》

代入「変数 = 式」も式であり、値（代入された値と同じ）を持つ。代入演算子は右結合である（右側から行われる）。つまり、`x = y = 0` は＿＿＿＿＿＿＿＿と解釈される。

Q 5.1.2 List 5-11 の 2 つめのループは、なぜ（`i = 0` ではなく）`i = 1` から始まるのか？

Warning 《教 p.131》

発音は /ˈwɔːnɪŋ/ で、カタカナでは＿＿＿＿＿＿が近い。警告という意味で、エラーではない（コンパイルができないということはない）が間違っている可能性が高いことを示す。

Q 5.1.3 次の break 文を使用したプログラム (breakTest.c) について考える。

```c
#include <stdio.h>
#define NUM 5

int main(void) {
    int i;
    int a[NUM] = {1, 2, -2, -4, 5};
    for (i = 0; i < NUM; i++) {
        if (a[i] < 0) {
            break; /* continue; も試せ。 */
        }
        printf("a[%d] = %2d\n", i, a[i]);
    }
    return 0;
}
```

上のプログラムの出力はどうなるか？改行は ↲ で表せ。

＿＿＿＿＿＿＿＿＿＿＿＿

また break を continue に変えたときはどうなるか？

＿＿＿＿＿＿＿＿＿＿＿＿

5.2 「多次元配列」のまとめ

多次元配列 《教 p.132》

各要素が配列であるような配列、言い替えれば 2 つ以上の添字を持つ配列のこと。ただし、物理的には一次元に配置される。（Fig.5-10参照） 《教 p.132》

```c
int x[2][3] = {{ 1, 2, 3 }, { 4, 5, 6 }};
```

Q 5.2.1 上の二次元配列 `x` のメモリ上の配置を Fig.5-10 のような図で表わせ。

Q 5.2.2 次のプログラムの断片の出力は?

```
int a[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
printf("%d", a[1][2]);
```

文法のまとめ

宣言 (declaration)

に以下を追加する。

分類	一般形	補足説明
配列宣言	*型 変数* [*定数*] = { *式* , … , *式* };	「=」以降の 灰色の部分は省略可能

式 (expression)

に以下を追加する。

分類	一般形	補足説明
配列アクセス	式 [式]	a[1],b[2][3] など

第6章 「関数」のまとめ

6.1 「関数とは」のまとめ

関数とは (教 p.142)

繰返し使うプログラムの一部の命令列を部品として、再利用できるようにしたもの。（他の言語ではサブルーチン・手続き・副プログラムなどとも呼ばれる。）

関数定義 (教 p.143)

分類	一般形
関数定義	***型*** *関数名*（ ***型*** *変数名*$_1$，…，***型*** *変数名*$_n$ ）*複合文*

関数定義には型が必要である。

かっこの中の変数（変数名$_1$～変数名$_n$）は＿＿＿＿(parameter) と呼ばれる。

C 言語の関数定義は必ずプログラムのトップレベルに書く。（つまり、関数定義の中に関数定義は書けない。他のブレース「`{`」～「`}`」の中に入らない。）また、後述のプロトタイプ宣言をしているときを除き、使うよりも先（上）に書く。

Q 6.1.1 次のような関数を定義せよ。

1. `int` 型の引数 `n` を受け取って、`2 * n + 1` を返す関数 `foo`

   ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

2. `double` 型の引数 `x` を受け取って、`x / 2` を返す関数 `bar`

   ＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

関数呼出し (教 p.144)

分類	一般形
関数呼出し式	*関数名*（ *式*$_1$，…，*式*$_n$ ）

関数呼出しには型は不要である。

かっこの中の式（式$_1$～式$_n$）は＿＿＿＿(argument) と呼ばれる。

これで文法上、式 (expression) になる。

注: ここのコンマはコンマ演算子ではない。

関数を呼出すと、プログラムの実行は呼び出された関数の定義の先頭に移り、実引数の値が仮引数の変数の初期値になる。

return 文 (教 p.145)

分類	一般形	補足説明
return 文	`return` 式 `;` `return ;`	値を返す場合 値を返さない場合

関数の呼出し元に値を返す。つまり、プログラムの実行が関数の呼出し元に戻り、 return 文の式の値が、関数呼出し式の値になる。

関数本体の最後の閉じブレース「}」にたどり着いたときも、プログラムの実行は関数の呼出し元に戻る。

値渡し（pass by value） (教 p.150)

値呼び（call by value）とも言う。引数は基本的に値がやりとりされる。関数呼出しのたびに仮引数のための新しいメモリ領域（"箱"）が用意される。仮引数の変数に値の代入を行なっても、呼出し元の実引数は＿＿＿＿＿＿。

Q 6.1.2 次のプログラム (cbv.c) の出力は?

```c
#include  <stdio.h>

void i_set(int v) {
    v = 0;
}

int main(void) {
    int a1 = 1, a2 = 3;

    i_set(a1);
    i_set(a2);

    printf("a1 = %d\n", a1);
    printf("a2 = %d\n", a2);

    return 0;
}
```

＿＿＿＿＿＿＿＿

6.2 「関数の設計」のまとめ

値を返さない関数 (教 p.152)

関数の定義の返却値型を書くところに ＿＿＿ と書く。

引数を受け取らない関数 (教 p.154)

関数の定義の仮引数のならびを書くところに＿＿＿と書く。関数の定義の仮引数のならびを空にすると、古い C 言語の規格で書かれていると見なされてしまい、意味が変ってしまう。関数を呼出すときは () のなかは空にする。

Q 6.2.1 引数を受け取らず `"Hey!"` と出力する（改行はしない）、値を返さない関数 `hey` を定義せよ。

ブロック有効範囲・ファイル有効範囲 (教 p.155)

