第 3 章

関数（1）

本章では、プログラムの部品を作る方法を説明します。プログラミングに限らず、世の中の複雑な大きな問題を解決する場合に、小さな問題に分割して、段階的に対処していくことがよくあります。駐車場の自動精算機のように複雑な作業を実現する時にも、やはりプログラムを小さな機能（部品）に分割して作っていきます。この小さな部品が「関数」です。

プログラミング言語の「関数」は、数学で習った「関数」と似たところもあります。関数をうまく使えば、再利用によってプログラム全体の長さを短くしたり、プログラム信頼性を高めたりすることも可能です。処理に名前をつけるので、**内部の細かな動きを忘れて**よくなり、プログラムの見通しがよくなります。

C 言語に限らず、ほとんどのコンピュータ言語に備わった機能ですので、ぜひとも身につけましょう。

なお、本文では関数の呼び方に次のようなバリエーションがありますが、すべて同じ意味です。

- f 関数
- f() 関数
- 関数 f
- 関数 f()
- 関数 f(x)
- f()
- f(x)
- double f(double x)

キーワード

- 関数，引数，戻り値
- 関数の入出力
- スコープ
- 値渡し，参照渡し

3.1　関数の入力＝引数・関数の出力＝戻り値

数学で習った関数を思い出すと、例えば 2 乗を返す関数は、$y = x^2$ とか、$f(x) = x^2$ のように表記していました。$f(x) = x^2$ のほうが C 言語の関数に近いので、この表記で説明すると、f が 関数名 、左辺の (x) が 入力となる変数 、右辺の x^2 が 関数の返す値 です。このように数学の関数は 3 つの要素で構成されています。**TODO:** 緑の下線を太く

C 言語での**関数** (function) も、やはり 3 つの要素があります。

（数学の文法）

関数名 (入力となる変数) = 関数の返す値

```
/* C言語の文法 */
型名 関数名 (引数) {
    return 戻り値;
}
```

関数の範囲

（数学の実例）

$f(x) = x^2$

```
/* C言語の実例 */
double f(double x) {
    return x * x;
}
```

関数名　数学ではアルファベット 1 文字にしますが、プログラムでは変数名と同じく識別子（☞**??**項）ですので、何文字の英数字でも構いません。

入力となる変数　関数名の後ろに ( ) を書いて、このカッコの中に型名と変数名のペアを書き並べます。この変数を**引数**(ひきすう)[*1]と呼びます。

関数の返す値　関数の出力に相当するもので、**戻り値**(もどり)とか**返り値**(かえり)と呼びます。 型名は関数名の前に書きます。値を書く場所は、その後に現れる `return` のすぐ後です。関数の手続きの範囲は { } で囲って示します。その中で、戻り値を計算する手続きが先にあって、最後に return 文と戻り値を書きます。

頻出ミス

戻り値の、型名は**関数名の前**、値は **return の後**と、離れたところに書くので注意してください。そして、関数名の前の型名と、return の後の値の型は、一致させておくべきです。（☞ 12 ページのコラム）

「関数の戻り値の型」を、簡単に「関数の型」ともいいます。「関数 f は double 型」という言い回しもあります。

[*1] 因数（いんすう）と区別するため、あえて湯桶読み（訓音の混じった変則的な読み）で「ひきすう」と呼ぶようです。

ソースコード 3.1 2 乗を求める関数

```
1 /* x の2乗を返す */
2 double f(double x) {
3     return x * x;
4 }
```

ソースコード 3.2 正 n 角形の中心角

```
1 /* 正n角形の中心角を返す */
2 double central_angle(int n) {
3     return 360.0 / (double)n;
4 }
```

$f(x) = x^2$ を C 言語で実現したものがソースコード 3.1 です。関数の**名前**は、数学の関数と同じ f にしました。**引数**を double 型の変数 x で受け取り、**戻り値**を double 型で返します。計算手順はあまりにも短いので、return 文に直接 `x*x` と[*2]書きました。

ソースコード 3.2 は、正 n 角形の中心角 $\frac{360}{n}$ を求めます。return 文の式は double 型ですから[*3]、関数の型も合わせて double にします。**引数 n の int とは異なっても構いません**。

3.1.1 ブロック

ブロックは、{ } で囲んだ、制御構造のひとかたまりです。関数の範囲を示すのにも用いますが、関数の内部に、新たなブロックをつくることもできます。

```
int f(void) {
    {
        /* ブロック1 */
    }
    {
        /* ブロック2 */
    }
}
```

```
int f(void) {
    {
        /* 外ブロック */
        {
            /* 内ブロック */
        }
    }
}
```

上記ではブロックを 2 個ずつ作りました。左のように**直列**に並べたり、右のように**入れ子**にもできます。この例のような単独のブロックを作る機会はそれほどありませんが[*4]、今後は条件分岐やループ処理などの構文と組み合わせて、頻繁に用いるようになります。

[*2] C 言語にべき乗演算子はありません。（`x^y` はビット演算の排他的論理和です。）そのため、2 乗や 3 乗なら掛け算を繰り返すのが慣用句になっています。数学関数にはべき乗関数 pow(x,y) がありますが、計算速度は掛け算の 5〜20 倍遅いのが通例で、誤差が混入する可能性もあるため、多用されません。

