C の入門書のようなもの（仮）
暫定版

鴨 浩靖



2026 年 6 月 22 日

目次

第 1 章　とにかく動かそう 2
1.1　何もしないプログラム 2
1.2　メッセージを出力するだけのプログラム 2
1.3　空白など 4
1.4　とても簡単な計算 6
1.5　数とその演算 14
1.6　変数名 28

第 2 章　制御構造 29
2.1　分岐と繰り返し 29
2.2　関数 39

第 3 章　配列とポインタ 60
3.1　配列 60
3.2　ポインタ 67
3.3　配列の配列 88
3.4　文字と文字列 93

第 4 章　構造体 106
4.1　構造体 106

第 5 章　標準入出力ライブラリ 117
5.1　標準入力と標準出力 117
5.2　ファイル 124
5.3　char の配列への操作 130

第 6 章　型いろいろ 135
6.1　型に名前をつける 135
6.2　標準型定義 136

第 1 章

とにかく動かそう

1.1 何もしないプログラム

まずは、何もしないプログラムです。プログラム 1.1 をコンパイルして実行しましょう[*1]。実行しても何も起きません。プログラムは、何もせずにすぐに終了します。

プログラムの最初の行にある

```
int main()
```

が、ここからこのプログラムの本体が始まることを示します。直後の{と対応する}の間に、プログラムの本体を書きます。

このプログラムは、本体に一行、

```
        return 0;
```

だけがあります。これが、プログラムを正常終了させます。

プログラム 1.1 何もしないプログラム

```
int main()
{
        return 0;
}
```

1.2 メッセージを出力するだけのプログラム

プログラム 1.2 をコンパイルして実行すると、出力 1.2.1 のように出力されます[*2]。

最初の

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

[*1] プログラムをコンパイルして実行する手順はシステムによって異なります。この資料とは別の資料で説明します。

[*2] どこに出力されるかはシステムによって異なります。いまどきの多くのシステムでは、特に設定しなければプログラムを実行しているターミナルウィンドウに表示されますが、設定による変更が可能です。そうでないシステムも存在します。
この辺りについては、この資料とは別の資料で説明します。

プログラム 1.2 Hello, world!と出力するだけのプログラム

```
#include <stdio.h>

int main()
{
        printf("Hello, world!\n");
        return 0;
}
```

出力 1.2.1: プログラム 1.2 を実行したときの出力

```
Hello, world!
```

プログラム 1.3 Hello, world!と出力するだけのプログラムの変種 1

```
#include <stdio.h>

int main()
{
        printf("Hello, ");
        printf("world!\n");
        return 0;
}
```

出力 1.3.1: プログラム 1.3 を実行したときの出力

```
Hello, world!
```

は、とりあえず、入出力をともなうプログラムではたいてい必要なものと覚えておいてください。

この例でも、

```
int main()
```

の部分が、このプログラムの本体の部分であることを示します。その直後の{と対応する}の間に、プログラムの本体があります。

```
        printf("Hello, world\n");
```

が文字列を出力する文です。文字列は二重引用符"で前後を囲みます。

\n は改行を意味します。改行も一つの文字として扱われます。詳細は次節で説明します。

ちょっと書き換えてみましょう。プログラム 1.3 をコンパイルして実行してみましょう。やはり、出力 1.3.1 のように出力されます。

では、プログラム 1.4 はどうでしょう。こちらは出力 1.4.1 のように出力されます。

プログラム 1.4 Hello, world!と出力するだけのプログラムの変種 2

```
#include <stdio.h>

int main()
{
        printf("Hello,\nworld!\n");
        return 0;
}
```

出力 1.4.1: プログラム 1.4 を実行したときの出力

```
Hello,
world!
```

1.3 空白など

1.3.1 空白と改行

C では、一部の例外を除いて、空白がいくつ連続しても一つの空白と同じ扱いとなります。また、改行は空白一つと同じ扱いです。プログラム 1.2 (3 ページ) もプログラム 1.5 もプログラム 1.6 も同じプログラムの扱いです。ただし、コンピュータにとっては同じでも人間にとっての見やすさは異なりますので、プログラム 1.5 やプログラム 1.6 のように書くことはお勧めしません。

プログラム 1.5 Hello, world!と出力するだけのプログラムの本体をむりやり一行にしたもの

```
#include <stdio.h>

int main() { printf("Hello, world!\n"); return 0; }
```

プログラム 1.6 Hello, world!と出力するだけのプログラムの本体に無駄な改行と空白を入れたもの

```
#include <stdio.h>

int main()
{
        printf("Hello, world!\n");
             return
                    0;
}
```

以下の二つについては例外的に空白の個数や改行が意味をもちます。

- 文字列の中
- #で始まる行

文字列の中

文字列の中の空白は個数が意味をもちます。プログラム 1.2 (3 ページ) とプログラム 1.7 は、実行すると違う出力がされます。

プログラム 1.7 Hello,　world!と出力するだけのプログラム

```
#include <stdio.h>

int main()
{
        printf("Hello,    world!\n");
        return 0;
}
```

出力 1.7.1: プログラム 1.7 を実行したときの出力

```
Hello,    world!
```

文字列の途中で改行することは文法違反です。システムによっては、文字列の途中で改行してもそれらしい動作をすることがありますが、偶然ですので頼ってはいけません。システムによって動いたり動かなかったりするプログラムになります。

#で始まる行

で始まる行は、原則としてその行限りです。途中で改行するには、特別な書き方が必要になります。したがって、

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

などは、必ず一行で書いてください。

で始まる行には、#include の他にもいろいろな種類があります。他にどんなものがあるかは、次章以降、説明します。

1.3.2 コメント

プログラム中に以下の二つのどちらかを書くと、コメントとみなされて無視してくれます。

- /* から */ まで。途中で改行しても可。
- // から行末まで。

正確にいえば、コメントは空白一個と同じものとして扱われます。

コメントは計算機ではなく人間が読むためのものです。プログラムを読む人のための解説や、自分のためのメモを自由に書いてください。

なお、/* */ は入れ子にできないので注意してください。

```
/*
This is /* a pen */ a pencil.
*/
```

は、最初にみつけた */ でコメントが終了して

```
  a pencil.
*/
```

と同じことになってしまい、/* に対応しない */ が出現したことになって文法エラーになります。

1.4 とても簡単な計算

1.4.1 計算結果を出力する

プログラム 1.8 は、11 と 3 の和と差と積と商（小数点以下切捨て）と割ったときの余りを計算をするプログラムです。

プログラム 1.8 11 と 3 の和と差と積と商（小数点以下切捨て）と割ったときの余りを計算をする

```
#include <stdio.h>

int
main()
{
        int     x = 11, y = 3;

        printf("%d\n", x + y);
        printf("%d\n", x - y);
        printf("%d\n", x * y);
        printf("%d\n", x / y);
        printf("%d\n", x % y);
        return 0;
}
```

プログラム 1.8 を実行すると、出力 1.8.1 のように出力されるでしょう。

出力 1.8.1: プログラム 1.8 を実行したときの出力

```
14
8
33
3
2
```

プログラム 1.8 にある

```
        int     x = 11, y = 3;
```

は、変数の定義です。ここで、変数が作られます。変数とは、値を入れる箱のようなものです。

最初の `int` は、これから定義する変数は int 型であるということを意味しています。つまり、これから定義する変数には値として整数値が入るということです。`x` と `y` が変数の名前です。`= 11` は、変数を作った直後にそこに整数値 11 を入れろということです。したがって、`x = 11` の部分で、名前が `x` である変数が作られ、そこに整数値 11 が入れられます。同じように、`y = 3` の部分で、名前が `y` である変数が作られ、そこに整数値 3 が入れられます。

ここの= (等号) は「等しい」という意味ではありません。変数に値を入れる意味になります。注意してください。なお、C で「等しい」を意味する記号は別にあります。

次の

```
        printf("sum %d\n", x + y);
```

は、$x + y$ を計算してその結果を出力します。

printf に与える文字列に % (パーセント記号) が含まれると、その後ろの何文字かとあわせて特別な意味を持ちます。%d は、別に与えられた整数値を 10 進で表示させます。

x + y は x と y の和を計算する式です。つまり、+ は C の式で使える演算子のひとつです。

C の式で使える演算子はたくさんありますが、まずは、次の五つを覚えてください。

+	足し算	-	引き算	*	掛け算	/	割り算	%	余り

計算の順番を指定するために丸括弧 () を使うことができます。たとえば、x + (y * z) は掛け算を先に行ってから足し算を行います。(x + y) * z は足し算を先に行ってから掛け算を行います。

括弧は何重にでも入れ子にすることができます。ただし、すべて丸括弧 () を使ってください。他の括弧は違う意味になります。

では、括弧なしで x + y * z と書いたらどうなるでしょうか？この場合は掛け算が先に行われます。一般の慣習と同じく、C でも掛け算は足し算よりも優先することになっているからです。

以下のルールを覚えておくと、とりあえず、困らないでしょう。

- 丸括弧 () はすべてに優先する。
- 掛け算*、割り算/、余り % は足し算 +、引き算- よりも優先する。
- 掛け算*、割り算/、余り % が連続するときは、左側が優先する。
- 足し算 +、引き算- が連続するときは、左側が優先する。

プログラム 1.8 の出力は寡黙すぎてどの数が何なのかわからないくて悲しい方は、プログラム 1.9 のように少し饒舌にすることもできます。

プログラム 1.9 11 と 3 の和と差と積と商（小数点以下切捨て）と割ったときの余りを計算をする

```
#include <stdio.h>

int
main()
{
        int     x = 11, y = 3;

        printf("sum %d\n", x + y);
        printf("difference %d\n", x - y);
        printf("product %d\n", x * y);
        printf("quotient %d\n", x / y);
        printf("remainder %d\n", x % y);
        return 0;
}
```

プログラム 1.9 を実行すると、出力 1.9.1 のように出力されるでしょう。

出力 1.9.1: プログラム 1.8 を実行したときの出力

```
sum 14
difference 8
product 33
quotient 3
remainder 2
```

printf に与える文字列に %d を複数回出現させることも可能です。プログラム 1.9 をプログラム 1.10 のように書き換えても、同じ動作をします。

プログラム 1.10 11 と 3 の和と差と積と商（小数点以下切捨て）と割ったときの余りを計算をする（別解 1）

```
#include <stdio.h>

int
main()
{
        int     x = 11, y = 3;

        printf("sum %d\ndifference %d\n", x + y, x - y);
        printf("product %d\nquotient %d\nremainder %d\n", x * y, x / y, x % y);
        return 0;
}
```

プログラム 1.11 のように、計算結果を変数にいったん代入してから使うこともできます。

計算結果を変数に代入しておくと、再利用できます。プログラム 1.12 は、$x+y$ と $x-y$ の計算結果を $(x+y)^2$ と $(x-y)^2$ と $(x+y)(x-y)$ の計算に再利用しています。

プログラム 1.13 もプログラム 1.12 と同じ動作をします。プログラム 1.12 では必要な変数を最初にまとめて作っていますが、プログラム 1.13 では必要になるぎりぎりのタイミングで作っています。どちらの書き方を採用してもかまいません。

プログラム 1.11 11 と 3 の和と差と積と商（小数点以下切捨て）と割ったときの余りを計算をする（別解 2）

```
#include <stdio.h>

int
main()
{
        int     x = 11, y = 3;
        int     s, d, p, q, r;

        s = x + y;
        printf("sum %d\n", s);
        d = x - y;
        printf("difference %d\n", d);
        p = x * y;
        printf("product %d\n", p);
        q = x / y;
        printf("quotient %d\n", q);
        r = x % y;
        printf("remainder %d\n", r);
        return 0;
}
```

プログラム 1.12 11 と 3 の和と差と積と商（小数点以下切捨て）と割ったときの余りと和の二乗と差の二乗と和と差の積を計算する（別解 1）

```
#include <stdio.h>

int
main()
{
        int     x = 11, y = 3;
        int     s, d, p, q, r, ss, dd, sd;

        s = x + y;
        printf("sum %d\n", s);
        d = x - y;
        printf("difference %d\n", d);
        p = x * y;
        printf("product %d\n", p);
        q = x / y;
        printf("quotient %d\n", q);
        r = x % y;
        printf("remainder %d\n", r);
        ss = s * s;
        printf("squared sum %d\n", ss);
        dd = d * d;
        printf("squared difference %d\n", dd);
        sd = s * d;
        printf("sum times difference %d\n", sd);
        return 0;
}
```

プログラム 1.13 11 と 3 の和と差と積と商（小数点以下切捨て）と割ったときの余りと和の二乗と差の二乗と和と差の積を計算する（別解 2）

```
#include <stdio.h>

int
main()
{
        int     x = 11, y = 3;

        int     s = x + y;
        printf("sum %d\n", s);
        int     d = x - y;
        printf("difference %d\n", d);
        int     p = x * y;
        printf("product %d\n", p);
        int     q = x / y;
        printf("quotient %d\n", q);
        int     r = x % y;
        printf("remainder %d\n", r);
        int     ss = s * s;
        printf("squared sum %d\n", ss);
        int     dd = d * d;
        printf("squared difference %d\n", dd);
        int     sd = s * d;
        printf("sum times difference %d\n", sd);
        return 0;
}
```

1.4.2 値をユーザが入力して計算結果を出力する

前節のプログラム例は、どれも、計算の対象となる値 11 と 3 がプログラムに埋め込まれています。これでは、他の値に対して計算しようとするといちいちプログラムを書き換えなくてはならず、不便です。

そこで、計算の対象となる値を実行時にユーザが入力するプログラムに書き換えてみたのが、プログラム 1.14 です。

プログラム 1.14 入力した整数の和と差と積と商（小数点以下切捨て）と割ったときの余りを計算をする

```
#include <stdio.h>

int
main()
{
        int     x, y;

        scanf("%d%d", &x, &y);
        printf("sum %d\n", x + y);
        printf("difference %d\n", x - y);
        printf("product %d\n", x * y);
        printf("quotient %d\n", x / y);
        printf("remainder %d\n", x % y);
        return 0;
}
```

プログラム 1.14 を実行すると、一見、何もいわずに黙り込んだようになりますが、実は、入力待ち状態です。そこで、ユーザがたとえば、

```
11 3
```

と入力すると、

```
sum 14
difference 8
product 33
quotient 3
remainder 2
```

が出力されるでしょう。

```
        scanf("%d%d", &x, &y);
```

が、入力を行う文です。ここでの `%d` は、`printf` での `%d` とほぼ同様に、10 進で表現された整数値を意味します。

`&`の意味については後回しにさせてください。ここでは、こう使うものだと覚えておいてください。

```
        int     x, y;
```

ではどちらの変数も=以降がありません。このような定義では、変数が作られた直後にはゴミの値が入っています。意味のある値を入れる前に誤って変数を使わないように注意してください。

1.4.3 浮動小数点数の計算

まずは、平面上の点の直交座標（図 1.1 右）と極座標（図 1.1 左）の変換をするプログラムを書いてみましょう。直交座標 (x, y) と極座標 (r, θ) の関係は、次の式で表せます。

$$\begin{cases} x = r\cos\theta \\ y = r\sin\theta \end{cases} \qquad \begin{cases} r = \sqrt{x^2 + y^2} \\ \theta = \arctan\dfrac{y}{x} \end{cases}$$

この公式をそのまま実装したのが、プログラム 1.15 と 1.16 です。プログラム 1.15 は、極座標を入力して直交座標に変換して出力します。プログラム 1.16 は、直交座標を入力して極座標に変換して出力します。

どちらのプログラムでも、変数を作るために、`double` を使っています。座標値は整数とは限らないので、変数が整数専用では困るからです。`int` は整数しか扱えません。`double` を使うと、小数を扱うことができるようになります。

`scanf` の書式にある `%lf` と `printf` の書式にある `%f` が、`double` 型の、それぞれ、入力と出力を指定する

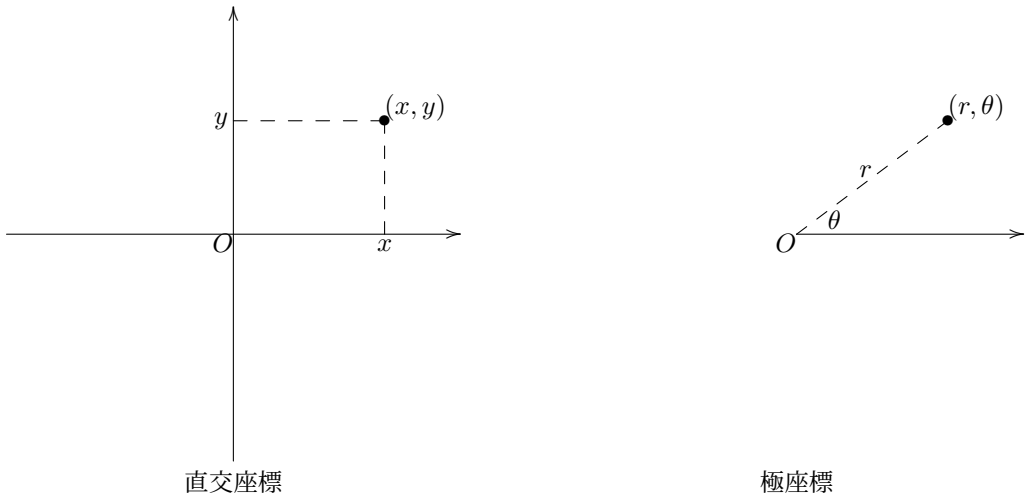


図 1.1: 平面上の点の座標

プログラム 1.15 極座標を直交座標に変換（二次元）

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

int
main()
{
        double  r, theta;
        double  x, y;

        scanf("%lf%lf", &r, &theta);
        x = r * cos(theta);
        y = r * sin(theta);
        printf("%f %f\n", x, y);
        return 0;
}
```

プログラム 1.16 直交座標を極座標に変換（二次元）

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

int
main()
{
        double  x, y;
        double  r, theta;

        scanf("%lf%lf", &x, &y);
        r = hypot(x, y);
        theta = atan2(y, x);
        printf("%f %f\n", r, theta);
        return 0;
}
```

ものです。詳細は、1.5.4 と 1.5.5 で説明します。

C には、標準的に使うことのできる数学関数が多数あります。その一部を 1.5.7 で並べています。cos と sin は、それぞれ、余弦関数と正弦関数を計算します。hypot は二乗和の平方根を計算します。atan2 は逆正接関数を計算します。

C の標準の数学関数を使うときには、プログラムの初めのほうに

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

が必要です。プログラム 1.15 やプログラム 1.16 では

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

も必要ですが、どちらを先に書いても良いと C の言語仕様で決まっています。

今後、

```
#include <なんとか.h>
```

がいろいろと出てきます。それらが複数必要なときにはどのような順番で書いても問題ないことも、C の言語仕様で決まっています。

次に、三次元空間上の点の直交座標と極座標の変換をするプログラムを書いてみましょう。直交座標 (x, y, z) と極座標 (r, θ, φ) の関係[*3]は、次の式で表せます。

$$\begin{cases} x = r\cos\theta\sin\varphi \\ y = r\sin\theta\sin\varphi \\ z = r\cos\varphi \end{cases} \qquad \begin{cases} r = \sqrt{x^2+y^2+z^2} \\ \theta = \arctan(y/x) \\ \varphi = \arctan(\sqrt{x^2+y^2}/z) \end{cases}$$

この公式をそのまま実装したのが、プログラム 1.17 と 1.18 です。プログラム 1.17 は、極座標を入力して直交座標に変換して出力します。プログラム 1.18 は、直交座標を入力して極座標に変換して出力します。

プログラム 1.17 も 1.18 も、中間結果を変数に代入して再利用することで計算回数を減らしています。プログラム 1.17 では $\sqrt{x^2+y^2}$ の計算結果を変数 hypot_x_y に代入して、r の計算と φ の計算で変数 hypot_x_y

[*3] 文献によっては θ と φ が入れ替わっているものもありますが、本質的には同じことです。そのような文献に出会っても、適宜、読み替えてください。

プログラム 1.17 極座標を直交座標に変換（三次元）

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

int
main()
{
        double  r, theta, phi;
        double  x, y, z;
        double  sin_phi;

        scanf("%lf%lf%lf", &r, &theta, &phi);
        sin_phi = sin(phi);
        x = r * cos(theta) * sin_phi;
        y = r * sin(theta) * sin_phi;
        z = r * cos(phi);
        printf("%f %f %f\n", x, y, z);
        return 0;
}
```

プログラム 1.18 直交座標を極座標に変換（三次元）

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

int
main()
{
        double  x, y, z;
        double  r, theta, phi;
        double  hypot_x_y;

        scanf("%lf%lf%lf", &x, &y, &z);
        hypot_x_y = hypot(x, y);
        r = hypot(hypot_x_y, z);
        theta = atan2(y, x);
        phi = atan2(hypot_x_y, z);
        printf("%f %f %f\n", r, theta, phi);
        return 0;
}
```

を利用することで、同じ計算を二度行なって計算時間を無駄にすることを防いでいます。r の計算では、

$$\sqrt{x^2+y^2+z^2} = \sqrt{\left(\sqrt{x^2+y^2}\right)^2+z^2}$$

をうまく利用していることに注意してください。プログラム 1.18 では $\sin\varphi$ の計算結果を変数 `sin_phi` に代入して、x の計算と y の計算変数 `sin_phi` を利用することで、同じ計算を二度行なって計算時間を無駄にすることを防いでいます。

なお、変数 `sin_phi` はコンピュータにとっては単にそういう名前の変数であって、その名前からコンピュータが何かを判断してくれたりはしません。人間が見て $\sin\varphi$ を意図していると推測しやすくしているだけです。人間の側の都合であって、コンピュータの知ったことではありません。

1.5 数とその演算

1.5.1 数値の型のいろいろ

本節では C で扱うことのできる数値について解説します。

C で標準的に使える数値には、整数と浮動小数点数があります。浮動小数点数には実浮動小数点数と複素数がありますが、本節では実浮動小数点数のみを解説します。

整数

「整数」と呼んでいますが、C で標準的に扱える整数は、実際には決められた最小値と最大値の間にある範囲の整数だけです[*4]。変数は作られる時にその型も決められます。その型に応じて格納できる整数の範囲が決まります。

少なくとも、以下の組み合わせで作られる整数型は使用できることが、C の言語仕様で決まっています。

$$\left\{\begin{array}{l}\texttt{signed}\\ \texttt{unsigned}\\ \text{（なし）}\end{array}\right\}\left\{\begin{array}{ll}\left\{\begin{array}{l}\texttt{short}\\ \text{（なし）}\\ \texttt{long}\\ \texttt{long long}\end{array}\right\} & \texttt{int}\\ & \texttt{char}\end{array}\right\}$$

前半が符号つきか符号なしかを、後半がビット幅を表します。ただし、2 語以上で最後が `int` のときは省略できます。システムによっては、これ以外の整数型が存在することもあります。

`signed` があるときは符号つきの型になります。`unsigned` があるときは符号なしの型になります。どちらもないときは、次のようになります。

- `short int` は `singed short int` と同じ。
- `int` は `singed int` と同じ。
- `long int` は `singed long int` と同じ。
- `long long int` は `singed long long int` と同じ。
- `char` は `singed char` と `unsinged char` のとちらかと同じだが、どちらと同じであるかはシステムによって異なる。

C の言語仕様では、整数型のビット幅について以下のように決まっています。

$$\begin{aligned}&\texttt{char}\ \text{のビット幅}\\ &\quad\leq \texttt{short int}\ \text{のビット幅}\\ &\quad\leq \texttt{int}\ \text{のビット幅}\\ &\quad\leq \texttt{long int}\ \text{のビット幅}\\ &\quad\leq \texttt{long long int}\ \text{のビット幅}\end{aligned}$$

実際のビット幅はシステムによって異なります。表 1.1 と表 1.2 と表 1.3 は実際に存在するコンピュータでの例です。これがすべてではありません。これ以外を採用しているコンピュータも多数存在します。

[*4] 大きな整数を扱いたければ特にプログラミングする必要があります。

表 1.1: あるコンピュータでの整数の実装 (1)

型	ビット幅	最小値	最大値
`char`	8 ビット	−128	127
`signed char`	8 ビット	−128	127
`unsigned char`	8 ビット	0	255
`short int`	16 ビット	−32768	32767
`unsigned short int`	16 ビット	0	65535
`int`	32 ビット	−2147483648	2147483647
`unsigned int`	32 ビット	0	4294967295
`long int`	64 ビット	−9223372036854775808	9223372036854775807
`unsigned long int`	64 ビット	0	18446744073709551615
`long long int`	64 ビット	−9223372036854775808	9223372036854775807
`unsigned long long int`	64 ビット	0	18446744073709551615

表 1.2: あるコンピュータでの整数の実装 (2)

型	ビット幅	最小値	最大値
`char`	8 ビット	−128	127
`signed char`	8 ビット	−128	127
`unsigned char`	8 ビット	0	255
`short int`	16 ビット	−32768	32767
`unsigned short int`	16 ビット	0	65535
`int`	32 ビット	−2147483648	2147483647
`unsigned int`	32 ビット	0	4294967295
`long int`	32 ビット	−2147483648	2147483647
`unsigned long int`	32 ビット	0	4294967295
`long long int`	64 ビット	−9223372036854775808	9223372036854775807
`unsigned long long int`	64 ビット	0	18446744073709551615

表 1.3: あるコンピュータでの整数の実装 (3)

型	ビット幅	最小値	最大値
`char`	8 ビット	0	255
`signed char`	8 ビット	−128	127
`unsigned char`	8 ビット	0	255
`short int`	16 ビット	−32768	32767
`unsigned short int`	16 ビット	0	65535
`int`	16 ビット	−32768	32767
`unsigned int`	16 ビット	0	65535
`long int`	32 ビット	−2147483648	2147483647
`unsigned long int`	32 ビット	0	4294967295
`long long int`	64 ビット	−9223372036854775808	9223372036854775807
`unsigned long long int`	64 ビット	0	18446744073709551615



浮動小数点数

コンピュータでは、本当の実数の計算をしてはいません。本当の実数は無限の精度をもっているので、そもそも、コンピュータで実現することはできないからです。

そのかわりに、コンピュータは浮動小数点数と呼ばれる数を使って計算します。多くのコンピュータでは

$$\pm f \times 2^e \qquad \text{ただし、} f \text{ は二進有限小数、} e \text{ は整数}$$

の形で表現できる数を使います。たまに、

$$\pm f \times 10^e \qquad \text{ただし、} f \text{ は十進有限小数、} e \text{ は整数}$$

の形で表現できる数を使うコンピュータもあります。これらを総称して、浮動小数点数と呼びます。二進でも十進でもない進法の浮動小数点数を採用することは理論的には可能ですが、現在では実際に採用されている例を見ません。

浮動小数点数については、以下の三つの型は存在することが C の言語仕様で決まっています。

```
float
double
long double
```

システムによっては、それ以外の浮動小数点数型が存在することもあります。

C の言語仕様では、浮動小数点数の精度について以下のように決まっています。

`float` の精度 ≤ `double` の精度 ≤ `long double` の精度

実際の精度はシステムによります。いまどきのコンピュータの大部分は、表 1.4 か表 1.5 か表 1.6 のいずれかを採用しています。

表 1.4: あるコンピュータでの浮動小数点数の実装 (1)

型	フォーマット	ビット幅	精度（2 進）	精度（10 進換算）
`float`	IEEE-754 単精度	32 ビット	24 桁	約 7 桁
`double`	IEEE-754 倍精度	64 ビット	53 桁	約 16 桁
`long double`	IEEE-754 四倍精度	128 ビット	113 桁	約 34 桁

表 1.5: あるコンピュータでの浮動小数点数の実装 (2)

型	フォーマット	ビット幅	精度（2 進）	精度（10 進換算）
`float`	IEEE-754 単精度	32 ビット	24 桁	約 7 桁
`double`	IEEE-754 倍精度	64 ビット	53 桁	約 16 桁
`long double`	拡張倍精度	80 ビット	64 桁	約 19 桁

表 1.6: あるコンピュータでの浮動小数点数の実装 (3)

型	フォーマット	ビット幅	精度（2 進）	精度（10 進換算）
`float`	IEEE-754 単精度	32 ビット	24 桁	約 7 桁
`double`	IEEE-754 倍精度	64 ビット	53 桁	約 16 桁
`long double`	IEEE-754 倍精度	64 ビット	53 桁	約 16 桁

1.5.2 プログラム中での数値の表現

整数

C では、整数定数を十進法、八進法、十六進法のいずれかで書くことができます。`0`（ゼロ）以外の十進数字の並びは十進法で解釈されます。`0` で始まると八進法で解釈されます。`0x` または `0X` に続く十六進数字の並びは十六進法で解釈されます。十六進数字のうちラテン文字を流用しているものについては、大文字と小文字を区別しません。

十進数字

```
0  1  2  3  4  5  6  7  8  9
```

八進数字

```
0  1  2  3  4  5  6  7
```

十六進数字

```
0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
                              A  B  C  D  E  F
```

例.

`125`	10 進法で 125
`0125`	8 進法で 125　（十進で 85）
`0x125`	16 進法で 125　（十進で 293）
`0x7beef`	16 進法で 7beef　（十進で 507631）

整数定数の末尾に以下の接尾辞をつけて型を強制できます。U 系と L 系を同時に使用することも可能です。

接尾辞	意味
`U` または `u`	unsigned
`L` または `l`	long
`LL` または `ll`	long long

例.

`31`	int 型の 31
`31U`	unsigned int 型の 31
`31L`	long int 型の 31
`31UL`	unsigned long int 型の 31
`31LL`	long long int 型の 31
`31ULL`	unsigned long long int 型の 31

浮動小数点数

浮動小数点定数には、いくつかの書き方があります。

まず、通常の小数表現と同じ形の表現が使えます。十進法で解釈されます。

整数部.小数部

さらに、大きな数や小さな数を表すのに便利な表現があります。

表現			意味
有効数字部`e`指数部	または	有効数字部`E`指数部	有効数字部 $\times 10^{\text{指数部}}$

ただし、有効数字部は十進整数表現か十進小数表現です。すなわち、以下のいずれかの形で、十進法で解釈されます。

表現	意味
整数部	（十進整数表現）
整数部.小数部	（十進小数表現）

指数部は十進整数表現です。

浮動小数点定数の接尾辞で型を指定できます。

接尾辞	型
F または f	float
（なし）	double
L または l	long double

例.

