# 6 関数

## 6.1 分割統治法

もし大きくて複雑な問題が、複数の小さくて簡単な問題に分割でき、この小さな問題の解から、元の大きな問題の解が得られるのなら、プログラムを小さなモジュール (module) に分割して完成させることができる。これを分割統治法 (divide and conquer algorithm) という。あるいは、簡単な問題に分割を繰り返すことから段階的詳細化 (stepwise refinement) という。

分割統治法で分割した部分は、一連のまとまった処理としてプログラミングを行うと読みやすいプログラムが書ける。一連のまとまった処理を独立したモジュール (module) などという言い方もする。

一連のまとまった処理を C では、関数を用いて実現できる。

## 6.2 関数とは

関数とは、所定の処理を実行する (分割統治法を実現する) モジュールのことである。C で言う関数とは数学で言う関数よりも広い意味で用い、独立したモジュールあるいは分割統治法で分割した部分の処理全般を指す。

関数を使うには、関数を使いたい場所に関数の名前を書いておく。関数の名前をプログラム中に書くことを、「関数を呼び出す」と言ったりする。関数は関数呼び出しによって起動され、所定の仕事を実行して、その結果を返す。

## 6.3 関数の定義

これまで見てきたプログラムはどれも `main()` と呼ばれるたった 1 つの関数だけから構成されたプログラムであった。しかし、通常 C のプログラムはいくつかの関数から構成されている。ここではプログラマが自分で必要な関数を書く方法を解説する。

### 6.3.1 関数の例

以下のプログラムは、二つの引数ををを受け取ってその和を返す関数 `add()` を含むプログラムの例である。関数の名前は、C の予約語やシステム関数と重なっていないかぎりプログラマが自由につけることができる。

```
1  /****************************************
2  * add-func.c                            *
3  * 2 つの double 型数の和を返す関数の使用例    *
4  ****************************************/
5  #include <stdio.h>
6  double add(double x,double y);
7
8  int main(void)
9  {
10   double a;
11
12   a=add(1.2,2.3);
13   printf("a=%f\n",a);
14
15   return 0;
16 }
17
18 double add(double x,double y)
19 {
20   double z;
21
22   z=x+y;
23
24   return z;
25 }
```

このプログラムは 3 つの関数

- `main()` 関数 (8-16 行目)
- `printf()` 関数 (13 行目)
- 関数 `add()` (18-25 行目)

を持つ。利用者が定義する関数 `add()` に関係する文を順に説明する。

1. 6 行目
   `double add(double x,double y);`
   **関数原型** (function prototype) で、使用する関数の仮引数の型と変数名、返却値の型を宣言している。関数原型にはセミコロン; が必要である。
2. 12 行目
   `add(1.2,2.3)`
   は**関数呼び出し** (function call) である。関数 `add()` を実引数 1.2,2.3 を使って呼び出している。1.2 が `x` に 2.3 が `y` に渡される。
3. 12 行目
   `a=add(1.2,2.3);`
   `add(1.2,2.3)` で返される 3.5 を `double` 型変数 `a` に代入している。
4. 18-25 行目
   **関数定義** (function definition) である。関数 `add()` を定義している。
5. 19-25 行目
   **関数本体** (function body) である。関数 `add()` の本体である。
6. 20 行目
   `double z;`
   関数 `add()` の中で `double` 型変数 `z` を用いることを宣言している。
7. 24 行目
   `return z;`
   `return` 文により `z` の値を呼び出しもとに返し、関数 `add()` の実行を終了する。

`add-func.c` の実行結果は次に示す。

```
$ cc -Wall add-func.c -o add-func
$ ./add-func
a=3.500000
```

関数には、利用者が定義して使うものとライブラリとして用意されているライブラリ関数とがある。main() 関数は利用者が定義して使うものであるが、プログラム開始時に呼び出される特別な関数である。

関数を呼び出すときに関数にデータを渡すが、渡されるデータを実引数 (actual arguments/actual parameters)、データを受け取る関数側の変数を仮引数 (formal arguments/formal parameters) という。データの受け渡しのない関数 (したがって引数を持たない関数) もある。