[*3] 7 角形を想定して、小数付きにしました。n を double にキャスト（☞**??**項）して計算しています。

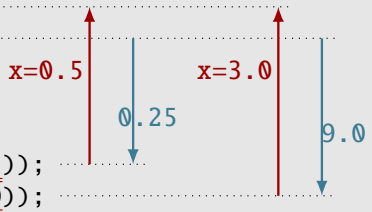
[*4] 変数のスコープ（☞3.3 節）を、わざと狭くするときに使います。

3.2　関数呼び出しと実行順序

では、作った関数 f() を呼び出してみましょう。数学では $f(0.5)$ の表記で $f(x)$ の $x = 0.5$ のときの値がわかります。プログラムでも同じような表記をします。ソースコード 3.3 では main() 関数から f(0.5) を呼び出して、printf() でその値「0.25」を表示してみました。f(3.0) なら「9」と、引数に応じて計算しなおされます。printf() 書式を"%g"としたので、小数部分の 0 が表示されていませんが、内部ではちゃんと double 型の値になっています。

ソースコード 3.3　2 乗を求める関数を呼び出す

```
1  #include <stdio.h>
2
3  /* x の2乗を返す */
4  double f(double x) {
5      return x * x;
6  }
7
8  int main(void) {
9      printf("f(0.5) は「%g」です\n", f(0.5));
10     printf("f(3.0) は「%g」です\n", f(3.0));
11     return 0;
12 }
```



ソースコード 3.3 の実行結果

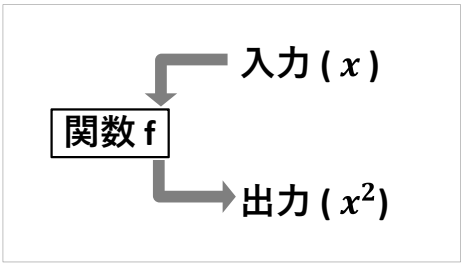
```
f(0.5) は「0.25」です
f(3.0) は「9」です
```

このように、main() から、何度でも f() を呼び出せて、引数の値は、その時ごとに違った値にできます。

ところで、ソースコードには f() のほうが先に現れていますが、**最初に実行されるのは main()** です。つまり関数の**「ソースコード上の記述順序」と「実行の順序」は無関係**ということです。ちなみに、ソースコード上の記述順序を入れ替えると、このままではコンパイルエラー（あるいは警告）になります。ひとまず main()（呼び出し側）を後に書くことにしてください。この記述順序の制約から逃れる方法は**??**節で説明します。

コラム：関数の内側・外側

すべての**文**は、どれかの関数に属するよう、関数ブロックの内側に書く必要があります。関数の外側に書くものは、#include や特別な変数の定義など、ごく限られたものです。



次は、関数 f() を様々な引数で呼び出してみましょう。**引数には計算式**を渡せるので、その例です。ソースコード 3.4 の f() は、これまでと同じですが、改行を省略しました。

ソースコード 3.4 $f(x)$ の呼び出しの組合せ

```
#include <stdio.h>
#define EPS 0.0001

/* x の2乗を返す */
double f(double x) { return x * x; }

/* f(x) の数値微分を返す */
double g(double x) { return (f(x + EPS) - f(x)) / EPS; }

/* x の4乗を返す */
double h(double x) { return f(f(x)); }

int main(void) {
    printf("f(3.0) = %.8f\n", f(3.0)); // f(3.0) = 9.00000000
    printf("g(3.0) = %.8f\n", g(3.0)); // g(3.0) = 6.00010000
    printf("h(3.0) = %.8f\n", h(3.0)); // h(3.0) = 81.00000000
    return 0;
}
```

$f(x)$ の数値微分、つまり微小な ϵ に対して

$$g(x) = \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

を求める関数 g(x) を作りました。2 行目で ϵ をマクロ定数の EPS として定義しています（☞**??**節）。8 行目では、数学と同じような記法で f() を 2 回呼び出しました。

2 乗を求める f() を 2 回連続で呼び出すと、4 乗になります。数学では

$$h(x) = f(f(x))$$

と表記しますから、11 行目で関数 h(x) を、ほぼ同じ表記で実現しました。

最後に main() から f(3.0), g(3.0), h(3.0) を呼び出しました。printf() では `%.8f` で表示したので、小数部分が 8 桁になります。数値微分の誤差も確認しておきましょう。

3.2.1　複数の引数・関数の組合せ

関数は、**式の一部**でありながら、**引数には式**をとるという、**再帰的な性質**を持っています。`f(f(x))` はその活用例ですが、他の例もみてみましょう。

標準ライブラリの**数学関数** (mathematical function) には、a と b の大きい方を返す `double fmax(double a, double b)` があります。これを利用して、ソースコード 3.5 では a, b, c, d の最大値を求める関数 `fmax4(a,b,c,d)` を作ってみました。