変数には有効範囲(スコープ、scope)がある。同じ変数名でも有効範囲が異なれば別の変数になる。

- ブロック（教 p.60）の中で宣言された変数（局所変数、ブロック有効範囲を持つ変数）は、宣言された場所から、＿＿＿＿＿＿までが有効範囲となる。
- 関数の仮引数は、その関数本体が有効範囲となる。
- 関数の外で宣言された変数（大域変数・グローバル変数、ファイル有効範囲を持つ変数）は、宣言された場所から＿＿＿＿＿＿までが有効範囲となる。どうしても必要でない限り、使わないこと。

関数プロトタイプ宣言 (function prototype declaration) (教 p.157)

関数を定義より前に（あるいは定義されているのと別のファイルで）使用する場合は、関数プロトタイプ宣言が必要である。

以下を“宣言”に追加する。

分類	一般形
関数プロトタイプ宣言	***型*** *関数名*（ ***型*** *変数名*，…，***型*** *変数名* ）;

変数名は省略可能である。

関数定義がその呼出しよりも前にある場合は、定義が宣言を兼ねるのでプロトタイプ宣言は不要である。（いずれにしても、実行は常に `main`から開始される。）

ヘッダーとインクルード (教 p.158)

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

の stdio.h は、`printf`, `putchar` などの関数のプロトタイプ宣言が集められたもの（通常はファイル）である。このようにプロトタイプ宣言やマクロの定義が集められたものを＿＿＿＿と呼ぶ。

`#include` はヘッダーの内容を、そっくりそのままその場所に読み込む（インクルードする）指令である。

処理系により標準のヘッダーがおかれる場所は異なる。

ライブラリー関数（前もって用意された関数）を利用するときは、ほとんどの場合、適切なヘッダーをインクルードする必要がある。例えば、 `sin`, `cos`,

sqrt などの数学関数を利用するときは math.h というヘッダーをインクルードする。

関数の汎用性

できるだけ大域変数を使わないようにする。 (教 p.159)

配列の受渡し (教 p.160)

関数の引数として配列を渡すこともできる。仮引数の宣言は、 型名　引数名[] としておき、実引数としては　　だけを書く。

- 関数に配列を引数として渡す場合、コピーではなく、配列そのもの（正確にいうと、配列の先頭要素のアドレス）が渡される。**（重要）**
    - 一方、int, double 型などの配列でない普通の型の引数の場合は、値がコピーされて渡される。（プログラム例の cbv.c 参照）
    - 関数の中で、配列の要素の値を変更すると、呼出し側の配列に反映される。 int, double 型などの普通の型の引数の場合は、呼出し側には反映されない。
- 引数として渡された配列の要素数を関数の中で知る方法はないので、通常は要素数も引数として渡す必要がある。

配列の受渡しと const 型修飾子 (教 p.162)

関数の引数の配列が書換えられないことを保証するためには、　　という型修飾子を仮引数の宣言につける。 const をつけているのに、その変数を書き換えようとするとコンパイル時にエラーになる。

Q 6.2.2 次のプログラム (cbr.c) の出力は?

```c
#include  <stdio.h>

void a_set(int v[]) {
    v[0] = 0;
}

int main(void) {
    int a1[1] = { 1 }, a2[1] = { 3 };

    a_set(a1);
    a_set(a2);

    printf("a1[0] = %d\n", a1[0]);
    printf("a2[0] = %d\n", a2[0]);

    return 0;
}
```

番兵法 (sentinel) (教 p.166)

探索の対象となっているデータ（　　(sentinel)）をデータの最後に付け加えること。探索範囲の終わりのチェックをする必要がなくなるので、少し効率が良くなる。

6.3 「有効範囲と記憶域期間」のまとめ

有効範囲と識別子の可視性 (教 p.172)

同名の変数の有効範囲が重なるとき、より内側のブロックで宣言されているものが優先する。

Q 6.3.1 次のプログラムの出力は？

```c
#include  <stdio.h>

int x = 9;

void foo(void) {
   printf("%d ", x);
}

int main(void) {
    int x = 5;
    printf("%d ", x);
    for (int i = 0; i < 2; i++) {
       int x = 2 * i;
       printf("%d ", x);
    }
    foo();
    printf("%d ", x);
    return 0;
}
```

記憶域期間 (教 p.174)

C 言語の変数の寿命（記憶クラス, storage class）には 2 種類のものがある。

- 自動変数 (automatic variable) — 自動記憶域期間を持つ変数
    - 　　定義された変数で `static` という修飾子がついていないもの
    - プログラムの流れが宣言を通過するときに、変数のための領域（箱）が確保され、初期化される。有効範囲を抜けるときに箱が回収される。
    - 初期化子が与えられていない場合、その値は　　となる。
- 静的変数 (static variable) — 静的記憶域期間を持つ変数

- ＿＿＿＿で定義・宣言された変数、または関数の中で宣言された変数で、 `static`という修飾子がついているもの
- ＿＿＿＿＿＿＿＿に変数のための領域（箱）が生成され、初期化される。プログラムの終了時まで回収されない。
- 初期化子が与えられていない場合、＿に初期化される。

静的変数は、過去の呼出しによって結果が変わるような関数の場合には必要となる。逆に言うと、必要なければ使ってはいけない。

Q 6.3.2 次のプログラムの出力は？

```c
#include <stdio.h>

void foo(void) {
    static int x = 0;
    printf("%d ", x++);
}

void bar(void) {
    int z = 9;
    printf("%d ", z--);
}

int main(void) {
    foo(); bar(); foo(); bar();
    return 0;
}
```

＿＿＿＿＿＿

第7章 「基本型」のまとめ

7.2 「整数型と文字型」のまとめ

整数型と文字型 (教 p.186)

`signed` は＿＿＿＿＿（負の数も扱える）整数、`unsigned` は＿＿＿＿＿（正の数と 0 のみ扱える）整数を宣言する際の型指定子である。 `short` と `long` はそれぞれ、標準よりも狭い、広い範囲を扱える（かもしれない）整数を宣言する際の型指定子である。`printf` 関数での `signed`, `long` に対する変換指定はそれぞれ＿, ＿＿である。

<limits.h> ヘッダー (教 p.188)

各数値型で表現できる値の最小・最大値をマクロとして集めたヘッダーである。

Q 7.2.1 教科書 List 7-1を実行し、 次の空欄を埋めよ。

『＿＿＿＿＿＿＿＿＿＿の場合、`INT_MIN` は＿＿＿＿＿＿、`INT_MAX` は＿＿＿＿＿＿、 `UINT_MAX` は＿＿＿＿＿＿である。』

sizeof演算子 (教 p.192)

```
sizeof(型名)
```

という形で指定した型のサイズ（単位: バイト）を返す。

```
sizeof 式    /* 通常は上の形にあわせて式を （ 〜 ） で囲む */
```

という形で、式（通常は変数）のサイズ（単位: バイト）を返す。特に、式が配列の場合は配列全体のサイズ（単位: バイト）を返す。

Q 7.2.2 配列 `a` の要素数を表す `sizeof` 演算子を使った式は?

＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿＿

ただし、関数の引数として渡された配列では、`sizeof` 演算子は別の値（ポインター型のサイズ）を返すので注意する。

Q 7.2.3 次の sizeof 演算子を使用するプログラム (sizeof.c) の出力はどうなるか? 実行して確かめよ。

```c
#include <stdio.h>

void foo(int x[]) {
    printf("sizeof(x) = %u\n", (unsigned)sizeof(x));
}
```

```
 6
 7 int main(void) {
 8     int a[] = { 1, 2, 3 };
 9
10     printf("sizeof(a) = %u\n", (unsigned)sizeof(a));
11     foo(a);
12     return 0;
13 }
```

```
sizeof(a) = ____
sizeof(x) = ____
```

関数の引数として渡される配列に対する `sizeof` はポインター型（第 10 章）のサイズを返す。

size_t 型と typedef 宣言 《教 p.194》

typedef 宣言は型の別名をつける。構造体・ポインターを学習したあとは頻繁に使う。

分類	一般形
typedef 宣言	`typedef` *型 新しい型名* `;`

例えば `typedef unsigned size_t;` と宣言すると、`size_t` が `unsigned` の別名となる。

ビット単位の論理演算 《教 p.202》

ビット単位の論理演算・シフト演算などは組込み用途では多用される。必要に応じて調べられるようにしておく。

整数定数 《教 p.210》

8 進定数は先頭に「 ___ 」を、 16 進定数は先頭に「 ___ 」をつけて表記する。

10 進	8 進	16 進
48	060	0x30
65	0101	0x41
97	0141	0x61

整数の表示 《教 p.212》

`printf` 関数で整数を 8 進数または 16 進数で表示するためには、それぞれ、 ___ , ___ （A ～ F を大文字にしたいときは ___ ）という書式指定を用いる。

7.3 「浮動小数点型」のまとめ

math.h ヘッダー 《教 p.217》

`sin`, `cos`, `tan`, `sqrt` （square root — 平方根）, `exp`, `log` などの 数学関数のプロトタイプ宣言が集められているヘッダーである。

gcc の場合、math.h ヘッダーの関数を使ったプログラムをコンパイルするときは、大抵 `-lm` オプションが必要である。

繰返しの制御 (教 p.218)

誤差が累積するので、繰返しを制御する変数に、できるだけ浮動小数点数型（`float`, `double`）は使わない。

> **Q 7.3.1** 次のプログラム（の断片）の出力はどうなるか？

```
1   printf("%.60f", 0.1);
```

7.4 「演算と演算子」のまとめ

演算子の優先順位と結合性 (教 p.221)

優先順位や結合性をすべてを覚える必要はないが、必要に応じて表を調べられるように、どのような演算子があるかくらいは覚えておきたい。(Table 7-13)

第8章 「いろいろなプログラムを作ってみよう」のまとめ

8.4 「再帰的な関数」のまとめ

関数と型 (教 p.240)

再帰 (recursion) とは、関数の定義の中で自分自身を呼び出すこと。一般に x の定義に x 自身を使用すること。

```
factorial(4)
  →  4 * factorial(3)
    →  4 * 3 * factorial(2)
      →  4 * 3 * 2 * factorial(1)
        →  4 * 3 * 2 * 1 * factorial(0)
          →  4 * 3 * 2 * 1 * 1
```

- "自分自身"と言っても、変数（仮引数や自動変数、 List 8-7 の `factorial` の場合 `n`）は別々に確保される。
- 繰返し（`for`, `while`）で簡単に実現できることを, 再帰で書くのは（C 言語の場合）良いこととはいえない。階乗の例題プログラムは、あくまでも再帰を説明するためのものと考えること。（もちろん、再帰を使わなければ簡単に書けないプログラムも今後たくさん出てくる。）
- 再帰関数には、特別な文法も特別な実行規則も必要ない。あくまでも C 言語の普通の関数で、普通の実行規則に基づいて計算される。

Q 8.4.1 次のように定義された関数 `foo` に対して、

```c
void foo(int n) {
    if (n > 0) {
        foo(n - 1);
        printf("%d ", n);
        foo(n / 2);
    }
}
```

`foo(1)`, `foo(2)`, `foo(3)`, `foo(4)` の出力はそれぞれどうなるか?

_____
_____
_____
_____

Q 8.4.2 次のプログラム (hanoi.c) は、ハノイの塔というパズルを解くためのプログラムである。

```c
#include <stdio.h>

void move(int n, int a, int b) {
```

```
    printf("ディスク %d を棒 %d から棒 %d へ\n", n, a, b);
}

/* n 枚のディスクを a から b に移動する手順 */
void hanoi(int n, int a, int b, int c) {
    if (n > 0) {
        hanoi(n - 1, a, c, b);
        move(n, a, b);
        hanoi(n - 1, c, b, a);
    }
}

int main(void) {
    int n;
    printf("円盤は何枚ですか? "); scanf("%d", &n);
    hanoi(n, 1, 2, 3);
    return 0;
}
```

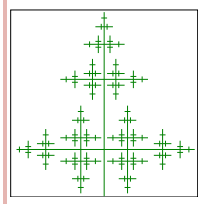
n が 3 のとき、このプログラムの出力はどうなるか?