関数は return 文により、返り値 (return value) を返し実行を終了する。値を返さない関数もある。値を返さない関数の宣言には void を用いる。

printf() 関数はライブラリ関数である。１つ以上の引数をとり、第 1 引数は文字列リテラル (char 型へのポインタ) である。hello.c の printf() 関数は、ただ 1 つの引数"Hello World\n"を持つ。int 型の値である転送文字数の 12 を返す。(空白、改行文字 \n もそれぞれ 1 文字と数える。) printf() 関数の戻り値は今まで使わずにきただけである。このことは、次のプログラムで確認することができる。

```
/* function printf() */
#include <stdio.h>

int main(void)
{
  int num;

  num=printf("Hello World\n");
  printf("転送文字数は%d です。\n",num);

  return 0;
}
```

### 6.3.2 return 文

return 文の構文規則は、

```
return 式;
```

または、

```
return;
```

である。

return 文は分岐文の一つである。実行中の関数の実行を終了し、制御をその呼び出し元に返す。

式を持つ return 文は、その式の値を関数呼び出し式の値 (返却値、または戻り値という) として呼び出し元に返す。式の型と関数の型が異なるときは、関数の型に変換される。

add-func.c の関数 add() は次のように書くこともできる。

```
double add(double x,double y)
{
  return x+y;
}
```

式のない return 文は関数を終了させる働きをする。関数本体の } に処理が到達したときは、return; に出会うのと同じである。

以下のプログラムは、0 度から 100 度まで、摂氏の温度を対応する華氏の温度に変換して表示するプログラムである。

```
#include <stdio.h>

double C2F(double cel);

int main(void)
{
    int c;

    printf("摂氏　華氏\n");
    printf("----------\n");
    for(c=0;c<=100;c++){
        printf("%4d %5.2f\n", c, C2F((double)c) );
    }

    return 0;
}

double C2F(double cel)
{
    return cel * 9.0 / 5.0 + 32.0;
}
```

### 6.3.3 main() 関数の return 文

main() 関数も関数の一つである。main() 関数の返り値 (戻り値とも言う) は int 型であり正常に終了した場合 0 を返すのが通常である。main() 関数の返り値はプログラムの終了ステータスを表し、シェル変数 $? によって参照することができる。たとえば hello.c

```
#include <stdio.h>

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

であれば、返り値を参照すると以下のような実行例が得られる。

```
$ cc -Wall -o hello hello.c
$ ./hello
Hello, world
$ echo $?
0
```

次のようなプログラムを使って、main() 関数の戻り値を変更してシェル変数 $? が変化する様子を見ることができる。

```
/* return_test.c */
#include <stdio.h>
```

```
int main(void)
{
    int ret;

    printf("main 関数の戻り値を入力してください");
    scanf("%d", &ret);
    return ret;
}
```

```
$ cc -Wall -o return_test return_test.c
$ ./return_test
main 関数の戻り値を入力してください 3
$ echo $?
3
$ ./return_test
main 関数の戻り値を入力してください-1
$ echo $?
255
```

このことからシェル変数 $? は unsigned char であることがわかる。

#### 6.3.4 関数定義

関数定義の構文規則は、

```
返り値の型 関数名 (仮引数の列)
{
    変数宣言

    文
}
```

である。{} 内が関数本体を構成する。

- 返り値の型を省略すると返り値は int 型とみなされる。
- 返り値がないときは返り値の型を void とする。
- 関数名の命名規則は変数名の命名規則と同じである。
- 仮引数の列のなかで用いた変数は、関数本体の中で宣言しないで用いる。
- 仮引数がないときは仮引数の列を void または空欄とする。

add-func.c の関数 add() を例に説明する。返り値の型は double、関数名 (識別子) は add、仮引数の列は double x,double y、関数本体は z=x+y;return z; である。add(double x,double y) を関数宣言子という。

### 6.4 関数原型 (関数プロトタイプ)

関数は呼び出す前にコンパイラに関数の仕様 (戻り値の型、受け取る引数の数、型、順序) を知らせるため、関数原型 (function prototype; プロトタイプ宣言) が必要である。関数原型では、仮引数の型 (と変数名)、返り値の型を宣言する。呼び出し時に仮引数と実引数の数を調べ、一致していないとエラーになる。

main() 関数の関数原型は不要である。ライブラリ関数は、関連するヘッダファイルに関数原型が記載されているので、それを include しておけば関数原型は不要である。例えば、printf() 関数は<stdio.h>に

関数原型が書かれているので、printf() 関数を使うプログラムでは、前処理指令に #include <stdio.h> と書けば良い。

関数原型の構文は次のようになる。