5 行 関数の名前は `fmax4`、型は 6 行目の return の式に合わせて double にします。引数には、4 つの変数を書き並べています。**複数の引数**は、このようにコンマで区切って書き並べます。今回は型がどれも double と共通ですが、「double a,b,c,d」とまとめることはできません。

6 行 return の式は最大値を求めています。**`fmax()` の引数を `fmax()` にする**のが思いつきにくいでしょう。`fmax(a,b)` と `fmax(c,d)` を先に計算して、これらの値の大きい方を、外側の `fmax()` で得ています。言ってみれば、4 チームのトーナメント形式で最大値を求めます。引数の組合せを変えれば、勝ち抜き形式にもできます。（もちろん結果は同じです。）

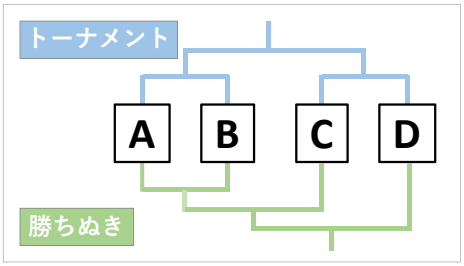
11 行 結果を表示します。関数を呼び出す際の**複数の引数**も、このようにコンマで区切って並べます。

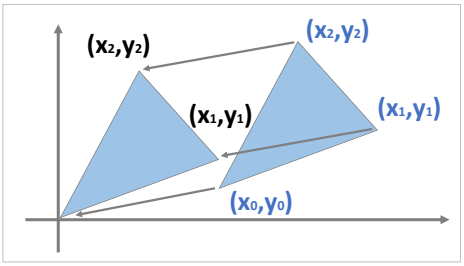
ソースコード 3.5　4 つの値の最大値を求める関数

```
1  #include <stdio.h>
2  #include <math.h> // fmax() に必要
3
4  /* a,b,c,d の最大値を返す */
5  double fmax4(double a, double b, double c, double d) {
6      return fmax(fmax(a,b), fmax(c,d));  // トーナメント
7  //  return fmax(fmax(fmax(a,b), c), d); // 勝ち抜き
8  }
9
10 int main(void) {
11     printf("fmax4(1, 3, 5, 7) = %g\n", fmax4(1, 3, 5, 7)); // 7
12 }
```

2 行目の「`#include <math.h>`」という記述は、標準ライブラリの数学関数を使うときに必要です[*5]。`#include` の役目は**??**項で説明します。数学関数の一覧は**??**節にあります。数学関数は、引数も戻り値も double のものが基本ですが、標準ライブラリ関数なら**型を確かめてから呼び出す**のが本来の姿です。確かめ方も**??**項で説明します。

[*5] Unix 系の gcc のように、さらにコンパイラに `-lm` のオプションが必要な場合もあります。（☞**??**節）





3.2.2 関数の役割分担

三角形の面積を求める関数を作りましょう。ソースコード 3.6 では、いくつかある公式の中で、**頂点座標**から求めるものを採用しました。関数名には vertex（頂点）の意味で 'v' をつけてみました。

3 頂点が**原点と 2 点** $(x_1, y_1), (x_2, y_2)$ であれば、面積 S は次の簡単な式になります。

$$S = \frac{|x_1 y_2 - x_2 y_1|}{2}$$

関数 `triangle_v2` では、引数で xy 座標を 2 組受け取り、公式通りの S を return で返します。`fabs()` も標準ライブラリの数学関数で、**絶対値** (absolute value) を返します。

3 頂点が**原点とは限らない** $(x_0, y_0), (x_1, y_1), (x_2, y_2)$ だとしましょう。この場合は、

$$\begin{cases} x_0' = x_0 - x_0 = 0 \\ y_0' = y_0 - y_0 = 0 \end{cases} \quad \begin{cases} x_1' = x_1 - x_0 \\ y_1' = y_1 - y_0 \end{cases} \quad \begin{cases} x_2' = x_2 - x_0 \\ y_2' = y_2 - y_0 \end{cases}$$

のように、全体を $(-x_0, -y_0)$ 方向に平行移動すると、(x_0', y_0') が原点になるので、$(x_1', y_1'), (x_2', y_2')$ に S の公式が適用できます。これを利用して、関数 `triangle_v3` では、**面積の計算はせず**に、11 行目で**平行移動の計算だけ**を行いました。最初の関数を呼び出して、得られた値をそのまま return で返すという**役割分担**にしたのがうまいところです。

ソースコード 3.6 三角形の面積を求める関数（座標版）

```
1  #include <math.h> // fabs() に必要
2
3  /* 三角形の面積（頂点が原点, (x1,y1), (x2,y2)）*/
4  double triangle_v2(double x1, double y1, double x2, double y2) {
5      return fabs(x1 * y2 - x2 * y1) / 2.0;
6  }
7
8  /* 三角形の面積（頂点が(x0,y0), (x1,y1), (x2,y2)）*/
9  double triangle_v3(double x0, double y0,
10                     double x1, double y1, double x2, double y2) {
11     return triangle_v2(x1-x0, y1-y0, x2-x0, y2-y0);
12 }                      //  (-x0,   -y0) 方向に平行移動して、下請けに出す
```

