8 モジュール化について

8–1 モジュール化

複雑な仕事内容を計算機で処理する際は、
処理のモジュール化、すなわち、

> 小さなプログラムを部品(module)として構成し、
> それらの部品を使うことによってより大きなプログラムを構築する、

というソフトウェア構築法が有効である。

補足：
プログラムの大きさが2倍になった場合、
分かり難さは2倍では済まない。

モジュール化の利点

- プログラムの構造化、系統化

  ⇨ プログラムが分かり易く修正し易くなる。

  補足：
  各モジュールの仕様が明らかになっていることが大切。

- 大きなプログラムを何人かで分担して作るのに都合が良い。

- 同一の処理単位を何回も重複してプログラムに組む込む必要がなくなる。

  ⇨ プログラムの簡素化

何をモジュールと考えるか：

通常のプログラミング言語では、機能的にまとまりのあるプログラム断片を1つの関数または手続きとして定義／登録でき、また、これらの関数や手続きを必要に応じて呼び出せる様になっている。

⇨ 一般にはプログラムを設計する際に
関数や手続きをモジュールと考えるのが最も素朴で自然。

例題8．1（Napier数eの1000桁計算） ネピア数(Napier number)

$$e = \lim_{x\to\infty}\left(1+\frac{1}{x}\right)^x = \sum_{i=0}^{\infty}\frac{1}{i!} = 2.718281828\cdots\cdots$$

を小数点以下1000桁まで計算して出力するCプログラムを作成せよ。

（考え方）　スターリング（Stirling）の公式

$$m! \approx \sqrt{2\pi}\, m^{m+1/2} e^{-m} \quad (m \longrightarrow \infty)$$

より

$$451! \approx \sqrt{2\pi}\, 451^{451.5} e^{-451} \approx 7.8 \times 10^{1002}$$

が得られることに注目しよう。これより、e の1001桁計算のためには近似式

$$e \approx \sum_{i=0}^{450} \frac{1}{i!}$$
$$= (((\cdots((1 \cdot \frac{1}{450}+1)\frac{1}{449}+1)\cdots)\frac{1}{3}+1)\frac{1}{2}+1)\frac{1}{1}+1$$

による計算を正確に行なえばよいことが分かる。この計算は、実数を正確に記憶できる変数 v_{ideal} があれば、次のように行うことが出来る。

$v_{ideal} \leftarrow 1$;
for $k \leftarrow 450$ **step** -1 **until** 1 **do**
　$v_{ideal} \leftarrow v_{ideal}/k + 1$;
v_{ideal} の値を出力 ;

しかし、実数を正確に記憶できる変数などあり得ないので、その代わ

りに小数点以下1000桁の10進小数

$$c_3c_2c_1c_0.d_1d_2d_3d_4d_5d_6d_7d_8\cdots\cdots d_{997}d_{998}d_{999}d_{1000}$$

（各c_i, d_iは 0~ 9 の整数を表す。）

を

$$\underbrace{c_3c_2c_1c_0}_{e[0]\text{に記憶}} . \underbrace{d_1d_2d_3d_4}_{e[1]\text{に記憶}} \underbrace{d_5d_6d_7d_8}_{e[2]\text{に記憶}} \cdots\cdots \underbrace{d_{997}d_{998}d_{999}d_{1000}}_{e[250]\text{に記憶}}$$