`1.22F`	float 型の 1.22
`1.22e-20F`	float 型の 1.22×10^{-20}
`1.22e+20F`	float 型の $1.22 \times 10^{+20}$
`1.22`	double 型の 1.22
`1.22e-20`	double 型の 1.22×10^{-20}
`1.22e+20`	double 型の $1.22 \times 10^{+20}$
`1.22L`	long double 型の 1.22
`1.22e-20L`	long double 型の 1.22×10^{-20}
`1.22e+20L`	long double 型の $1.22 \times 10^{+20}$

浮動小数点定数を `0x`（ゼロ・エックス）または `0X` で始めると十六進で解釈します。

`0x`整数部`.`小数部　または　`0X`整数部`.`小数部

浮動小数点定数の十六進表現に指数部をつけることもできます。その場合は以下のように書きます。

表現			意味
有効数字部`p`指数部	または	有効数字部`P`指数部	有効数字部 $\times 2^{\text{指数部}}$

ただし、有効数字部は十六進整数表現か十六進小数表現です。すなわち、以下のいずれかの形で、十六進法で解釈されます。

表現			意味
`0x`整数部	または	`0X`整数部	（十六進整数表現）
`0x`整数部`.`小数部	または	`0X`整数部`.`小数部	（十六進小数表現）

指数部は十進整数表現です。有効数字部が十六進でも指数部は十進であることに注意してください。指数部は $2^{\text{指数部}}$ であって $16^{\text{指数部}}$ でないことにも注意してください。

浮動小数点定数の接尾辞で型を指定できることは、十六進表現も十進表現と同じです。

接尾辞	型
F または f	float
（なし）	double
L または l	long double

例.

`0x1.22F`	float 型の $(1.22)_{16\text{ 進}}$
`0x1.22p-20F`	float 型の $(1.22)_{16\text{ 進}} \times 2^{-20}$
`0x1.22p+20F`	float 型の $(1.22)_{16\text{ 進}} \times 2^{+20}$
`0x1.22`	double 型の $(1.22)_{16\text{ 進}}$
`0x1.22p-20`	double 型の $(1.22)_{16\text{ 進}} \times 2^{-20}$
`0x1.22p+20`	double 型の $(1.22)_{16\text{ 進}} \times 2^{+20}$
`0x1.22L`	long double 型の $(1.22)_{16\text{ 進}}$
`0x1.22p-20L`	long double 型の $(1.22)_{16\text{ 進}} \times 2^{-20}$
`0x1.22p+20L`	long double 型の $(1.22)_{16\text{ 進}} \times 2^{+20}$

1.5.3 C の演算子の一部

整数

■単項演算子（前置）

`+` 符号そのまま　`-` 符号反転

`~` ビットごとの not

■二項演算子

`+` 足し算　`-` 引き算　`*` 掛け算　`/` 割り算　`%` 余り

`&` ビットごとの and　`|` ビットごとの or　`^` ビットごとの xor

浮動小数点数

■単項演算子（前置）

`+` 符号そのまま　`-` 符号反転

■二項演算子

`+` 足し算　`-` 引き算　`*` 掛け算　`/` 割り算

いろいろ

■単項演算子（前置）

（型名）　型名 への強制型変換

1.5.4 scanf でのフォーマット

整数

`%d` 符号つき 10 進　`%u` 符号なし 10 進　`%o` 符号なし 8 進

`%x` `%X` 符号なし 16 進

duoxX のどれかの前につけるもの

hh	char
h	short
l	long
ll	long long

例.

```
int     n;
unsigned long x;
scanf("%d%lu", &n, &x);
```

浮動小数点数

%a %A %e %E %f %F %g %G

aAeEfFgG のどれかの前につけるもの

（なし）	float
l	double
L	long double

例.

```
double  t;
long double x;
scanf("%lf%Lf", &t, &x);
```

1.5.5 printf でのフォーマット

整数

%d	符号つき 10 進	%u	符号なし 10 進	%o	符号なし 8 進
%x	符号なし 16 進（小文字）	%X	符号なし 16 進（大文字）		

duoxX のどれかの前につけるもの

hh	char
h	short
l	long
ll	long long

% とアルファベットの間に十進整数で 幅 を入れることで、表示する幅（文字数）を指定できます。たとえば、%10d だと、int 型の値を 10 文字分の幅で表示しようとします。ただし、素直に表示すれば幅が 10 文字に足りないときは、左に空白を埋めて表示します。10 文字を超えるときは、超えた分だけ必要な文字数で表示します。

幅 を 0 から始めた場合は例外です。その場合は、足りないときに左に数字の 0 を埋めて表示します。

幅 の先頭に + をつけると、数値の正負にかかわらず必ず符号（+ か-）を前につけて表示します。これがない場合は、数値が負の場合は符号-をつけますが、非負の場合は何もつけません。

例.

```
        unsigned n;
        long x;
  中略

        printf("%3u: %18ld\n", n, x);
```

浮動小数点数

%f	%F	十進小数	$ddd.ddd$
%e		指数部つき十進小数	$d.ddd$e$\pm dd$
%E		指数部つき十進小数	$d.ddd$E$\pm dd$
%g		値に応じて %f か %e を選択	
%G		値に応じて %F か %E を選択	
%a		指数部つき十六進小数（小文字）	0x$h.hhh$p$\pm dd$
%A		指数部つき十六進小数（大文字）	0x$h.hhh$P$\pm dd$

aAeEfFgG のどれかの前につけるもの

（なし）	double
L	long double

% とアルファベットの間に十進整数で 幅 を入れることで、表示する幅（文字数）を指定できます。たとえば、%12f だと、double 型の値を 12 文字分の幅で表示しようとします。ただし、素直に表示すれば幅が 10 文字に足りないときは、左に空白を埋めて表示します。10 文字を超えるときは、超えた分だけ必要な文字数で表示します。

幅 を 0 から始めた場合は例外です。その場合は、足りないときに左に数字の 0 を埋めて表示します。

幅 の先頭に + をつけると、数値の正負にかかわらず必ず符号（+ か-）を前につけて表示します。これがない場合は、数値が負の場合は符号-をつけますが、非負の場合は何もつけません。

% とアルファベットの間に .桁数 を入れることで、小数点以下の桁数を指定できます。たとえば、%.10f だと、小数点以下 10 桁の十進小数として表示されます。

幅 と .桁数 の両方を指定する際は、幅.桁数 の順です。

例.

```
        double x, y;
  中略

        printf("%12.6f%+12.6f\n", x, y);
```

1.5.6 最小値と最大値

ある型の変数に格納できる数値の範囲はシステムごとに異なります。その最大値・最小値がプログラムの中で必要になったときに、そのシステムでの値をプログラムに直接書き込むと、他のシステムに移植するたびにそこを書き換える必要が生じ、不便ですしトラブルの元です。

数値型それぞれについて、その型での最小値・最大値を表す方法が提供されています。その方法を使って書いておけば、システムごとに適切な値として扱ってくれます。違うシステムで動作させようとするたびに書き換える必要がなくなって、便利です。

整数

```
#include <limits.h>
```

が必要です。

型	最小値	最大値
char	CHAR_MIN	CHAR_MAX
signed char	SCHAR_MIN	SCHAR_MAX
unsigned char	0	UCHAR_MAX
short int	SHRT_MIN	SHRT_MAX
unsigned short int	0U	USHRT_MAX
int	INT_MIN	INT_MAX
unsigned int	0U	UINT_MAX
long int	LONG_MIN	LONG_MAX
unsigned long int	0UL	ULONG_MAX
long long int	LLONG_MIN	LLONG_MAX
unsigned long long int	0ULL	ULLONG_MAX

浮動小数点数

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

が必要です。

型	最小値	最大値
float	-HUGE_VALF	HUGE_VALF
double	-HUGE_VAL	HUGE_VAL
long double	-HUGE_VALL	HUGE_VALL

1.5.7 C の標準ライブラリに含まれる関数の一部

整数

```
#include <stdlib.h>
```

が必要です。

■int を受け取り int を返す整数計算の関数

abs(x)	$\lvert x \rvert$	x の絶対値
div(x, y)		x を y で割った商と余り[*5]

■long を受け取り long を返す整数計算の関数および long long を受け取り long long を返す整数計算の関数　int 版のそれぞれに対応して、long 版と long long 版があります。

- long 版の関数の名前は int 版の関数の名前の前に l（小文字のエル）を付け加えたものです。
- long long 版の関数の名前は int 版の関数の名前の前に ll（小文字のエル二つ）を付け加えたものです。

たとえば、絶対値を計算する関数は、次のように三つあります。

	x の型	計算結果の型
abs(x)	int	int
labs(x)	long	long
llabs(x)	long long	long long

浮動小数点数

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

が必要です。

■double を受け取り double を返す数学関数

実数操作

copysign(x, y)		x と絶対値が等しく y と符号が等しい数
nextafter(x, y)		x から y を見た向きで次の位置にある表現可能な数

丸め

ceil(x)	$\lceil x \rceil$	x 以上の最小の整数
floor(x)	$\lfloor x \rfloor$	x 以下の最大の整数
round(x)		x に最も近い整数[*6]
trunc(x)		x から 0 に向かって最も近い整数

剰余

fmod(x, y)	x を y で割った余り[*7]
remainder(x, y)	x を y で割った余り[*8]

[*5] 返値は構造体です。

```
div_t   result;
```

で、変数 result が定義されている状態で、

```
result = div(x, y);
```

が実行されると、result.quot に商が、result.rem に余りが格納されます。

[*6] x が隣り合った整数のちょうど真ん中の値のときは、round(x) の値は 0 から遠いほうの整数です。

[*7] $y \neq 0$ のとき、fmod(x, y) の値は y と同符号でその絶対値は $|y|$ よりも小さい。

[*8] $y \neq 0$ のとき、x/y に最も近い整数を n とおくと、remainder(x, y) の値は $x - ny$ となります。x/y が隣り合った整数のちょうど真ん中の値のときは、n として偶数のほうが選ばれます。

最大・最小に関連する関数

`fmax(`x`, `y`)`	$\max\{x, y\}$	x と y の大きいほう
`fmin(`x`, `y`)`	$\min\{x, y\}$	x と y の小さいほう
`fabs(`x`)`	$\lvert x\rvert$	x の絶対値
`fdim(`x`, `y`)`	$\max\{x-y,\, 0\}$	x, y が $x > y$ をみたすとき $x-y$、そうでない数値のとき $+0.0$

積和関数

`fma(`x`, `y`, `z`)`	$xy+z$	x と y の積に z を加えたもの。x`*`y`+`z よりも正確

冪関数

`sqrt(`x`)`	$\sqrt{x}$	x の平方根
`hypot(`x`, `y`)`	$\sqrt{x^2+y^2}$	x の 2 乗と y の 2 乗の和の平方根。`sqrt(`x`*`x`+`y`*`y`)` よりも正確
`cbrt(`x`)`	$\sqrt[3]{x}$	x の立方根
`pow(`x`, `y`)`	x^y	x の y 乗

指数関数

`exp(`x`)`	e^x	e の x 乗
`exp2(`x`)`	2^x	2 の x 乗
`expm1(`x`)`	e^x-1	e の x 乗から 1 を引いた数。`exp(`x`)-1` よりも正確

対数関数

`log(`x`)`	$\log_e x$	x の自然対数
`log1p(`x`)`	$\log_e(1+x)$	x に 1 を加えた数の自然対数。`log(1+`x`)` よりも正確
`log10(`x`)`	$\log_{10} x$	x の 10 を底とする対数
`log2(`x`)`	$\log_2 x$	x の 2 を底とする対数

三角関数

`sin(`x`)`	$\sin x$	x（ラジアン）の正弦
`cos(`x`)`	$\cos x$	x（ラジアン）の余弦
`tan(`x`)`	$\tan x$	x（ラジアン）の正接

逆三角関数

`asin(`x`)`	$\arcsin x$	x の逆正弦（$[-\pi/2, +\pi/2]$ ラジアン）
`acos(`x`)`	$\arccos x$	x の逆余弦（$[0, +\pi]$ ラジアン）
`atan(`x`)`	$\arctan x$	x の逆正接（$[-\pi/2, +\pi/2]$ ラジアン）
`atan2(`y`, `x`)`	$\arctan(y/x)$	y/x の逆正接（$[-\pi, +\pi]$ ラジアン）

双曲線関数

`sinh(`x`)`	$\sinh x$	x の双曲線正弦
`cosh(`x`)`	$\cosh x$	x の双曲線余弦
`tanh(`x`)`	$\tanh x$	x の双曲線正接

逆双曲線関数

`asinh(`x`)`	$\operatorname{arcsinh} x$	x の逆双曲線正弦
`acosh(`x`)`	$\operatorname{arccosh} x$	x の逆双曲線余弦
`atanh(`x`)`	$\operatorname{arctanh} x$	x の逆双曲線正接

ガンマ関数

`tgamma(`x`)`	$\Gamma(x)$	x のガンマ関数
`lgamma(`x`)`	$\log_e\lvert\Gamma(x)\rvert$	x のガンマ関数の絶対値の自然対数

誤差関数

`erf`(x)	$\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$　x の誤差関数
`erfc`(x)	$\frac{2}{\sqrt{\pi}}\int_x^\infty e^{-t^2}\,dt$　x の余誤差関数

■`long double` を受け取り `long double` を返す数学関数および `float` を受け取り `float` を返す数学関数

`double` 版の標準数学関数それぞれに対応して、`long double` 版と `float` 版があります。

- `long double` 版の関数の名前は `double` 版の関数の名前の後ろに `l`（小文字のエル）を付け加えたものです。
- `float` 版の関数の名前は `double` 版の関数の名前の後ろに `f` を付け加えたものです。

たとえば、正弦関数を計算する関数は、次のように三つあります。

	x の型	計算結果の型
`sin`(x)	`double`	`double`
`sinl`(x)	`long double`	`long double`
`sinf`(x)	`float`	`float`

1.5.8 例

■整数の足し算　プログラム 1.19〜1.24 は、整数の足し算をするだけのプログラムを整数の型だけを変えていろいろと書いたものです。

プログラム 1.19

```
#include <stdio.h>

int
main()
{
        int     x, y;
        scanf("%d%d", &x, &y);
        printf("%d + %d = %d\n", x, y, x + y);
        return 0;
}
```

プログラム 1.20

```
#include <stdio.h>

int
main()
{
        long    x, y;
        scanf("%ld%ld", &x, &y);
        printf("%ld + %ld = %ld\n", x, y, x + y);
        return 0;
}
```

プログラム 1.21

```
#include <stdio.h>

int
main()
{
        long long       x, y;
        scanf("%lld%lld", &x, &y);
        printf("%lld + %lld = %lld\n", x, y, x + y);
        return 0;
}
```

プログラム 1.22

```
#include <stdio.h>

int
main()
{
        unsigned        x, y;
        scanf("%u%u", &x, &y);
        printf("%u + %u = %u\n", x, y, x + y);
        return 0;
}
```

プログラム 1.23

```
#include <stdio.h>

int
main()
{
        unsigned long   x, y;
        scanf("%lu%lu", &x, &y);
        printf("%lu + %lu = %lu\n", x, y, x + y);
        return 0;
}
```

プログラム 1.24

```
#include <stdio.h>

int
main()
{
        unsigned long long      x, y;
        scanf("%llu%llu", &x, &y);
        printf("%llu + %llu = %llu\n", x, y, x + y);
        return 0;
}
```

1.6 変数名

変数名の長さ（文字数）に、特に制限はありません。

変数名に使うことができるのは、原則としては以下の文字だけです。

ラテン文字小文字

a b c d e f g h i j k l m n o p q r s t u v w x y z

ラテン文字大文字

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

数字

0 1 2 3 4 5 6 7 8 9

下線

_

ただし、変数名の最初の文字だけは数字であってはいけません。また、以下の語は特別な役割があって、変数名には使えません。

auto	do	goto	return	typedef	_Complex
break	double	if	short	union	_Imaginary
case	else	inline	signed	unsigned	
char	enum	int	sizeof	void	
const	extern	long	static	volatile	
continue	float	register	struct	while	
default	for	restrict	switch	_Bool	

上記以外の文字（たとえば、漢字）を変数名に使えるシステムもありますが、使わないほうが無難です。



第 2 章

制御構造

2.1 分岐と繰り返し

2.1.1 コラッツ予想

正整数 n に対して、次の操作を考えます。

- n が偶数ならば 2 で割る。
- n が奇数ならば 3 を掛けて 1 を足す。

最初に一つの数を決めて、この操作を繰り返します。

たとえば、7 から始めるとこうなります。

$$7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$$

この操作を繰り返すと、最初の数が何であってもいずれは $4 \to 2 \to 1$ の循環に入るというのがコラッツ予想です。有名な未解決問題です。

コラッツ予想の 1 ステップ

コラッツ予想の 1 ステップを実行するのが、プログラム 2.1 です。

プログラム 2.1 コラッツ予想の 1 ステップを実行する

```
#include <stdio.h>

int
main()
{
        int     x;

        scanf("%d", &x);
        if (x % 2 == 0) {
                x = x / 2;
        } else {
                x = x * 3 + 1;
        }
        printf("%d\n", x);
        return 0;
}
```

このプログラムでは if 文を使っています。if 文は、

if（式$_1$）文$_1$ else 文$_2$

の形をしています。まず、式$_1$ の計算を行って、結果が真ならば 文$_1$ を実行し、結果が偽ならば 文$_2$ を実行します。なお、else 文$_2$ の部分は省略できます。

真のとき、あるいは、偽のときに実行したい文が複数のときには、ブロックが使えます。ブロックとは、

{文$_1$ ... 文$_n$}

の形で、いくつかの文を一つの文にまとめるものです。

if 文で本体の文をブロックにするときは、

if（式$_1$）{
    文$_1$
    ⋮
    文$_m$
} else {
    文$'_1$
    ⋮
    文$'_n$
}

の形式で書くと読みやすくなります。ただし、これは人間にとっての読みやすさであって、コンピュータにとっては知ったことではありません。

上のプログラム例では、真のときに実行する文も偽のときに実行する文もそれぞれ一つだけですので、ブロックにしなくてもかまいません。もちろん、ブロックにして困ることもありません。ここでは、スタイルでブロックにしています。

比較演算子として、とりあえず、以下の六つを覚えておいてください。

==	等しい	<	小さい	<=	以下
!=	等しくない	>	大きい	>=	以上

後で出てきますが、条件を組み合わせて複雑な条件を作ることもできます。

!	……でない	&&	……かつ……	||	……または……

たとえば、「x は正の偶数である」は

```
x > 0 && x % 2 == 0
```

と書くことができます。

「x は正の偶数でない」は

```
!(x > 0 && x % 2 == 0)
```

と書くことができます。ド・モルガン則を使うと、これを

プログラム 2.2 コラッツ予想のステップを 1 に達するまで実行する

```
#include <stdio.h>

int
main()
{
        int     x;

        scanf("%d", &x);
        printf("%d\n", x);
        while (x != 1) {
                if (x % 2 == 0) {
                        x = x / 2;
                } else {
                        x = x * 3 + 1;
                }
                printf("%d\n", x);
        }
        return 0;
}
```

プログラム 2.3 コラッツ予想の 1 に達するまでのステップ数を数える

```
#include <stdio.h>

int
main()
{
        int     x;
        int     counter = 0;

        scanf("%d", &x);
        printf("%d\n", x);
        while (x != 1) {
                if (x % 2 == 0) {
                        x = x / 2;
                } else {
                        x = x * 3 + 1;
                }
                counter ++;
        }
        printf("%d\n", counter);
        return 0;
}
```

```
        x <= 0 || x % 2 != 0
```

と書くこともできます。いまどきのコンピュータでは、どちらの書き方をしても、計算時間やメモリ使用量に大差はありません。

コラッツ予想の 1 に達するまで

コラッツ予想のステップを 1 に達するまでのステップをすべて出力するのが、プログラム 2.2 です。

繰り返しを実現する方法の一つが、while 文を使うことです。while 文は

`while` （式$_1$） 文$_1$

の形です。式$_1$ を計算して真である間、文$_1$ を実行します。繰り返しの始めに、毎回、式$_1$ を計算します。

while 文でも、ブロックと組み合わせれば、本体に複数の文を使うことができます。

while 文で本体の文をブロックにするときでも、if 文のときと同じく、

```
while（式1）{
    文1
     ⋮
    文n
}
```

の形式で書くと読みやすくなります。これが人間にとっての読みやすさであってコンピュータにとっては知ったことではないことも、if 文のときと同じです。

コラッツ予想の手順で 1 に達するまでのステップ数を出力するのが、プログラム 2.3 です。プログラム 2.2 と比較すると、わかりやすいでしょう。プログラム 2.2 にはない変数 `counter` が、プログラム 2.3 にはあります。プログラム 2.2 では while 文の中で `printf` しているのが、プログラム 2.3 では while 文の終了後に `printf` しています。

変数 `counter` は、ステップを数えるために使われます。変数が作られたときに 0 に初期化され、while 文の本体が 1 回実行されるごとに 1 増えます。

```
counter ++;
```

が、変数 `counter` の値を 1 増やします。一般に

```
変数 ++;
```

で、変数の値を 1 増やします。

なお、1 減らしたいときは、

```
変数 --;
```

が使えます。

2.1.2 表を作る

繰り返し構文には for 文もあります。for 文は、

```
for（式1; 式2; 式3）文1
```

の形です。

```
{式1; while（式2）{文1 式3;}}
```

とほぼ同じ意味です。

プログラム 2.4 は、for 文を使って表を作る例です。$n = 1, \ldots, 99$ に対して、

$n \quad n^2 \quad n^3$

の書式の行を順に出力します。

プログラム 2.4 n, n^2, n^3 の表を作る

```
#include <stdio.h>

int
main()
{
        int     n;

        for (n = 1; n < 100; n ++) {
                int     n2, n3;
                n2 = n * n;
                n3 = n2 * n;
                printf("%d %d %d\n", n, n2, n3);
        }
        return 0;
}
```

$n^3 = n^2 \times n$ を利用して掛け算の回数を減らしています。

ブロックの中で定義された変数は、そのブロックの中でだけ存在しています。ブロックに入るときに生成され、ブロックから出るときに消滅します。プログラム 2.4 で if 文の本体のブロックの中に

```
                int     n2, n3;
```

があるのは、その例です。この変数定義で作られた変数 n2 と n3 はブロックの中でのみ有効です。for 文の外では存在しません。

プログラム 2.5 掛け算九九の表を作る

```
#include <stdio.h>

int
main()
{
        int     i, j;

        for (i = 1; i <= 9; i ++) {
                for (j = 1; j <= 9; j ++) {
                        printf("%3d", i * j);
                }
                printf("\n");
        }
        return 0;
}
```

出力 2.5.1: プログラム 2.5 の出力

```
  1  2  3  4  5  6  7  8  9
  2  4  6  8 10 12 14 16 18
  3  6  9 12 15 18 21 24 27
  4  8 12 16 20 24 28 32 36
  5 10 15 20 25 30 35 40 45
  6 12 18 24 30 36 42 48 54
  7 14 21 28 35 42 49 56 63
  8 16 24 32 40 48 56 64 72
  9 18 27 36 45 54 63 72 81
```

プログラム 2.5 は、for 文を使って掛け算九九の表を作る例です。for 文の中で for 文を使うのがミソです。`printf()` で横をそろえるために、横幅の指定を行っています。

```
printf("%3d", i * j);
```

にある `%3d` の 3 は、「3 桁で表示してほしい」と指示します。表示する整数が 3 桁にみたないときは左を空白で埋めます。3 桁を越えるときは、しかたないので 3 桁を超えて表示します。



2.1.3 素因数分解

プログラム 2.6 は、与えられた整数を素因数分解します。

プログラム 2.6 素因数分解（遅い）

```
#include <stdio.h>

int
main()
{
        unsigned        n;
        unsigned        k, p;

        scanf("%u", &n);
        printf("%u:", n);
        k = n;
        p = 2;
        while (k > 1) {
                while (k % p == 0) {
                        k /= p;
                        printf(" %u", p);
                }
                p ++;
        }
        printf("\n");
        return 0;
}
```

プログラム 2.6 では、素数判定を行うことなく素因数分解を実現しています。素朴に考えると、素因数分解は与えられた整数を小さな素数から順に割れるだけ割ることで実現します。ところが、割る数として素数だけ選ぶのは面倒です。実は、素数かどうかを気にせずにとにかく小さい数から順に割っていくとことで十分です。なぜなら、合成数で割ろうとしたときには、その素因数ですでに割られていているので、一度も割ることなく次に進むからです。

プログラム 2.6 では、整数を格納する変数を unsigned 型にしています。int 型は負の整数値をとることもできますが、unsigned 型は非負整数のみです。そのかわり、int 型よりも unsigned 型のほうがとれる値の上限が大きくなっています。詳細は 1.5.1 で説明しています。

unsigned 型の値を scanf() で入力するには %u を使います。unsigned 型の値を printf() で出力するには %u を使います。

プログラム 2.6 に出てくる

```
k /= p;
```

は

```
k = k / p;
```

とほぼ同じ意味です。+, -, *, /, % に対応して +=, -=, *=, /=, %= があります。

プログラム 2.6 を改良してみましょう。2 以外の素数はすべて奇数です。そこで、2 で割れるだけ割ったあとは奇数だけで割っていくと、ちょっと高速化できます（プログラム 2.7）。

プログラム 2.7 素因数分解（2 を除く素数は奇数であることを利用して高速化）

```
#include <stdio.h>

int
main()
{
        unsigned        n;
        unsigned        k, p;

        scanf("%u", &n);
        printf("%u:", n);
        k = n;
        p = 2;
        while (k % p == 0) {
                k /= p;
                printf(" %u", p);
        }
        p = 3;
        while (k > 1) {
                while (k % p == 0) {
                        k /= p;
                        printf(" %u", p);
                }
                p += 2;
        }
        printf("\n");
        return 0;
}
```

さらに改良してみましょう。2 と 3 を除く素数は 6 で割った余りが 1 か 5 です。この事実を利用すると、も

うちょっと高速化できます（プログラム 2.8）。

プログラム 2.8 素因数分解（2 と 3 を除く素数は 6 で割った余りが 1 か 5 であることを利用して高速化）

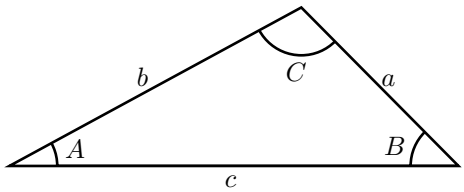
```
#include <stdio.h>

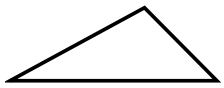
int
main()
{
        unsigned        n;
        unsigned        k, p, d;

        scanf("%u", &n);
        printf("%u:", n);
        k = n;
        p = 2;
        while (k % p == 0) {
                k /= p;
                printf(" %u", p);
        }
        p = 3;
        while (k % p == 0) {
                k /= p;
                printf(" %u", p);
        }
        p = 5;
        d = 2;
        while (k > 1) {
                while (k % p == 0) {
                        k /= p;
                        printf(" %u", p);
                }
                p += d;
                d = 6 - d;
        }
        printf("\n");
        return 0;
}
```

2.1.4 三角形の面積

三角形の面積を求める方法はいくつかあります。以下は、三つの辺の長さ a, b, c から計算するもの（三辺）、二つの辺の長さ a, b とそれらが挟む角の大きさ C から計算するもの（二辺挟角）、一つの辺の長さ a とその両端の角の大きさ B, C から計算するもの（一辺両端角）の三つです。





$$S = \sqrt{s(s-a)(s-b)(s-c)} \qquad \text{ただし、} s = \frac{a+b+c}{2} \qquad \text{(三辺)}$$



$$S = \frac{1}{2}ab\sin C \qquad \text{(二辺挟角)}$$



$$S = \frac{a^2 \sin B \sin C}{2\sin(B+C)} \qquad \text{(一辺両端角)}$$

一つのプログラムで、この三つのどれでも計算できるものを書いてみます。どれであるかを区別するために、ユーザには最初に 3 か 2 か 1 を入力してもらいます。つまり、入力は以下のいずれかの形式であるとします。

3 a b c

2 a b C

1 a B C

プログラム 2.9 三角形の面積

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

int
main()
{
        int     t;
        double  a, b, c, s;
        double  B, C;
        double  S;

        scanf("%d", &t);
        switch (t) {
        case 3:
                scanf("%lf%lf%lf", &a, &b, &c);                          //   △
                s = (a + b + c) / 2;                                     //   △
                S = sqrt(s * (s - a) * (s - b) * (s - c));               //   △
                break;                                                   //   △
        case 2:
                scanf("%lf%lf%lf", &a, &b, &C);                          //   ♢
                S = a * b * sin(C) / 2;                                  //   ♢
                break;                                                   //   ♢
        case 1:
                scanf("%lf%lf%lf", &a, &B, &C);                          //   ♡
                S = a * a * sin(B) * sin(C) / (2 * sin(B + C));          //   ♡
                break;                                                   //   ♡
        default:
                return 1;                                                //   ○
        }
        printf("%f\n", S);
        return 0;
}
```

プログラム 2.10 三角形の面積（その 2）

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

int
main()
{
        int     t;
        double  a, b, c, s;
        double  B, C;
        double  S;

        scanf("%d", &t);
        switch (t) {
        case 3:
        case 0:
                scanf("%lf%lf%lf", &a, &b, &c);                          //   △
                s = (a + b + c) / 2;                                     //   △
                S = sqrt(s * (s - a) * (s - b) * (s - c));               //   △
                break;                                                   //   △
        case 2:
        case -1:
                scanf("%lf%lf%lf", &a, &b, &C);                          //   ♢
                S = a * b * sin(C) / 2;                                  //   ♢
                break;                                                   //   ♢
        case 1:
        case -2:
                scanf("%lf%lf%lf", &a, &B, &C);                          //   ♡
                S = a * a * sin(B) * sin(C) / (2 * sin(B + C));          //   ♡
                break;                                                   //   ♡
        default:
                return 1;                                                //   ○
        }
        printf("%f\n", S);
        return 0;
}
```

switch 文は、通常はブロックと組み合わせて、

`switch` （式） {文$_1$ ...文$_n$}

の形で使います。ただし、ブロックの中の文には case ラベルか default ラベルが 1 個以上ついてもかまいません。case ラベルは

`case` 数値:

の形です。default ラベルは

`default:`

の形です。

switch 文は、まず、式を計算します。計算結果がどれかの case ラベルの数値に一致すれば、その case ラベルの位置にとんでいき、そこから順に文を実行します。計算結果がどの case ラベルの数値にも一致せず、default ラベルがあれば、その default ラベルの位置にとんでいき、そこから順に文を実行します。計算結果がどの case ラベルの数値にも一致せず、default ラベルがなければ、どの文も実行しません。

たいていの場合、switch 文の中で途中で抜けるために break 文が必要になります。case ラベルや default ラベルに達しても、それだけではそこで switch 文の実行が終了することはありません。そのまま、ラベルの次の文に進んでしまいます。それを防ぐために、break 文を使って switch 文から抜けさせる必要があるのです。

プログラム 2.9 の switch 文では、t の値が 3 のときは △ の部分が実行され、2 のときは ♢ の部分だけが実行され、1 のときは ♡ の部分だけが実行され、それ以外の値のときは ○ の部分だけが実行されます。break 文を忘れるとそうはなりません。たとえば、△ の部分の最後の break 文を書き忘れると、t の値が 3 のときに △ の部分のあとで続けて ♢ の部分も実行されてしまいます。

何もなければ次の case ラベルまたは default ラベルを超えてさらに実行を続けることを利用すれば、複数の選択肢で同じ動作をすることを簡潔に書くこともできます。プログラム 2.10 はその例です。これは、入力の種類を表す数として 1, 2, 3 に加えて −2, −1, 0 も許すように、プログラム 2.9 を書き換えたものです。プログラム 2.10 の switch 文では、t の値が 3 のときと 0 のときは △ の部分が実行され、2 のときと −1 のときは ♢ の部分が実行され、1 のときと −2 のときは ♡ の部分が実行され、それ以外の値のときは ○ の部分が実行されます。



2.2 関数

関数は、同じようなことを繰り返し書かずにすむようにする仕組みです。数学でいう「関数」とは近いですが違います。

2.2.1 三角形の辺の長さ

プログラム 2.11 は、三角形の三頂点の座標を入力して三辺の長さを出力します。三角形の辺は頂点を端点とする線分です。頂点の座標を線分の長さの公式に放り込むだけで、辺の長さは計算できます。そこで、プログラム 2.11 では、端点の座標から線分の長さを計算する関数を作り、それを三回呼び出すことで三角形の三辺の長さを計算しています。

プログラム 2.11 三角形の頂点の座標を入力して辺の長さを出力する

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

double  segment_length(double, double, double, double);                    //   ♡

double                                                                     //   ♢
segment_length(double x1, double y1, double x2, double y2)                //   ♢
{                                                                          //   ♢
        return hypot(x1 - x2, y1 - y2);                                    //   ♢
}                                                                          //   ♢

int
main()
{
        double  xa, ya, xb, yb, xc, yc;
        double  a, b, c;

        scanf("%lf%lf%lf%lf%lf%lf", &xa, &ya, &xb, &yb, &xc, &yc);
                // 頂点 A の x 座標、頂点 A の y 座標、頂点 B の x 座標、頂点 B の y 座標、
                // 頂点 C の x 座標、頂点 C の y 座標をこの順に入力
        a = segment_length(xb, yb, xc, yc);
                // 辺 BC の長さを計算
        b = segment_length(xc, yc, xa, ya);
                // 辺 CA の長さを計算
        c = segment_length(xa, ya, xb, yb);
                // 辺 AB の長さを計算
        printf("%12.8f %12.8f %12.8f\n", a, b, c);
                // 辺 BC の長さ、辺 CA の長さ、辺 AB の長さをこの順に出力
        return 0;
}
```

関数 segment_length が線分の長さを計算します。♡ の部分は関数 segment_length の型を宣言しています。**プロトタイプ宣言**と呼ばれます。♢ の部分は関数 segment_length の実体を作ります。**関数定義**と呼ばれます。下線＿＿の引かれた三カ所では、関数 segment_length を呼び出します。詳しくいうと、関数に制御を渡してその本体を実行させます。**関数呼び出し**と呼ばれます。

プロトタイプ宣言は関数の型を知らせるだけで、関数の実体はここでは作られません。まずは、こんな形をしたプロトタイプ宣言が使えるようになってください。

返値の型　関数名(仮引数の型の並び);

プログラム 2.11 の ♡ の行はプロトタイプ宣言です。以下のことを宣言しています。

- 関数名が segment_length である関数がある。この関数は以下の通りである。
  - 返値は double 型。
  - 引数を四つ取る。
    - 第一引数は double 型。
    - 第二引数は double 型。
    - 第三引数は double 型。
    - 第四引数は double 型。

関数定義は、関数の実体を作ります。まずは、こんな形をした関数定義が書けるようになってください。

返値の型
関数名(仮引数の並び)
{
　　本体
}

返値の型は、その関数を呼び出した関数呼び出しの式としての型になります。仮引数の並びは、仮引数の定義をコンマで区切って並べたものです。仮引数は特殊な変数だと考えて問題ありません。関数の中で定義される変数とほぼ同じものだが、初期値が呼び出し側に決められることだけが違うと考えておいてください。

本体は、関数が呼ばれたときに実行される文や変数定義などの並びです。

関数定義の本体で変数が定義されたとき、その変数はその関数の中でのみ有効です。同じ名前の変数が違う関数の中で定義されていても、それらは無関係です。

関数 segment_length の関数定義では、返値の型が double で、関数名が segment_length で、仮引数の並びが

```
double x1, double y1, double x2, double y2
```

で、本体が

```
    return hypot(x1 - x2, y1 - y2);
```

です。ここで関数 segment_length の実体を作っています。線分の二つの端点の座標が、それぞれ、(x_1, y_1) と (x_2, y_2) のとき、長さは $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ です。それをそのまま実装しています。

関数が定義されていると、式の中で関数呼び出しの形で使うことができます。もっとも良く使われるのは、こんな形をした関数呼び出しです。

関数名（式$_1$，...，式$_n$）

関数呼び出しの値の計算では、まず、式$_1$, ..., 式$_n$ の値が計算されます。ただし、計算する順序は決まっていません。次に、仮引数が作られ、呼び出した側でさきほど得られた値で初期化されます。それから、関数の本体が実行されます。

関数の本体の中で変数定義があると、そこで変数が作られます。作られた変数は、関数の実行の終了時には破棄されます。

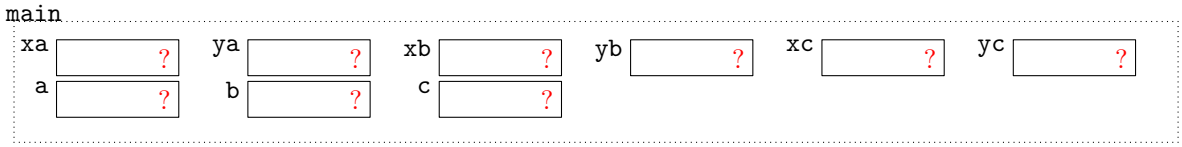
関数の本体の中で

```
return 式1;
```

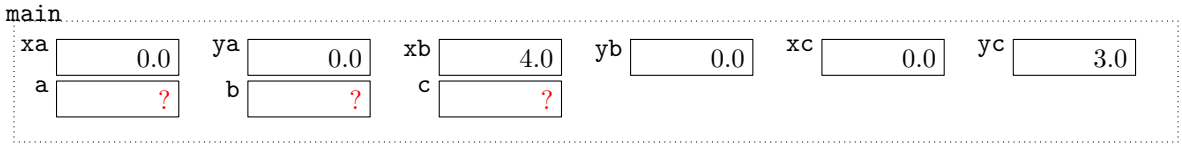
の形の return 文が実行されると、そこで関数の実行が終了します。関数を呼び出した側に制御が戻り、式$_1$ の計算結果が、関数呼び出しの値となります。

プログラム 2.11 を実行した際の変数の作られ方やその値の変遷を図示すると、以下のようになります。

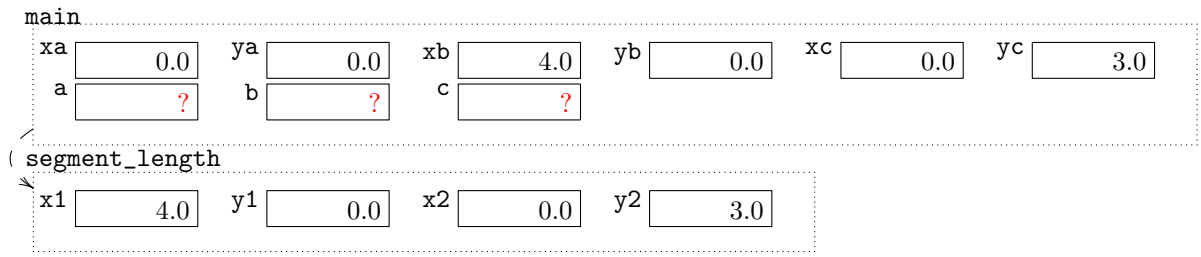
- main 実行開始直後



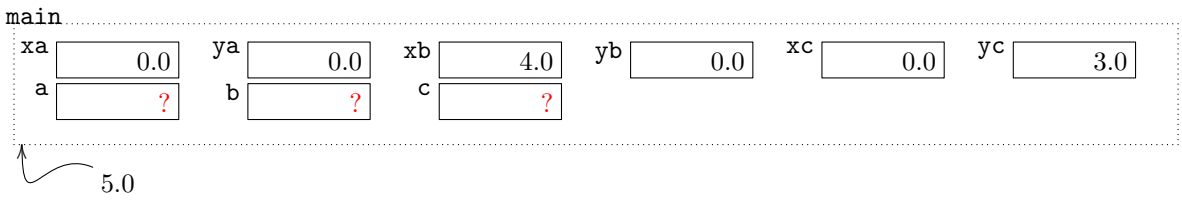
- scanf() から戻ってきた直後



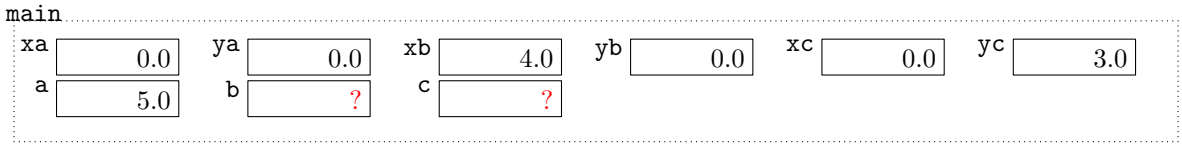
- 1 回目の segment_length 呼び出し直後



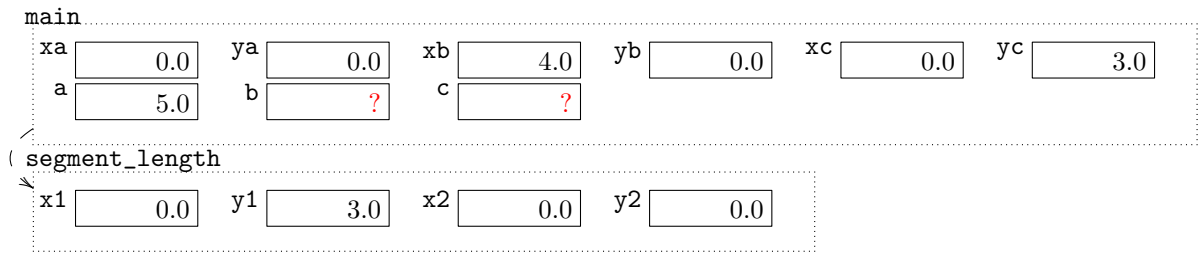
- 1 回目の segment_length 呼び出しからの返値



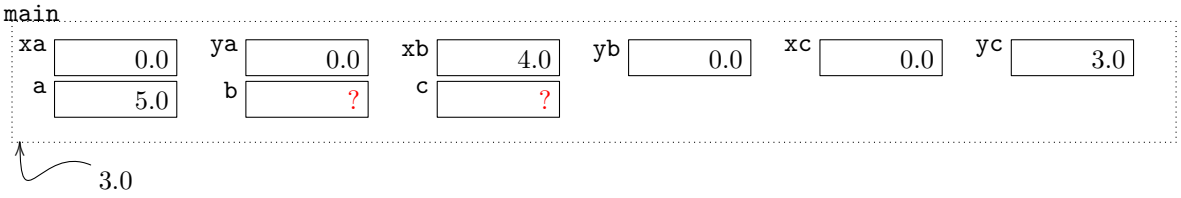
- 返値を a に代入した直後



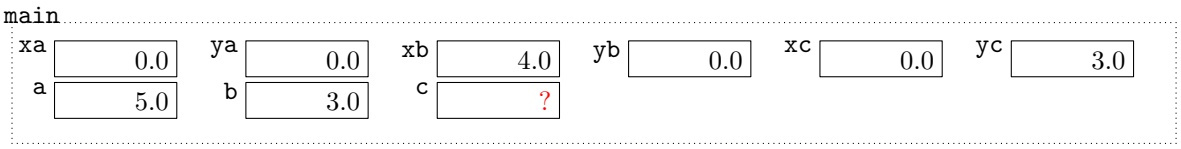
- 2 回目の segment_length 呼び出し直後



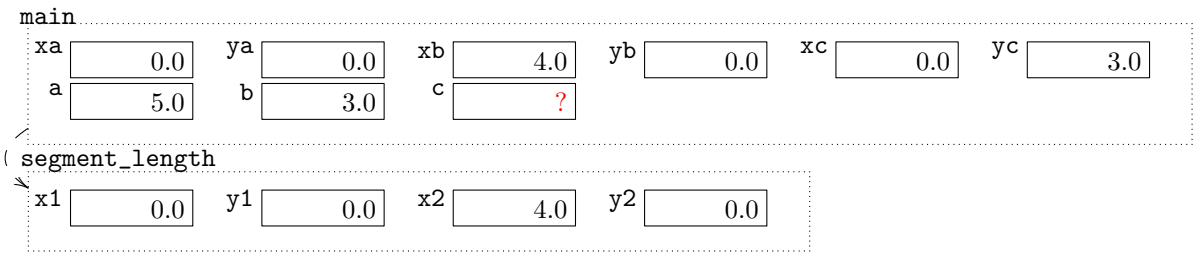
- 2 回目の segment_length 呼び出しからの返値



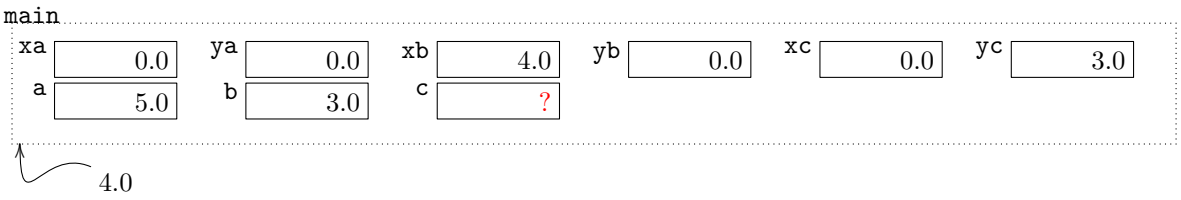
- 返値を b に代入した直後



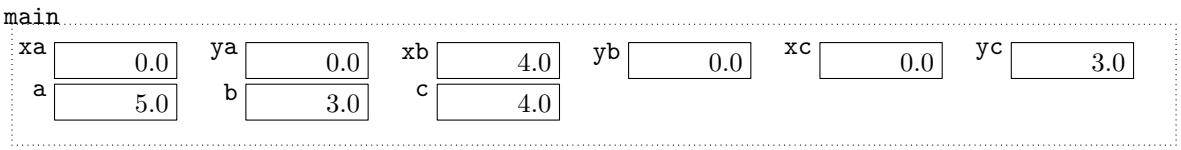
- 3 回目の segment_length 呼び出し直後



- 3 回目の segment_length 呼び出しからの返値



- 返値を c に代入した直後



関数のプロトタイプ宣言と関数定義と関数呼び出しの順序

関数のプロトタイプ宣言の必要性について、具体例で考えてみましょう。プログラム 2.11 は正しいプログラムです。プログラム 2.11 から ♡ で示すプロトタイプ宣言を削ったプログラム 2.12 も、正しいプログラムです。プログラム 2.11 の ♢ で示す関数定義を後ろに動かしてプログラム 2.13 に書き換えても、やはり正しいプログラムです。ところが、その両方を行うと、つまり、プログラム 2.13 から ♡ で示すプロトタイプ宣言を削ると、正しいプログラムではなくなります。なぜそうなるかを説明します。

関数のプロトタイプ宣言と関数定義がそれぞれ何をするかをまとめると、以下のようになります。

- 関数のプロトタイプ宣言は、関数の型を宣言するのみ。
- 関数定義は、関数の実体を作ると同時に関数の型を宣言する。

関数の型の宣言については以下のような制限があります。

- 関数呼び出しの出現よりもプログラムの字面上で前で、その関数の型が宣言されている必要がある。
- 関数の型の宣言は複数回行ってよい。ただし、それらが矛盾してはならない。

プログラム 2.12 三角形の頂点の座標を入力して辺の長さを出力する（関数 segment_length のプロトタイプ宣言を省略）

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

double                                                                  //  ♢
segment_length(double x1, double y1, double x2, double y2)             //  ♢
{                                                                       //  ♢
        return hypot(x1 - x2, y1 - y2);                                 //  ♢
}                                                                       //  ♢

int
main()
{
        double  xa, ya, xb, yb, xc, yc;
        double  a, b, c;

        scanf("%lf%lf%lf%lf%lf%lf", &xa, &ya, &xb, &yb, &xc, &yc);
                // 頂点 A の x 座標、頂点 A の y 座標、頂点 B の x 座標、頂点 B の y 座標、
                // 頂点 C の x 座標、頂点 C の y 座標をこの順に入力
        a = segment_length(xb, yb, xc, yc);
                // 辺 BC の長さを計算
        b = segment_length(xc, yc, xa, ya);
                // 辺 CA の長さを計算
        c = segment_length(xa, ya, xb, yb);
                // 辺 AB の長さを計算
        printf("%12.8f %12.8f %12.8f\n", a, b, c);
                // 辺 BC の長さ、辺 CA の長さ、辺 AB の長さをこの順に出力
        return 0;
}
```

プログラム 2.13 三角形の頂点の座標を入力して辺の長さを出力する（関数 segment_length の定義を後ろに移動）

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

double  segment_length(double, double, double, double);                    //   ♡

int
main()
{
        double  xa, ya, xb, yb, xc, yc;
        double  a, b, c;

        scanf("%lf%lf%lf%lf%lf%lf", &xa, &ya, &xb, &yb, &xc, &yc);
                // 頂点 A の x 座標、頂点 A の y 座標、頂点 B の x 座標、頂点 B の y 座標、
                // 頂点 C の x 座標、頂点 C の y 座標をこの順に入力
        a = segment_length(xb, yb, xc, yc);
                // 辺 BC の長さを計算
        b = segment_length(xc, yc, xa, ya);
                // 辺 CA の長さを計算
        c = segment_length(xa, ya, xb, yb);
                // 辺 AB の長さを計算
        printf("%12.8f %12.8f %12.8f\n", a, b, c);
                // 辺 BC の長さ、辺 CA の長さ、辺 AB の長さをこの順に出力
        return 0;
}

double                                                                     //   ♢
segment_length(double x1, double y1, double x2, double y2)                //   ♢
{                                                                          //   ♢
        return hypot(x1 - x2, y1 - y2);                                    //   ♢
}                                                                          //   ♢
```

関数のプロトタイプ宣言と関数定義の出現についての制約としてまとめて書くと、次のようになります。

- 関数呼び出しの出現よりもプログラムの字面上で前に、その関数のプロトタイプ宣言か関数定義がある必要がある。
- 関数定義はちょうど一つ必要である。
- プロトタイプ宣言は必須ではない。また、矛盾しなければ一つの関数のプロトタイプ宣言が複数あっても良い。

下線＿＿での関数 segment_length の呼び出しに対して、それよりも字面上で前での対応する関数の型の宣言は、それぞれ、次のようになります。

1. [プログラム 2.11] ♡ と ♢ の 2 回、対応する関数の型が宣言されている。宣言は矛盾していない。
2. [プログラム 2.12] ♢ の 1 回、対応する関数の型が宣言されている。
3. [プログラム 2.13] ♡ の 1 回、対応する関数の型が宣言されている。
4. [プログラム 2.13 から ♡ を削除] 字面上で前では、対応する関数の型は宣言されていない。

したがって、1 と 2 と 3 は正しいプログラムですが、4 は正しいプログラムではありません。

言語仕様上の制約を満たした上で、関数定義の順序の自由さを享受したければ、次の方針をお勧めします。

- ファイルの最初のほうに主な関数のプロトタイプ宣言をまとめて書く。
- 必要なプロトタイプ宣言をすべて並べた部分より後ろで、プログラマにとってわかりやすい順序で関数定義を並べる



2.2.2 最大公約数

この節では、三つの非負整数の最大公約数を計算するプログラムを四つ例示します。プログラム 2.14（47 ページ）とプログラム 2.15（50 ページ）とプログラム 2.16（51 ページ）とプログラム 2.17（52 ページ）です。どれも、二つの非負整数の最大公約数を計算する関数 `gcd()` を定義しています。この関数は、第一引数の値と第二引数の値の最大公約数を計算し、計算結果を関数の返値として返します。それを二回呼び出すことで三つの非負整数の最大公約数を計算します[*1]。

ユークリッドの互除法にはいくつかの変種があり、四つのプログラムはそれによって二種類に分けることができます。プログラム 2.14 とプログラム 2.15 とプログラム 2.16 では、関数 `gcd()` は剰余演算を用いる通常のユークリッドの互除法[*2]を使って最大公約数を計算します。プログラム 2.17 では、剰余演算を用いずに減

[*1] 二つの整数の最大公約数を計算することができれば、それを使って三つの整数の最大公約数を

$$\gcd(x, y, z) = \gcd(x, \gcd(y, z))$$

で計算できます。

[*2] 最大公約数には以下の二つの性質があります。

- a が非負整数で b が正整数であり、a を b で割った余りが r のとき、$\gcd(a, b) = \gcd(b, r)$ が成り立つ。
- 任意の整数 a について $\gcd(a, 0) = a$ が成り立つ。

ユークリッドの互除法は、これらの性質をうまく利用して効率よく最大公約数を求めるアルゴリズムの一つです。たとえば、60 と 42 の最大公約数は、

$$\gcd(60, 42) = \gcd(42, 18) = \gcd(18, 6)) = \gcd(6, 0) = 6$$

と、計算できます。

なお、$\gcd(x, 0) = \gcd(0, x) = x$ ですが、このアルゴリズムはその場合も正しい結果を返します。

算と 1/2 倍と 2 倍だけで計算する変種[*3]を使って計算します。

プログラム 2.14 とプログラム 2.15 の違いは、二つの整数の最大公約数をユークリッドの互除法によって計算する関数 gcd() の実装方法です。プログラム 2.14 では、関数 gcd() の中から関数 gcd() 自身を呼び出しています。プログラム 2.15 では、関数 gcd() の中で while 文によりループを実装しています。

プログラム 2.15 の関数 gcd のローカル変数 z を while 文の本体に閉じ込めるように書き換えたのが、プログラム 2.16 です。プログラム 2.15 では、ローカル変数 z は変数定義の後ろ、関数 gcd の定義本体の最後まで存在し続けますが、プログラム 2.16 では、while 文の本体の中でのみ存在し、制御がその外に移ると存在しなくなります。

プログラム 2.14 三つの整数の最大公約数を計算するプログラム（再帰呼び出し）

```
#include <stdio.h>

unsigned        gcd(unsigned, unsigned);

unsigned
gcd(unsigned x, unsigned y)
{
        unsigned        g;
        if (y != 0) {
                g = gcd(y, x % y);
        } else {
                g = x;
        }
        return g;
}

int
main()
{
        unsigned        k, m, n, g;
        scanf("%u%u%u", &k, &m, &n);
        g = gcd(k, gcd(m, n));
        printf("%u\n", g);
        return 0;
}
```

[*3] 最大公約数には以下の性質があります。

- 任意の奇数 m, n と任意の非負整数 u, v について、$\gcd(2^u m, 2^v n) = 2^{\min\{u,v\}} \gcd(m, n)$ が成り立つ。
- 任意の整数 a と b について、$\gcd(a, b) = \gcd(a - b,\ b) = \gcd(a, b - a)$ が成り立つ。
- 任意の整数 a について、$\gcd(a, a) = a$ が成り立つ。

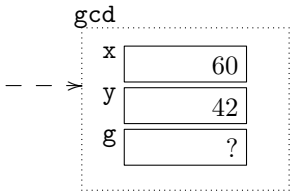
これを巧妙に利用すると、引き算と 1/2 倍と 2 倍だけを用いて最大公約数を計算する、ユークリッドの互除法の変種のアルゴリズムが得られます。

$$\gcd(60, 42) = 2^1 \gcd(15, 21) = 2^1 \gcd(15, 6) = 2^1 \gcd(15, 3) = 2^1 \gcd(12, 3) = 2^1 \gcd(3, 3) = 2 \cdot 3$$

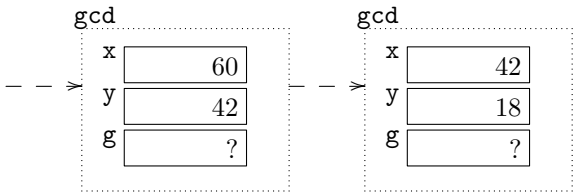
なお、$\gcd(x, 0) = \gcd(0, x) = x$ については、このアルゴリズムでは特別扱いが必要になります。

プログラム 2.14 での関数 gcd() の呼び出しを追いかけてみると、以下のようになります。

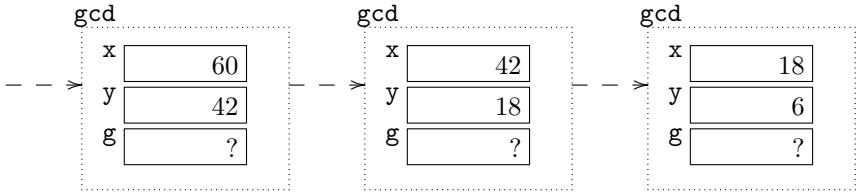
- 1 段目の呼び出し



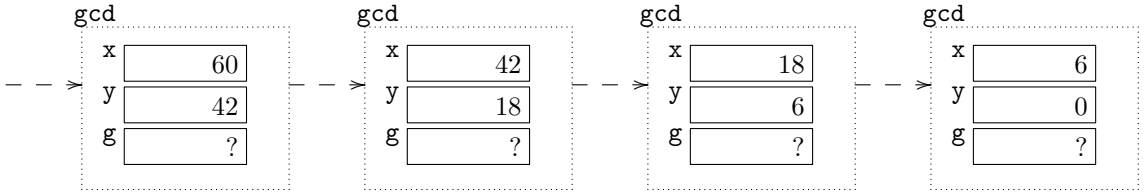
- 2 段目の呼び出し



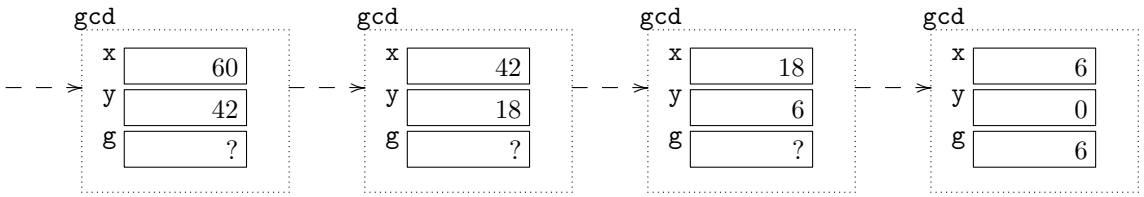
- 3 段目の呼び出し



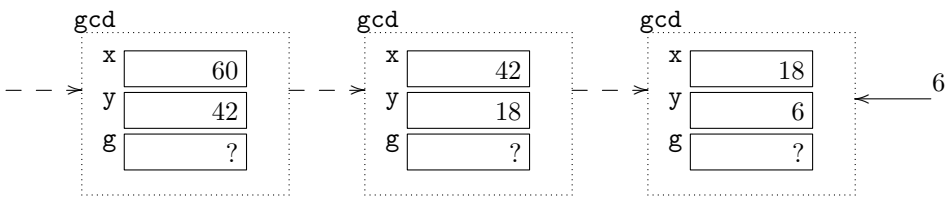
- 4 段目の呼び出し



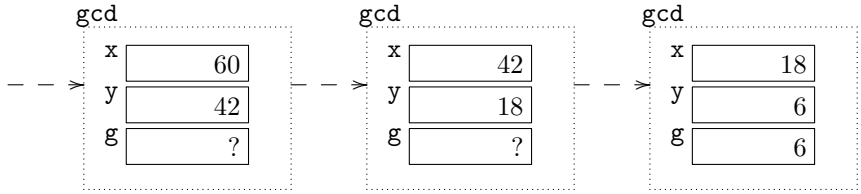
- 4 段目から戻る直前



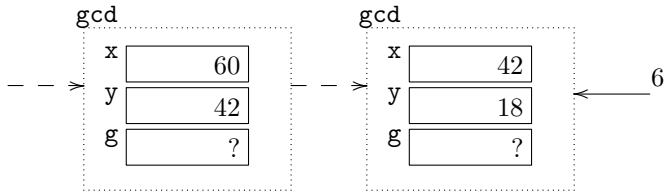
- 4 段目からの戻り



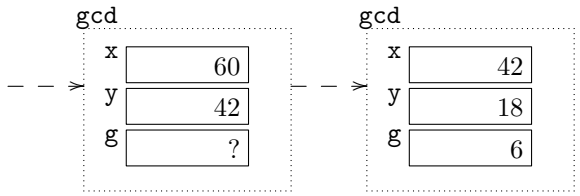
- 3 段目から戻る直前



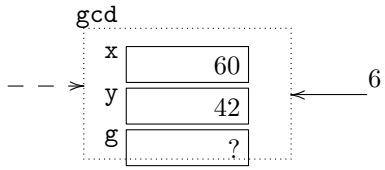
- 3 段目からの戻り



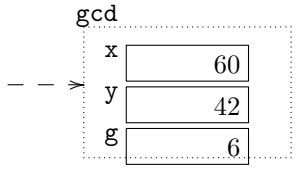
- 2 段目から戻る直前



- 2 段目からの戻り



- 1 段目から戻る直前



- 1 段目からの戻り



プログラム 2.15 三つの整数の最大公約数を計算するプログラム（ループ、変数 z が関数にローカル）