```
返り値の型 関数名 (仮引数の列);
```

関数定義のヘッダをセミコロン; で置き換えればよい。仮引数の列には型だけで変数名は省略してもよい。

関数定義が呼び出し文よりも前にあれば、プロトタイプ宣言は必要ない。add-func.c は次のように表わすことができる。

```
/*****************************************
* add-func1.c                             *
* 関数原型を使わない例              *
*****************************************/
#include <stdio.h>

double add(double x,double y)
{
  double z;

  z=x+y;

  return z;
}

int main(void)
{
  double a;

  a=add(1.2,2.3); /* 関数呼び出し */
  printf("a=%f\n",a);

  return 0;
}
```

関数原型があると関数の呼び出し方を誤ったときコンパイラが誤りを指摘してくれるので、最初に関数原型を書くことを勧める。

## 6.5 関数の引数

関数は任意の数の引数をとることができる。3 つの引数をとって、その最大値を返す関数 maximum() は次のようになる。

```
#include <stdio.h>

int maximum(int a, int b, int c);

int main(void)
{
    int x, y, z, max;

    printf("整数を 3 つ入力してください:");
    scanf("%d%d%d", &x, &y, &z);
    max = maximum(x, y, z);
    printf("最大値は %d です\n", max);
```

```
    return 0;
}

int maximum(int a, int b, int c)
{
    int ret = a;

    if ( b > ret )
        ret = b;
    if ( c > ret )
        ret = c;

    return ret;
}
```

実行結果は以下のようになる。

```
$ cc maximum.c -o maximum -Wall
$ ./maximum
整数を 3 つ入力してください:30 25 11
最大値は 30 です
$ ./maximum
整数を 3 つ入力してください:15 8 33
最大値は 33 です
```

## 6.6 関数呼び出し: call by value と call by reference

関数の呼び出し文は、

```
関数名 (実引数の列);
```

となる。引数を持たない関数を呼び出すときは、実引数の列は空欄にする。

```
関数名 ();
```

たとえば、getchar() 関数を呼び出すのは、

```
getchar();
```

である。

```
/*****************************************
* add-func2.c                            *
* 値による呼び出しの例                   *
*****************************************/
#include <stdio.h>
double add(double x,double y);

int main(void)
{
  double a=1.2,b=2.3,c;
  c=add(a,b); /* 関数呼び出し */
  printf("c=%f\n",c);

  return 0;
}

double add(double x,double y)
{
  double z;

  z=x+y;

  return z;
}
```

add() には a と b の値が渡されるだけで、a と b そのものが渡されるわけではない。**値による呼び出し (call by value)** という。関数の本体で呼び出し側の変数の値を変更するには**参照による呼び出し (call by reference)** を用いる。参照による呼び出しはポインタの節で学ぶ[16]。値による呼び出し (call by value) のために、呼び出した側の変数が、呼び出された関数内で書き換えられても呼び出した側では変わらないことは以下のプログラムで確認できる。

```
/* call_by_value.c */
#include <stdio.h>
void func(int x, int y);

int main(void)
{
  double x=1, y=2;

  printf("main 関数内 x=%d, y=%d\n", x, y);
  func(x, y);
  printf("main 関数内 x=%d, y=%d\n", x, y);

  return 0;
}

void func(double x,double y)
{

  x=3;
  y=0;

  printf("func 関数内 x=%d, y=%d\n", x, y);

  return ;
}
```

**演習 6.1** キーボードから double 型の 2 数を入力し、関数 add() を呼び出し、入力した 2 数の和を出力するプログラム add-func3.c を作れ。

[16] scanf() 関数の第 2 引数に & をつけるのは参照による呼び出し (call by reference) を行って、呼び出し側の変数の値を変更させるためである

**演習 6.2** 商品代を入力すると消費税と合計額を出力するプログラムをつくれ。商品代を引数とし消費税を返却値とする関数 tax() を作り、main() 関数から tax() を呼び出すことにより実現せよ。

**演習 6.3**

$$f(x) = \frac{1}{1+x^2}$$

で定義される関数 double f(double x) を作り、これを用いて 0 から 1 までの 0.1 きざみの値 $f(0.0), f(0.1), \ldots, f(1.0)$ を出力せよ。

**演習 6.4** 絶対値を与える関数

$$\mathrm{absolute}(x) = \begin{cases} x & x \geqq 0 \\ -x & x < 0 \end{cases}$$

double absolute(double x) を作れ。次にキーボードから double 型の数を入力し、absolute() を呼び出して入力した数の絶対値を出力するプログラムをつくれ。

また、引数を持たない関数も作ることができる。引数を持たない関数定義の仮引数は void または空欄とし、呼び出すときは