Q 8.4.3 次のプログラム (tree.c)は、再帰を利用して、樹のような図形を描画する。

```
#include <stdio.h>
#include <math.h>

void drawTree(int d, double x, double y,
                     double r, double t) {
    /* d       --- 再帰の深さ、
       (x, y)  --- 枝の根元の座標、
       r       --- 枝の長さ、
       t       --- 枝の伸びる向き（ラジアン）*/
    double r1;
    if (d == 0) return; /* 打切り */

    printf("%6.3f %6.3f %6.3f %6.3f\n",
            x,    y,
            x + r * cos(t), y + r * sin(t));
    drawTree(d - 1, x + r  * cos(t), y + r  * sin(t),
                    0.5 * r, t);
    r1 = 0.5 * r;
    drawTree(d - 1, x + r1 * cos(t), y + r1 * sin(t),
                    0.5 * r, t + 0.5 * 3.1416);
    drawTree(d - 1, x + r1 * cos(t), y + r1 * sin(t),
                    0.5 * r, t - 0.5 * 3.1416);
}

int main(void) {
    drawTree(6, 128, 255, 128, - 0.5 * 3.1416);
    return 0;
```

```
28 }
```

次の図はこのプログラムが出力する座標を結ぶ線分を描画したものである。



`drawTree` 関数の中の定数の値を少しずつ変えて、図形がどのように変化するか確かめよ。

8.5 「入出力と文字列」のまとめ

`getchar` 関数と `EOF` (教 p.244)

`getchar` は標準入力から＿＿＿＿を読み込んで返す関数。

`EOF` は `getchar` などが、入力の終わり（＿＿＿＿＿＿に由来）に達した場合に返す値をマクロで `EOF` と書く。（stdio.h に定義されている。）この値は、通常の文字とは区別される。

リダイレクト (教 p.247)

標準入出力をファイルへの入出力につなぎかえることで、C 言語ではなく OS (Unix, MS-DOS など)の機能になる。

コマンド `<` *ファイル名*	— ファイルの内容をコマンドの標準入力に渡す
コマンド `>` *ファイル名*	— コマンドの標準出力をファイルに書込む
コマンド `>>` *ファイル名*	— コマンドの標準出力をファイルの最後に追加する形で書込む

文字コードと数字 (教 p.248)

C 言語では文字は、単にその文字に与えられたコード（整数値）で表す。具体的な値は機種に依存する。

ASCII での文字コードの抜粋:

文字	10 進	16 進
`'0'`	48	0x30
`'A'`	65	0x41
`'a'`	97	0x61

ただし、数字 `'0'`, `'1'`, `'2'`, ..., `'9'`, については、文字コードもこの順に連続していることが保証されている。

Q 8.5.1 次の文の出力はそれぞれどうなるか? ただし、文字コードに依存するものには✗と書け。

```
putchar('1' + 3);
putchar(2 + '4');
putchar('2' + '3');
printf("%d", '9' - '7');
printf("%d", '5' - 2);
```

拡張表記 (教 p.250)

「\n」の他に、「\t」,「\a」,「\b」などいくつかの特殊文字を表す表記がある。特に、バックスラッシュ（円記号）そのものを表す時には「　　」と書く。

ブックカバー作成用グラフィックス関数解説

あらまし

C または Java 言語から（文庫本サイズの）ブックカバー用の SVG ファイルを作成するための特定用途専用の自家製グラフィックスライブラリーです。Processing (https://processing.org/) という言語とできるだけ同じ名前で描画関数を用意しています。しかし、いくつかの関数は簡略化されていますし、当然ながら、アニメーションやインタラクション関係の関数はありません。

座標系

初期状態では、紙の左上が原点で、x 軸は右向き、y 軸は下向きにのびています。（ふつう数学で使う座標系と y 軸の向きが逆です。）長さの単位は mm（ミリメートル）です。

サイズは A4 紙横向きがデフォルトになっています。 A4 紙のサイズは 横 297mm × 縦 210mmです。ブックカバーにしたとき、描いた図形が本の表面に現れるようにするには、だいたい (43, 31)—(253, 179) の座標の範囲（横 210mm × 縦 148mm）に図形がおさまるようにして下さい。

#include

C 版では、

```
#include "svg.h"
```

としてください

```
/* 間違い */
#include <svg.h>
```

ではありませんので気を付けてください。

関数一覧

初期設定・その他

`void start(void);`
　　描画の開始のときに必ず呼び出します。
`void finish(void);`
　　描画の終了のときに必ず呼び出します。

色・属性設定

初期状態は、線なし・塗潰し黒です。

`void stroke(unsigned int color);`
線の色を設定します。色は 0xRRGGBB の形式で指定します。

`void strokeWeight(double w);`
線の太さを設定します。

`void strokeOpacity(double opacity);`
線の透明度を 0（完全透明）～ 1（完全不透明）の値で指定します。

`void noStroke(void);`
線を描きません。

`void fill(unsigned int color);`
塗潰しの色を設定します。色は 0xRRGGBB の形式で指定します。

`void fillOpacity(double opacity);`
塗潰しの透明度を 0（完全透明）～ 1（完全不透明）の値で指定します。

`void noFill(void);`
塗潰ししません。

`void textFont(char* fontName, double size);`
文字のフォントとサイズ（単位 mm）を設定します。（初期値は "MS-Mincho", 12mm です。） Windows上で SVGを閲覧する場合、fontName としては "serif", "sans-serif", "monospace", "cursive", "fantasy", "ＭＳ 明朝", "ＭＳ ゴシック", "MS Pゴシック", "MS P明朝",（注: ＭとＳとＰは全角、空白は半角） "Arial", "Times New Roman", "Verdana", "Courier New", "Andale Mono", "Comic Sans MS", "Garamond", "Georgia", "Impact", "Tahoma", "Trebuchet MS" などが使えるはずです。

基本図形

`void line(double x1, double y1, double x2, double y2);`
(x1, y1) から (x2, y2) へ線分を描きます。