```
#include <stdio.h>

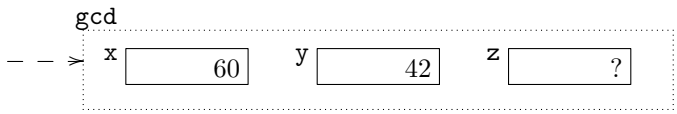
unsigned        gcd(unsigned, unsigned);

unsigned
gcd(unsigned x, unsigned y)
{
        unsigned        z;
        while (y != 0) {
                z = x % y;
                x = y;
                y = z;
        }
        return x;
}

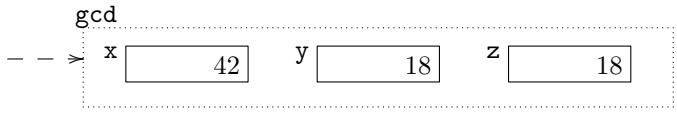
int
main()
{
        unsigned        k, m, n, g;
        scanf("%u%u%u", &k, &m, &n);
        g = gcd(k, gcd(m, n));
        printf("%u\n", g);
        return 0;
}
```

プログラム 2.15 での関数 gcd() 内の while ループの実行を追いかけてみると、以下のようになります。

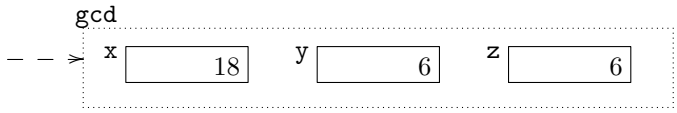
- ループに入る直前



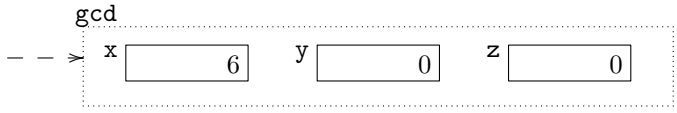
- ループ 1 周目最後



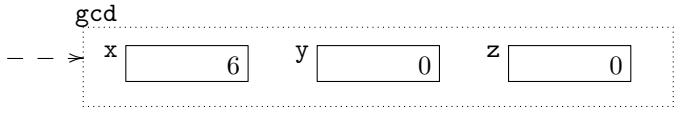
- ループ 2 周目最後



- ループ 3 周目最後



- ループを抜けた直後



プログラム 2.16 三つの整数の最大公約数を計算するプログラム（ループ、変数 z がブロックにローカル）

```
#include <stdio.h>

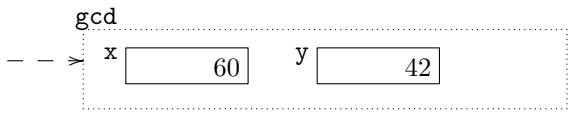
unsigned        gcd(unsigned, unsigned);

unsigned
gcd(unsigned x, unsigned y)
{
        while (y != 0) {
                unsigned        z = x % y;
                x = y;
                y = z;
        }
        return x;
}

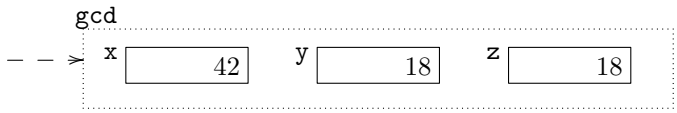
int
main()
{
        unsigned        k, m, n, g;
        scanf("%u%u%u", &k, &m, &n);
        g = gcd(k, gcd(m, n));
        printf("%u\n", g);
        return 0;
}
```

プログラム 2.16 での関数 gcd() 内の while ループの実行を追いかけてみると、以下のようになります。

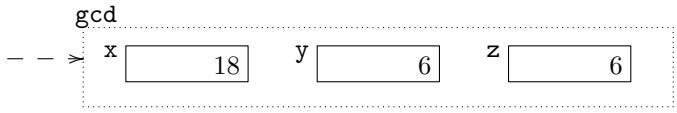
- ループに入る直前



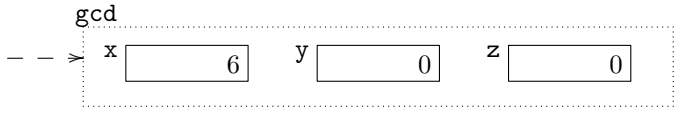
- ループ 1 周目最後



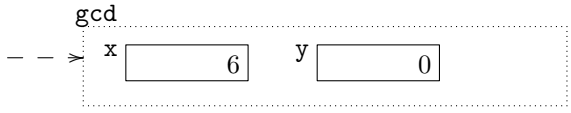
- ループ 2 周目最後



- ループ 3 周目最後



- ループを抜けた直後



プログラム 2.17 三つの整数の最大公約数を計算するプログラム（剰余演算なし）

```
#include <stdio.h>

unsigned        gcd(unsigned, unsigned);

unsigned
gcd(unsigned x, unsigned y)
{
        if (x == 0)
                return y;
        if (y == 0)
                return x;
        int     u = 0;
        while (x % 2 == 0 && y % 2 == 0) {
                x /= 2;
                y /= 2;
                u ++;
        }
        while (x % 2 == 0) {
                x /= 2;
        }
        while (y % 2 == 0) {
                y /= 2;
        }
        while (x != y) {
                if (x < y) {
                        y = (y - x) / 2;
                        while (y % 2 == 0) {
                                y /= 2;
                        }
                } else {
                        x = (x - y) / 2;
                        while (x % 2 == 0) {
                                x /= 2;
                        }
                }
        }
        while (u > 0) {
                u --;
                x *= 2;
        }
        return x;
}

int
main()
{
        unsigned        k, m, n, g;
        scanf("%u%u%u", &k, &m, &n);
        g = gcd(k, gcd(m, n));
        printf("%u\n", g);
        return 0;
}
```

2.2.3 ゴールドバッハ予想

「4 以上の偶数は二つの素数の和として表すことができる」をゴールドバッハ予想といいます。有名な未解決問題です。4 以上の偶数すべてについて考えると未解決問題ですが、4 以上の偶数が具体的に一つ与えられたときにそれについて成り立つかどうかを確認するのは単なる計算です。ついでなので、二つの素数の和として表せるかどうかだけでなく、表し方すべてを求めるプログラムを書いてみましょう。

プログラム 2.18 は、入力された整数の素数の和としての表し方をすべて表示します。ちょっと遅いプログラムです。

このプログラムは関数を定義して使っています。関数 is_prime は与えられた整数が素数かそうでないかを判定する関数です[*4]。関数 print_pair は二つの整数を間に + をはさんで出力する関数です。

プログラム 2.18 入力された整数の二つ素数の和としての表し方をすべて表示する（ちょっと遅い）

```
#include <stdio.h>

int     is_prime(int);                                          //   ♡
void    print_pair(int, int);                                   //   ♡

int                                                             //   ♢
is_prime(int x)                                                 //   ♢
{                                                               //   ♢
        int     i;                                              //  ⋆♢
                                                                //  ⋆♢
        if (x < 2)                                              //  ⋆♢
                return 0;                                       //  ⋆♢
        for (i = 2; i * i <= x; i ++) {                         //  ⋆♢
                if (x % i == 0)                                 //  ⋆♢
                        return 0;                               //  ⋆♢
        }                                                       //  ⋆♢
        return 1;                                               //  ⋆♢
}                                                               //   ♢

void
print_pair(int x, int y)
{
        printf("%d + %d\n", x, y);
}

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

        scanf("%d", &n);
        for (i = 2, j = n - 2; i <= j; i ++, j --) {
                if (is_prime(i) && is_prime(j)) {               //   †
                        print_pair(i, j);
                }
        }
        return 0;
}
```

[*4] 素数とは、2 より大きな自然数で、1 とその数自身だけが約数であるもののことです。ところで、正整数 x が素数であってもなく

マーク ♡ をつけた 2 行は、いずれもプロトタイプ宣言と呼ばれるもので、関数の型を宣言しています。プロトタイプ宣言は関数の型を知らせるだけで、関数の実体はここでは作られません。ここでは、以下のことを示しています。

♡ `int is_prime(int);`
- 返値は `int` 型。
- 引数を一つ取る。第一引数は `int` 型。

♡ `void print_pair(int, int);`
- 値を返さない。
- 引数を二つ取る。第一引数は `int` 型。第二引数は `int` 型。

典型的なプロトタイプ宣言は、一般的に

型 関数名 (型の並び);

の形をしています。

マーク ♢ をつけた 13 行は、関数 `is_prime` の定義です。そのうち、さらに ⋆ をつけて ⋆♢ になっている 9 行が関数定義の本体です。

典型的な関数定義は、一般的に

```
型
関数名 (仮引数の並び)
{
    本体
}
```

の形をしています。「本体」と書いている部分が関数定義の本体で、ここに処理の本体を書きます。

関数定義の本体に出現する

`return` 式$_1$`;`

は、return 文です。そこで関数の実行を終了し、式$_1$ の計算結果を関数の返値として返します。

関数の中で定義された変数はその関数の中でだけ有効です。同じ名前の変数が他の関数の中で定義されていても無関係であることに、注意しましょう。たとえば、プログラム 2.18 の関数 `is_prime` と関数 `main` では、どちらもその中で名前が `i` である変数を定義していますが、この二つの変数はたまたま名前が同じなだけで、何の関係もない別々の変数です。

関数を使うことを「関数の呼び出し」といいます。マーク † をつけた行にある `is_prime(i)` も `is_prime(j)` も関数の呼び出しです。

関数が呼び出されると、まず、引数 (括弧内の式) が計算され、その値を渡して関数本体に制御が移ります。

ても、1 と x は x の約数です。したがって、$2, 3, \ldots, n-1$ の中に x が約数がなければ x は素数であり、あれば素数でないことがわかります。

ところが、$2, 3, \ldots, n-1$ の全部を調べる必要はありません。というのは、x に $\sqrt{x}$ よりも大きな約数があれば、$\sqrt{x}$ よりも小さな約数もあるからです。i が $\sqrt{x}$ よりも大きな x の約数ならば、ある整数 j が存在して $x = ij$ が成り立ちます。この j は $\sqrt{x}$ よりも小さな x の約数です。$2, 3, \ldots, \lfloor\sqrt{x}\rfloor$ の中に x の約数があるかを調べるだけで十分です。

プログラム 2.19 入力された整数の二つ素数の和としての表し方をすべて表示する（2 を除く素数は奇数であることを利用して高速化）

```
#include <stdio.h>

int     is_prime(int);
void    print_pair(int, int);

int
is_prime(int x)
{
        int     i;

        if (x <= 2)
                return x == 2;
        if (x % 2 == 0)
                return 0;
        for (i = 3; i * i <= x; i += 2) {
                if (x % i == 0)
                        return 0;
        }
        return 1;
}

void
print_pair(int x, int y)
{
        printf("%d + %d\n", x, y);
}

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

        scanf("%d", &n);
        for (i = 2, j = n - 2; i <= j; i ++, j --) {
                if (is_prime(i) && is_prime(j)) {
                        print_pair(i, j);
                }
        }
        return 0;
}
```

関数本体で return 文が実行されると、関数の実行が終了し、関数を呼び出したところに制御を戻し、return 文の式を計算した値が関数呼び出しの値になります。return 文を実行することなく関数本体の最後に到達したときも、そこで関数本体の実行が終了します。

プログラム 2.18 を高速化してみましょう。プログラム 2.19 は、2 を除く素数は奇数であることを利用して高速化しています[*5]。プログラム 2.20 は、2 と 3 を除く素数は 6 で割った余りが 1 か 5 であることを利用して高速化しています[*6]。

[*5] 整数 x が奇数ならば、x の約数に偶数はありません。この事実を利用して、3 以上の奇数については奇数でだけ試し割りすることで、試し割りの回数を減らして高速化しています。

[*6] 整数 x が 2 でも 3 でも割り切れなければ、x の約数に 2 の倍数も 3 の倍数もありません。2 の倍数でも 3 の倍数でもないことは 6 で割った余りが 1 か 5 であることと同値です。この事実を利用して、2 の倍数でも 3 の倍数でもない 5 以上の自然数に対しては 6 で割った余りが 1 か 5 である自然数でだけ試し割りすることで、試し割りの回数を減らして高速化しています。

プログラム 2.20 入力された整数の二つ素数の和としての表し方をすべて表示する（2 と 3 を除く素数は 6 で割った余りが 1 か 5 であることを利用して高速化）

```
#include <stdio.h>

int     is_prime(int);
void    print_pair(int, int);

int
is_prime(int x)
{
        int     i, d;

        if (x <= 3)
                return x >= 2;
        if (x % 2 == 0 || x % 3 == 0)
                return 0;
        for (i = 5, d = 2; i * i <= x; i += d, d = 6 - d) {
                if (x % i == 0)
                        return 0;
        }
        return 1;
}

void
print_pair(int x, int y)
{
        printf("%d + %d\n", x, y);
}

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

        scanf("%d", &n);
        for (i = 2, j = n - 2; i <= j; i ++, j --) {
                if (is_prime(i) && is_prime(j)) {
                        print_pair(i, j);
                }
        }
        return 0;
}
```

プログラム 2.18 とプログラム 2.19 とプログラム 2.20 を見比べると、関数を使うことの利点の一つが見えてきます。これら三つのプログラムは、関数 `is_prime` の定義だけが異なっていて、他はまったく同じです。素数判定という機能を一つの関数にまとめているので、素数判定の高速化はその関数の定義だけを書き換えれば良いのです。

コンマ演算子

プログラム 2.18 の

```
        for (i = 2, j = n - 2; i <= j; i ++, j --) {
```

に出てくる二つの ，（コンマ）は、コンマ演算子と呼ばれる、式を逐次実行するためのものです。

`i = 2, j = n - 2` は、最初に `i = 2` を実行して次に `j = n - 2` を実行します。同様に、`i ++, j --` は、最初に `i ++` を実行して次に `j --` を実行します。

コンマ演算子で結ばれた式では、一つめの式の計算結果は捨てて二つめの式の計算結果を全体の結果とします。この例では、計算結果はそもそも必要ないので、気にする必要はありません。いつでも気にしなくて良いのではありませんので、必要があるときは気にしてください。

2.2.4 チェビシェフの定理

「n が正の整数ならば，n より大きく $2n$ 以下の素数が 1 個以上存在する」はチェビシェフの定理として知られています。1850 年にチェビシェフが証明しました。これを確かめるプログラムを書いてみましょう。

プログラム 2.21 正の整数 n を入力して n より大きく $2n$ 以下の素数の個数を出力する（ちょっと遅い）

```
#include <stdio.h>

int     is_prime(int);
int     count_chebyshev(int);

int
is_prime(int x)
{
        int     i;

        if (x < 2)
                return 0;
        for (i = 2; i * i <= x; i ++) {
                if (x % i == 0)
                        return 0;
        }
        return 1;
}

int
count_chebyshev(int x)
{
        int     i;
        int     n;

        n = 0;
        for (i = x + 1; i <= 2 * x; i ++) {
                if (is_prime(i)) {
                        n ++;
                }
        }
        return n;
}

int
main()
{
        int     n;

        scanf("%d", &n);
        printf("%d\n", count_chebyshev(n));
        return 0;
}
```

プログラム 2.22 正の整数 n を入力して n より大きく $2n$ 以下の素数の個数を出力する（2 を除く素数は奇数であることを利用して高速化）

```
#include <stdio.h>

int     is_prime(int);
int     count_chebyshev(int);

int
is_prime(int x)
{
        int     i;

        if (x <= 2)
                return x == 2;
        if (x % 2 == 0)
                return 0;
        for (i = 3; i * i <= x; i += 2) {
                if (x % i == 0)
                        return 0;
        }
        return 1;
}

int
count_chebyshev(int x)
{
        int     i;
        int     n;

        n = 0;
        for (i = x + 1; i <= 2 * x; i ++) {
                if (is_prime(i)) {
                        n ++;
                }
        }
        return n;
}

int
main()
{
        int     n;

        scanf("%d", &n);
        printf("%d\n", count_chebyshev(n));
        return 0;
}
```

プログラム 2.21 は、正の整数 n を入力して n より大きく $2n$ 以下の素数の個数を出力します。ちょっと遅いプログラムです。

main から関数 count_chebyshev を呼び出し、関数 count_chebyshev からさらに関数 is_prime を呼び出していることに注意してください。このように、関数呼び出しが数珠つなぎになることは、ごく普通のことです。

プログラム 2.18 とプログラム 2.21 を比較すると、関数を使うことの利点がもう一つ見えてきます。二つのプログラムは関数 is_prime の定義が共通です。素数判定の機能を一つの関数にまとめているので、その機能だけを切り出して他のプログラムで再利用することが、関数定義を切り貼りするだけで可能になるのです。

プログラム 2.23 正の整数 n を入力して n より大きく $2n$ 以下の素数の個数を出力する（2 と 3 を除く素数は 6 で割った余りが 1 か 5 であることを利用して高速化）

```
#include <stdio.h>

int     is_prime(int);
int     count_chebyshev(int);

int
is_prime(int x)
{
        int     i, d;

        if (x <= 3)
                return x >= 2;
        if (x % 2 == 0 || x % 3 == 0)
                return 0;
        for (i = 5, d = 2; i * i <= x; i += d, d = 6 - d) {
                if (x % i == 0)
                        return 0;
        }
        return 1;
}

int
count_chebyshev(int x)
{
        int     i;
        int     n;

        n = 0;
        for (i = x + 1; i <= 2 * x; i ++) {
                if (is_prime(i)) {
                        n ++;
                }
        }
        return n;
}

int
main()
{
        int     n;

        scanf("%d", &n);
        printf("%d\n", count_chebyshev(n));
        return 0;
}
```

プログラム 2.18 を高速化したのと同じ方法で、プログラム 2.21 を高速化してみましょう。プログラム 2.22 は、2 を除く素数は奇数であることを利用して高速化しています。プログラム 2.23 は、2 と 3 を除く素数は 6 で割った余りが 1 か 5 であることを利用して高速化しています。

第 3 章

配列とポインタ

3.1 配列

同じ型のデータが並んでいるデータ型が配列です。

3.1.1 平均および平均との差

2016 個のデータを入力し、平均を計算して出力し、各データと平均との差を出力するプログラムを書いてみます。

素朴な方法

素朴な方法で実装したのがプログラム 3.1 です。

1. 2016 個のデータを順に入力して配列に格納
2. データの総和を素朴なアルゴリズムで計算[*1]
3. 総和を個数で割って平均を計算
4. 平均を出力
5. 配列に格納したデータについて、データそのものとデータから平均を引いた値を順に出力

の手順で実行します。

マクロ定義

プログラム 3.1 のマーク ■ をつけた行はマクロ定義です。この行があることによって、それ以降に出現する `NUM` はすべて `2016` に置き換えられます。

[*1] なんらかの総和を求める素朴なアルゴリズムは以下の通りです。
1. 仮の総和を記録する変数（等）を用意し 0 で初期化する。
2. 各項目について、その値を仮の総和に加える。
3. すべての項目について処理が終わったら、仮の総和が実際の総和になっている。

大量のデータに対して高精度で総和を求めるにはもう一工夫が必要になりますが、2016 個ぐらいなら素朴なアルゴリズムで十分でしょう。

プログラム 3.1 2016 個のデータを入力し、平均を出力し、各データと平均との差を出力する

```
#include <stdio.h>

#define NUM     2016                                                    //   ■

int
main()
{
        double  data[NUM];
        int     i;
        double  total, average;

        for (i = 0; i < NUM; i ++) {                                    //   ◀
                scanf("%lf", &data[i]);                                 //   ◀
        }                                                               //   ◀
        total = 0;                                                      //   ★
        for (i = 0; i < NUM; i ++) {                                    //   ★
                total += data[i];                                       //   ★
        }                                                               //   ★
        average = total / NUM;                                          //   ★
        printf("###  %10.3f\n", average);                              //   ▶
        for (i = 0; i < NUM; i ++) {                                    //   ▶
                printf("%3d: %10.3f %10.3f\n", i, data[i], data[i] - average);  //   ▶
        }                                                               //   ▶
        return 0;
}
```

プログラム 3.2 2016 個のデータを入力し、平均を出力し、各データと平均との差を出力する（改良版）

```
#include <stdio.h>

#define NUM     2016

int
main()
{
        double  data[NUM];
        int     i;
        double  total, average;
        double  c;

        for (i = 0; i < NUM; i ++) {
                scanf("%lf", &data[i]);
        }
        total = 0;
        c = 0;
        for (i = 0; i < NUM; i ++) {
                double  x = data[i] - c;
                double  s = total + x;
                c = s - total - x;
                total = s;
        }
        average = total / NUM;
        printf("###  %10.3f\n", average);
        for (i = 0; i < NUM; i ++) {
                printf("%3d: %10.3f %10.3f\n", i, data[i], data[i] - average);
        }
        return 0;
}
```

一般に、

`#define` マクロ名 展開先

でマクロ定義できます。それ以降、プログラム中のマクロ名は文字列の中を除いて展開先に書き換えられて処理されます。

プログラム 3.1 は、マクロ定義を使っていることでデータの個数の変更が簡単に行えるようにできています。たとえば、データの個数を 57517 個に増やす必要が生じたら、マクロ定義を

```
#define NUM     57517
```

に書き換えるだけで、他にまったく手を入れずに変更できます。マクロ定義を使わなかったら、プログラム中に散在する複数の数値を変更しなくてはなりません。見落としが生じたり、関係ないところまで書き換えたりする危険がおおいにあります。

マクロの使用は単に置き換えられるだけで、置き換えの際に C の文法には配慮してくれません。そのことで落とし穴にはまることがあります。今回の例のような単純なマクロ定義では気にする必要はありませんが、複雑なマクロ定義では注意が必要になることがあります。

たとえば、

```
#define AYAYA 3 + 8
```

とマクロ定義したとします。マクロ AYAYA は 3 + 8、すなわち、11 であるとプログラマは意図して書いたのでしょう。ここで、

```
x = AYAYA * 2;
```

があったとします。これが実行されると x には 22 が代入されてほしいところですが、実際には 19 が代入されます。

```
x = 3 + 8 * 2;
```

に展開され、掛け算は足し算よりも優先されるからです。

これを防ぐには、冗長でも括弧をつけて

```
#define AYAYA (3 + 8)
```

とマクロ定義します。

配列型の変数の定義

プログラム 3.1 の

```
double  data[NUM];
```

は、コンパイル時に

```
double  data[2016];
```

に書き換えて処理されます。これは、int 型の要素をもつ長さが 2016 の配列を定義します。

配列は、同じ型のデータが連なったものです。配列の要素には、0 から始まる通し番号がついています。その番号を添字（そえじ）と呼びます。上で定義した配列 data は 0 番から 2015 番までの要素があることになります。

プログラム 3.1 で式の中に何度か出てくる data[i] は、配列の要素をアクセスしています。配列 data の i 番目の要素をアクセスします。

浮動小数点数の出力書式

printf の書式の中で、何度か %10.3f が出てきています。これは、出力の幅 10 文字分、小数点以下 3 桁で出力してほしいということです。

高精度な方法

プログラム 3.1 の総和を計算する部分を改良したのが、プログラム 3.2 です。この改良で総和の計算の誤差が小さくなることが知られています[*2]。

3.1.2 最小値および最小値との差

プログラム 3.3 は、2016 個のデータを入力し、最小値を計算して出力し、各データと最小値との差を出力します。以下の手順を実装しています。

プログラム 3.3 2016 個のデータを入力し、最小値を出力し、各データと最小値との差を出力する

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

#define NUM     2016

int
main()
{
        double  data[NUM];
        int     i;
        double  minvalue;

        for (i = 0; i < NUM; i ++) {
                scanf("%lf", &data[i]);
        }
        minvalue = HUGE_VAL;
        for (i = 0; i < NUM; i ++) {
                if (minvalue > data[i]) {
                        minvalue = data[i];
                }
        }
        printf("###  %10.3f\n", minvalue);
        for (i = 0; i < NUM; i ++) {
                printf("%3d: %10.3f %10.3f\n", i, data[i], data[i] - minvalue);
        }
        return 0;
}
```

[*2] 単純に足し算していくだけではなく、毎回の足し算で生じる計算誤差をできる範囲で検出して補償しています。この方法はカハンのアルゴリズムと呼ばれています。詳細は数値解析の教科書を参照してください。

1. 2016 個のデータを順に入力して配列に格納
2. データの最小値を計算[*3]
3. 最小値を出力
4. 配列に格納したデータについて、データそのものとデータから最小値を引いた値を順に出力

この例では、仮の最小値の初期値として用いる十分に大きな値として HUGE_VAL を[*4]採用しています。

3.1.3 素数のリスト

素数のリストを二通りの方法で作ってみます。

試し割り法

プログラム 3.4 は、試し割り法です。ある数よりも小さな素数の一覧があれば、試し割りする数はその範囲に限定できます。そこで、小さいほうから素数判定をし、素数であるとわかったものを順に保存していきます。

プログラム 3.4 は最初の 78498 個の素数を出力します。この個数を変更するには、

```
#define TABLE_SIZE      78498
```

を書き換えます。

無引数関数のプロトタイプ宣言

引数が 0 個の関数のプロトタイプ宣言では、関数名の後ろの () の中に void と書く必要があります。歴史的な事情で () の中が空の場合には別の意味があるため、引数が 0 個の場合だけ例外になっています[*5]。

プログラム 3.4 の ♦ の部分は実例です。関数 init_table も関数 print_primes も、引数が 0 個で返値がない関数であることを宣言しています。

大域変数

プログラム 3.4 では、配列 prim_table を関数定義の外で定義していることに注意してください。このように、関数の外で定義された変数を**大域変数** (**グローバル変数**) と呼びます。大域変数は次のような性質をもちます。

- 定義された場所以降のどこからでも見える。
- プログラムの実行開始から存在し、実行終了まで存続する。

一方、関数の中で定義された変数を**局所変数** (**ローカル変数**) と呼びます。局所変数は、特に指定しない限り、次のような性質をもちます。

[*3] 以下のアルゴリズムで最小値を計算することができます。
1. 仮の最小値を記録する変数（等）を用意し十分に大きな値で初期化する。
2. 各項目について、その値を仮の最小値と比較し、その値が小さければ仮の最小値を更新する。
3. すべての項目について処理が終わったら、仮の最小値が実際の最小値になっている。

大小を逆にすれば、最大値も同様に求めることができます。

[*4] §1.5.6 を参照。

[*5] C と違ってそのあたりに歴史的なしがらみのない C++ では異なります。C と C++ の両方を使う人は注意してください。

プログラム 3.4 素数のリスト（試し割り法）

```
#include <stdio.h>

#define TABLE_SIZE      78498

void    init_table(void);                                       //  ♦
void    print_primes(void);                                     //  ♦
int     none_is_divisible(int, int);
int     prime_table[TABLE_SIZE];

void
init_table()
{
        int     i;
        int     n;

        n = 2;
        for (i = 0; i < TABLE_SIZE; i ++) {
                while (!none_is_divisible(i, n)) {
                        n ++;
                }
                prime_table[i] = n;
                n ++;
        }
}

int
none_is_divisible(int size, int x)
{
        int     i;

        for (i = 0; i < size; i ++) {
                if (x % prime_table[i] == 0)
                        return 0;
        }
        return 1;
}

void
print_primes()
{
        int     i;

        for (i = 0; i < TABLE_SIZE; i ++) {
                printf("%d\n", prime_table[i]);
        }
}

int
main()
{
        init_table();
        print_primes();
        return 0;
}
```

- 変数定義が含まれる最内の{}（関数の仮引数の場合はその関数定義の{}）のうち、定義以降の部分からのみ見える。
- 変数定義が含まれる最内の{}が実行されるたびに作成され、その実行が終了すると消滅する。

大域変数はどこからでも見えて便利ではありますが、逆に、どこで触られるかわからなくて怖いです。プログラムが巨大化してくると、想定外のところで大域変数の値が変更されて思わぬバグの元となる危険も増します。大域変数は、使いすぎないように注意して使ってください。

エラトステネスの篩

プログラム 3.5 は、有名なエラストテネスの篩[*6]を実装したものです。素数かどうかを判定するための表として、大域変数 prime_table[] を使っています。関数 init_table() が、表 prime_table[] に適切な値を設定します。この関数を一度呼び出すと、prime_table[i] の値は、i が素数のときは 1、そうでないときは 0 になります。

プログラム 3.5 は 1000000 未満の素数を出力します。範囲を変更したいときは、

```
#define TABLE_SIZE      1000000
```

を書き換えます。

多重代入

プログラム 3.5 での関数 init_table の定義にある

```
    prime_table[0] = prime_table[1] = 0;
```

では、prime_table[0] と prime_table[1] の両方に 0 を代入しています。

仕組みはこうなっています。

=が並ぶと、右側が優先されます。したがって、優先順位を括弧を使って表すと、次のようになります。

```
    prime_table[0] = (prime_table[1] = 0);
```

C では、代入を意味する=も演算子の一つとして扱われます。その式としての値は代入した値となります。したがって、0 を prime_table[1] に代入した上で、さらに同じ値を prime_table[1] に代入することになります。