```
関数名 ();
```

とする。関数原型の仮引数並びには void と書く。

返却値を持たない関数の返却値の型は void である。以下は画面に四角形を描くプログラムである。戻り値がないため void が使われていることに注意。

```
#include <stdio.h>

void printEdge(int size); /* 関数のプロトタイプ宣言 */

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

    printf("Input size :");
    scanf("%d", &l);

    if ( l < 3 || l > 80) {
        printf("Inpossible\n");
        return 0;
    }

    printEdge(l); /* 関数の呼び出し */

    for(i=2;i<l;i++){
        printf("*");
        for(j=2;j<l;j++){
            printf(" ");
```

```
        }
        printf("*\n");
    }

    printEdge(l); /* 関数の呼び出し */

    return 0;
}

void printEdge(int size)
{
    int i;

    for(i=1;i<=size;i++){
        printf("*");
    }
    printf("\n");

    return;
}
```

## 6.7 変数の記憶クラス

C には auto, register, extern, static の 4 つの記憶クラスがある。記憶クラスが異なると変数の生存期間 (有効期間) が異なる。プログラムの実行中ずっと有効な変数もあれば、生成消滅を繰り返す変数もある。4 つの記憶クラスは、自動記憶クラスである auto と register、静的記憶クラスである extern と static に分けられる。

通常の変数は自動記憶クラスである。自動記憶クラスであることを明示的に宣言する場合には auto をつけて宣言する double 型の変数 x, y を明示的に自動記憶クラスとして宣言したければ、

```
auto double x, y;
```

と書く。通常ローカルな変数は自動記憶クラスとなるのでキーワード auto を用いることは滅多にない。キーワード register はコンパイラに、もし利用可能ならば CPU のレジスターを割り当てること要求する。コンパイラは CPU のレジスタに余裕があれば、その変数をあるレジスタに割り当てる。近年、CPU が高速、複雑化していることと、コンパイラが高度な最適化を自動的に行うこととあいまってプログラマがキーワード register を積極的に使う場面は少なくなった。

キーワード static と extern とは静的記憶クラスを宣言するために使われる。静的記憶クラスの変数はプログラムが始まった時点で生成され、プログラムが終了するまでメモリ上に存在する。

静的記憶クラスの識別子には 2 種類ある。一つは extern 宣言されたグローバルな変数名と関数名とからなる外部識別子であり、もう一つは static 宣言された変数である。

変数をローカルではなくグローバルとして宣言すると予期せぬ副作用が生じる危険がある。たとえば、その変数にアクセスする必要のない関数が偶然その値を変更してしまうなどである。一般にグローバル変数は特別な必要性がない限り使うべきではない。特定の関数の中でしか使わない変数はグローバルではなくローカル変数として宣言すべきである。

キーワード static をつけて宣言された変数は、それが定義された関内でしか適用されないが、その関数から処理が出ても値を保持し続ける。したがって次のプログラム

```
#include <stdio.h>

int counter(void);

int main(void)
{
    int i;

    for ( i=0; i<10; i++) {
        counter();
    }

    return 0;
}

int counter(void)
{
    static int counter = 0;

    printf("関数 counter が %d 回呼び出されました。\n", ++counter);