という風に記憶する整数型（4バイト以上）配列 e を用意する。

そして、上で示した計算手順に現われる処理の基本単位

① 配列e ← 1 （i.e. 配列eの表す10進小数を 1 に設定する処理），

② 配列e ← 配列eの保持する値 / k ，

③ 配列e ← 配列eの保持する値 + 1 ，

④ 配列eの保持する値 の出力

を一般化したものを関数モジュールとして用意することにすれば、これらの関数モジュールを用いて上で示した計算手順を見通し良く書き表すことが出来る。

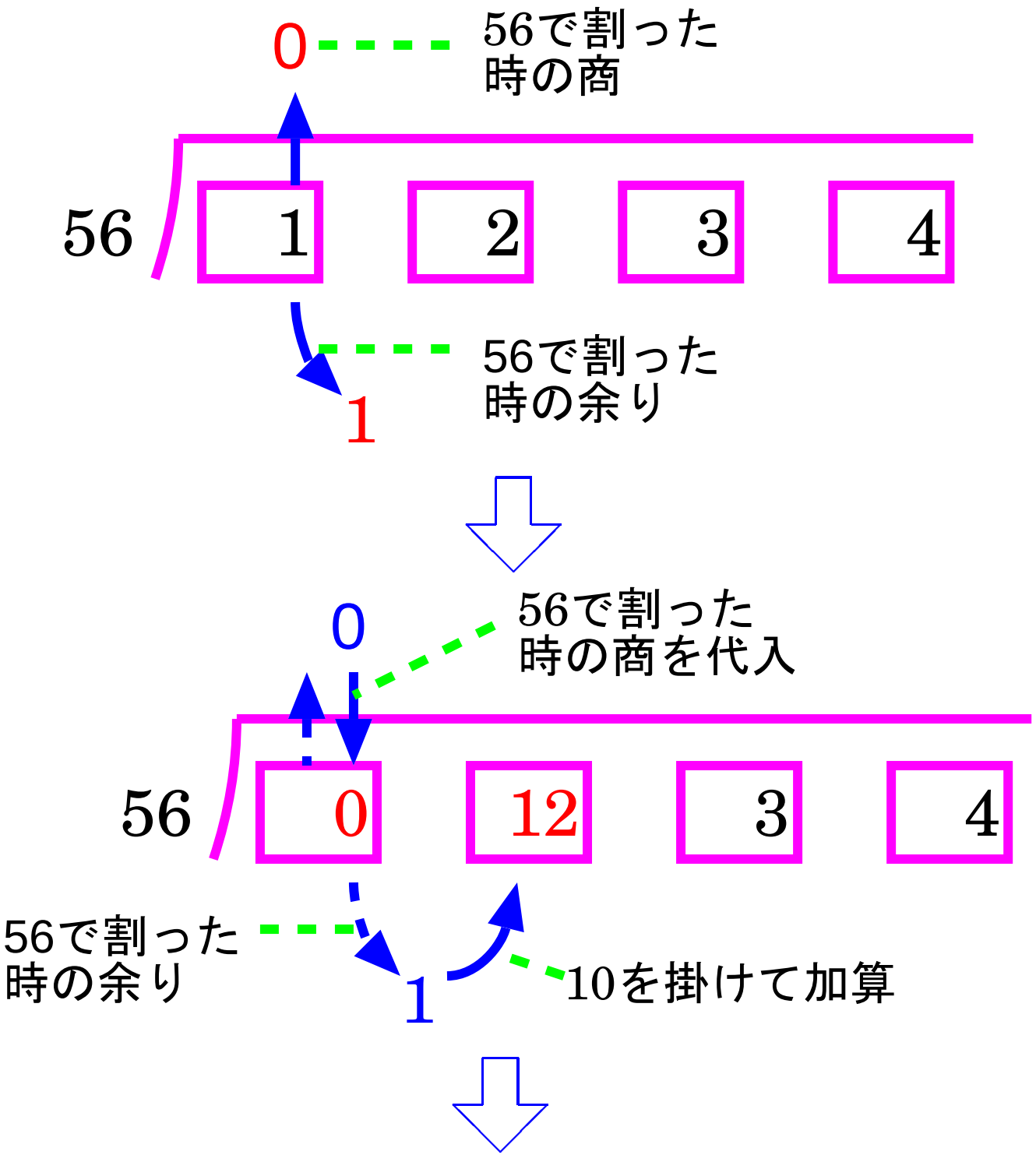
補足：
このアルゴリズムは、結局は、10000進法のそれぞれの桁を記憶するために配列要素 e[0]~ e[250] を使い、10000進法でNapier数を計算するものである。