[*6] エラストテネスの篩は、以下のようなアルゴリズムです。2 以上 N 未満の整数について、素数か合成数かの判定表を作るものとします。

- 2 以上 N 未満の整数を未確定の状態にしておく。
- p の初期値を 2 として、p が $\sqrt{N}$ より小さい間、以下を繰り返す。
  - p を素数と確定する。
  - p^2 以上の p の倍数をすべて合成数と確定する。
  - p をこの時点で未確定な最小値に変更する。
- 残った未確定の整数をすべて素数と確定する。

プログラム 3.5 素数のリスト（エラトステネスの篩）

```
#include <stdio.h>

#define TABLE_SIZE      1000000

void    init_table(void);
void    print_primes(void);
char    prime_table[TABLE_SIZE];

void
init_table()
{
        int     i, j;

        prime_table[0] = prime_table[1] = 0;
        for (i = 2; i < TABLE_SIZE; i ++) {
                prime_table[i] = 1;
        }
        for (i = 2; i * i < TABLE_SIZE; i ++) {
                if (prime_table[i]) {
                        for (j = i * i; j < TABLE_SIZE; j += i) {
                                prime_table[j] = 0;
                        }
                }
        }
}

void
print_primes()
{
        int     i;

        for (i = 0; i < TABLE_SIZE; i ++) {
                if (prime_table[i]) {
                        printf("%d\n", i);
                }
        }
}

int
main()
{
        init_table();
        print_primes();
        return 0;
}
```

3.2 ポインタ

3.2.1 ポインタとは

ポインタとは、データがメモリのどこにあるかを表すデータです[*7]。

int を指すポインタ型の変数を定義するには、たとえば、

[*7] 変数をデータを格納する箱に例えれば、ポインタは箱が設置してある場所の住所表記にあたります。したがって、ポインタ型の変数は、住所表記を格納する箱にあたります。

```
int *p;
```

のように書きます[*8]。

式の前に*をつけると、ポインタの指している実体を表します。

*ポインタ

は ポインタ の指している実体を表します[*9]。逆に、式の前に&をつけることで、その式が表す実体を指すポインタを表します。

&変数

は 変数 を指すポインタを表します[*10]。

図 3.1 の左側のコードが実行されると、図 3.1 の右側の図の状態になります。まず、int 型の変数 x が作られ、初期値は 5 になります。次に、int *型の変数 p が作られ、初期値は変数 x を指すポインタになります。

NULL は特別なポインタで、どこも指していないことを表します。図 3.2 の左側のコードが実行されると、図 3.2 の右側の図の状態になります。変数 p には「どこも指していない」を意味する特別なポインタが格納されています。どこも指していないので、実体は存在しませんが、ポインタではあるので、他のポインタと等しいか等しくないかの判定は可能です。

NULL を使うには、

```
#include <stddef.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <wchar.h>
#include <locale.h>
```

のうちの少なくとも 1 行をプログラムの最初のほうに書く必要があります。

```
int x = 5;
int *p = &x;
```

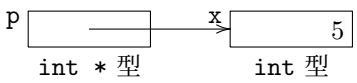


図 3.1: ポインタ操作の例

```
int *p = NULL;
```

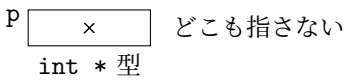


図 3.2: NULL ポインタ

[*8] int 型のデータを格納する箱の設置場所の住所表記を格納する箱が用意され、p という名札が貼られている状態に例えられます。
[*9] ポインタを住所表記に例えると、その住所表記の場所に設置されている箱にあたります
[*10] 変数を箱に例えると、その箱が設置されている場所の住所表記にあたります。

```
int *p;
```

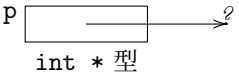


図 3.3: 初期化されていないポインタ変数

初期化されていない局所変数の初期値がゴミの値であることは、ポインタ型の変数でも変わりません。図 3.3 の左側のコードが実行されると、図 3.3 の右側の図の状態になります。変数 p にはなんらかのポインタ値が入っていますが、その指している先がどこであるかはわかりません。その状態で*p をアクセスしようとすると、運良く存在しないメモリやアクセス不可なメモリを指していれば実行時にエラーが報告されるかもしれませんが、運悪く実在する変数などが使っているメモリ領域を指していれば全然関係ない変数の値が知らないうちに変わってしまうことがあります。とても取りにくいバグの原因になりますので、ご注意ください。

3.2.2 ポインタの用途 (1) 参照呼び

ポインタの重要な用途の一つが、参照呼びの実現です。関数を呼び出す側で変数を用意し、その変数の値を呼び出した先で変更させたいとき、参照呼びで実現します。

関数から呼び出し元に一つの値を返したいときは、返したい値を関数の返り値にすれば簡単です。たとえば、二つの整数の最大公約数だけを計算したいなら、プロトタイプ宣言が

```
unsigned        gcd(unsigned, unsigned);
```

となるような関数を定義することで実現できます。

では、関数から複数の値を返したいときは、どうすれば良いでしょうか。たとえば、二つの整数の最大公約数と最小公倍数をいっしょに計算したいときはどうしましょう。最大公約数を計算する関数と最小公倍数を求める関数を別々に作ると、途中の計算の共通する部分を無駄に二度行うことになり、非効率です。それを解決するのが、参照呼びです。

C で参照呼びを実現するには、呼び出し元で受け取り用の変数を用意して、その変数を指すポインタを引数として関数呼び出しを行います。たとえば、プログラム 3.6 を実行すると

プログラム 3.6 参照呼びの実現

```
#include <stdio.h>

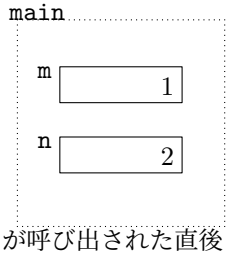
void
ayaya(int *x, int *y)
{
    *x = 3; *y = 5;
}

int
main()
{
    int m = 1, n = 2;
    ayaya(&m, &n);
    printf("%d %d\n", m, n);
    return 0;
}
```

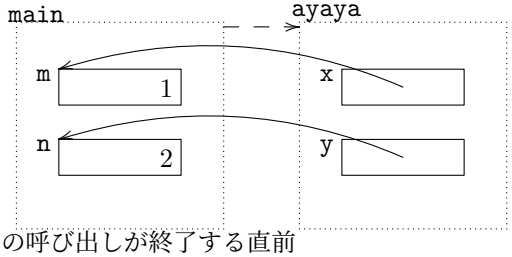
```
3 5
```

が出力されます。この時の変数の値の変遷を追いかけると、以下のようになります。

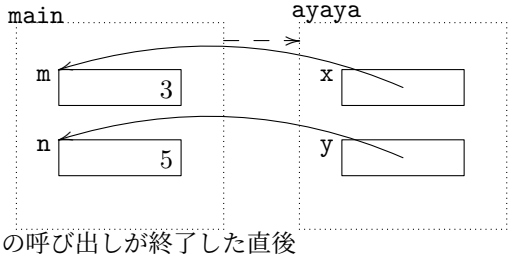
- ayaya が呼び出される直前



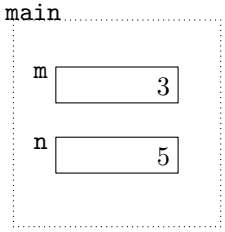
- ayaya が呼び出された直後



- ayaya の呼び出しが終了する直前



- ayaya の呼び出しが終了した直後

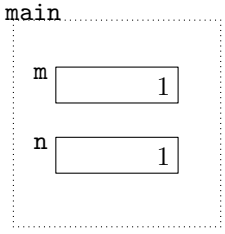


なお、C の関数呼び出しでは、呼び出された関数で仮引数の値を変更しても、呼び出した側の実引数には影響しません。たとえば、プログラム 3.7 を実行すると

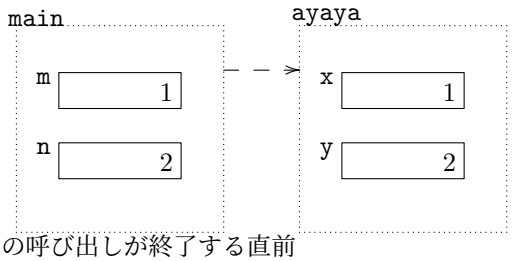
```
1 2
```

が出力されます。この時の変数の値の変遷を追いかけると、以下のようになります。

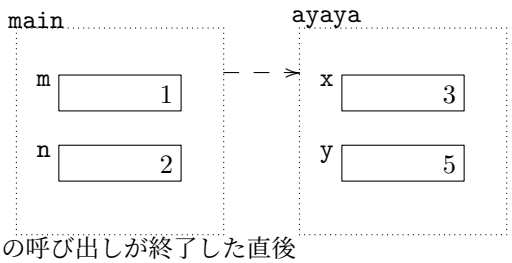
- ayaya が呼び出される直前



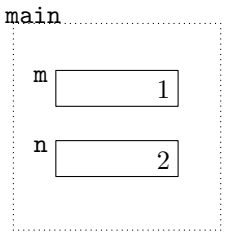
- ayaya が呼び出された直後



- ayaya の呼び出しが終了する直前



- ayaya の呼び出しが終了した直後



標準ライブラリ関数 scanf

ライブラリ関数 scanf を使う際に

```
scanf("%d%d", &m, &n);
```

と & をつけるのは、実は、参照呼びです。scanf は引数としてポインタを受け取り、その指す先に入力した値を代入する関数です。したがって、どの変数に変数に値を代入するかを scanf に知らせるためには、その変数を指すポインタを渡す必要があります。変数名の前の & はそのためのものだったのです。

つまり、scanf の引数はポインタであることが本質的であり、& をつけるのがお約束ということではありま

プログラム 3.7 仮引数の値を変更してみる

```
#include <stdio.h>

void
ayaya(int x, int y)
{
    x = 3; y = 5;
}

int
main()
{
    int m = 1, n = 2;
    ayaya(m, n);
    printf("%d %d\n", m, n);
    return 0;
}
```

プログラム 3.8 scanf を間接的に呼ぶ

```
#include <stdio.h>

void
get_int_pair(int *p, int *q)
{
        scanf("%d%d", p, q);
}

int
main()
{
        int     m, n;
        get_int_pair(&x, &y);
        printf("%d\n", m + n);
        return 0;
}
```

せん。そのことは、プログラム 3.8 を見るとよくわかります。scanf の呼び出しで p と q の前に & をつけていません。プログラム 3.8 は整数を二つ入力してその和を出力するだけのプログラムですが、main から scanf を直接呼び出すのではなく、関数 get_int_pair に二つのポインタを中継させています。関数 get_int_pair の仮引数 p と q はもともとポインタなので、そのまま、scanf に渡せば良いのです。

例：正方形の周長と面積

プログラム 3.9 は、正方形の一辺の長さが与えられたとき、周長と面積を計算します[*11]。計算を行う部分を一つの関数にまとめて、main() からその関数を呼び出す形で実装しています。

関数 square_perimeter_area() が、正方形の一辺の長さから周長と面積を計算します。第一引数の値として一辺の長さを受け取ります。第二引数と第三引数がポインタで、計算結果をそれぞれの指している実体に代入して返します。

プログラム 3.10 は、プログラム 3.9 をより安全にしたものです。第二引数と第三引数についてのいずれについても、ポインタ値が NULL であるかないかを判定して、NULL でないときにのみ計算結果をポインタの指す実体に代入します。このように、値が不要なことを意図してポインタ値 NULL を用いることは、しばしば行われます。

一般に、ポインタ値が NULL である可能性が完全に排除される状況でなければ、ポインタの実体を取る前に NULL 検査を行うのは良いことです。関数 square_perimeter_area() が切り出されて別のプログラムで使われる可能性がちょっとでもあれば、プログラム 3.9 よりもプログラム 3.10 のほうがお勧めです。

なお、NULL の指す実体の参照は、当然、エラーですが、そのエラーでプログラムの動作がどうなるかについては、C の言語仕様では特に規定されていません。エラーメッセージを出力して停止するシステムも、何事もなかったかのように動作を続けるシステムも、どちらも実在します。

例：最大公約数と最小公倍数

プログラム 3.11 は、二つの整数が与えられたとき、最大公約数と最小公倍数を計算します。

関数 gcd() は、第一引数の値と第二引数の値の最大公約数を計算し、計算結果を関数の返値として返しま

[*11] 一辺の長さが a の正方形の周長は $4a$ で面積は a^2 です。

プログラム 3.9 正方形の一辺の長さから周長と面積を計算するプログラム

```
#include <stdio.h>
void    square_perimeter_area(double, double *, double *);

void
square_perimeter_area(double a, double *p, double *q)
{
        *p = 4 * a;
        *q = a * a;
}

int
main()
{
        double  x, y, z;
        scanf("%lf", &x);
        square_perimeter_area(x, &y, &z);
        printf("%12.5f %12.5f\n", y, z);
        return 0;
}
```

プログラム 3.10 正方形の一辺の長さから周長と面積を計算するプログラム（NULL 検査つき）

```
#include <stdio.h>
void    square_perimeter_area(double, double *, double *);

void
square_perimeter_area(double a, double *p, double *q)
{
        if (p != NULL) {
                *p = 4 * a;
        }
        if (q != NULL) {
                *q = a * a;
        }
}

int
main()
{
        double  x, y, z;
        scanf("%lf", &x);
        square_perimeter_area(x, &y, &z);
        printf("%12.5f %12.5f\n", y, z);
        return 0;
}
```

す。プログラム 2.15 から定義を切り出し、再利用したものです。

関数 gcd_lcm() は、関数 gcd() を下請けに使って、最大公約数と最小公倍数を計算します[*12]。第一引数の値と第二引数の値の最大公約数と最小公倍数を計算します。第三引数と第四引数がポインタで、計算結果をそれぞれの指している実体に代入して返します。いずれでも、NULL 検査を行っています。

関数 gcd() の定義をプログラム 2.15 のもの（剰余演算を使用）からプログラム 2.17 のもの（減算と 2 倍と 1/2 倍だけを使用）に置き換えることも可能です。紙面の都合で、置き換えたプログラムの掲載は省略します。

[*12] 整数 a と b の最大公約数が g で最小公倍数が l のとき、$ab = gl$ が成り立ちます。この事実を利用すると、最大公約数の計算を下請けにして最小公倍数を計算できます。

プログラム 3.11 二つの整数の最大公約数と最小公倍数を計算するプログラム

```
#include <stdio.h>

unsigned        gcd(unsigned, unsigned);
void    gcd_lcm(unsigned, unsigned, unsigned *, unsigned *);

unsigned
gcd(unsigned x, unsigned y)
{
        unsigned        z;
        while (y != 0) {
                z = x % y;
                x = y;
                y = z;
        }
        return x;
}

void
gcd_lcm(unsigned x, unsigned y, unsigned *p, unsigned *q)
{
        unsigned        g = gcd(x, y);
        if (p != NULL) {
                *p = g;
        }
        if (q != NULL) {
                *q = x / g * y;
        }
}

int
main()
{
        unsigned        m, n, g, l;
        scanf("%u%u", &m, &n);
        gcd_lcm(m, n, &g, &l);
        printf("%u %u\n", g, l);
        return 0;
}
```

3.2.3 ポインタの用途 (2) 配列の要素のアクセス

C では、配列の要素（先頭とは限らない）のアクセスにポインタを用いることができます。その際には、ポインタの加減算と大小比較が役立ちます。

ポインタに整数を足す

配列の要素を指しているポインタ値に整数値を足すと、結果は加えた整数値だけ添字を進めた要素を指すポインタ値になります。たとえば、図 3.4 の左側のコードを実行すると、`p` の値は `a[3]` を指すポインタ（図 3.4 実線の矢印）になります。この直後に `p + 2` を計算すると、結果は `a[5]` を指すポインタ（図 3.4 破線の矢印）になります。したがって、関数 `ayaya` の呼び出しでは引数として `a[5]` を指すポインタが渡されます。

配列の要素を指すポインタに整数を足して得られたポインタが配列の範囲からはみ出した時の動作については、後述の「配列の最後の要素の次」（77 ページ）の場合を除いて、処理系に依存します。実際、システムと

状況によって、セグメンテーション違反で異常終了することもあれば、知らん顔をして関係ない変数をアクセスしてしまうこともあります。

```
int a[10];
int *p = &a[3];

ayaya(p + 2);
```

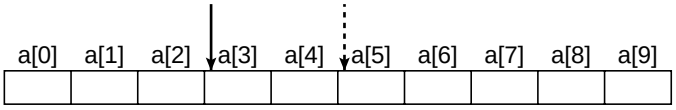


図 3.4: ポインタへの整数の足し算

ポインタから整数を引く

配列の要素を指しているポインタ値から整数値を引くと、結果は引いた整数値だけ添字を戻した要素を指すポインタ値になります。たとえば、図 3.5 の左側のコードを実行すると、p の値は a[3] を指すポインタ（図 3.5 実線の矢印）になります。この直後に p - 2 を計算すると、結果は a[1] を指すポインタ（図 3.5 破線の矢印）になります。したがって、関数 ayaya の呼び出しでは引数として a[1] を指すポインタが渡されます。

配列の要素を指すポインタから整数を引いて得られたポインタが配列の範囲からはみ出した時の動作については、後述の「配列の最後の要素の次」（77 ページ）の場合を除いて、処理系に依存します。実際、システムと状況によって、セグメンテーション違反で異常終了することもあれば、知らん顔をして関係ない変数をアクセスしてしまうこともあります。

```
int a[10];
int *p = &a[3];

ayaya(p - 2);
```

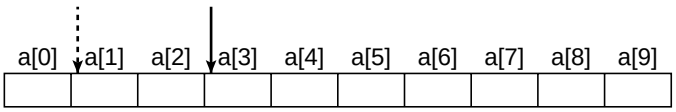


図 3.5: ポインタから整数の引き算

ポインタからポインタを引く

配列の要素を指しているポインタ値から同じ配列の要素を指しているポインタ値を引くと、結果は添字の差にあたる整数値になります。たとえば、図 3.6 の左側のコードを実行すると、p の値は a[5] を指すポインタ（図 3.6 実線の矢印）になり、q の値が a[1] を指すポインタ（図 3.6 破線の矢印）になります。その直後に p - q を計算すると結果は 4 になり、q - p を計算すると結果は −4 になります。したがって、関数 hoyoyo の 1 回目の呼び出しでは 4 が引数として渡され、2 回目の呼び出しでは −4 が渡されます。

二つのポインタ値が同じ配列の要素を指すものでなければ、引き算してはいけません。してはいけないのにしてしまうと何が起きるかは、処理系に依存します。たとえば、図 3.7 の左側のコードを実行すると、p の値は a[5]（図 3.7 実線の矢印）を指すポインタになり、q の値は b[1] を指すポインタ（図 3.7 破線の矢印）に

```
int a[10];
int *p = &a[5];
int *q = &a[1];

hoyoyo(p - q);
hoyoyo(q - p);
```

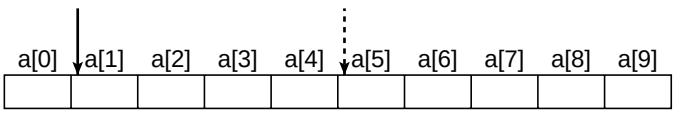


図 3.6: ポインタからポインタの引き算

```
int a[10];
int b[12];
int *p = &a[5];
int *q = &b[1];

hoyoyo(p - q);
hoyoyo(q - p);
```

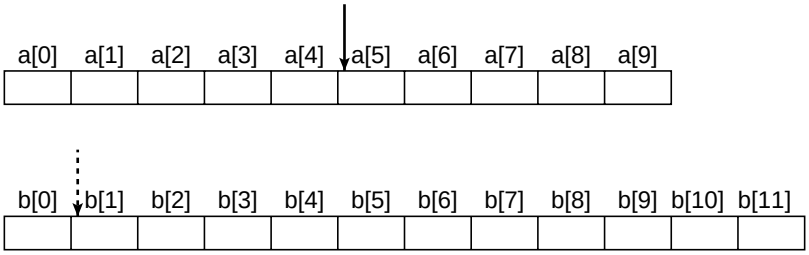


図 3.7: ポインタからポインタの引き算の困った事態

なります。この状況で `p - q` や `q - p` を計算して何が起きるかはわかりません。

ポインタとポインタの大小比較

配列の要素を指しているポインタ値と同じ配列の要素を指しているポインタ値の大小比較の結果は、添字の大小比較の結果と一致します。たとえば、図 3.8 の左側のコードを実行すると、`p` の値は `a[5]` を指すポインタ（図 3.8 実線の矢印）になり、`q` の値は `a[1]` を指すポインタ（図 3.8 破線の矢印）になります。その直後に `p < q` を計算すると、結果は 0 になり、`p >= q` を計算すると、結果は 1 になります[*13]。したがって、関数 `nahaha` の 1 回目の呼び出しでは引数として 1 が渡され、2 回目の呼び出しでは 0 が渡されます。

二つのポインタ値が同じ配列の要素を指すものでなければ、大小比較してはいけません。してはいけないのにしてしまうと何が起きるかは、処理系に依存します。たとえば、図 3.9 の左側のコードを実行すると、`p` の値は `a[5]` を指すポインタ（図 3.9 実線の矢印）になり、`q` の値は `b[1]` を指すポインタ（図 3.9 破線の矢印）になります。この直後に `p < q` を計算して何が起きるかはわかりません。

```
int a[10];
int *p = &a[5];
int *q = &a[1];

nahaha(p < q);
nahaha(p >= q);
```

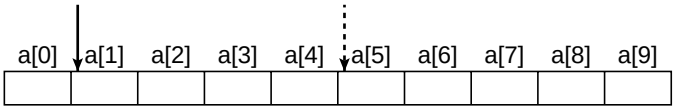


図 3.8: ポインタとポインタの大小比較

```
int a[10];
int b[12];
int *p = &a[5];
int *q = &b[1];

nahaha(p < q);
nahaha(p >= q);
```

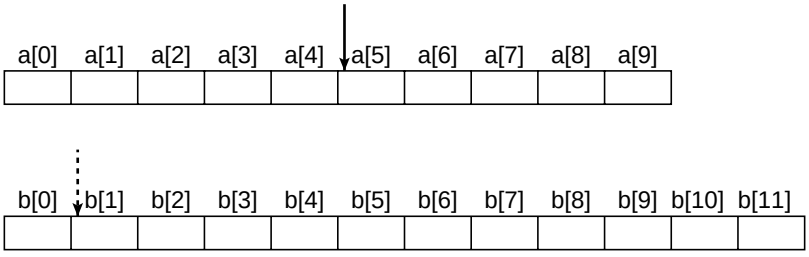


図 3.9: ポインタとポインタの大小比較の困った事態

[*13] if 文や while 文などでの条件部で、0 は偽を、0 以外の値は真を表すことを思い出しましょう

配列の最後の要素の次

配列の最後の要素の次には、当然、要素はありません。しかし、そこを指すポインタは、ポインタとしては有効と C の言語仕様で規定されています（図 3.10）。ただし、ポインタとして有効なだけで、実体をとること（単項*演算子の適用）をしてはいけません。してはいけないのにしてしまうと何が起きるかは、処理系に依存します。

このことは、前四つの小小節で説明して演算すべてに適用されます。ポインタと整数かポインタとポインタの演算に限れば、引数や結果が配列の最後の要素の次を指すものであるとき、あたかも配列の最後の次にもう一つ要素があるかのように処理されます。しかし、本当にそこに要素があるのではないので、そのポインタの実体をとってはいけません。

```
int a[10];
int *p = &a[10];
```

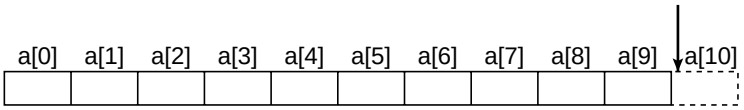


図 3.10: 配列の最後の要素の次を指すポインタ

配列名

C では、式の中で配列名が使われると、一部の例外を除いて[*14]、その配列の最初の要素へのポインタと解釈されます。たとえば、

```
int     a[10];
int     *p;
```

と定義されているとき、

```
p = a;
```

と

```
p = &a[0];
```

は同じ意味になります。

[] 演算子

p がポインタのとき、p[i] は*(p+i) の略記とみなされます。この略記を利用すると、ポインタ p が配列の最初の要素を指しているとき、見かけ上、p が配列名であるかのようにプログラムを書くことができます。

size_t と ptrdiff_t

C には複数の整数型があり、表せる整数の範囲が異なります[*15]。

データのバイト数や配列の大きさや配列の添字などメモリサイズに関係する整数を表すのにどの整数型が適

[*14] 後述の sizeof 演算子は例外の一つです。
[*15] 1.5.1 節参照

切かは、システムによって異なります。例えば、配列の添字を表す変数を小さすぎる整数型で定義すると、大きな配列の後ろのほうがアクセスできなくなる危険があります。大きすぎる整数型で定義すると、計算時間が無駄になります。システムごとに適切な整数型を選ばなくてはなりません。ところが、そうなると、あるシステムで動かしていたプログラムを他のシステムで使う際にいちいち書き直す必要が生じて、手間がかかり間違いのもとになります。

そこで、C の標準ライブラリには size_t 型が用意されています。これは、実際にはそのシステムで使える整数型のどれかなのですが、どれなのかはシステムごとに異なります。そのシステムでメモリサイズに関係する整数を表すのに最適な型が選ばれます。つまり、プログラマは size_t 型で変数定義しておけば、システムごとに最適な型が自動的に選ばれることになります。これを使えば、システムごとにプログラムを書き換える必要がなくなります。

prtdiff_t は、ポインタとポインタの差を表すのに適した整数型です。ある意味で size_t の類似品ですが、たいていのシステムでは size_t は符号なし整数であるのに対して、prtdiff_t は符号つき整数であることが違います。

size_t 型を使うには、

```
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <wchar.h>
```

のいずれかが必要です。

prtdiff_t 型を使うには、

```
#include <stddef.h>
```

が必要です。

sizeof 演算子

sizeof 型 は、型が示す型の変数のバイト数を表します。具体的な値はシステムごとに異なります。ただし、char と signed char と unsigned char については、sizeof の値は 1 であると決まっています。

sizeof 式 と書くこともできます。この場合は、式 の型のバイト数を表します。

sizeof 演算子は、配列とポインタに関する C の特別ルールの例外です。sizeof 型 または sizeof 式 で型や式が配列の場合、sizeof の値は配列全体のバイト数になります。配列の先頭へのポインタとは解釈されません。

なお、上の例では型を丸括弧でくるんで sizeof(int) と書いていますが、見やすくするためだけで、意味は変わりません。

例：配列の要素の総和

a[] の総和を計算して、s に代入します。3 通りの実装です。

- 添字を動かす

```
        size_t  i;
        s = 0;
        for (i = 0; i < 100; i ++) {
                s += a[i];
        }
```

- ポインタを動かす

```
        int     *p;
        s = 0;
        for (p = &a[0]; p < &a[100]; p ++) {
                s += *p;
        }
```

- ポインタを動かす

```
        int     *p;
        s = 0;
        for (p = a; p < a + 100; p ++) {
                s += *p;
        }
```

ポインタを動かす実装二つは、書き方が違うだけで同じ意味です。

例：配列から配列へのコピー

a[] から b[] に要素をコピーします。

- 添字を動かす

```
        size_t  i;
        for (i = 0; i < 100; i ++) {
                b[i] = a[i];
        }
```

- ポインタを動かす

```
        int     *p, *q;
        for (p = a, q = b; p < a + 100; p ++, q ++) {
                *q = *p;
        }
```

- memcpy() を使う。

  #include <string.h>

  が必要。

```
        memcpy(b, a, sizeof(int) * 100);
```

memcpy は C の標準ライブラリ関数で、メモリ領域からメモリ領域へその内容をコピーします。

memcpy(コピー先の先頭を指すポインタ，コピー元の先頭を指すポインタ，コピーするバイト数)

の形で使います。

memcpy では、コピー元のメモリ領域とコピー先のメモリ領域に重なりがあってはいけません。重なりがある場合に memcpy を使うと何が起こるかはわかりません。

重なりがある可能性がある場合は、memcpy のかわりに memmove を使います。たとえば、int の配列 a に対して a[5]〜a[14] を a[0]〜a[9] にコピーしたいときは、

```
memmove(a, a + 5, sizeof(int) * 10);
```

とします。

例：配列から配列へ各要素を 2 個ずつコピー

a[] から b[] に要素をコピーしますが、コピー元の要素 1 個がコピー先に 2 個書き込まれるようにします。

- 添字を動かす

```
size_t  i;
for (i = 0; i < 100; i ++) {
        b[i * 2] = a[i];
        b[i * 2 + 1] = a[i];
}
```

- 添字を動かす（別解）

```
size_t  i, j;
for (i = 0, j = 0; i < 100; i ++, j += 2) {
        b[j] = a[i];
        b[j + 1] = a[i];
}
```

- 添字を動かす（別解 2）

```
size_t  i, j;
for (i = 0, j = 0; i < 100; i ++) {
        b[j] = a[i];
        j ++;
        b[j] = a[i];
        j ++;
}
```

- ポインタを動かす

```
int     *p, *q;
for (p = a, q = b; p < a + 100; p ++, q += 2) {
        *q = *p;
        *(q + 1) = *p;
}
```

- ポインタを動かす（別解）

```
int     *p, *q;
for (p = a, q = b; p < a + 100; p ++) {
        *q = *p;
        q ++;
        *q = *p;
        q ++;
}
```

例：配列から二つの配列へ交互にコピー

a[] から b[] と c[] に交互にコピーします。

- 添字を動かす

```
size_t  i;
for (i = 0; i < 50; i ++) {
        b[i] = a[i * 2];
        c[i] = a[i * 2 + 1];
}
```

- 添字を動かす（別解）

```
size_t  i, j;
for (i = 0, j = 0; j < 50; i += 2, j ++) {
        b[j] = a[i];
        c[j] = a[i + 1];
}
```

- 添字を動かす（別解 2）

```
size_t  i, j;
for (i = 0, j = 0; j < 50; j ++) {
        b[j] = a[i];
        i ++;
        c[j] = a[i];
        i ++;
}
```