3.3　変数の有効範囲（スコープ）

三角形の面積の、別の公式を使ってみましょう。ヘロンの公式は少し複雑です。3 辺の長さを a, b, c としたときに、$s = \dfrac{a+b+c}{2}$ という変数を用意して、面積 T は

$$T = \sqrt{s(s-a)(s-b)(s-c)}$$

となります。これをソースコード 3.7 の関数 `triangle_side(a,b,c)` で実現しました。

6 行目では s、7 行目では area2 という変数を定義して、公式の計算を段階的に行います。area2 は面積の 2 乗（つまりルートの中身）です。

8 行目の return 文の `sqrt()` は、やはり数学関数で、**平方根** (square root) を double 型で返します。

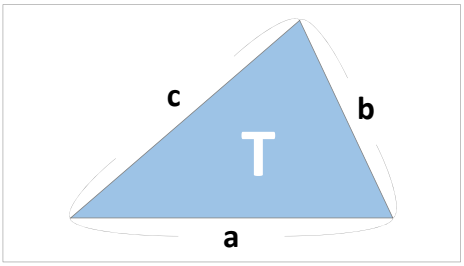
ソースコード 3.7　三角形の面積を求める関数（辺長版）

```
1  #include <stdio.h>
2  #include <math.h> // sqrt() に必要
3
4  /* 三角形の面積を返す（３辺の長さをa,b,cとする） */
5  double triangle_side(double a, double b, double c) { a,b,c のスコープ
6      double s = (a + b + c) / 2.0;                      s のスコープ
7      double area2 = s * (s-a) * (s-b) * (s-c); area2 のスコープ
8      return sqrt(area2);
9  }
10
11 int main(void) {
12     printf("3辺の長さが%g,%g,%gの三角形の面積は%gです。\n",
13            1.0, 1.0, 1.0, triangle_side(1.0, 1.0, 1.0));
14     printf("3辺の長さが%g,%g,%gの三角形の面積は%gです。\n",
15            3.0, 4.0, 5.0, triangle_side(3.0, 4.0, 5.0));
16     return 0;   スペース文字で切れ目を目立たせる
17 }
```

最後に main() から `triangle_side()` を呼び出します。**コンマの後にスペース文字**を入れるのは、そもそもの**基本的な習慣**ですが、引数に小数ばかりが並ぶ時には、切れ目が目立つので**特に有用**です。（ピリオドの後のスペース文字はコンパイルエラーになるので、場所を間違えたらすぐにわかります。）

ソースコード 3.7 の実行結果

```
3辺の長さが1,1,1の三角形の面積は0.433013です。
3辺の長さが3,4,5の三角形の面積は6です。
```



関数で定義した変数は、プログラムのどこから使えるのでしょうか。変数の使える範囲を**有効範囲**[*6]あるいは**スコープ** (scope) といい、**定義したブロックの最後まで**と決まっています。ブロックを抜け出すと、変数は代入も参照もできなくなります。

つまり、ソースコード 3.7 の 6 行目の変数 s は、ブロックの終わりである 9 行目までで使えます（実際に 7 行目でも使っています）が、それ以外の main() のような場所では使えないということです。たとえ main() で同じ名前の s という変数を用意したとしても、triangle_side() の s とは異なる、別の値を記憶する変数として扱われます。

引数のスコープは、（引数名が最初に現れるのは関数のブロックの外側ですが）**関数のブロックの内側の先頭**で定義された場合と同じです。つまり引数は、関数ごとに別のものになります。実際に、5 ページのソースコード 3.4 の関数は f(x), g(x), h(x) と、引数の名前がどれも同じですが、実体は違っていて、数学の関数のような計算ができました。

このおかげで、関数ごとの変数名に重複があるかどうか気にせず、**その関数のことだけを考えて**プログラムを作ればよいことになります。引数の名前を変更した場合でも、影響範囲はその関数内だけで、呼び出す側には何の関係もありません。

コラム： スコープは狭く

小さなプログラムを作っている間は、1 つの変数をいろいろな関数で共有できたほうが便利に思えるかもしれません。しかし、プログラムの間違いで共有の変数の値をおかしくしてしまうことを想像してみてください。プログラムが大きいほど、共有している関数が多いほど、間違いを探すのに手間がかかります。

スコープを狭くしておけば、変数名の重複を気にしなくてよくなったり、コンパイル時に初期化忘れを検出してくれたりと、良いことがたくさんあります。

[*6] 日本語訳はあまり一定せず、「可視範囲」「通用範囲」などとも呼ばれます。プログラミング用語としてもっとも定着しているのは、カタカナの「スコープ」でしょう。

3.4　void 型の関数

8 ページのソースコード 3.7 を、**実行結果は同じ**のまま、関数の構成を変更してみたのがソースコード 3.8 です。行数は少し増えていますが、どんな利点があるのでしょうか。

まずは新しい文法です。11 行目の print_triangle_side() は、関数の型が void と宣言されています。void とは、日常生活ではあまり使わない言葉ですが、「無効の」「空っぽの」というような意味があります。空っぽを返すということは**何も返さない**ということです。