    return 0;
}
```

をコンパイルして実行すると、以下のようになる。

```
$ cc -Wall -o static_variable static_variable.c
$ ./static_variable
関数 counter が 1 回呼び出されました。
関数 counter が 2 回呼び出されました。
関数 counter が 3 回呼び出されました。
関数 counter が 4 回呼び出されました。
関数 counter が 5 回呼び出されました。
関数 counter が 6 回呼び出されました。
関数 counter が 7 回呼び出されました。
関数 counter が 8 回呼び出されました。
関数 counter が 9 回呼び出されました。
関数 counter が 10 回呼び出されました。
```

static 宣言を取ると、変数 counter はその都度初期化されるので毎回 1 となる。

## 6.8 変数の有効範囲

変数の有効範囲とは、プログラムの中でその変数を参照できる範囲のことである。プログラムのどこからでも参照できる変数もあれば、部分的にしか参照できない (ローカライズ localize された) 変数もある。

関数の先頭で宣言されたローカル変数 (局所変数とも言う) は、関数の引数と共にその関数内でのみ有効である。関数を呼び出した側はその関数の内部にどのようなローカル変数があるのかはわからない。こうした情報隠蔽はソフトウエア工学の重要な基本原理である。情報隠蔽によって考慮すべき変数の数が少なくなり、解くべき問題に焦点をあてることができるようになるからである。

次のプログラムは、ローカル変数の有効範囲を示すものである。

```
#include <stdio.h>

void func(void);

int main(void)
{
    int x = 5; /* main の局所変数 */

    printf("main の局所変数 x は %d\n", x);
    func();
    func();
    printf("main の局所変数 x は %d\n", x);

    return 0;
}

void func(void)
{
    int x = 20; /* func の局所変数 */

    printf("\nfunc の局所変数 x は %d\n", x);
    x++;
    printf("\nfunc の局所変数 x は %d\n", x);
}
```

このプログラムの実行結果は以下のとおり。

```
$ cc -Wall -o scope scope.c
$ ./scope
main の局所変数 x は 5

func の局所変数 x は 20

func の局所変数 x は 21

func の局所変数 x は 20

func の局所変数 x は 21
main の局所変数 x は 5
```

関数呼び出しによって x の値が変化しないことに注意。

## 6.9 数学関数

数学で用いる三角関数や指数関数はヘッダ<math.h>で宣言され、関数は double 型の実引数を受取り、double 型の値を返す。

数学関数を用いるときは、前処理指令に

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

と書き、コンパイルは

```
$ cc -lm ファイル名 -o 実行ファイル名
```

などとする。-lm はリンクの際、数学ライブラリ libm.a, libm.dylib を使用するオプションである。

主な数学関数を次表に示す。数学の実数値関数として定義されない実引数を与え呼び出したとき (例えば、sqrt(-1.2) としたとき) にはエラーになる。

表 1: <math.h>で定義されている主な数学関数

| 形式 | 機能 | 数学 |
|---|---|---|
| double acos(double x); | x の逆余弦の主値 (ラジアン) | $\cos^{-1} x$ |
| double asin(double x); | x の逆正弦の主値 (ラジアン) | $\sin^{-1} x$ |
| double atan(double x); | x の逆正接の主値 (ラジアン) | $\tan^{-1} x$ |
| double cos(double x); | x ラジアンの余弦 | $\cos x$ |
| double sin(double x); | x ラジアンの正弦 | $\sin x$ |
| double tan(double x); | x ラジアンの正接 | $\tan x$ |
| double exp(double x); | x の指数 | $e^x$ |
| double log(double x); | x の自然対数 | $\log_e x$ |
| double log10(double x); | x の常用対数 | $\log_{10} x$ |
| double pow(double x,double y); | x の y 乗 | $x^y$ |
| double sqrt(double x); | x の平方根 | $\sqrt{x}$ |
| double fabs(double x); | x の絶対値 | $\lvert x \rvert$ |
| double floor(double x); | x 以下の最大の整数 | $[x]$ |

**例 6.1** 長方形の縦と横の長さを与え対角線の長さを求めるプログラム。

```
/* diagonal.c */
#include <stdio.h>
#include <math.h>

int main(void)
{
  double x,y,d;

  printf("縦の長さ=");
  scanf("%lf",&x);
  printf("横の長さ=");
  scanf("%lf",&y);

  d=sqrt(x*x+y*y);
  printf("対角線の長さ=%f\n",d);

  return 0;
}
```

コンパイルと実行結果は、

```
$ cc -lm diagonal.c -o diagonal
$ ./diagonal
縦の長さ=3
横の長さ=4
対角線の長さ=5.000000
```

である。

**演習 6.5** 実数係数の 2 次方程式 $ax^2 + bx + c = 0,\ (a \neq 0)$ の解を出力するプログラムを作成せよ。($D = b^2 - 4ac$ の符号で分ける。)

**例 6.2** 次のプログラムは画面にサインカーブを描くプログラムである。