- ポインタを動かす

```
int     *p, *q, *r;
for (p = a, q = b, r = c; q < b + 50; p += 2, q ++, r ++) {
        *q = *p;
        *r = *(p + 1);
}
```

- ポインタを動かす（別解）

```
int     *p, *q, *r;
for (p = a, q = b, r = c; q < b + 50; q ++, r ++) {
        *q = *p;
        p ++;
        *r = *p;
        p ++;
}
```

3.2.4 ポインタの用途 (3) 配列を関数に渡す

C では、配列の値をまるごと関数に渡すことはできません。しかし、呼び出し元で用意した配列を関数に渡して何かを計算させたいことは、よくあります。そこで、配列とポインタに関する C の以下の仕様を利用します。

- 配列名が式の中に現れたら、その先頭の要素を指すポインタと解釈される。
- 配列の先頭を指すポインタを使うと、ポインタ[添字] で、配列の要素をアクセスできる。

これを利用すると、次のような形式でプログラムを書くことができます。

呼び出し元 関数名(配列名) の形式で、先頭の要素を指すポインタを渡す。

呼び出し先 ポインタ型の仮引数で配列の先頭の要素を指すポインタを受け取り、その仮引数名を配列名と同じように使用する。

仮引数では、たとえば、`int *a` を `int a[]` と書き換えても、同じ意味になります[*16]。図 3.11 の左のコードと右のコードは同じ意味です。右のほうが、いかにも配列を渡しているかように見えるかもしれません。

```
void
ayaya(int *a)
{
        a[0] = 5;
}
```

同じ意味

```
void
ayaya(int a[])
{
        a[0] = 5;
}
```

図 3.11: 配列の最初の要素を指すポインタを渡すときの仮引数の書き方

配列を関数に渡そうとすると、必然的に参照呼びになります。そのため、そのままでは、呼び出し先の関数で渡された配列の要素に代入を行うと、呼び出し元の配列の要素の値が変わってしまいます。それが意図することである場合はそれでかまいません。そうでない場合は、`const` を併用します。

図 3.12 は `const` の効果を説明する例です。左側のコードは正しいプログラム（の一部）ですが、右側のコードはコンパイル時にエラーとなります。右側では、代入してはいけないところに代入しようとしているからです。

```
void
ayaya(int *a)
{
        a[0] = 5;
}
```

正しくコンパイルされる

```
void
ayaya(const int *a)
{
        a[0] = 5;
}
```

コンパイル時にエラーとなる

図 3.12: ポインタと const へのポインタ

[*16] 仮引数以外のところでこの書き換えを行うと意味が変わります。

3.2.5　続 平均および平均との差

プログラム 3.12 は、2016 個のデータを入力し、平均を計算して出力し、各データと平均との差を出力するプログラムの別バージョンです。プログラム 3.1 で入力を行なっている部分（行末に ◀ をつけた部

プログラム 3.12 2016 個のデータを入力し、平均を出力し、各データと平均との差を出力する（別バージョン）

```
#include <stdio.h>

#define NUM     2016

void    input_data(double *, int);                                          //   *
double  average_data(const double *, int);                                  //  **
void    print_data_with_differences(const double *, int, double);           // ***

void
input_data(double *data, int size)                                          //   †
{
        int     i;

        for (i = 0; i < size; i ++) {
                scanf("%lf", &data[i]);
        }
}

double
average_data(const double *data, int size)                                  //  ††
{
        double  sum = 0;
        int     i;

        for (i = 0; i < size; i ++) {
                sum += data[i];
        }
        return sum / size;
}

void
print_data_with_differences(const double *data, int size, double x)         // †††
{
        int     i;

        printf("###  %10.3f\n", x);
        for (i = 0; i < size; i ++) {
                printf("%3d: %10.3f %10.3f\n", i, data[i], data[i] - x);
        }
}

int
main()
{
        double  data[NUM];
        double  average;

        input_data(data, NUM);
        average = average_data(data, NUM);
        print_data_with_differences(data, NUM, average);
        return 0;
}
```

分）がプログラム 3.12 では関数 input_data に、平均の計算を行なっている部分（行末に ★ をつけた部分）が関数 average_data に、差の計算と出力を行なっている部分（行末に ▶ をつけた部分）が関数 print_data_with_differences に、それぞれ、切り出されています。複数の関数で同一の配列を操作しなくてはならないので、main() で定義した配列 data を引数で各関数に渡しています。

プログラム 3.13 2016 個のデータを入力し、平均を出力し、各データと平均との差を出力する（別パージョン）

```
#include <stdio.h>

#define NUM     2016

void    input_data(double [], int);                                             //   *
double  average_data(const double [], int);                                     //  **
void    print_data_with_differences(const double [], int, double);              // ***

void
input_data(double data[], int size)                                             //   †
{
        int     i;

        for (i = 0; i < size; i ++) {
                scanf("%lf", &data[i]);
        }
}

double
average_data(const double data[], int size)                                     //  ††
{
        double  sum = 0;
        int     i;

        for (i = 0; i < size; i ++) {
                sum += data[i];
        }
        return sum / size;
}

void
print_data_with_differences(const double data[], int size, double x)            // †††
{
        int     i;

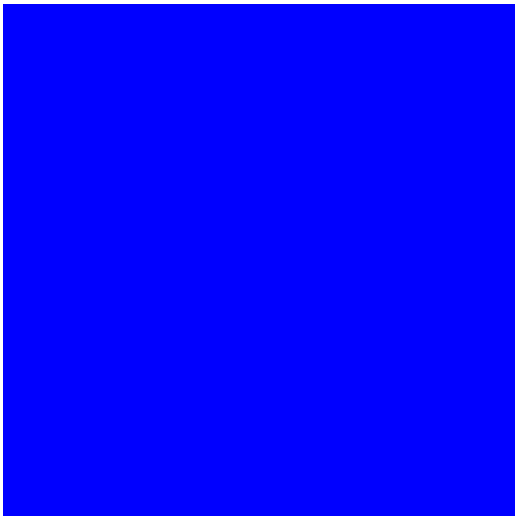
        printf("###  %10.3f\n", x);
        for (i = 0; i < size; i ++) {
                printf("%3d: %10.3f %10.3f\n", i, data[i], data[i] - x);
        }
}

int
main()
{
        double  data[NUM];
        double  average;

        input_data(data, NUM);
        average = average_data(data, NUM);
        print_data_with_differences(data, NUM, average);
        return 0;
}
```

入力・計算・出力のそれぞれ関数は、配列の大きさも引数として受け取っています。それによって、異なる大きさの配列に対しても同じ関数が使えるようになります。この例ではそのことの利点は見えてきませんが、関数の再利用を頻繁に行うと、さまざまな大きさの配列に適用できることのありがたみが見えてきます。

プログラム 3.13 は、仮引数の書き方だけを変えて、プログラム 3.12 を書き換えたものです。両者はまったく同じ意味になります。



3.2.6　続 素数の表

プログラム 3.14 は、試し割り法で素数の表を作るプログラムの別バージョンです。§3.1.3 のプログラム 3.4（65 ページ）では素数の表を大域変数 `prime_table` として定義してどの関数からも見えるようにしていますが、プログラム 3.14 では、`main` 関数の局所静的変数として定義して、下請けの関数に渡しています。

関数内またはブロック内で変数を定義する際に型の前に `static` をつけると、その変数は**局所静的変数**になります。局所静的変数は、プログラムの実行開始から実行終了まで存在し続けますが、定義された関数またはブロックの中からしか見えません。つまり、局所静的変数は、生存期間は大域変数と同じで、スコープ（可視範囲）は普通の局所変数と同じである変数です。

変数をプログラムの実行開始から実行終了まで存在し続けさせるには、大域変数にする方法もあります。しかし、大域変数は、プログラムのどこからも見えてしまいます。どこからでも見えることは、時には便利ですが、時には思いもよらないところで書き換えられてバグを見つけるのが困難になる原因になることもあります。プログラムの実行開始から実行終了まで存在し続ける必要があるが、特定の関数の中でしか使わない変数が必要な場合、局所静的変数にすることが一つの方法です。

プログラム 3.14 では、素数の表は各関数に引数として渡されるので、表の変数そのものが各関数から見える必要はありません。したがって、main 関数の局所静的変数にするほうが、意図せぬ書き換えをしてしまう危険を減らすことができます。

プログラム 3.15 は、エラトステネスの篩の実装の別バージョンです。プログラム 3.5（67 ページ）では素数判定の表を大域変数にしていますが、このプログラムでは、`main` 関数の局所静的変数にして、下請けの関数に渡しています。

プログラム 3.14 素数のリスト（試し割り法）（別バージョン）

```
#include <stdio.h>

#define TABLE_SIZE      78498

void    init_table(int *, int);
void    print_primes(const int *, int);
int     none_is_divisible(const int *, int, int);

void
init_table(int *table, int size)
{
        int     i;
        int     n;

        n = 2;
        for (i = 0; i < size; i ++) {
                while (!none_is_divisible(table, i, n)) {
                        n ++;
                }
                table[i] = n;
                n ++;
        }
}

int
none_is_divisible(const int *table, int size, int x)
{
        int     i;

        for (i = 0; i < size; i ++) {
                if (x % table[i] == 0)
                        return 0;
        }
        return 1;
}

void
print_primes(const int *table, int size)
{
        int     i;

        for (i = 0; i < size; i ++) {
                printf("%d\n", table[i]);
        }
}

int
main()
{
        static int      prime_table[TABLE_SIZE];

        init_table(prime_table, TABLE_SIZE);
        print_primes(prime_table, TABLE_SIZE);
        return 0;
}
```

プログラム 3.15 素数のリスト（エラトステネスの篩）（別バージョン）

```
#include <stdio.h>

#define TABLE_SIZE      1000000

void    init_table(char *, int);
void    print_primes(const char *, int);

void
init_table(char *table, int size)
{
        int     i, j;

        table[0] = table[1] = 0;
        for (i = 2; i < size; i ++) {
                table[i] = 1;
        }
        for (i = 2; i * i < size; i ++) {
                if (table[i]) {
                        for (j = i * i; j < size; j += i) {
                                table[j] = 0;
                        }
                }
        }
}

void
print_primes(const char *table, int size)
{
        int     i;

        for (i = 0; i < size; i ++) {
                if (table[i]) {
                        printf("%d\n", i);
                }
        }
}

int
main()
{
        static char     prime_table[TABLE_SIZE];

        init_table(prime_table, TABLE_SIZE);
        print_primes(prime_table, TABLE_SIZE);
        return 0;
}
```

3.3 配列の配列

3.3.1 配列の配列

C で縦横に並んだデータを扱うには、配列の配列を使うと便利です。たとえば、図 3.13 の上のコードは、int の長さ 4 の配列の長さ 3 の配列を定義しています。いいかえれば、int が 4 個並んだものが 3 個並んだものです。図 3.13 の下に図示したように格納されます。

プログラマは、あたかも図 3.14 のように縦横に並んでいるとみなしてプログラムを書いても、困らないことが普通です。

なお、配列の配列のことを二次元の配列と呼ぶこともあります。正確な表現ではありませんが、許容できる表現でしょう。本当は図 3.13 の下のようにメモリ中に一列の配置されているのですが、図 3.14 のように縦横に並んでいると無理矢理思い込んでも困らない場合のみ、二次元の配列と呼んでも困りません。

```
int a[3][4];
```

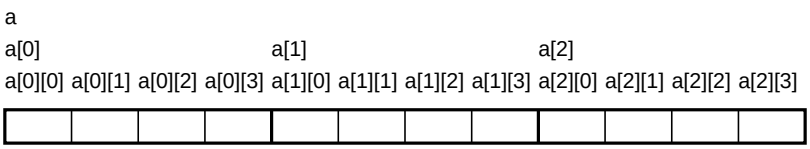


図 3.13: 配列の配列の例

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

図 3.14: 縦横に並んでいるとみなす

残念ながら、いつも困らないのではありません。配列の配列が二次元の配列ではないことが面倒の原因となる微妙な状況も、たまにはあります。そういう状況に出会ったときだけは違いを思い出してください。

二次元の表から行を切り出すことと列を切り出すことが非対称になることは、その例です。図 3.13 にある配列の配列 a を図 3.14 のような二次元の表と思って扱っているとします。行を配列として切り出すことは可能です。たとえば、第 1 行は a[1] で表せます（最初の行を第 0 行とする）。しかし、列を配列として切り出すことはできません。図 3.13 を見るとわかるように、列の要素はメモリ上でとびとびに並んでいるからです。

例：横の和と縦の和

プログラム 3.16 は、横 10、縦 6 の長方形状に並ぶ整数値を入力し、各行（横方向）の和を上から順に出力し、各列（縦方向）の和を左から順に出力します。

データの個数が 60 個と少ないので、配列の配列を局所変数にしています。データの個数がもっと多いとき

プログラム 3.16 横の和と縦の和

```
#include <stdio.h>

#define HSIZE   10
#define VSIZE   6

int
main()
{
        int     a[VSIZE][HSIZE];
        size_t  i, j;
        int     sum;

        for (i = 0; i < VSIZE; i ++) {
                for (j = 0; j < HSIZE; j ++) {
                        scanf("%d", &a[i][j]);
                }
        }
        for (i = 0; i < VSIZE; i ++) {
                sum = 0;
                for (j = 0; j < HSIZE; j ++) {
                        sum += a[i][j];
                }
                printf("%d\n", sum);
        }
        for (j = 0; j < HSIZE; j ++) {
                sum = 0;
                for (i = 0; i < VSIZE; i ++) {
                        sum += a[i][j];
                }
                printf("%d\n", sum);
        }
        return 0;
}
```

は、局所変数にするとメモリ不足を起こす危険があるので[*17]、他の方法を考える必要があります[*18]。

3.3.2 配列を指すポインタ

式に現れる配列の名前がその先頭の要素へのポインタを表す C の特別ルールは、配列の配列にも適用されます。したがって、

```
int a[3][4];
```

と定義されているとき、式の中に a が現れると、a の最初の要素へのポインタを表します。それは、int の長さ 4 の配列を指すポインタです。int を指すポインタとは別のものです。

配列を指すポインタ型の変数を定義するには、たとえば、

[*17] 多くのシステムでは、局所変数に使えるメモリは大域変数に使えるメモリよりもはるかに小さいです。そのため、巨大な配列を局所変数として確保すると、全体としてはメモリが足りていてもメモリ不足で異常終了することがあります。

[*18] 他の方法の候補としては次のものが有力です。

- 大域変数か局所静的変数として確保する。
- malloc() か calloc() で確保する。

他の方法もあります。

```
int     (*p)[4];
```

のように書きます。

この場合、丸括弧 () は必須です。丸括弧がないと意味が変わるので注意しましょう[*19]。

配列の配列 a と配列を指すポインタ p が上記の通りに定義されているとき、

```
p = a;
```

は正しい文です。この文が実行されると、変数 p には a の最初の要素を指すポインタが代入されます。

例：横の和と縦の和

プログラム 3.17 は、プログラム 3.16 の計算部分を独立した関数にしたものです。

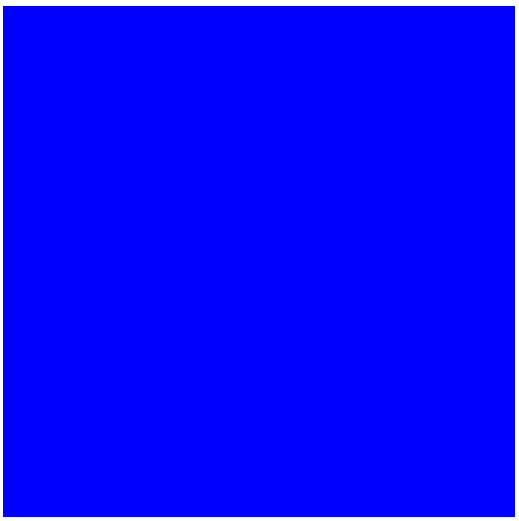
仮引数の型については、

```
const int (*a)[HSIZE]
```

を

```
const int a[][HSIZE]
```

と書いても、ほぼ同じ意味になります。プログラム 3.17 をそのように書き換えたものがプログラム 3.18 です。



[*19] 丸括弧なしの

```
int     *p[4];
```

は、int を指すポインタからなる長さ 4 の配列の定義になります。これは、*よりも [] が優先順位が高いので、括弧を補うと

```
int     *(p[4]);
```

と解釈されるからです。

プログラム 3.17 横の和と縦の和（計算を独立）

```
#include <stdio.h>

#define HSIZE   10
#define VSIZE   6

void
vsum(const int (*a)[HSIZE], int *s)
{
        int     sum;
        size_t  i, j;

        for (i = 0; i < VSIZE; i ++) {
                sum = 0;
                for (j = 0; j < HSIZE; j ++) {
                        sum += a[i][j];
                }
                s[i] = sum;
        }
}

void
hsum(const int (*a)[HSIZE], int *s)
{
        int     sum;
        size_t  i, j;

        for (j = 0; j < HSIZE; j ++) {
                sum = 0;
                for (i = 0; i < VSIZE; i ++) {
                        sum += a[i][j];
                }
                s[j] = sum;
        }
}

int
main()
{
        int     a[VSIZE][HSIZE];
        int     x[VSIZE];
        int     y[HSIZE];
        size_t  i, j;

        for (i = 0; i < VSIZE; i ++) {
                for (j = 0; j < HSIZE; j ++) {
                        scanf("%d", &a[i][j]);
                }
        }
        vsum(a, x);
        hsum(a, y);
        for (i = 0; i < VSIZE; i ++) {
                printf("%d\n", x[i]);
        }
        for (j = 0; j < HSIZE; j ++) {
                printf("%d\n", y[j]);
        }
        return 0;
}
```

プログラム 3.18 横の和と縦の和（計算を独立）（プログラム 3.17 から仮引数の型の書き方だけを変更）

```
#include <stdio.h>

#define HSIZE   10
#define VSIZE   6

void
vsum(const int a[][HSIZE], int s[])
{
        int     sum;
        size_t  i, j;

        for (i = 0; i < VSIZE; i ++) {
                sum = 0;
                for (j = 0; j < HSIZE; j ++) {
                        sum += a[i][j];
                }
                s[i] = sum;
        }
}

void
hsum(const int a[][HSIZE], int s[])
{
        int     sum;
        size_t  i, j;

        for (j = 0; j < HSIZE; j ++) {
                sum = 0;
                for (i = 0; i < VSIZE; i ++) {
                        sum += a[i][j];
                }
                s[j] = sum;
        }
}

int
main()
{
        int     a[VSIZE][HSIZE];
        int     x[VSIZE];
        int     y[HSIZE];
        size_t  i, j;

        for (i = 0; i < VSIZE; i ++) {
                for (j = 0; j < HSIZE; j ++) {
                        scanf("%d", &a[i][j]);
                }
        }
        vsum(a, x);
        hsum(a, y);
        for (i = 0; i < VSIZE; i ++) {
                printf("%d\n", x[i]);
        }
        for (j = 0; j < HSIZE; j ++) {
                printf("%d\n", y[j]);
        }
        return 0;
}
```

3.3.3 配列の配列の配列、……

配列、配列の配列が可能であるのと同様に、配列の配列の配列、配列の配列の配列の配列、配列の配列の配列の配列の配列、配列の配列の配列の配列の配列の配列の配列、……も可能です。それぞれ、三次元の配列、四次元の配列、五次元の配列、六次元の配列、……だと思って使うことができます。

```
int     a[3];                   // int の配列
int     b[20][3];               // int の配列の配列
int     c[10][20][3];           // int の配列の配列の配列
int     d[5][10][20][3];        // int の配列の配列の配列の配列
int     e[8][5][10][20][3];     // int の配列の配列の配列の配列の配列
int     f[2][8][5][10][20][3];  // int の配列の配列の配列の配列の配列の配列
```

対応して、配列を指すポインタが可能であるのと同様に配列の配列を指すポインタ、配列の配列の配列を指すポインタ、配列の配列の配列の配列を指すポインタ、配列の配列の配列の配列の配列を指すポインタ、……も可能です。

```
int     *p;                     // int を指すポインタ
int     (*q)[3];                // int の配列を指すポインタ
int     (*r)[20][3];            // int の配列の配列を指すポインタ
int     (*s)[10][20][3];        // int の配列の配列の配列を指すポインタ
int     (*t)[5][10][20][3];     // int の配列の配列の配列の配列を指すポインタ
int     (*u)[8][5][10][20][3];  // int の配列の配列の配列の配列の配列を指すポインタ
```

3.4 文字と文字列

この節では C での文字列処理の解説をしますが、昔はともかく、今は、文字列処理が主のプログラムを C で書くことはお勧めしません。今となっては、C よりも文字列処理に適したプログラミング言語が多数存在します。

3.4.1 文字

C で文字を扱うには、その文字コードを使って整数値として処理します。C には、厳密な意味では文字型は存在しません。

文字定数

文字を’（一重引用符）で囲むと、その文字の文字コード（整数値）を表します。以下はその例です。

’a’　a の文字コード
’#’　# の文字コード
’3’　3 の文字コード　(数値の 3 とは異なる)

文字コードを直接書き込むこともできます。'\八進' と'\x 十六進' が使えます。

'\075'　八進法で 075　（十進法で 61）
'\x3d'　十六進法で 3d　（十進法で 61）

引用符や逆斜線との干渉を避けるための特殊な表現があります。

'\''　'（一重引用符）の文字コード
'\"'　"（二重引用符）の文字コード
'\\'　\（逆斜線）の文字コード
'\?'　?（疑問符）の文字コード

制御文字を表すための特殊な表現があります。

'\n'　改行の文字コード
'\r'　復帰の文字コード
'\b'　後退の文字コード
'\t'　水平タブの文字コード
'\v'　垂直タブの文字コード
'\f'　紙送りの文字コード
'\a'　警報の文字コード
'\0'　ヌル文字の文字コード

一般には、文字コードをプログラム中に直接書くのではなく、文字定数を使うほうが、移植性の良いプログラムになります。文字コードはシステムによって異なる可能性があります。

なお、数字の文字コードは順番に並んでいることが C の言語仕様で決まっています。つまり、以下の表の上段の各式は下段の対応する式と同じ値をもちます。

'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
'0' + 1	'0' + 2	'0' + 3	'0' + 4	'0' + 5	'0' + 6	'0' + 7	'0' + 8	'0' + 9

アルファベットについてはそのような保証はないので注意してください。アルファベットの文字コードが順番に並んでいるかどうかはシステムによります。移植生が重視される場合は、アルファベットの文字コードが順番に並んでいることを前提としてプログラミングしてはいけません。

3.4.2 文字の入出力

文字入力

getchar() は、標準入力から一文字入力してその文字コードを返す関数です。5.1.1 節で説明します。

文字出力

putchar() は、文字コードを指定して標準出力に対応する文字を出力する関数です。5.1.1 節で説明します。

文字を scanf で入力する

scanf() で文字を入力するには %c を使います。対応する引数は char * です。つまり。格納先は char です。

```
char    c;
scanf("%c", &c);
```

文字を printf で出力する

printf() で文字を出力するには %c を使います。対応する引数は int です。scanf() と printf() で文字の扱いについて型の食い違いがありますので、注意してください。

```
int     c = 'a';
printf("%c\n", c);
```

3.4.3 C の標準ライブラリにある文字処理の関数

文字種判定

使用には、#include <ctype.h> が必要。文字コードを int 型で受け取り、条件が真ならば 0 以外を返し、偽ならば 0 を返す。

isalpha(c)	c がアルファベットならば真
islower(c)	c がアルファベット小文字ならば真
isupper(c)	c がアルファベット大文字ならば真
isdigit(c)	c が数字ならば真
isalnum(c)	c がアルファベットまたは数字ならば真
isxdigit(c)	c が 16 進数字ならば真
isgraph(c)	c が図形文字（空白' 'を除く）ならば真
isprint(c)	c が図形文字（空白' 'を含む）ならば真
iscntrl(c)	c が制御文字ならば真
isspace(c)	c が空白文字ならば真
isblank(c)	c が間隔文字ならば真
ispunct(c)	c が特殊文字ならば真

文字変換

使用には、#include <ctype.h> が必要。文字コードを int 型で受け取り、変換後の文字コードを int 型で返す。

toupper(c)	c を対応する大文字に変換
tolower(c)	c を対応する小文字に変換

3.4.4 文字列

文字列と char の配列

C での文字列は char の配列で表されます。ただし、文字列の終端にヌル文字を置いて、どこまで続くかがわかるようにする約束です。C は、文字列型を特に用意していはいません。

文字列リテラルは"（二重引用符）で囲まれた文字の並びで表現します。引用符や逆斜線との干渉を避けるための特殊な表現や制御文字を表すための特殊な表現は、文字列でも使えます。文字列リテラルでも、ヌル文字で終端された char の配列が自動的に作られます。さらに、式の中に配列の名前が現れたら先頭の要素へのポインタと解釈されるルールがここでも適用されますので、最初の char 型のデータを指すポインタになります。

たとえば、プログラム中に "abc" と書くと、図 3.15 のような長さ 4 の char の配列が作られます。ヌル文字が加わるので、配列の長さは"で囲まれた文字数よりも 1 多いことに注意してください。

'a'	'b'	'c'	'\0'

図 3.15: 文字列の実体の例

文字列の連結

C には、文字列を単に並べると勝手に連結してくれる機能があります。たとえば、

```
"abcd" "xyz"
```

は

```
"abcdxyz"
```

と同じ意味になります。

この機能は、#define と組み合わせて使うと便利です。文字列の共通部分をマクロにするのです。たとえば、プログラム 3.19 は、マクロ展開の結果、プログラム 3.20 のようになり、文字列が連結されてプログラム 3.21 と同じ意味になります。プログラム 3.19 は 1 行目の "ayaya" を "hoyoyo" に変えるだけで name0 の初期値と name1 の初期値を、それぞれ、"hoyoyo0" と "hoyoyo1" にいっせいに変更できるので、最初からプログラム 3.21 のように書くよりも便利です。

プログラム 3.19

```
#define PREFIX "ayaya"
const char      *name0 = PREFIX "0";
const char      *name1 = PREFIX "1";
```

プログラム 3.20

```
const char      *name0 = "ayaya" "0";
const char      *name1 = "ayaya" "1";
```

プログラム 3.21

```
const char      *name0 = "ayaya0";
const char      *name1 = "ayaya1";
```

文字列を scanf で入力する

scanf() で文字列を入力するには %s を使います。入力は空白文字で区切られます。対応する引数は char * で、入力した文字列を格納するための配列の先頭を指します。

生の %s は危険なので、通常は長さの上限を指定する形式で使います。

以下は使用例です。入力した文字列を格納する配列の長さが入力する文字数の上限よりも 1 大きいことに注意してください。ヌル文字も格納する必要があるためです。

```
char    buf[17];

        （中略）

scanf("%16s", buf);
```

文字列を printf で出力する

printf() で文字列を出力するには %s を使います。対応する引数は char * で、出力する文字列が格納されている配列の先頭を指します。

以下は使用例です。

```
char    buf[17];

        （中略）

printf("ayaya%shoyoyo\n", buf);
```

printf() で %s を使って文字列を出力するときは、出力するものは文字列であること、つまり、必ずヌル文字で終端されていることが前提となります。そうでない可能性がある場合は、生の %s ではなく長さの上限つきのものを使うと安全になります。その例です。

```
char    buf[17];

        （中略）

printf("ayaya%.16shoyoyo\n", buf);
```

文字列のよくある処理

C の文字列は char の配列ですので、C の文字列の処理は配列の処理をそのまま応用すればよいだけです。

プログラム 3.22 は、文字列 str に含まれる文字を最初から順に処理する典型的なパターンです。文字列に含まれる文字について最初から順に関数 use に適用させています。この例では、文字列 str は書き換えないことに注意してください。

プログラム 3.22 文字列のよくある処理

```
void
use_each(const char *str)
{
        const char      *p;
        for (p = str; *p != '\0'; p ++) {
                use(*p);
        }
}
```

C の標準ライブラリに文字列の長さを求める関数がありますが（3.4.5 節を参照)、なんらかの理由でこれが使えず、自力で書かなければならないときは、たとえば、プログラム 3.23 かプログラム 3.24 のように関数定義します。

プログラム 3.23 文字列の長さを求める

```
#include <stddef.h>

size_t
strlen(const char *str)
{
        size_t  counter = 0;
        for (p = str; *p != '\0'; p ++) {
                counter ++;
        }
        return counter;
}
```

プログラム 3.24 文字列の長さを求める

```
#include <stddef.h>

size_t
strlen(const char *str)
{
        const char      *p = str;
        while (*p != '\0') {
                p ++;
        }
        return p - str;
}
```

他の例もあげましょう。プログラム 3.25 は、文字列に含まれるアルファベットの個数を求める関数の定義です。プログラム 3.26 は、文字列に最初に出現するアルファベットの位置を求める関数の定義です。

プログラム 3.25 文字列に含まれるアルファベットの個数を求める

```
#include <stddef.h>
#include <ctype.h>

size_t
alphacount(const char *str)
{
        size_t  counter;
        const char      *p;

        counter = 0;
        for (p = str; *p != '\0'; p ++) {
                if (isalpha(*p)) {
                        counter ++;
                }
        }
        return counter;
}
```

プログラム 3.26 文字列に出現する最初のアルファベットを見つける

```
#include <stddef.h>
#include <ctype.h>

char *
alphafind(const char *str)
{
        char    *p;

        for (p = str; *p != '\0'; p ++) {
                if (isalpha(*p))
                        return (char *)p;
        }
        return NULL;
}
```



3.4.5 C の標準ライブラリにある文字列処理の関数の一部

文字列の長さ

使用には、`#include <string.h>` が必要。返値の型は `size_t` 。

`strlen(str)` str が指す文字列の長さを返す。

文字列の比較

使用には、`#include <string.h>` が必要。返値は表 3.1 のとおり。返値の型は `int` 。

`strcmp(s1, s2)` s1 が指す文字列と s2 が指す文字列を辞書式順序で比較する。
`strncmp(s1, s2, n)` s1 が指す文字列と s2 が指す文字列の先頭の最大 n 文字を辞書式順序で比較する。