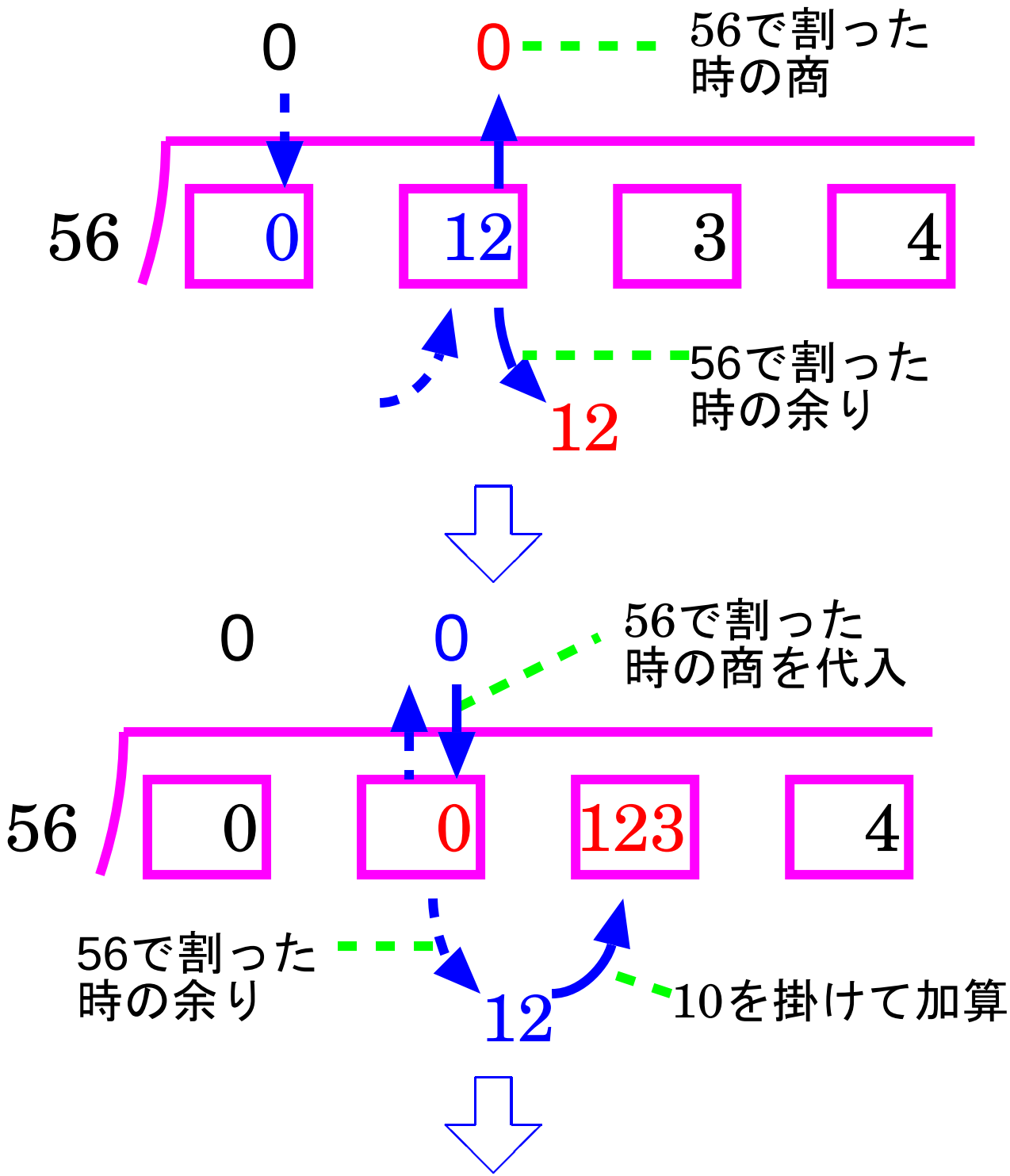
基本処理「配列e ← 配列eの保持する値 / k 」の実装：