```
/* sin_curve.c */
#include <stdio.h>
#include <math.h>

int main(void)
{
    double x, y, Pi=3.14159;
    int i, y2;

    for(x=0.0; x<=2.0 * Pi; x+=0.1){
        y=sin(x);
        y2 = (int)((y + 1.0)/2.0 * 80.0 );
        for(i=0;i<y2;i++){
            printf(" ");
        }
        printf("*\n");
    }

    return 0;
}
```

**演習 6.6** 2 点 $(x_1, y_1),\ (x_2, y_2)$ の距離を計算するプログラム `distance.c` を作成せよ。すべての数値は `double` 型であると仮定する。

## 6.10 関数の再帰呼び出し

以下のプログラムは 5 の階乗を求めるプログラムである。

```
#include <stdio.h>

int Fact( int n );

int main(void)
{
  int n;

  n = 5;
  printf("Factorial of %d is %d.\n", n, Fact(n));
  return 0;
}

int Fact( int n )
{
  if ( n == 1 )
    return 1;
  else
    return n * Fact( n - 1 );
  }
}
```

ここでのポイントは Fact() 関数の中で自分自身である Fact() が呼び出されていることである。関数 Fact() は呼び出されるごとに 1 だけ少ない値を引数として自分自身を呼び出す。引数 n が 1 の場合だけ 1 を返す。このような関数を再帰的 (recursive) な関数と呼ぶ。呼び出された関数と呼び出した側の関数で引数 n が異なる値になっていることを理解して欲しい。また、main() の中にある変数 n と Fact() 側の引数 n は全く別物であることにも注意が必要である。

**演習 6.7** 上の階乗を求めるプログラムを再帰関数を使わずに，繰り返し for() 文を用いて書き換えよ。

ベトナムの首都ハノイのとある寺院では、僧侶たちが 1 本の棒にさしてある 64 枚のディスクの束を別の棒に移そうとしている。最初、64 枚のディスクは左端の棒に下のほうから大きい順にさしてある。僧侶たちはこの円盤の束を右端の棒に移そうとする。ただしディスクは一度に一枚しか移してはいけない。どんなときでも大きなディスクを小さなディスクの上に置いてはならないという制約がある。真ん中の棒は一時的にディスクを退避するために使われる。僧侶たちの努力が報われるのはいつだろうか。

ハノイの塔の問題は再帰を使うとスッキリと解ける。n 枚のディスクを A から B に移すには n - 1 枚のディスクを A から C に移しておき n 番目のディスクを A から B に移してから、C に移しておいた n - 1 枚のディスクを C から B に移せばよい。

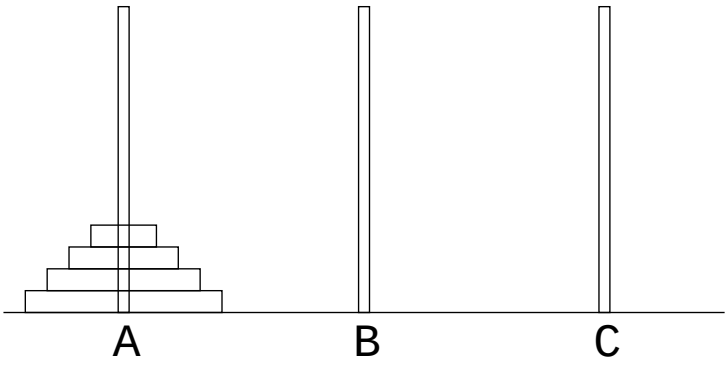


図 1: ハノイの塔

```
#include<stdio.h>

int hanoi(int disks, char a, char b, char c);

int main(void)
{
    int disks;
    char a = 'A', b='C', c='B';

    printf("何枚のディスクを動かしますか?:");
    scanf("%d", &disks);

    while (disks > 65 || disks < 0 ) {
        printf("1 から 64 までの数字で指定してください :");
        scanf("%d", &disks);
    }

    hanoi(disks, a, b, c);
```

```
    return 0;
}

int hanoi(int disks, char a, char b, char c)
{

    if (disks == 1) {
        printf("ディスク 1 を %c から %c に動かせ\n", a, b);
    } else {
        hanoi(disks - 1, a, c, b);
        printf("ディスク %d を %c から %c に動かせ\n", disks, a, b);
        hanoi(disks - 1, c, b, a);
    }
    return 0;
}
```