表 3.1: 文字列比較関数の返値

比較結果	返値
辞書式で前	負の整数
等しい	0
辞書式で後	正の整数

文字列から探す

使用には、`#include <string.h>` が必要。返値は表 3.2 のとおり。返値の型は `char *` 。

`strchr(s, c)` s が指す文字列での文字 c の最初の出現を探す。
`strrchr(s, c)` s が指す文字列での文字 c の最後の出現を探す。
`strstr(s1, s2)` s1 が指す文字列での s2 が指す文字列の最初の出現を探す。

表 3.2: 文字列から探す関数の返値

	返値
見つかったとき	出現したところを指すポインタ
見つからなかったとき	NULL

文字列の先頭の特定文字の個数

使用には、`#include <string.h>` が必要。返値は `size_t` 型。

`strspn(s1, s2)` s1 が指す文字列の先頭から始まる、s2 が指す文字列に**含まれる**文字だけからなる部分の最大長を返す。
`strcspn(s1, s2)` s1 が指す文字列の先頭から始まる、s2 が指す文字列に**含まれない**文字だけからなる部分の最大長を返す。

3.4.6 ハノイの塔

「ハノイの塔」として知られる有名なパズル[*20]を解くプログラムを書きましょう。

ハノイの塔の概要を説明します。柱が三本立っています。柱には穴の空いた大きさの違う何枚かの円盤を刺されています。大きな円盤は必ず小さな円盤よりも下に置かれます。円盤を一つずつ動かして、与えられた初期状態から目標状態にもっていくパズルです。

たとえば、図 3.16 の上の状態のように 7 枚の円盤がすべて左の柱（以下、柱 A と呼びます）に刺さっていて、その全部を中の柱（以下、柱 B と呼びます）へ移動させて図 3.16 の下の状態にする手順を考えましょう。もちろん、右の柱（以下、柱 C と呼びます）を中継に使ってかまいません。

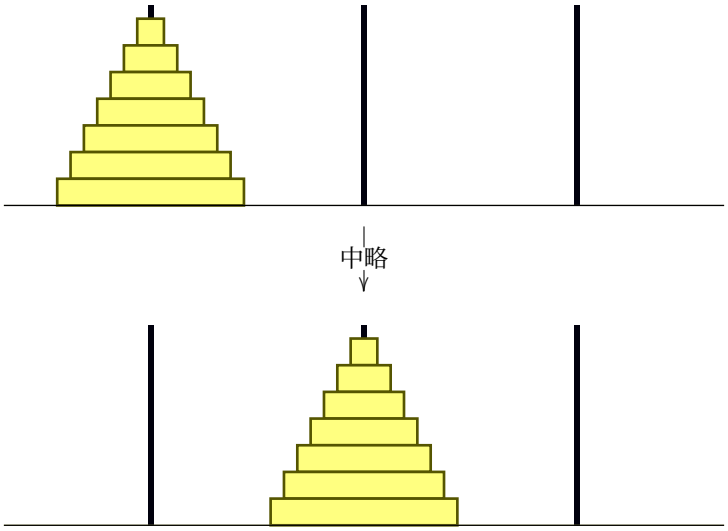


図 3.16: ハノイの塔（円盤 7 枚）

手順を分解して考えてみましょう。説明を簡単にするため、円盤に小さい順に通し番号をふります。
円盤 1〜7 を柱 A から柱 B に柱 C を中継に使って動かすには：

- 円盤 1〜6 を柱 A から柱 C へ柱 B を中継に使って動かす
- 円盤 7 を柱 A から柱 B へ動かす
- 円盤 1〜6 を柱 C から柱 B へ柱 A を中継に使って動かす

となります（図 3.17）。

7 枚の円盤を動かす手順は、上の 6 枚の円盤を動かす手順に帰着できることがわかりました。同じように考えると、6 枚の円盤を動かす手順は、上の 5 枚の円盤を動かす手順に帰着できます。5 枚の円盤を動かす手順は、……。これは再帰呼び出しで実装できますね。

[*20] このパズルは実在の都市ハノイとは無関係ですが、最初に販売された時のパッケージに「La tour d'Hanoï」と書かれていたことから、今も「ハノイの塔」と呼ばれています。考案者の名を冠して「リュカの塔」と呼ぶのがふさわしいところですが、「ハノイの塔」があまりに広まっていることから、やむなくそう呼ぶことにします。

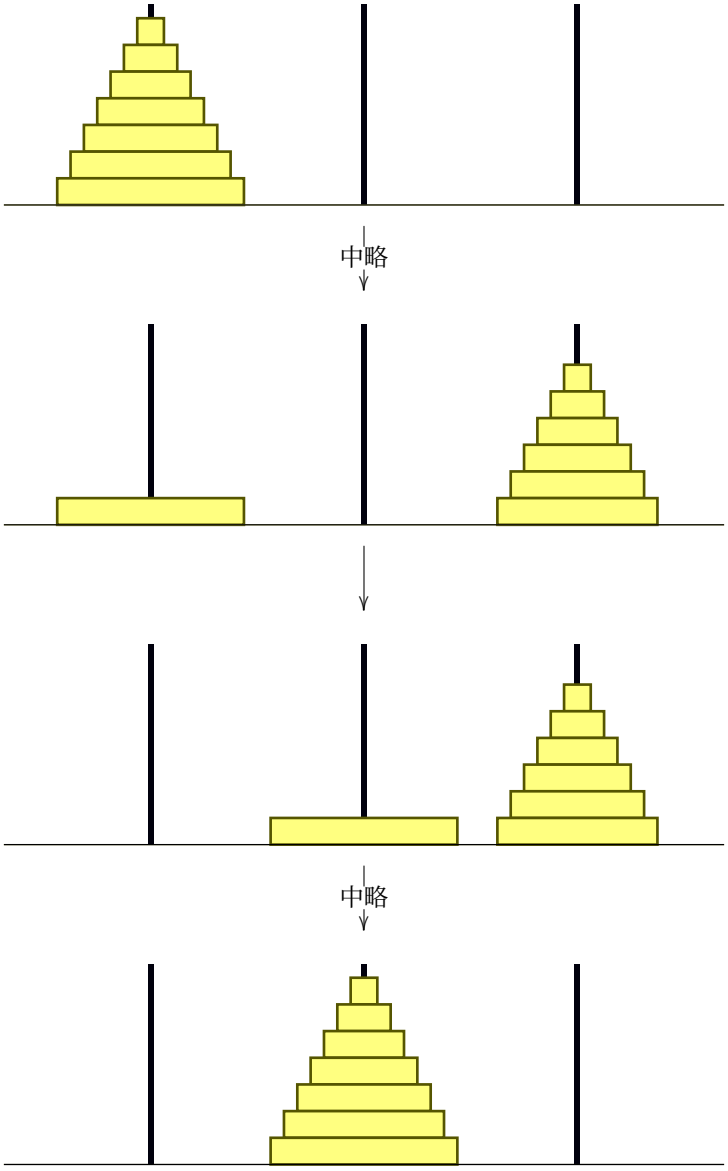


図 3.17: ハノイの塔（ステップをちょっと分解）

以上の方針で実装したのがプログラム 3.27 です。

ここまでは、どの円盤もどの柱からどの柱へも動かせる場合を考えましたが、円盤の動きにもっと制限をつけることもできます。円盤の動きを $A \underset{\leftarrow}{\rightarrow} B \underset{\leftarrow}{\rightarrow} C$ に限定した場合がプログラム 3.28 で、$A \underset{\leftarrow}{\leftarrow} B \underset{\leftarrow}{\leftarrow} C$ に限定した場合がプログラム 3.29 です。

プログラム 3.27 ハノイの塔を解くプログラム

```
#include <stdio.h>

void
hanoi_print_step(int n, const char *source, const char *target)
{
        printf("Move #%d from %s to %s.\n", n, source, target);
}

void
hanoi_print(int n, const char *source, const char *target, const char *spare)
{
        if (n > 0) {
                hanoi_print(n - 1, source, spare, target);
                hanoi_print_step(n, source, target);
                hanoi_print(n - 1, spare, target, source);
        }
}

int
main()
{
        int     n;
        scanf("%d", &n);
        hanoi_print(n, "A", "B", "C");
        return 0;
}
```

プログラム 3.28 巡回的順方向の制限つきハノイの塔を解くプログラム

```
#include <stdio.h>

void    hanoix_print(int, const char *, const char *, const char *);
void    hanoiy_print(int, const char *, const char *, const char *);

void
hanoi_print_step(int n, const char *source, const char *target)
{
        printf("Move #%d from %s to %s.\n", n, source, target);
}

void
hanoix_print(int n, const char *source, const char *target, const char *spare)
{
        if (n > 0) {
                hanoiy_print(n - 1, source, spare, target);
                hanoi_print_step(n, source, target);
                hanoiy_print(n - 1, spare, target, source);
        }
}

void
hanoiy_print(int n, const char *source, const char *target, const char *spare)
{
        if (n > 0) {
                hanoiy_print(n - 1, source, target, spare);
                hanoi_print_step(n, source, spare);
                hanoix_print(n - 1, target, source, spare);
                hanoi_print_step(n, spare, target);
                hanoiy_print(n - 1, source, target, spare);
        }
}

int
main()
{
        int     n;
        scanf("%d", &n);
        hanoix_print(n, "A", "B", "C");
        return 0;
}
```

プログラム 3.29 巡回的逆方向の制限つきハノイの塔を解くプログラム

```
#include <stdio.h>

void    hanoix_print(int, const char *, const char *, const char *);
void    hanoiy_print(int, const char *, const char *, const char *);

void
hanoi_print_step(int n, const char *source, const char *target)
{
        printf("Move #%d from %s to %s.\n", n, source, target);
}

void
hanoix_print(int n, const char *source, const char *target, const char *spare)
{
        if (n > 0) {
                hanoiy_print(n - 1, source, spare, target);
                hanoi_print_step(n, source, target);
                hanoiy_print(n - 1, spare, target, source);
        }
}

void
hanoiy_print(int n, const char *source, const char *target, const char *spare)
{
        if (n > 0) {
                hanoiy_print(n - 1, source, target, spare);
                hanoi_print_step(n, source, spare);
                hanoix_print(n - 1, target, source, spare);
                hanoi_print_step(n, spare, target);
                hanoiy_print(n - 1, source, target, spare);
        }
}

int
main()
{
        int     n;
        scanf("%d", &n);
        hanoiy_print(n, "A", "B", "C");
        return 0;
}
```

第 4 章

構造体

4.1 構造体

4.1.1 構造体

C の構造体は、いくつかのデータを束ねて一つのデータにしたものです。束ねられたおのおののデータは**メンバ**と呼ばれ、名前がつけられます。

プログラム 4.1 構造体の型宣言の例

```
struct ayaya {
        int     n;
        double  x;
        char    *s;
};
```

構造体の型宣言は、文字通り、型を宣言するだけです。**実体は作りません**。

コード 4.2 は、構造体型の変数 a と構造体を指すポインタ型の変数 p を定義する例です。

プログラム 4.2 構造体型の変数の定義と構造体を指すポインタ型の変数の定義の例

```
struct ayaya    a;
struct ayaya    *p;
```

構造体を指すポインタ型の変数を定義しただけでは、構造体の実体は作られません。そのことは、int を指すポインタや double を指すポインタと変わりません。

```
struct ayaya  a;
struct ayaya  *p = &a;
```

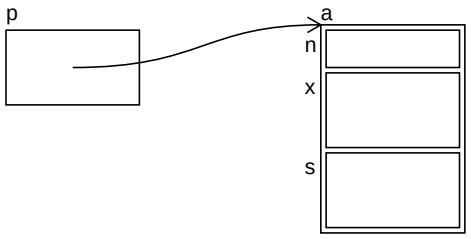


図 4.1

構造体のメンバは

構造体型の式.**メンバ名**

でアクセスできます。プログラム 4.1, 4.2 の組み合わせでは、

```
a.n    a.x    a.s    (*p).n    (*p).x    (*p).s
```

などが使えます[*1]。

なお、

（***構造体へのポインタ型の式**）.**メンバ名**

は

構造体へのポインタ型の式->**メンバ名**

と略記できます。たとえば、p->n は (*p).n の略記です。

C には、型に別名をつける機能があります。変数宣言の前に typedef をつけた形の typedef 宣言で行いま

プログラム 4.3 typedef 宣言の例

```
typedef int     Number;
typedef int     *Pointer;
```

す。たとえば、プログラム 4.3 のように書くと、以後、int 型に Number という別名がつき、int を指すポインタ型に Pointer という別名がつきます。

typedef 宣言は、構造体型を struct なしの一語で表すことにも使えます。プログラム 4.4 は struct ayaya に Ayaya という別名を与えている例です。

プログラム 4.4 構造体の型宣言の例

```
struct ayaya {
        int     n;
        double  x;
        char    *s;
};
typedef struct ayaya    Ayaya;
```

構造体宣言と typedef 宣言をまとめることもできます。プログラム 4.4 の構造体宣言と typedef 宣言をまとめたものが、プログラム 4.5 です。

プログラム 4.1 をプログラム 4.4 かプログラム 4.5 に書き換ると、変数宣言は、プログラム 4.2 のように書いてもプログラム 4.6 のように書いてもよくなります。

プログラム 4.7 のように typedef 宣言することもできます。この場合、変数宣言をプログラム 4.2 のように書くことはできなくなり、プログラム 4.6 のように書かなくてはならなくなります。

[*1] ここで、(*p).n などの丸括弧を省略することはできません。. は * よりも優先順位が高いからです。*p.n だと *(p.n) とみなされ、メンバを取ることのできないもののメンバを取ろうとした構文エラーになります。

プログラム 4.5 構造体の型宣言の例

```
typedef struct ayaya {
        int     n;
        double  x;
        char    *s;
}       Ayaya;
```

プログラム 4.6 変数の定義の例

```
Ayaya   a;
Ayaya   *p;
```

プログラム 4.7 構造体の型宣言の例

```
typedef struct {
        int     n;
        double  x;
        char    *s;
}       Ayaya;
```

4.1.2 構造体を引数や返値とする関数

構造体は、関数の引数にも返値にもできます。それについては、基本的には int や double と変わりません。

プログラム 4.8 はその例です。関数 ptoo は、struct polar 型の引数で値を受け取り、計算結果を struct orthogonal 型の返値で返しています。

4.1.3 構造体へのポインタを引数とする関数

巨大な構造体を関数の引数や返値にすると、処理に時間がかかることがあります。ポインタを活用するとそれを避けることができることがあります。

プログラム 4.9 はその例です。関数 ptoo は、struct polar を指す const ポインタ型の第一引数と struct orthogonal を指すポインタ型の第二引数をとります。第一引数の指す先から値を取り出し、計算結果を第二引数の指す先に代入します。

4.1.4 構造体のメンバの型いろいろ

構造体のメンバが構造体

プログラム 4.10、プログラム 4.11

構造体のメンバが配列

プログラム 4.12、プログラム 4.13

構造体のメンバが構造体を指すポインタ

プログラム 4.14

構造体のメンバが構造体を指すポインタの配列

プログラム 4.15

プログラム 4.8 平面上の点の極座標から直交座標への変換（関数の引数と返値が構造体）

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

struct polar {
        double  r, theta;
};

struct orthogonal {
        double  x, y;
};

struct orthogonal
ptoo(struct polar p)
{
        struct orthogonal       result;

        result.x = p.r * cos(p.theta);
        result.y = p.r * sin(p.theta);
        return result;
}

int
main()
{
        struct polar    p_in;
        struct orthogonal       p_out;

        scanf("%lf%lf", &p_in.r, &p_in.theta);
        p_out = ptoo(p_in);
        printf("%f %f\n", p_out.x, p_out.y);
        return 0;
}
```

プログラム 4.9 平面上の点の極座標から直交座標への変換（関数の引数が構造体へのポインタ）

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

struct polar {
        double  r, theta;
};

struct orthogonal {
        double  x, y;
};

void
ptoo(const struct polar *p, struct orthogonal *result)
{
        result->x = p->r * cos(p->theta);
        result->y = p->r * sin(p->theta);
}

int
main()
{
        struct polar    p_in;
        struct orthogonal       p_out;

        scanf("%lf%lf", &p_in.r, &p_in.theta);
        ptoo(&p_in, &p_out);
        printf("%f %f\n", p_out.x, p_out.y);
        return 0;
}
```

プログラム 4.10 ジェルゴンヌ三角形のジェルゴンヌ三角形を計算するプログラム (1)

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

struct point {
        double  x, y;
};

struct triangle {
        struct point    A, B, C;
};

struct triangle
gergonne_triangle(struct triangle ref)
{
        double  a = hypot(ref.B.x - ref.C.x, ref.B.y - ref.C.y);
        double  b = hypot(ref.C.x - ref.A.x, ref.C.y - ref.A.y);
        double  c = hypot(ref.A.x - ref.B.x, ref.A.y - ref.B.y);
        double  sa = (b + c - a) / 2;
        double  sb = (c + a - b) / 2;
        double  sc = (a + b - c) / 2;
        struct triangle result;
        result.A.x = fma(ref.B.x, sc, ref.C.x * sb) / a;
        result.A.y = fma(ref.B.y, sc, ref.C.y * sb) / a;
        result.B.x = fma(ref.C.x, sa, ref.A.x * sc) / b;
        result.B.y = fma(ref.C.y, sa, ref.A.y * sc) / b;
        result.C.x = fma(ref.A.x, sb, ref.B.x * sa) / c;
        result.C.y = fma(ref.A.y, sb, ref.B.y * sa) / c;
        return result;
}

int
main()
{
        struct triangle t, t1, t2;

        scanf("%lf%lf%lf%lf%lf%lf",
              &t.A.x, &t.A.y, &t.B.x, &t.B.y, &t.C.x, &t.C.y);
        t1 = gergonne_triangle(t);
        t2 = gergonne_triangle(t1);
        printf("%15.10f %15.10f %15.10f %15.10f %15.10f %15.10f\n",
               t2.A.x, t2.A.y, t2.B.x, t2.B.y, t2.C.x, t2.C.y);
        return 0;
}
```

プログラム 4.11 ジェルゴンヌ三角形のジェルゴンヌ三角形を計算するプログラム (2)

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

struct point {
        double  x, y;
};

struct triangle {
        struct point    A, B, C;
};

void
gergonne_triangle(const struct triangle *ref, struct triangle *result)
{
        double  a = hypot(ref->B.x - ref->C.x, ref->B.y - ref->C.y);
        double  b = hypot(ref->C.x - ref->A.x, ref->C.y - ref->A.y);
        double  c = hypot(ref->A.x - ref->B.x, ref->A.y - ref->B.y);
        double  sa = (b + c - a) / 2;
        double  sb = (c + a - b) / 2;
        double  sc = (a + b - c) / 2;
        result->A.x = fma(ref->B.x, sc, ref->C.x * sb) / a;
        result->A.y = fma(ref->B.y, sc, ref->C.y * sb) / a;
        result->B.x = fma(ref->C.x, sa, ref->A.x * sc) / b;
        result->B.y = fma(ref->C.y, sa, ref->A.y * sc) / b;
        result->C.x = fma(ref->A.x, sb, ref->B.x * sa) / c;
        result->C.y = fma(ref->A.y, sb, ref->B.y * sa) / c;
}

int
main()
{
        struct triangle t, t1, t2;

        scanf("%lf%lf%lf%lf%lf%lf",
              &t.A.x, &t.A.y, &t.B.x, &t.B.y, &t.C.x, &t.C.y);
        gergonne_triangle(&t, &t1);
        gergonne_triangle(&t1, &t2);
        printf("%15.10f %15.10f %15.10f %15.10f %15.10f %15.10f\n",
               t2.A.x, t2.A.y, t2.B.x, t2.B.y, t2.C.x, t2.C.y);
        return 0;
}
```

プログラム 4.12 ジェルゴンヌ三角形のジェルゴンヌ三角形を計算するプログラム (3)

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

struct point {
        double  coordinates[2];
};

struct triangle {
        struct point    vertices[3];
};

struct triangle
gergonne_triangle(struct triangle ref)
{
        size_t  i, j, k, n;
        double  a[3];
        for (i = 0, j = 1, k = 2; i < 3; j = k, k = i, i ++) {
                double  d[2];
                for (n = 0; n < 2; n ++) {
                        d[n] = ref.vertices[j].coordinates[n]
                                - ref.vertices[k].coordinates[n];
                }
                a[i] = hypot(d[0], d[1]);
        }
        double  sa[3];
        for (i = 0, j = 1, k = 2; i < 3; j = k, k = i, i ++) {
                sa[i] = (a[j] + a[k] - a[i]) / 2;
        }
        struct triangle result;
        for (i = 0, j = 1, k = 2; i < 3; j = k, k = i, i ++) {
                for (n = 0; n < 2; n ++) {
                        result.vertices[i].coordinates[n]
                                = fma(ref.vertices[j].coordinates[n], sa[k],
                                      ref.vertices[k].coordinates[n] * sa[j]) / a[i];
                }
        }
        return result;
}

int
main()
{
        struct triangle t[3];

        scanf("%lf%lf%lf%lf%lf%lf",
              &t[0].vertices[0].coordinates[0], &t[0].vertices[0].coordinates[1],
              &t[0].vertices[1].coordinates[0], &t[0].vertices[1].coordinates[1],
              &t[0].vertices[2].coordinates[0], &t[0].vertices[2].coordinates[1]);
        t[1] = gergonne_triangle(t[0]);
        t[2] = gergonne_triangle(t[1]);
        printf("%15.10f %15.10f %15.10f %15.10f %15.10f %15.10f\n",
               t[2].vertices[0].coordinates[0], t[2].vertices[0].coordinates[1],
               t[2].vertices[1].coordinates[0], t[2].vertices[1].coordinates[1],
               t[2].vertices[2].coordinates[0], t[2].vertices[2].coordinates[1]);
        return 0;
}
```

プログラム 4.13 ジェルゴンヌ三角形のジェルゴンヌ三角形を計算するプログラム (4)

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

struct point {
        double  coordinates[2];
};

struct triangle {
        struct point    vertices[3];
};

void
gergonne_triangle(const struct triangle *ref, struct triangle *result)
{
        size_t  i, j, k, n;
        double  a[3];
        for (i = 0, j = 1, k = 2; i < 3; j = k, k = i, i ++) {
                double  d[2];
                for (n = 0; n < 2; n ++) {
                        d[n] = ref->vertices[j].coordinates[n]
                                - ref->vertices[k].coordinates[n];
                }
                a[i] = hypot(d[0], d[1]);
        }
        double  sa[3];
        for (i = 0, j = 1, k = 2; i < 3; j = k, k = i, i ++) {
                sa[i] = (a[j] + a[k] - a[i]) / 2;
        }
        for (i = 0, j = 1, k = 2; i < 3; j = k, k = i, i ++) {
                for (n = 0; n < 2; n ++) {
                        result->vertices[i].coordinates[n]
                                = fma(ref->vertices[j].coordinates[n], sa[k],
                                      ref->vertices[k].coordinates[n] * sa[j]) / a[i];
                }
        }
}

int
main()
{
        struct triangle t[3];

        scanf("%lf%lf%lf%lf%lf%lf",
              &t[0].vertices[0].coordinates[0], &t[0].vertices[0].coordinates[1],
              &t[0].vertices[1].coordinates[0], &t[0].vertices[1].coordinates[1],
              &t[0].vertices[2].coordinates[0], &t[0].vertices[2].coordinates[1]);
        gergonne_triangle(&t[0], &t[1]);
        gergonne_triangle(&t[1], &t[2]);
        printf("%15.10f %15.10f %15.10f %15.10f %15.10f %15.10f\n",
               t[2].vertices[0].coordinates[0], t[2].vertices[0].coordinates[1],
               t[2].vertices[1].coordinates[0], t[2].vertices[1].coordinates[1],
               t[2].vertices[2].coordinates[0], t[2].vertices[2].coordinates[1]);
        return 0;
}
```

プログラム 4.14 ジェルゴンヌ三角形のジェルゴンヌ三角形を計算するプログラム (5)

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

struct point {
        double  x, y;
};

struct triangle {
        struct point    *A, *B, *C;
};

void
gergonne_triangle(const struct triangle *ref, struct triangle *result)
{
        double  a = hypot(ref->B->x - ref->C->x, ref->B->y - ref->C->y);
        double  b = hypot(ref->C->x - ref->A->x, ref->C->y - ref->A->y);
        double  c = hypot(ref->A->x - ref->B->x, ref->A->y - ref->B->y);
        double  sa = (b + c - a) / 2;
        double  sb = (c + a - b) / 2;
        double  sc = (a + b - c) / 2;
        result->A->x = fma(ref->B->x, sc, ref->C->x * sb) / a;
        result->A->y = fma(ref->B->y, sc, ref->C->y * sb) / a;
        result->B->x = fma(ref->C->x, sa, ref->A->x * sc) / b;
        result->B->y = fma(ref->C->y, sa, ref->A->y * sc) / b;
        result->C->x = fma(ref->A->x, sb, ref->B->x * sa) / c;
        result->C->y = fma(ref->A->y, sb, ref->B->y * sa) / c;
}

int
main()
{
        struct point    A, B, C;
        scanf("%lf%lf%lf%lf%lf%lf",
              &A.x, &A.y, &B.x, &B.y, &C.x, &C.y);
        struct triangle t = {&A, &B, &C};
        struct point    A1, B1, C1;
        struct triangle t1 = {&A1, &B1, &C1};
        gergonne_triangle(&t, &t1);
        struct point    A2, B2, C2;
        struct triangle t2 = {&A2, &B2, &C2};
        gergonne_triangle(&t1, &t2);
        printf("%15.10f %15.10f %15.10f %15.10f %15.10f %15.10f\n",
               A2.x, A2.y, B2.x, B2.y, C2.x, C2.y);
        return 0;
}
```

プログラム 4.15 ジェルゴンヌ三角形のジェルゴンヌ三角形を計算するプログラム (6)

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

struct point {
        double  coordinates[2];
};

struct triangle {
        struct point    *vertices[3];
};

void
gergonne_triangle(const struct triangle *ref, struct triangle *result)
{
        size_t  i, j, k, n;
        double  a[3];
        for (i = 0, j = 1, k = 2; i < 3; j = k, k = i, i ++) {
                double  d[2];
                for (n = 0; n < 2; n ++) {
                        d[n] = ref->vertices[j]->coordinates[n]
                                - ref->vertices[k]->coordinates[n];
                }
                a[i] = hypot(d[0], d[1]);
        }
        double  sa[3];
        for (i = 0, j = 1, k = 2; i < 3; j = k, k = i, i ++) {
                sa[i] = (a[j] + a[k] - a[i]) / 2;
        }
        for (i = 0, j = 1, k = 2; i < 3; j = k, k = i, i ++) {
                for (n = 0; n < 2; n ++) {
                        result->vertices[i]->coordinates[n]
                                = fma(ref->vertices[j]->coordinates[n], sa[k],
                                      ref->vertices[k]->coordinates[n] * sa[j]) / a[i];
                }
        }
}

int
main()
{
        struct point    p[3][3];
        struct triangle t[3] = {
                {&p[0][0], &p[0][1], &p[0][2]},
                {&p[1][0], &p[1][1], &p[1][2]},
                {&p[2][0], &p[2][1], &p[2][2]},
        };

        scanf("%lf%lf%lf%lf%lf%lf",
              &p[0][0].coordinates[0], &p[0][0].coordinates[1],
              &p[0][1].coordinates[0], &p[0][1].coordinates[1],
              &p[0][2].coordinates[0], &p[0][2].coordinates[1]);
        gergonne_triangle(&t[0], &t[1]);
        gergonne_triangle(&t[1], &t[2]);
        printf("%15.10f %15.10f %15.10f %15.10f %15.10f %15.10f\n",
               p[2][0].coordinates[0], p[2][0].coordinates[1],
               p[2][1].coordinates[0], p[2][1].coordinates[1],
               p[2][2].coordinates[0], p[2][2].coordinates[1]);
        return 0;
}
```

第 5 章

標準入出力ライブラリ

5.1 標準入力と標準出力

この節で紹介するライブラリ関数は、いずれも、使用するには

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

が必要です。

5.1.1 文字単位入出力

文字単位入力

標準入力から 1 文字入力し、int 型の変数 c に文字コードを格納するには、次のような式を使います。

```
c = getchar()
```

getchar() は標準入力から 1 文字入力し、その文字コードを返します。入力する文字には、空白も改行などの制御文字も含まれます。ただし、入力が尽きた場合や入力エラーがあった場合は、EOF を返してそのことを知らせます。

上の例で変数 c は int 型でなくてはなりません。EOF が char 型の範囲外の値である可能性があるからです。

文字単位出力

int 型の変数 c に格納された文字コードの 1 文字を標準出力に出力するには、次のような式を使います。

```
putchar(c)
```

この式の値は、出力に成功したときは出力した文字の文字コードを返します。つまり、c の値を返します。何らかの原因で出力に失敗したときは EOF を返します。

次の式も同じ動作をしますが、返り値が異なります。

```
printf("%c", c)
```

printf() は、出力に成功したときは出力したバイト数を返します。何らかの原因で出力に失敗したときは何

か負の整数値を返します。

使用例

プログラム 5.1 は、標準入力からテキストを入力してそのまま標準出力に出力します。一文字入力してはその文字を出力することを、入力が尽きるまで繰り返します。これは、文字単位入出力によるテキスト処理の基本パターンです。文字単位入出力によるテキスト処理のプログラムの多くは、これに書き足す形で実装できます。

プログラム 5.1 で変数 c が int 型であることに注意してください。間違って char 型にすると、getchar() が EOF を返したときに想定外の動作をすることがありえます。EOF は char で表現できる値の範囲の外であるかもしれないからです。

以降のプログラムも同じ理由で getchar() の返値を格納する変数は int 型にしています。

プログラム 5.2 は、標準入力からテキストを入力して、#は@に書き換え、その他はそのまま標準出力に出力します。入力された文字が#であるかそうでないかで挙動を変える方法で実装しています。#であるときは@を出力し、そうでないときは入力した文字をそのまま出力します。