`void rect(double x, double y, double w, double h);`
左上の頂点の座標が (x, y)、幅 w、 高さ h の長方形を描きます。

`void ellipse(double x, double y, double w, double h);`
中心の座標が (x, y)、幅 w、 高さ h の楕円を描きます。

`void triangle(double x1, double y1, double x2, double y2, double x3, double y3);`
3つの頂点の座標が (x1, y1), (x2, y2), (x3, y3) の 三角形を描きます。

`void text(char* str, double x, double y, ...);`
文字列 str を座標 (x, y) に表示します。

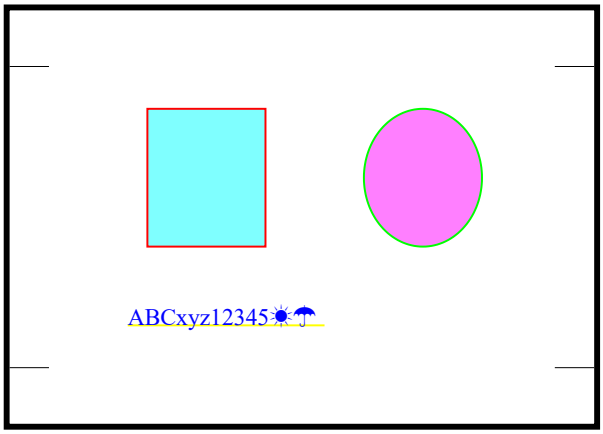
サンプルプログラム

C 版 C/FirstSample.c

```c
#include "svg.h"

int main(void){
    start();                        /* 最初に必要 */
    rulers();                       /* 四隅の線   */

    strokeWeight(1);
    stroke(hsb360(0, 100, 100)); /* 線の色 */
    fill(hsb360(180, 50, 100));  /* 塗りの色 */
    rect(70, 50, 60, 70);        /* 長方形 */

    stroke(hsb360(120, 100, 100));
    fill(hsb360(300, 50, 100));
    ellipse(210, 85, 60, 70);    /* 楕円 */
```

```
15
16      stroke(hsb360(60, 100, 100));
17      line(60, 160, 160, 160);      /* 直線 */
18
19      noStroke();
20      fill(hsb360(240, 100, 100));
21      textFont("Times New Roman", 12);
22      text("ABCxyz12345☀☂", 60, 160);
23      /* 文字列 */
24      finish();                     /* 最後に必要 */
25      return 0;
26  }
```

上記のプログラムが生成する図形



色関連のユーティリティ

`unsigned int hsl360(double h, double s, double l);`
　　fill 関数や stroke 関数の引数として与えるための色の値を h（色相）, s（彩度）, l（輝度）から計算します。 h（色相）は 0 から 360 の範囲、 s（彩度）, l（輝度）はそれぞれ 0 から 100 の範囲の数で指定します。

`unsigned int rgb255(double r, double g, double b);`
　　fill 関数や stroke 関数の引数として与えるための色の値を 光の三原色 r（赤）, g（緑）, b（青）から計算します。 r（赤）, g（緑）, b（青）はそれぞれ 0 から 255 の範囲の数で指定します。

`unsigned int rotateH360(unsigned int color, double a);`
　　色相を a 度変えた新しい色を計算します。

タートルグラフィックス関数

"亀"は最初ページの真ん中 `(148.5, 105)` にペンを下げた状態で 0 度の向き（右）を向いています。

`void forward(double len);`
　　現在、向いている向きに `len` だけ移動します。

`void turn(double angle);`
　　右方向に `angle` 度回転します。 （左方向に回転する時は負の数を渡します。）

`void penUp(void);`
　　ペンを上げます。（この状態で移動しても線を描きません。）

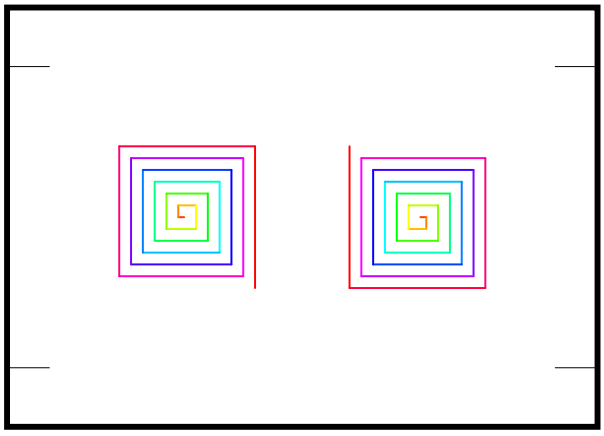
`void penDown(void);`
　　ペンを下げます。（この状態で移動すると線を描きます。）

タートルグラフィックスのサンプルプログラム

C 版 C/TurtleSample.c

```c
#include "svg.h"

int main(void) {
    int i;

    start();                                /* 最初に必要 */
    rulers();                               /* 四隅の線   */

    direction(0);
    penUp();
    forward(60);
    penDown();
    for (i = 1; i <= 24; i++) {
        stroke(hsb360(i * 15, 100, 100)); /* 色 */
        forward(i * 3);                     /* 進む距離 */
        turn(90);                           /* 曲がる角度 */
    }

    center();
    direction(180);
    penUp();
    forward(60);
    penDown();
    for (i = 1; i <= 24; i++) {
        stroke(hsb360(i * 15, 100, 100)); /* 色 */
        forward(i * 3);                     /* 進む距離 */
        turn(90);                           /* 曲がる角度 */
    }

    finish();                               /* 最後に必要 */
    return 0;
}

```

上記のプログラムが生成する図形



第P章 プログラミング言語 Python

Python は 1990 年に Guido van Rossum により発表された、マルチパラダイム（手続き型 + ＿＿＿＿＿＿＿＿ + ＿＿＿＿）・動的型付けのプログラミング言語である。ライブラリー（前もって用意されている関数など）が豊富で、Web アプリケーション のほか、＿＿＿＿＿＿ などデータサイエンス分野で広く用いられるため、 2010 年代後半から急速に人気が高まってきている。