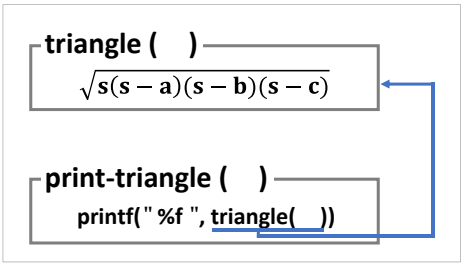
何も返さない関数を、数学の関数から連想すると、存在価値を疑いたくなりますが[*7]、この関数では printf() で表示するという動作をしていますので、C 言語で考えると意味のある作業をしています。

ソースコード 3.8　三角形の面積を表示する関数（辺長版）

```
1  #include <stdio.h>
2  #include <math.h> // sqrt() に必要
3
4  /* 三角形の面積を返す（３辺の長さをa,b,cとする） */
5  double triangle_side(double a, double b, double c) {
6      double s = (a + b + c) / 2.0;
7      double area2 = s * (s-a) * (s-b) * (s-c);
8      return sqrt(area2);
9  }
10
11 /* 三角形の面積を表示する（３辺の長さをa,b,cとする） */
12 void print_triangle_side(double a, double b, double c) {
13     printf("3辺の長さが%g,%g,%gの三角形の面積は%gです。\n",
14             a, b, c, triangle_side(a, b, c));
15 }       //  表示用                  計算用
16
17 int main(void) {
18     print_triangle_side(1.0, 1.0, 1.0);  // void 型の関数なので、
19     print_triangle_side(3.0, 4.0, 5.0);  // 関数呼び出しのみを書く
20     return 0;
21 }
```

void 型の関数は返す値がないので、return 文がありません（15 行目）。呼び出し側では、この関数の戻り値を受け取ることはできないので、変数に代入したりせずに、単に関数呼び出しを書き並べます（18–19 行目）。

[*7] Pascal という言語では、値を返さないものは関数ではなく、手続き（プロシージャ）として、区別して扱います。



3.4.1 プログラムの信頼性

では、この 2 つのプログラムを比べましょう。どちらも三角形の面積を計算しますが、もっと計算したいと思ったときに、どうすればよいでしょうか。ソースコード 3.7 なら main() の printf() を増やそうと思うかもしれません。もちろん、注意深く増やすことは可能です。テキストエディタのコピー機能を使って、printf() 部分をコピーして、3 辺の数値を編集すればよいのですが、同じ数値が表示用と計算用の **2 度出現**することに注意してください。もし間違って異なる数値を書いてしまったら、実行結果はおかしくなります。

```
    printf("3辺の長さが%g,%g,%gの三角形の面積は%gです。\n",
           3.0, 4.0, 5.0, triangle_side(3.0, 4.0, 5.0));
//             表示用                    計算用
```

この不整合の危険を**プログラムの構造**によって防ぐ方法があります。それがソースコード 3.8 で作った関数 print_triangle_side()（12 行〜15 行）です。引数で受け取った a,b,c を、printf() の表示と、triangle_side() の計算の**両方**に使いますから、この関数を呼び出す限り、原理的に不整合が起こらないというわけです。このような手法の積み重ねが、プログラムの信頼性向上に貢献します。

```
    printf("3辺の長さが%g,%g,%gの三角形の面積は%gです。\n",
           a, b, c, triangle_side(a, b, c));
//         表示用                 計算用
```

この 2 つのプログラムの長さは、今は関数を余分に作ったほうが長いですが、表示する面積が増えていくと逆転します。これも処理を関数に独立させるメリットです。

コラム： バグ，デバッグ

プログラミング用語の**バグ** (bug) の語源は「虫」で、プログラムの動作不良や欠陥を指しています。その虫 (bug) を取り除いて (de-)、正しいプログラムにすることを**デバッグ** (debug) という造語で呼んでいます。

3.5　出力＝ return

return 文は、関数の処理の最後に書いて、戻り値を示すのに使いました。return に書いた値は、関数の型（関数名の前に書いた型）で返されるので、この 2 つの型は一致させておくのが正統です。（しかし現実は甘くはありません。☞下のコラム）

void 型の関数には、return 文を書く必要はありませんが、書きたければ書いても構いません。その場合の return 文には、返す値がないので、式を省略してすぐに ; を書きます。

TODO: 複数の値を返すことに言及するか

void 型に限らず、関数には**複数の return** を書いても構いません。もし関数の途中に書くと、それ以降の処理を行わずに、関数を抜け出します。もちろん単独で用いることはありませんが、**??**章の条件分岐と組み合わせると、if 文の else 節を書かなくてよくなるので、インデントが深くならずにすみます。

コラム：return の書き忘れ、型違い

void 型以外の関数では return 文が必要なのは当然ですが、書き忘れてもエラーにならずに警告止まりです[a]。条件分岐（**??**章）の結果、return 文にたどり着かないという場合もあるので、見抜けるかどうか、プログラマは注意力を試されてるようなものです。ちなみに、return のないままで呼び出し元に返される値は、初期化されてない変数と同様で、どんな値になるのかの保証がありません[b]。