プログラム 5.3 は、標準入力からテキストを入力して、アルファベットと数字と空白文字のみを残し、他は削除して標準出力に出力します。入力された文字がアルファベットか数字か空白文字である場合とそうでない場合で挙動を変える方法で実装しています。アルファベットか数字か空白文字のときは入力した文字をそのまま出力し、そうでないときは何もしません。改行文字も空白文字に含まれるので、改行も正しく処理されます。

プログラム 5.1 入力をそのまま出力する（文字単位処理、エラー処理を省略）

```
#include <stdio.h>

int
main()
{
        int     c;
        while ((c = getchar()) != EOF) {
                putchar(c);
        }
        return 0;
}
```

プログラム 5.2 #を@に書き換える（文字単位処理、エラー処理を省略）

```
#include <stdio.h>

int
main()
{
        int     c;
        while ((c = getchar()) != EOF) {
                if (c == '#') {
                        putchar('@');
                } else {
                        putchar(c);
                }
        }
        return 0;
}
```

プログラム 5.3 アルファベットと数字と空白文字のみを残し、他の文字を削除する（文字単位処理、エラー処理を省略）

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

int
main()
{
        int     c;
        while ((c = getchar()) != EOF) {
                if (isalnum(c) || isspace(c)) {
                        putchar(c);
                }
        }
        return 0;
}
```

プログラム 5.4 大文字を小文字に変換する（文字単位処理、エラー処理を省略）

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

int
main()
{
        int     c;
        while ((c = getchar()) != EOF) {
                putchar(tolower(c));
        }
        return 0;
}
```

プログラム 5.5 各行の末尾に$を追加する（文字単位処理、エラー処理を省略）

```
#include <stdio.h>

int
main()
{
        int     c;
        while ((c = getchar()) != EOF) {
                if (c == '\n') {
                        putchar('$');
                }
                putchar(c);
        }
        return 0;
}
```

プログラム 5.4 は、標準入力からテキストを入力して、アルファベットのみ対応する小文字に変換し、その他はそのまま標準出力に出力します。アルファベットの小文字への変換に、標準ライブラリ関数 `tolower` を使っています。

プログラム 5.5 は、標準入力からテキストを入力して、各行の末尾に `$` を追加して標準出力に出力します。改行も一つの文字であることに注意してください。行の末尾は改行文字の直前です。したがって、改行文字が入力されたらそれを出力する前に `$` を出力すればよいのです。

プログラム 5.6 各行の先頭に^を追加する（文字単位処理、エラー処理を省略）

```
#include <stdio.h>

int
main()
{
        int     c, c0 = '\n';
        while ((c = getchar()) != EOF) {
                if (c0 == '\n') {
                        putchar('^');
                }
                putchar(c);
                c0 = c;
        }
        return 0;
}
```

プログラム 5.7 行数を数える（文字単位処理、エラー処理を省略）

```
#include <stdio.h>

int
main()
{
        int     c;
        long    counter;

        counter = 0;
        while ((c = getchar()) != EOF) {
                if (c == '\n') {
                        counter ++;
                }
        }
        printf("%ld\n", counter);
        return 0;
}
```

プログラム 5.6 は、標準入力からテキストを入力して、各行の先頭に ^ を追加して標準出力に出力します。

ここでは、一つ前の文字を常に覚えておく技法を使っています。最初の行を除き、行の先頭は改行文字の直後です。したがって、行の先頭に ^ を追加して出力するには、一つ前の文字を覚えておいて、それが改行文字であるときに現在の文字を出力する前に ^ を出力すれば良いのです。プログラム 5.6 では、変数 c0 が一つ前の文字を覚えておく役割を果たしています。while ループの本体の最後で、変数 c の値を変数 c0 に退避させることで、c0 に一回前での c の値を残しておくことができます。したがって、c0 は c と同じく int 型にします。

さらにもう一つ、巧妙な工夫をしています。一つ前の文字を覚えておくための変数 c0 の初期値を改行文字'\n' にしています。これで、最初の行についても特別扱いが不要になります。最初の行の先頭、つまり、入力全体の最初のついても、あたかもその直前に改行文字があったかのように処理できるからです。

プログラム 5.7 は、標準入力から入力したテキストの行数を数えて、それを標準出力に出力するプログラムです。テキストにおいて、改行文字は行の末端を示します。ということは、テキストに出現する改行文字の数を数えれば、行の数がわかります。ゼロクリアされたカウンタを用意し、改行文字に出会うたびにカウンタを 1 増やし、テキストの入力がすべて終わってからカウンタの値を出力すれば良いのです。

プログラム 5.8 は、標準入力から入力したテキストの行ごとにそこの含まれる文字数（改行文字を除く）を

プログラム 5.8 改行文字を除く文字数を行ごとに数える（文字単位処理、エラー処理を省略）

```
#include <stdio.h>

int
main()
{
        int     c;
        long    counter;

        counter = 0;
        while ((c = getchar()) != EOF) {
                if (c == '\n') {
                        printf("%ld\n", counter);
                        counter = 0;
                } else {
                        counter ++;
                }
        }
        return 0;
}
```

数えて、行ごとにそれを標準出力に出力するプログラムです。テキストにおいて、改行文字は行の末端を示します。そこで、改行文字が出現するまで文字を数えれば、その行の文字数がわかります。ゼロクリアされたカウンタを用意し、改行文字以外が出現したらカウンタを 1 増やし、改行文字に出会うたびにカウンタの値を出力してカウンタをクリアすれば良いのです。

5.1.2 行単位入出力

行単位入力

標準入力から 1 行入力し、str が指す長さ size の領域に文字列として格納するには、次の式が使えます。

```
fgets(str, size, stdin)
```

入力は最大 size − 1 文字です。格納先はヌル文字'\0' で終端されます。ただし、入力の途中でエラーが生じた場合は終端されるとは限りません。

この式の値は、入力に成功したときは str の値そのものです。入力が尽きて一文字も入力できなかったときは NULL です。入力の途中で何らかのエラーが生じたときも NULL です。

改行文字'\n' を読み込むと、そこで一回分の fgets() の入力は終りです。また、入力が尽きたり入力エラーが起きたときもそこで入力は終ります。scanf() で %s を使う場合と違って、空白文字で区切ることはしません。

改行文字が入力されたときは、改行文字も格納されます。したがって、改行文字が格納されないのは、次のいずれかが生じたときです。

- 行が長過ぎた
- 改行文字なしで入力の終りに達した
- 改行文字を読む前にエラーが生じた

廃止された gets について

標準入力から一行入力する関数として昔は gets がありましたが、使うと確実にプログラムが脆弱になる関数なので廃止されました。gets を安全に使う方法は全く存在しません。多くの処理系では諸般の事情で今でも gets を残しています。古い教科書などに gets の使用例が載っていることもあります。しかし、決して使ってはいけません。

行単位出力

str が指す文字列を標準出力に出力するには、次の式が使えます。

```
fputs(str, stdout)
```

この式の値は、出力に成功したときは何か非負の整数値です。失敗したときは EOF です。
次の式も同じ動作をしますが、返り値が異なります。

```
printf("%s", str)
```

printf() は、出力に成功したときは出力したバイト数を返します。何らかの原因で出力に失敗したときは何か負の整数値を返します。

puts について

標準出力に一行出力する関数としては、puts もあります。特に危険ではないので廃止されてはいません。ただ、fputs(str, stdout) と puts(str) の挙動が違い、不慣れな人は混乱しやすく使いづらいです。

使用例

プログラム 5.9 は、標準入力からテキストを入力してそのまま標準出力に出力します。

プログラム 5.9 入力をそのまま出力する（行単位処理、エラー処理を省略）

```
#include <stdio.h>

#define BUFSIZE 132

int
main()
{
        char    buf[BUFSIZE];

        while (fgets(buf, BUFSIZE, stdin) != NULL) {
                fputs(buf, stdout);
        }
        return 0;
}
```

プログラム 5.10 は、標準入力からテキストを入力して、アルファベットのみ対応する小文字に変換し、その他はそのまま標準出力に出力します。

なお、プログラム 5.9 もプログラム 5.10 も、想定よりも長い行が入力されても、たまたま、期待通りの動作をします。これは、「たまたま」であって、一般には想定より長い行が入力すると期待とは異なる動作をします。行単位入出力を使うときには想定より長い行への対策が一般には必要です。

プログラム 5.10 入力の大文字を小文字に変換し、その他はそのまま出力する（文字単位処理、エラー処理を省略）

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

#define BUFSIZE 132

int
main()
{
        char    buf[BUFSIZE];
        char    buf_out[BUFSIZE];
        const char      *p;
        char    *q;

        while (fgets(buf, BUFSIZE, stdin) != NULL) {
                q = buf_out;
                for (p = buf; *p != '\0'; p ++) {
                        *q = tolower(*p);
                        q ++;
                }
                *q = '\0';
                fputs(buf_out, stdout);
        }
        return 0;
}
```

5.1.3 フォーマットつき入出力

フォーマットつき入力

すでに何度も出てきていますが、標準入力からフォーマット付き入力をするには、scanf() が使えます。

```
scanf(フォーマット, ...)
```

のように使います。

scanf() は int を返す関数です。返り値は、実際に入力して変換に成功した項目の個数です。ただし、変換が行われる前に入力が失敗したときには、特別な値 EOF が返されます。

たとえば、

```
n = scanf("%d%d", &x, &y);
```

を実行して実際に入力して変換に成功したのが 1 個のとき、つまり、x には入力値が代入されたが y には変更がなかったとき、n には 1 が代入されます。

フォーマットつき出力

すでに何度も出てきていますが、標準出力にフォーマット付き出力をするには、printf() が使えます。

```
printf(フォーマット, ...)
```

のように使います。

printf() は int を返す関数です。返り値は、出力した文字数です。ただし、出力が失敗したときには、何か負の値が返されます。

5.2 ファイル

この節で紹介するライブラリ関数は、いずれも、使用するには

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

が必要です。

5.2.1 ファイルアクセス

ファイルのオープン

fopen() はファイルをオープンして、FILE 構造体へのポインタを返します。オープンに失敗したときは、NULL を返します。

```
FILE    *fp;
```

と定義されているとして、

```
fp = fopen(ファイル名, モード);
```

のように使います。ファイル名はオープンしたいファイルの名前を表す文字列を指すポインタです。モードはどのようにオープンするかを示す文字列です。表 5.1 の意味を持ちます。システムによっては、この表にないものにも対応していることがあります。

表 5.1: fopen のモード

モード（文字列）	対象	（読み書き）	存在しないとき	存在するとき
r	テキストファイル	読み取りモード	失敗	（読むだけ）
w	テキストファイル	書き込みモード	生成	長さ 0 に切り詰め
wx	テキストファイル	書き込みモード	生成	失敗
a	テキストファイル	書き込みモード	生成	末尾に追加
r+	テキストファイル	更新モード	失敗	（中身を残して先頭から）
w+	テキストファイル	更新モード	生成	長さ 0 に切り詰め
w+x	テキストファイル	更新モード	生成	失敗
a+	テキストファイル	更新モード	生成	末尾に追加

標準入出力

通常は、プログラムの実行を開始した時点で表 5.2 の三つがオープンされています。標準入力は通常は特に指示がなければキーボードから入力しますが、入力リダイレクションで変更できます。標準出力は通常は特に指示がなければ画面に出力しますが、出力リダイレクションで変更できます。

表 5.2: 標準入出力

stdin	標準入力
stdout	標準出力
stderr	標準エラー出力

標準入力・標準出力・標準エラー出力

UNIX 系 OS（macOS や Linux も含みます）では、標準入力と標準出力はリダイレクションの影響を受けますが、標準エラー出力は通常の出力リダイレクションの影響を受けません。エラーメッセージは標準出力ではなく標準エラー出力に出力すると、出力リダイレクションをしていてもエラーメッセージが画面に表示されて、便利です。

ファイルのクローズ

fclose() はファイルをクローズします。

```
FILE    *fp;
```

と定義されているとして、

```
fclose(fp);
```

のように使います。

ファイルは、以後読み書きのどちらもすることがなくなったら、クローズするのが原則です。プログラムの実行中に同時にオープンしたままにしておけるファイルの個数には上限があります。そのため、多数のファイルをオープンしたままクローズせずに放置することを繰り返すと、そのうちに上限に達してファイルがそれ以上オープンできなくなり、プログラムが正常に動作しなくなります。それを防ぐために、使い終えたファイルはこまめにクローズしましょう。

プログラムの実行終了の直前まで読み書きのいずれかを行う場合は例外です。プログラムの終了時にまだクローズしていないファイルがあれば、自動的にクローズするようになっています。状況によっては、プログラムで明示的にクローズするよりも、自動的にクローズする機能を頼ったほうが便利なこともあります。

プログラム 5.11 （文字単位処理、エラー処理を省略）

```
#include <stdio.h>

int
main()
{
        int     c;

        while ((c = getchar()) != EOF) {
                putchar(c);
        }
        fclose(stdin);
        fclose(stdout);
        fclose(stderr);
        return 0;
}
```

標準入力と標準出力と標準エラー出力も、プログラムの終了時に自動的にクローズするファイルに含まれます。プログラマが明示的にこれらをクローズする必要がないのは、そのためです。必要がなくなったときに明示的にクローズしても害はありません。プログラム 5.11 は標準入力と標準出力と標準エラー出力をわざわざクローズしている例です。

5.2.2 エラーメッセージ

エラーメッセージを出力するには、ライブラリ関数 perror() が便利です。引数として NULL を渡すと、直前に起きたエラーに対応するエラーメッセージが標準エラー出力に出力されます。引数として文字列を指すポインタを渡すと、その文字列、:、直前に起きたエラーに対応するエラーメッセージの順に、標準エラー出力に出力されます。

5.2.3 文字単位入出力

文字単位入力

ファイル構造体へのポインタ fp に対応するファイルから 1 文字入力し、int 型の変数 c に文字コードを格納するには、次のような式を使います。

```
c = getc(fp)
```

getc() はファイルから 1 文字入力し、その文字コードを返します。ただし、入力が尽きた場合や入力エラーがあった場合は、EOF を返してそのことを知らせます。

上の例で変数 c を char 型にしてはいけません。システムによっては、EOF が char 型の範囲外の値である可能性があるからです。

実は、getchar() は getc(stdin) と同じです。

文字単位出力

int 型の変数 c に格納された文字コードの 1 文字を、ファイル構造体へのポインタ fp に対応するファイルに出力するには、次のような式を使います。

```
putc(c, fp)
```

この式の値は、出力に成功したときは出力した文字の文字コードを返します。つまり、c の値を返します。何らかの原因で出力に失敗したときは EOF を返します。

実は、putchar(c) は putc(c, stdout) と同じです。

使用例

プログラム 5.12 は、ファイル ayaya の中身をそのままファイル hoyoyo に複写します。文字単位入出力の繰り返しで実装しています。

11 行目の

```
in_fp = fopen(in_file, "r");
```

で、ファイル ayaya を読み出しモードでオープンします。
16 行目の

```
out_fp = fopen(out_file, "w");
```

で、ファイル hoyoyo を書き込みモードでオープンします。ファイル hoyoyo が存在しなければ新たに作成され、すでに存在していれば元の内容を消去して上書きされます。16 行目を

```
out_fp = fopen(out_file, "a");
```

に書き換えると、ファイル hoyoyo が存在しなければ新たに作成され、すでに存在していれば元の内容の後ろに追記するようになります。

```
out_fp = fopen(out_file, "wx");
```

に書き換えると、ファイル hoyoyo が存在しなければ新たに作成され、すでに存在していればオープンに失敗します。
22〜24 行目の while 文

```
while ((c = getc(in_fp)) != EOF) {
        putc(c, out_fp);
}
```

が処理の本体です。1 文字入力してはそれをそのまま出力することを、入力が尽きるまで繰り返します。入力元と出力先は、さきほどオープンしたファイル、すなわち、それぞれ、ayaya と hoyoyo です。

プログラム 5.12 ファイルからファイルに複写する（文字単位処理、エラー処理を一部省略）

```
#include <stdio.h>

int
main()
{
        FILE    *in_fp, *out_fp;
        const char      *in_file = "ayaya";
        const char      *out_file = "hoyoyo";
        int     c;

        in_fp = fopen(in_file, "r");
        if (in_fp == NULL) {
                perror(in_file);
                return 1;
        }
        out_fp = fopen(out_file, "w");
        if (out_fp == NULL) {
                perror(out_file);
                fclose(in_fp);
                return 1;
        }
        while ((c = getc(in_fp)) != EOF) {
                putc(c, out_fp);
        }
        fclose(in_fp);
        fclose(out_fp);
        return 0;
}
```

25〜26 行目でオープンしたファイルをクローズします。

5.2.4 行単位入出力

行単位入力

ファイル構造体へのポインタ fp に対応するファイルから 1 行入力し、str が指す長さ size の領域に文字列として格納するには、次の式が使えます。

```
fgets(str, size, fp)
```

入力は最大 size − 1 文字です。格納先はヌル文字'\0' で終端されます。ただし、入力の途中でエラーが生じた場合は終端されるとは限りません。

この式の値は、入力に成功したときは str の値そのものです。入力が尽きて一文字も入力できなかったときは NULL です。入力の途中で何らかのエラーが生じたときも NULL です。

改行文字'\n' を読み込むと、そこで一回分の fgets() の入力は終りです。また、入力が尽きたり入力エラーが起きたときもそこで入力は終ります。scanf() で %s を使う場合と違って、空白文字で区切ることはしません。

改行文字が入力されたときは、改行文字も格納されます。したがって、改行文字が格納されないのは、次のいずれかが生じたときです。

- 行が長過ぎた
- 改行文字なしで入力の終りに達した
- 改行文字を読む前にエラーが生じた

行単位出力

str が指す文字列を、ファイル構造体へのポインタ fp に対応するファイルに出力するには、次の式が使えます。

```
fputs(str, fp)
```

この式の値は、出力に成功したときは何か非負の整数値です。失敗したときは EOF です。

使用例

プログラム 5.13 は、ファイル ayaya の中身をそのままファイル hoyoyo に複写します。行単位入出力の繰り返しで実装しています。行が長すぎて fgets 一回で行全体を入力できない場合の特別処理を省略していますが、このプログラムでは、たまたま、うまくいきます。長すぎる行があってもなにごともなかったかのように最終的に入力ファイルの中身がすべてそのまま出力ファイルに書き込まれます。

5.2.5 フォーマット付き入出力

フォーマット付き入力

ファイルからフォーマット付き入力をするには、fscanf() が使えます。

プログラム 5.13 ファイルからファイルに複写する（行単位処理、エラー処理を一部省略）

```
#include <stdio.h>

#define BUFSIZE 132

int
main()
{
        FILE    *in_fp, *out_fp;
        const char      *in_file = "ayaya";
        const char      *out_file = "hoyoyo";
        char    buf[BUFSIZE];

        in_fp = fopen(in_file, "r");
        if (in_fp == NULL) {
                perror(in_file);
                return 1;
        }
        out_fp = fopen(out_file, "w");
        if (out_fp == NULL) {
                perror(out_file);
                fclose(in_fp);
                return 1;
        }
        while (fgets(buf, BUFSIZE, in_fp) != NULL) {
                fputs(buf, out_fp);
        }
        fclose(in_fp);
        fclose(out_fp);
        return 0;
}
```

```
fscanf(fp, フォーマット, ...)
```

のように使います。

fscanf() は int を返す関数です。返り値は、実際に入力して変換に成功した項目の個数です。ただし、変換が行われる前に入力が失敗したときには、特別な値 EOF が返されます。

実は、scanf(フォーマット, ...) は fscanf(stdin, フォーマット, ...) と同じです。

フォーマット付き出力

ファイルにフォーマット付き出力をするには、fprintf() が使えます。

```
fprintf(fp, フォーマット, ...)
```

のように使います。

fprintf() は int を返す関数です。返り値は、出力した文字数です。ただし、出力が失敗したときには、何か負の値が返されます。

実は、printf(フォーマット, ...) は fprintf(stdout, フォーマット, ...) と同じです。

使用例

プログラム 5.14 は、ファイル ayaya から整数値を読み込み、その値をファイル hoyoyo に書き込みます。

プログラム 5.14 ファイルからファイルに整数値を書き写す（エラー処理を一部省略）

```
#include <stdio.h>

int
main()
{
        FILE    *in_fp, *out_fp;
        const char      *in_file = "ayaya";
        const char      *out_file = "hoyoyo";
        long    x;

        in_fp = fopen(in_file, "r");
        if (in_fp == NULL) {
                perror(in_file);
                return 1;
        }
        out_fp = fopen(out_file, "w");
        if (out_fp == NULL) {
                perror(out_file);
                fclose(in_fp);
                return 1;
        }
        while (fscanf(in_fp, "%ld", &x) == 1) {
                fprintf(out_fp, "%ld\n", x);
        }
        fclose(in_fp);
        fclose(out_fp);
        return 0;
}
```

5.3 char の配列への操作

5.3.1 フォーマット付き入出力

フォーマット付き入力

フォーマット付き入力は、メモリ上の文字列から行うこともできます。

```
sscanf(文字列を指すポインタ, フォーマット, ...)
```

のように使います。

sscanf() は int を返す関数です。返り値は、実際に入力して変換に成功した項目の個数です。ただし、変換が行われる前に入力が失敗したときには、特別な値 EOF が返されます。

フォーマット付き出力

フォーマット付き出力は、メモリ上の char の配列に対して文字列を書き込む形で行うこともできます。

```
snprintf(ポインタ, サイズ, フォーマット, ...)
```

のように使います。

書き込み先の終端には'\0'（ヌル文字）を書き込みます。その結果、書き込んだ結果はヌル文字で終端されている C の文字列になります。

書き込む文字数が サイズ − 1 を超えたときは、超えた分は捨てられます。その場合でも、ヌル文字による終端は行われます。

snprintf() は int を返す関数です。返り値は、サイズが十分に大きくければ出力したであろう文字数（末尾のヌル文字を除く）です。実際に出力した文字数ではありません。ただし、出力が失敗したときには、何か負の値が返されます。

snprintf() の返り値をサイズと比較することで、書き込めなくて捨てられた部分があるかどうかを判定できます。

sprintf について

フォーマットつきで char の配列に文字列を書き込む関数としては、snprintf の他に sprintf もありますが、新規に開発するプログラムでは snprintf を使ってください。わざわざ、sprintf を使う利点はありません。

sprintf は安易に使うとプログラムの脆弱性の原因になりえます。慎重に使えば、安全に使うことが不可能ではありません。ただ、sprintf が安全に使えるようにがんばるよりも、snprintf を使うほうが楽です。

snprintf は sprintf よりも標準に取り込まれたのが遅いので、古いプログラムで sprintf が使われていることはよくあります。それを見つけた場合、余裕があれば snprintf を使う形に書き直すと良いでしょう。

使用例

プログラム 5.15 は、標準入力からテキストを入力し、6 個の整数を含む行についてはその和を標準出力に出力します。そうでない行に対しては、* を出力します。行の長さは最大で 1021 文字（改行文字を除く）であると仮定しています。

fgets() で 1 行入力してそれを buf[] に格納し、sscanf() で 6 個の整数値を切り出そうとし、成功したら和を計算して出力し、切り出しに失敗したら*を出力することを、入力が尽きるまで繰り返しています。

プログラム 5.15　6 個の整数の和を求めることを繰り返す（エラー処理を簡略化）

```
#include <stdio.h>

#define BUFSIZE 1023

int
main()
{
        long long       u, v, w, x, y, z;
        char    buf[BUFSIZE];

        while (fgets(buf, BUFSIZE, stdin) != NULL) {
                if (sscanf(buf, "%lld%lld%lld%lld%lld%lld", &u, &v, &w, &x, &y, &z) == 6) {
                        printf("%lld\n", u + v + w + x + y + z);
                } else {
                        printf("*\n");
                }
        }
        return 0;
}
```

使用例

プログラム 5.16 は、標準入力からテキストを入力し、行ごとにバラバラにして、別々のファイルに出力します。出力ファイルの名前は、a0000, a0001, a0002, . . . のように、a の後ろに 4 桁の枝番を 0000 から順につけたものとなります。a9999 まで使い切ったら、残りの入力は無視します。

プログラム 5.17 とプログラム 5.18 はプログラム 5.16 の改良版です。ファイル名を a9999 まで使い切ったら、b0000 から始めます。b9999 まで使い切ったら、今度は c0000 から始めます。同様にアルファベット小文字を順に使い、z9999 までのファイル名を必要なら使います。z9999 まで使い切ったら、残りの入力は無視します。

プログラム 5.16 テキストを行ごとにバラバラにするプログラム

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

#define FILENAME_LENGTH 5
#define MAXLINENUM      10000
#define MAXLINELEN      1023

int
main()
{
        int     i;
        char    filename[FILENAME_LENGTH + 1];
        char    buf[MAXLINELEN + 2];
        FILE    *fp;

        for (i = 0; i < MAXLINENUM && fgets(buf, sizeof(buf), stdin) == buf; i ++) {
                snprintf(filename, sizeof(filename), "a%04d", i);
                fp = fopen(filename, "w");
                if (fp == NULL)
                        return 1;
                fputs(buf, fp);
                fclose(fp);
        }
        return 0;
}
```

プログラム 5.17 テキストを行ごとにバラバラにするプログラム（改良版）

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

#define FILENAME_LENGTH 5
#define MAXLINENUM      10000
#define MAXLINELEN      1023

void
split1()
{
        const char      *p;
        int     i;
        char    filename[FILENAME_LENGTH + 1];
        char    buf[MAXLINELEN + 2];
        FILE    *fp;

        for (p = "abcdefghijklmnopqrstuvwxyz"; *p != '\0'; p ++) {
                for (i = 0; i < MAXLINENUM; i ++) {
                        if (fgets(buf, sizeof(buf), stdin) != buf)
                                return;
                        snprintf(filename, sizeof(filename), "%c%04d", (int)*p, i);
                        fp = fopen(filename, "w");
                        if (fp == NULL)
                                exit(1);
                        fputs(buf, fp);
                        fclose(fp);
                }
        }
}

int
main() {
        split1();
        return 0;
}
```

プログラム 5.18 テキストを行ごとにバラバラにするプログラム（改良版）

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

#define FILENAME_LENGTH 5
#define MAXLINENUM      10000
#define MAXLINELEN      1023

const char      *const prefixes[26] = {
        "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m",
        "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z"
};

void
split1()
{
        int     n;
        int     i;
        char    filename[FILENAME_LENGTH + 1];
        char    buf[MAXLINELEN + 2];
        FILE    *fp;
        for (n = 0; n < sizeof(prefixes) / sizeof(prefixes[0]); n ++) {
                for (i = 0; i < MAXLINENUM; i ++) {
                        if (fgets(buf, sizeof(buf), stdin) != buf)
                                return;
                        snprintf(filename, sizeof(filename), "%s%04d", prefixes[n], i);
                        fp = fopen(filename, "w");
                        if (fp == NULL)
                                exit(1);
                        fputs(buf, fp);
                        fclose(fp);
                }
        }
}

int
main() {
        split1();
        return 0;
}
```

第 6 章

型いろいろ

6.1 型に名前をつける

変数定義の前に typedef をつけると、型の定義になります。変数定義での変数名が型定義の型名です。たとえば、

```
typedef long long      INTEGER;
```

は、long long 型に INTEGER という別名をつけます。

```
typedef long long      **INTEGER_PTR_PTR;
```

は、long long を指すポインタを指すポインタ型に INTEGER_PTR_PTR という別名をつけます。

```
typedef struct stella  STELLA;
```

は、struct stella 型に STELLA という別名をつけます。

型定義は、複雑な型を 1 語で表してプログラムの見通しをよくするのに使えます。たとえば、

```
typedef int     (*array_ptr)[10];
```

と定義しておくと、int を要素とする長さ 10 の配列を指すポインタに、array_ptr という別名を与えられます。これを使うと、int を要素とする長さ 10 の配列を指すポインタを要素とする長さ 100 の配列を

```
array_ptr       a[100];
```

で定義できます。typedef を使わないと、

```
int     (*a[100])[10];
```

と書かなくてはなりません。

6.2 標準型定義

6.2.1 整数

データの大きさに関連する整数

size_t	sizeof の値の型
ptrdiff_t	ポインタとポインタの差の型

ビット幅

int8_t	8 ビット符号つき整数	（持たない処理系もあり得る）
int16_t	16 ビット符号つき整数	（持たない処理系もあり得る）
int32_t	32 ビット符号つき整数	（持たない処理系もあり得る）
int64_t	64 ビット符号つき整数	（持たない処理系もあり得る）
uint8_t	8 ビット符号なし整数	（持たない処理系もあり得る）
uint16_t	16 ビット符号なし整数	（持たない処理系もあり得る）
uint32_t	32 ビット符号なし整数	（持たない処理系もあり得る）
uint64_t	64 ビット符号なし整数	（持たない処理系もあり得る）

int_least8_t	少なくとも 8 ビット幅の符号つき整数
int_least16_t	少なくとも 16 ビット幅の符号つき整数
int_least32_t	少なくとも 32 ビット幅の符号つき整数
int_least64_t	少なくとも 64 ビット幅の符号つき整数
uint_least8_t	少なくとも 8 ビット幅の符号なし整数
uint_least16_t	少なくとも 16 ビット幅の符号なし整数
uint_least32_t	少なくとも 32 ビット幅の符号なし整数
uint_least64_t	少なくとも 64 ビット幅の符号なし整数

int_fast8_t	少なくとも 8 ビット幅で最速の符号つき整数
int_fast16_t	少なくとも 16 ビット幅で最速の符号つき整数
int_fast32_t	少なくとも 32 ビット幅で最速の符号つき整数
int_fast64_t	少なくとも 64 ビット幅で最速の符号つき整数
uint_fast8_t	少なくとも 8 ビット幅で最速の符号なし整数
uint_fast16_t	少なくとも 16 ビット幅で最速の符号なし整数
uint_fast32_t	少なくとも 32 ビット幅で最速の符号なし整数
uint_fast64_t	少なくとも 64 ビット幅で最速の符号なし整数

intmax_t	その処理系で扱える最大のビット幅の符号つき整数
uintmax_t	その処理系で扱える最大のビット幅の符号なし整数