ここでは C 言語と異なる部分を中心に説明する。

P.1 Python の実行

対話的な処理系は ＿＿＿＿＿＿ というコマンドで起動できる。対話的な処理系では、プロンプト（通常、>>>）のあとに式を入力すれば、その値を出力する。

```
>>> 1 + 1
2
```

＿＿＿＿＿＿ または ＿＿＿＿＿＿ と入力すれば対話的な処理系を終了する。

また `python` ファイル名 というかたちで直接実行することもできる。（「>」はシェルのプロンプト）

```
> python factorial.py
```

Python を実行する環境として、IDLE（Python をインストールしていれば `idle` というコマンドで起動する）, PyCharm, Spyder などいくつかの統合開発環境 (Integrated Development Environment, IDE) や Jupyter Notebook というアプリケーションもよく使われるが、ここでは説明を割愛する。

P.2 代入と関数定義

変数と代入

変数は、次のように記号「＿＿」の左辺に書き、右辺の値を代入する。（宣言は特になく、初めて使う変数に代入したときに変数が用意される。このため、綴りの間違いには気を付ける必要がある。）

```
>>> x = 2
>>> x * 3
6
```

関数定義

関数はキーワード ＿＿＿＿ により定義することができる。

```
>>> def fact(n):
...     if n == 0:
```

```
...         return 1
...     else:
...         return n * fact(n - 1)
...
>>> fact(10)
3628800
```

入力が完了していないと判断すると Python 処理系は、上のように第２プロンプト「...」を表示して入力の継続を促す。

関数定義は、キーワード def のあとに関数名（この例では fact）、丸括弧「(」～「)」の中の仮引数のコンマ区切りの並び（この例では n のみ）、コロン「:」で始まる。改行後に関数の本体を記述する。

関数の本体はインデント（字下げ）する。つまり def の位置よりも数文字（この例では４文字）分下げる。インデントしている限り関数の本体が続いていると見なされる。

このように Python はインデンテーションが文法上＿＿＿＿＿言語である。（一方、C や Java はどのようにインデントしてもプログラムの意味は＿＿＿＿＿。）

インポート

通常は、ファイルに関数などの定義を記述して、import 文で読み込む。また Python のソースファイルには＿＿＿という拡張子をつけるのが通例である。

ファイル factorial.py

```
1  def fact(n):
2      if n == 0:
3          return 1
4      else:
5          return n * fact(n - 1)
```

```
>>> from factorial import *
>>> fact(20)
2432902008176640000
```

この from ～ import * という形式の import 文は、～というモジュール（ファイル名から拡張子 .py を除いたもの）から変数や関数などの定義を読み込むことを意味する。

P.3 リテラル

数値リテラル

整数や浮動小数点数のリテラルは他の言語と大きな違いはない。また、接尾辞 j を整数や浮動小数点数リテラルにつけることで＿＿＿を表すことができる。

```
>>> (1 + 1j) / (1 - 2j)
(-0.2+0.6j)
```

文字列リテラル

文字列リテラルは、二重引用符「"」 または一重引用符「'」 で囲まれた文字列である。文字リテラルは存在しないので一重引用符／二重引用符どちらを使っても意味は変わらない。

```
>>> "hello"
'hello'
>>> 'world'
'world'
```

また三連続の二重引用符「"""」または一重引用符「'''」で囲むと改行や一重引用符・二重引用符を含む文字列を表すことができる。

```
>>> """Mike said "Hello!"
... and Anne said "Good Bye!"
... """
'Mike said "Hello!"\nand Anne said "Good Bye!"\n'
```

引用符の前に接頭辞 `f` または `F` をつけると、波括弧「{」～「}」に囲まれた部分が式として評価され、文字列中に埋め込まれる。

```
>>> x = 13
>>> f"The factorial of {x} is {fact(x)}."
'The factorial of 13 is 6227020800.'
```

問 P.3.1 では、f 文字列のなかに波括弧（「{」または「}」）を含めたいときはどうすれば良いか調べよ。

文字列は「+」演算子で連接することができ、「*」演算子で自身を繰り返し連接することができる。

```
>>> "hello, " + 'world'
'hello, world'
>>> "hey!" * 3
'hey!hey!hey!'
```

P.4 演算子と組み込み関数

算術演算子

演算子は C や Java の演算子と似ているが、「/」は整数同士の演算でも通常の（小数になる可能性のある）除算である。整数としての除算の商を求めるには「`//`」を用いる。余りを求める演算子は「%」である。

```
>>> 1 / 3
0.3333333333333333
>>> 17 // 6
2
>>> 17 % 6
5
```

「**」は__を求める演算子である。

```
>>> 2 ** 10
1024
>>> 2 ** 0.5
1.4142135623730951
```

print 関数と input 関数

画面に出力するには print 関数を用いる。いくつかの引数をコンマで区切って与えるとそれらを順に出力する。

```
>>> y = 23
>>> print(x, "+", y, "=", x + y)
13 + 23 = 36
```

最後に、sep=～ と指定すると、区切り文字を～に変えることができる。このように「キーワード=式」のカタチで与える引数をキーワード引数という。（区切り文字を何も指定しないと上の例のように空白文字が区切りに使われる。）

```
>>> print(x, "+", y, "=", x + y, sep=',')
13,+,23,=,36
```

一方、input 関数はキーボードから文字列を読みこむ関数である。＿＿関数（整数への変換）や＿＿＿関数（浮動小数点数への変換）と組み合わせることで、文字列をそれぞれ整数や浮動小数点数に変換することができる。

ファイル名 temp.py

```
1 x   = float(input('x を入力してください: '))
2 y   = float(input('y を入力してください: '))
3 z   = int(input('z を入力してください: '))
4 age = int(input('年齢を入力してください: '))
```

P.5 制御構造

if 文

条件分岐を表す if 文は if というキーワードのあと、条件式、コロン「:」で始まり、改行後に条件が成り立つときに実行する文を並べて書く。

```
1 if x < 0:
2     print('x は負の数です。')
3     print('正の数や 0 ではありません。')
```