さらに、関数の型と return の式の型が一致しなければ、黙ってキャストされて、警告すら出ないこともあります。例えば int 型の関数で「`return 3.14;`」とすれば、**int の 3** が返されます[c]。

```
double f(double x) { // × 不定
    x = x * x;   // return なし
}

int pi(void) {   // × 3
    return 3.14; // intになる
}
```

このように、コンパイルに成功してもまったく安心できません。せめて警告レベルを上げる（☞**??**節）などして対処しましょう。

[a] Java や C# のように、設計の新しい言語では、当然エラーになります。

[b] 最後に評価した式の値が（偶然にも）採用されることがあり、return 忘れを気づきにくくしています。

[c] Java では、このような情報の欠落する暗黙のキャストは、文法エラーになります。C 言語では、char と int を（わざと）混同してきている場面があるため、網羅的に検査することは難しいのですが、限られた場面で警告を出すコンパイラ（例えば Clang）もあります。

表 3.1 仮引数と実引数の違い

	仮引数 (formal) parameter	実引数 (actual) argument
ソースコード上の場所	`int f(int x) {...}`	`y = f(2);`
ソースに現れる回数	1 回	何回でも
文法上の役割	変数	式（値）
具体的な値の決まり方	呼び出された関数が受け取る	呼び出すときに関数に与える

3.6 入力＝引数

関数にとっての入力である引数について、もう少し掘り下げてみます。

3.6.1 仮引数・実引数

これまで、ぼんやりと「引数」と書いてきましたが、呼び出し側と、呼び出された側とでは役割が違うので、表 3.1 のように、区別する呼び方があります。

仮引数 (formal parameter) 呼び出された関数から見た引数です。必ず変数であって、具体的な値は、関数が呼び出されるまでわかりません。

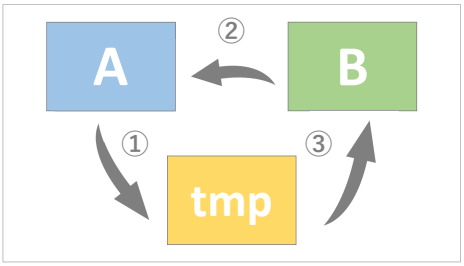
実引数 (actual argument) 関数を呼び出す側が与える値です。定数を与えている場合など、具体的な値が事前にわかることがあります。プログラムに何度現れてもよくて、その度毎に値が異なっても構いません。

この 2 種類は、文脈からも判断できるので、区別はそれほど重要ではありません[*8]が、使い分けると意味が明確になる場面もあります。本書でも、ここからは必要に応じて使い分けてみます。

コラム： 引数の型違い

return 文にも型にまつわる話題がありましたが、引数にもあります。関数呼び出しの際に、実引数と仮引数の型が異なる場合は、自動的に仮引数（受け取り側）の型にキャストされます。double のところに int の数値を渡したときはよいのですが、逆だと困るのにエラーにならないのも、return と同じです。

[*8] 英単語でも parameter と argument の区別は曖昧です。区別したとしても formal parameter と actual parameter のように、両方とも parameter にしたり、逆に両方とも argument にする例もみられます。



3.6.2　値渡し・参照渡し（発展的内容）

??項の変数の値を入れ替える動作（スワップ）を関数にしましょう。しかしソースコード 3.9 では、main() の変数の入れ替えが起きません。確かに swap() 内では入れ替わっているのに、なぜでしょうか。

ソースコード 3.9　変数の値を入れ替える (?)

```
1  #include <stdio.h>
2
3  /* aとbの値を入れ替える(?) */
4  void swap(int a, int b) {                                    a,b のスコープ
5      printf("swap:(before) a=%d, b=%d\n", a, b); // a=1, b=2
6      int tmp = a; a = b; b = tmp;
7      printf("swap:(after)  a=%d, b=%d\n", a, b); // a=2, b=1
8  }
9
10 int main(void) {
11     int a = 1, b = 2;                                        a,b のスコープ
12     swap(a, b); // aとbの値を入れ替えたい
13     printf("main: a=%d, b=%d\n", a, b);          // a=1, b=2 のまま
14     swap(1, 2); // 1と2の値は（当然）入れ替わらず、エラーにもならない
15     return 0;
16 }
```

ここで思い出してほしいことは、まずは変数のスコープです。4 行目の a と、11 行目の a は同じ変数でしょうか。変数のスコープは、変数の定義されたブロック内（**引数は関数のブロック内**）でしたから、4 行目の a は関数の終わる 8 行目まで、11 行目の a は 16 行目までがスコープです。つまり、2 つの a は**異なる変数**です。

次に、12 行目で swap() を呼び出すときの実引数は、（a と b という変数ではなく）1 と 2 という**値がコピーされて引き渡された**、ということです。受け取った swap() は、仮引数の a を 1 で、b を 2 で**初期化**して、5–7 行目の関数ブロックを実行します。main() の a と b の変数には、影響を及ぼせないのです。

表 3.2 引数の渡し方の戦略

	値渡し (call by value)	**参照渡し (call by reference)**
渡されるもの	値	変数を特定するアドレスやポインタ
呼び出し側の変数の変更	できない	できる
C 言語での実現方法	通常の変数	配列やポインタを利用
再帰呼び出し	できる	できない

多くの人の想像するように、呼び出し側（ここでは main() 関数）の変数まで影響が及ぶコンピュータ言語もあります。このような引数の渡し方を**参照渡し** (call by reference) といいます。対して、C 言語の採用するのは**値渡し** (call by value) といいます。コピーを作ってから渡して、呼び出し元への影響をなくしています。

この 2 通りの戦略は、言語によっても違いますし、使い分けのできる場合もある微妙な違いですから、正確に見分けるのは難しいかもしれません。見分けるヒントとして、14 行目のように `swap(1,2)` というような呼び出しをしてみましょう。これが正常に実行できる（つまりコンパイルエラーにならない）なら、そもそも 1 と 2 の値が入れ替わるはずはありませんから、12 行目の a と b の値も変化しないだろうと類推できるでしょう。

それでは C 言語では、swap() のようなものは作れないのでしょうか。それには技があって、**??**章の配列とか、**??**章ポインタの知識で参照渡し相当を実現すれば、呼び出し元の変数を変更できます（☞**??**項）。今のところは、**変数の値が変化するのは、その関数で代入したときのみ**と思っておいてください。

コラム： 値渡しと再帰呼び出し

C 言語は、多くの人の想像する「参照渡し」ではなく、手間をかけてコピーを作って渡す「値渡し」を採用していますが、なぜでしょうか。関数の独立性が高まるというメリットはもちろんありますが、理由はそれだけでしょうか。

C 言語が登場した 1970~1980 年代、新しい言語の特徴として求められた機能には、 (1) コンパイラ型 (2) 構造化プログラミング（goto 文を使わない） (3) 局所変数 (4) 再帰呼び出し可能、といったものがありました。この中の、再帰呼び出しを実現するために必要な機能だという側面もあるでしょう。参照渡しは「値渡し＋ポインタ」で実現できるので、C 言語では値渡し一本に割り切ったのでしょう。

再帰呼び出しは本書の守備範囲を越えますが、ある関数が自分自身を（引数を変えながら）呼び出す手法で、場面によっては非常に威力を発揮します。

3.6.3　引数なし

引数のない関数は、我々が普段から書いている main() 関数が代表格でしょう。「ない」ことを表すのに、関数の仮引数部分に (void) と書きます。() と何も書かないと、呼び出し側（実引数）との整合性が検査されなくなります（☞**??**項の脚注）。つまり、無駄な実引数をつけて呼び出してもコンパイルエラーになりません。

もちろん、関数を呼び出す側では、実引数を空にします。（void を書いてはいけません。）

コラム：(void) vs. ()

引数なしを示すのに、Java 言語では () と空にします。C 言語では、過去との互換性のため、苦肉の策で void のキーワードを目印にしていました。その C 言語でも、C23 からは旧規格の関数宣言が廃止され、C++ と同様、void を省略できます。

3.7　関数を作る意義

ある手続きを関数にする意義は、**手続きに名前をつける**ことにあります。手続きが単純か複雑かは別次元の話です。呼び出し側は、名前を頼りに関数を呼び出して、実際に行われる**具体的な処理内容を知らず**にすむのです。

手続きを関数にまとめると**「入力」と「出力」が明確**になります。ソースコード 3.7 の

```
double triangle_side(double a, double b, double c) { ... }
```

を見ると、引数が 3 辺の長さですから、三角形の面積は「3 辺の長さから計算できて、場所には影響されない」ことがわかります。さらに「辺の長さが double ならば面積も double で得られる」こともわかります。ソースコード 3.6 の

```
double triangle_v3(double x0, double y0,
                   double x1, double y1, double x2, double y2) { ... }
```

であれば、同じ三角形の面積を得るのにも「3 頂点の座標がわかればよくて、辺の長さは不要」ということです。関数の型や引数から、これだけのことがわかります。

5 ページのソースコード 3.4 では $f(x) = x^2$ という単純な関数を作りました。この関数の数値微分を、5 行目の g(x) では

```
    return (f(x + EPS) - f(x)) / EPS;
```

と、f() を呼び出す形で求めました。もし、f() が単純だからといって

```
    return ((x+EPS)*(x+EPS) - x*x) / EPS;
```

と展開してしまうと、f() を書き換えるたびに、g() まで修正せねばなりません。「f() がどんな関数でも、数値微分を求める」ために、**f() の処理内容がどんなに単純であっても**、g() からは f() を呼び出す形にしておきましょう[*9]。

コラム： main も関数

我々がいつも書いている「`int main(void) ...`」も関数ですが、引数なし、戻り値が int 型と決められています。return ではいつも 0 ばかり返す、変わったものです。そのためか、C99 からは main() の「`return 0;`」は**省略可能**になりました。

実は printf() も関数です。C 言語のシステムが用意してくれているものですが、我々が自作した関数と区別は（本当は）ありません。

3.8 練習問題

1. **[関数の入出力（☞3.1 節、3.5 節）]**
   右の関数 central_angle(n) は正 n 角形の中心角 $\frac{360}{n}$ を返すが、不自然である[*10]。(i) **アイウ**の型のうち一致させるべきものを述べよ。(ii) 修正方法を提案せよ。

```
                                イ
/* 正n角形の中心角360/nを返す */
int central_angle(int n) {
  ア  return 360.0 / (double)n;
}                      ウ
```