条件が成り立つときに実行する文はインデント（字下げ）する。つまり if の位置よりも数文字（この例では 4 文字）分開始を下げる。ここに複数の文を並べることもできる。同じ字下げ幅である限りは、条件が成り立つときには実行する。

条件が成り立たないときに実行する文は、キーワード＿＿、コロン「:」のあとに改行して書く。

```
1 if y < 0:
```

```
    print('y は負の数です。')
    print('正の数や 0 ではありません。')
else:
    print('y は正の数または 0 です。')
    print('負の数ではありません。')
```

また、if と else の間に elif というキーワード、条件式、コロン「:」で始まり、改行後にインデントした文の並び、というかたちをはさむことができる。この場合、上から順に条件式を評価し、成り立つときに対応する文の並びを実行する。

```
if z <= 0:
    print('z は負の数か 0 です。')
elif z < 10:
    print('z は一桁の正の数です。')
elif z < 100:
    print('z は二桁の正の数です。')
else:
    print('z は三桁以上の正の数です。')
```

論理演算子

条件式は and（～かつ～）、or（～または～）、not（～でない）などの論理演算子で組み合わせることができる。

```
if 13 <= age and age <= 19:
    print('あなたはティーンエイジャーです。')
```

Python では（C や Java と異なり）比較演算子は連ねることができるので、この例は次のように書くこともできる。

```
# x < y < z は x < y and y < z と同じ。
# ただし、前者では y は一度しか評価されない。
if 13 <= age <= 19:
    print('あなたはティーンエイジャーです。')
```

コメント

上の例で使われているように、「#」から行末まではコメントである。

for 文

決まった回数の繰り返しを実現するときには for 文が使われる。次のようにキーワード for、変数、キーワード in、式、コロン「:」という形式で使われる。

ファイル名 temp2.py

```
for i in range(5):
    print('Hello')
    print('  ', i, '回目')
```

ここで range 関数は、range(n) が、0 から n - 1 までの整数の列を返すような関数である。

これを実行すると、変数 i に 0, 1, ..., 4 が順に代入され、次のように出力される。

```
Hello
   0 回目
Hello
   1 回目
Hello
   2 回目
Hello
   3 回目
Hello
   4 回目
```

他に range 関数は次のようなカタチでも使われる。

range(n)	$0, 1, ..., n-1$ を返す。
range(m, n)	_____ を返す。
range(m, n, s)	_____ を返す。

ただし、k は $m + k \cdot s$ が n 未満となるような最大の k である。

while 文

繰り返しを表す while 文は while というキーワードのあと、条件式、コロン「:」で始まり、改行後に繰り返す文をインデントして並べて書く。

ファイル名 while.py

```
1 n = 1000
2 while n > 0:
3     print(n, end=', ')
4     n = n // 2    # n //= 2 と書いてもよい
5 print()          # 改行のみ出力する
```

ここで print 関数に、end=～ とキーワード引数を指定すると最後に出力する文字を変えることができる。（何も指定しないと改行文字が最後に出力される。）

これを実行すると、次のように出力される。

```
1000, 500, 250, 125, 62, 31, 15, 7, 3, 1,
```

なお、Python に do～while 文に相当する構文はない。

P.6 リスト

リストは単純だが有用性の高いデータ型で、関数型言語などで多用されるデータ型である。配列と同様に（同種の）データを集めたものだが、要素の追加・削除が可能である。ただし、配列のように各要素に高速にアクセスすることはできない。

リストリテラル

リストを構成するためには、＿＿＿＿（[～]）で囲み、各要素をコンマ「,」で区切って並べる。例えば、 [] は空リストを表し、[2, 3, 5] は 3 つの要素からなるリストを表す。

また、リストは「+」演算子や「+=」演算子で連接することができる。

```
>>> w = [1, 3] + [0]
>>> w += [6, 2, 3]
>>> w
[1, 3, 0, 6, 2, 3]
```

「*」演算子や「*=」で、自身を繰り返し連接することもできる。

```
>>> v = [0]
>>> v * 5
[0, 0, 0, 0, 0]
>>> u = [1, 2, 3]
>>> u *= 3
>>> u
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

リスト内包表記

リスト内包表記 (list comprehension) は数学で使われる集合の表記に似た糖衣構文 (syntax sugar) である。

```
>>> [x * y for x in range(1, 5) for y in range(5, 8)]
[5, 6, 7, 10, 12, 14, 15, 18, 21, 20, 24, 28]
>>> [x * x for x in range(1,11) if x % 2 == 1]
[1, 9, 25, 49, 81]
```

リスト内包表記は、角括弧（[～]）のなかに最初に式を一つ書き、そのあとに、「for 変数 in 式」というカタチか「if 式」というカタチを並べたものである。（ただし並びの最初は「for ～」のカタチでなければいけない。）その値は「for 変数 in 式」というカタチで与えられた繰り返しの中で「if 式」というカタチで与えられた条件が成り立つときの最初の式の値を順に並べたものになる。

Q P.6.1 次のリスト内包表記の値は何か?

① [x * y for x in [1,2] for y in [3,5,7]]

P.7 タプル

タプル（tuple, 組）は要素を「,」（コンマ）で区切って並べ、丸括弧「(」と「)」で囲んで表す。（文脈によっては丸括弧を省略できる場合がある。）リストは通常、各要素は同種のものからなるが、タプルは要素の種類が同一である必要はない。

タプルは以下の例のように関数の戻り値に使うこともできるし、代入文の左辺に書くこともできる。

ファイル名 temp3.py

```
def sort2(m, n):
    if m > n:
        return (m, n)
    else:
        return (n, m)

i1 = int(input('整数 1 を入力してください: '))
i2 = int(input('整数 2 を入力してください: '))

(j1, j2) = sort2(i1, i2)
print('大きいほうは', j1, '小さいほうは', j2, 'です。')
```

特に次のように書くと左辺の 2 つの変数への代入が同時に行われるので、2 つの変数の内容を入れ替えることができる。