2. **[単語の読み（☞3.1 節、3.6 節）]**
   「引数」と「仮引数」の、専門用語としての読みを述べよ。「いんすう」ではない。
3. **[関数を作る意義（☞3.7 節）]**
   下の rectangle() 関数は、引数として int 型の幅と高さを受け取り、長方形の面積を int 型で返す。このことから、長方形の面積についてわかることを述べよ。（例えば、面積は何から計算できるのか、どのような条件でどのような値をとるのか。）

```
/* 長方形の面積を返す（幅をw, 高さをhとする）*/
int rectangle(int w, int h) { /* 省略 */ }
```

4. **[計算式による実引数・浮動小数点の誤差（☞3.2 節、??項）]**
   a, b を適当に定めて $f(x) = ax^2 + bx$ を求める関数を作れ。微小な定数 ϵ をマクロ EPS で定め、$g(x) = \dfrac{f(x+\epsilon) - f(x)}{\epsilon}$ と $h(x) = \dfrac{g(x+\epsilon) - g(x)}{\epsilon}$ を求める関数を作れ。main() 関数で、p を適当に定めて $g(p), h(p)$ を表示し、手計算で求めた微係数の理

[*9] C99 では、関数のインライン展開を指示する予約語 `inline` が新設されました。コンパイラが、単純な関数の展開を肩代わりすることも期待できます。
[*10] Java ならコンパイルエラーになる。（しかし C 言語では、例えば GCC では警告にもならない。）

論値 $f'(p)$, $f''(p)$ と比較せよ。また、EPS を調整すると何が起こるか観察せよ[*11]。

5. **[関数の組合せ（☞3.2.1 項）]**
   標準ライブラリの数学関数 <math.h> に、2 つの double の値 a, b の大きい方を返す関数 fmax(a, b) がある。これを用いて、3 つの double の値 a, b, c の最大値を返す関数 `double fmax3(double a, double b, double c)` を作れ。

6. **[関数の役割分担・座標の平行移動（☞3.2.2 節）]**
   7 ページのソースコード 3.6 の `triangle_v2()` 関数を用いて（改造するのではなく、呼び出して）、以下の関数を作れ。
   - `double incircle_v2(double x1, double y1, double x2, double y2)` は、原点と 2 点 $(x_1, y_1), (x_2, y_2)$ を頂点とする三角形の**内接円の半径**を返す。

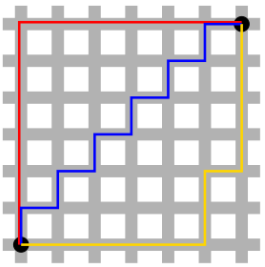
   なお、内接円の半径 r と、面積 S、3 辺の長さの合計 ℓ の間には $r = \dfrac{2S}{\ell}$ の関係がある。次の ℓ を求める関数を利用してよい。（hypot() は ☞**??**節）

```
/* 三角形の3辺の長さの合計（頂点が原点，(x1,y1)，(x2,y2)）*/
double perimeter_v2(double x1, double y1, double x2, double y2) {
    return hypot(x1, y1) + hypot(x2, y2) + hypot(x1-x2, y1-y2);
}
```

   そして、(x_0, y_0) を含む 3 頂点版の `incircle_v3(x0,y0, x1,y1, x2,y2)` も作れ。
   ヒント：`incircle_v2()` を呼び出せ。S や ℓ の 3 頂点版は不要。

7. **[スコープ（☞3.3 節）]**
   ソースコード 3.7 に「`printf("s=%g\n", s);`」を挿入できる行の範囲を述べよ。

8. **[関数の役割分担・void 関数（☞3.2.2 節、3.4.1 節、3.4 節）]**
   マンハッタン距離とは、マンハッタン島にあるような、格子状の道路のみを通ったときの最短距離のことである。したがって 2 点 $(x_1, y_1) - (x_2, y_2)$ の間のマンハッタン距離は $|x_2 - x_1| + |y_2 - y_1|$ と表される。（道順は、右図のように何通りもある。）以下の関数を作れ。



   - `int mdist(int x1, int y1, int x2, int y2)` は $(x_1, y_1) - (x_2, y_2)$ のマンハッタン距離を返す。
   - `int sum_mdist(int x, int y)` は $(0, 0) - (x, y)$ と $(x, y) - (10, 10)$ の、それぞれのマンハッタン距離の合計を返す。
   - `void print_sum_mdist(int x, int y)` は sum_mdist(x,y) の値を表示する。

   いくつかの x, y について `sum_mdist(x,y)` の値を調査し、気のついた顕著な性質を 1 つ述べよ。 <stdlib.h> の絶対値関数 `int abs(int j)` を用いてよい。

9. **[機能を独立させる]** ☞**??**項

[*11] EPS を小さくしすぎると、有効精度におさまりきらず、かえって精度が落ちることに注意せよ。

索引