```
y, x = x, y
```

（この例ではタプルのまわりの丸括弧が省略されている。）

Q P.7.1 次のリスト内包表記の値は何か?

① `[(x, y) for x in [1,4,7] for y in [2,5,8] if x < y]`

問 P.7.2 3 つの実数 a, b, c を受け取り、二次方程式 $ax^2 + bx + c = 0$ の 2 つの解を組として返す関数 `quadratic` を定義せよ。

標準ライブラリー関数の `zip` は 2 つ以上のイテラブルの同じ位置の要素をタプルにしたもののイテラブルを返す関数である。イテラブルの長さが異なる場合は、短いほうにあわせる。（イテラブル (iterable) は、いくつかの要素を列挙したデータと考えておけば良い。リストもイテラブルの一種である。）

```
>>> list(zip([1,3,5,7,11], [2,4,6,10]))
[(1, 2), (3, 4), (5, 6), (7, 10)]
```

P.8 高階関数とラムダ式

高階関数は　　関数である。

リストに対しては、いくつかの高階関数が標準ライブラリーに用意されている。例えば、`map` は、リストの要素に一斉に関数を適用し、その戻り値のリストを返す関数である。（正確にはイテラブルを受け取って、イテラブルを返す関数である。一般にイテラブルを表示するためには `list` 関数でリストに変換する必要がある。）

このメソッドは次のように使用することができる。

```
>>> list(map(chr, [97, 98, 99, 100]))
['a', 'b', 'c', 'd']
>>> def twice(n):
```

```
...     return 2 * n
>>> list(map(twice, [97, 98, 99]))
[194, 196, 198]
```

ここで、chr は文字コードに対し対応する一文字からなる文字列を返す関数である。

ところで高階関数の引数として使う twice のような小さな関数にいちいち名前をつけるのは面倒なので、名前をつけずに関数を表現する記法が用意されている。これを＿＿＿＿＿＿という。（この名前は、かつて数学の一分野で、この目的のためにギリシャ文字の「λ」が使われたことに由来する。）

例えば、lambda x: 2 * x という式で twice と同等の関数を表す。キーワード lambda とコロン「:」の間に仮引数のコンマ区切りの並びを、コロンの右側に戻り値の式を書く。次はラムダ式の使用例である。

```
>>> list(map(lambda x: x * x, [2, 3, 5]))
[4, 9, 25]
```

また、＿＿＿＿＿＿は、リストの要素の中で、与えられた関数の値を真にする要素だけのリストを返すメソッドである。

```
>>> list(filter(lambda x: x % 2 ==  0, [2, 3, 5, 8]))
[2, 8]
```

P.9 ジェネレーター

Python のジェネレーター関数 (generator function) は＿＿＿＿＿＿＿(coroutine) の一種を提供する。コルーチンとは、2 つ以上のプログラムの実行単位が、＿＿＿制御を受け渡しながら実行されていく方式のことである。通常の関数（サブルーチン）はリターンする（戻り値を返す）と、次に実行するときはもう一度最初からになるが、コルーチンは次に実行するときに前回リターンした地点の続きから実行する。

ファイル名 fib.py

```
1 def gfib(n):
2     a = 1
3     b = 1
4     while a < n:
5         yield a
6         a, b = b, a + b
```

Python のジェネレーター関数では ＿＿＿＿というキーワードを使って値を生成する。

ジェネレーター関数を呼び出すと、すぐに関数内部のコードが実行されるのではなく、一旦、ジェネレーターイテレーター (generator iterator) が作られて返される。このジェネレーターイテレーターを引数として ＿＿＿関数を呼び出すと、ジェネレーター関数内部のコードが実行され、yield された値を返す。さ

らに next 関数を呼び出すと yield 文の＿＿＿＿実行が再開され、やはり、次の yield された値を返す。

```
gen = gfib(100)
print(next(gen))        # 1 を出力する
print(next(gen))        # 1 を出力する
print(next(gen))        # 2 を出力する
print(next(gen))        # 3 を出力する
print(next(gen))        # 5 を出力する
print(next(gen))        # 8 を出力する
```

ジェネレーターイテレーターは for 文の in の次の式でも使うことができる。

for 文はジェネレーターイテレーターを受け取った next 関数によって返される値を順に変数に代入してループする。ジェネレーター関数の中のコードの実行が return 文によりリターンするか、関数を抜けるとループを終了する。

```
for v in gfib(20):
    print(v, end=' ')
```

この部分は「＿＿＿＿＿＿＿＿」と出力する。

ジェネレーターは必ずしも有限個の要素で終わる必要はない。次の例は無限に yield する例である。

ファイル名 ifib.py

```
def ifib():
    a = 1
    b = 1
    while True:
        yield a
        a, b = b, a + b
```

次のようにすると、

```
for i, v in zip(range(15), ifib()):
    print(v, end=' ')
print()
```

この部分は「1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 」と出力する。

問 P.9.1 整数 n を引数として受け取り、最初は n を yield し、以降は、

① 直前に yield した値が偶数ならば $n/2$ を yield する、
② 直前に yield した値が奇数ならば $3n+1$ を yield する、

という処理を繰り返すジェネレーター関数 hailstone を定義せよ。

P.10 例: エラトステネスの篩（ふるい）

最後に、素数列を生成するプログラムを例として挙げる。（ただし、この定義は効率面での改良の余地は大いにあると思われる。）

ファイル名 primes.py

```
def ifrom(n):
    while True:
        yield n
        n += 1

def sieve(n, xs):
    for i in xs:
        if i % n != 0:
            yield i

def primes():
    xs = ifrom(2)
    while True:
        n = next(xs)
        yield n
        xs = sieve(n, xs)
```

この `primes()` は無限に素数を生成するので、例えば `range` のように有限のものと `zip` する。

```
for i, p in zip(range(20), primes()):
    print(p, end=' ')
```

この for 文は、

```
 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
```

と 20 個の素数を出力する。