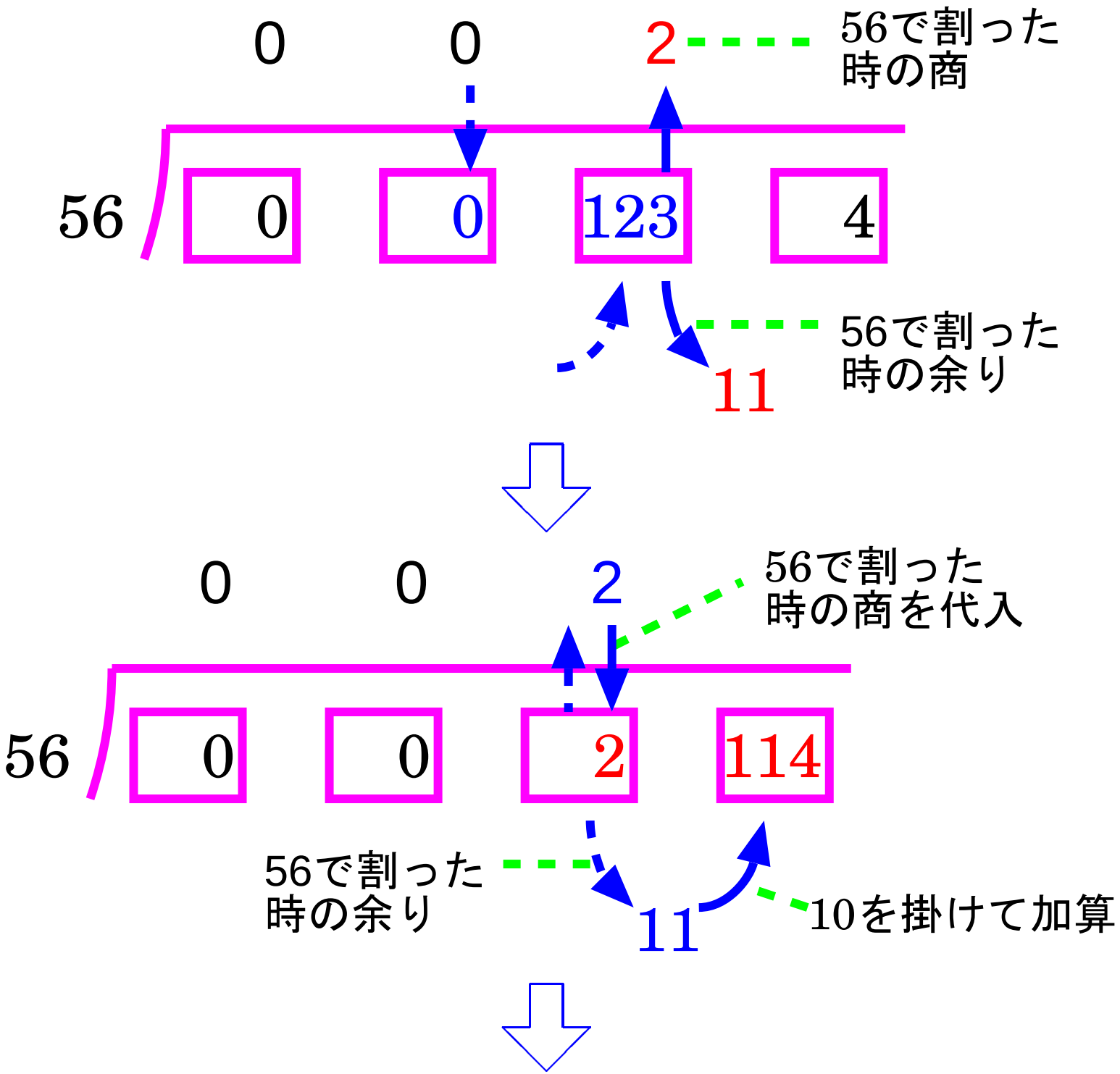
例えば、
配列要素に10進の1桁ずつを保持することにして

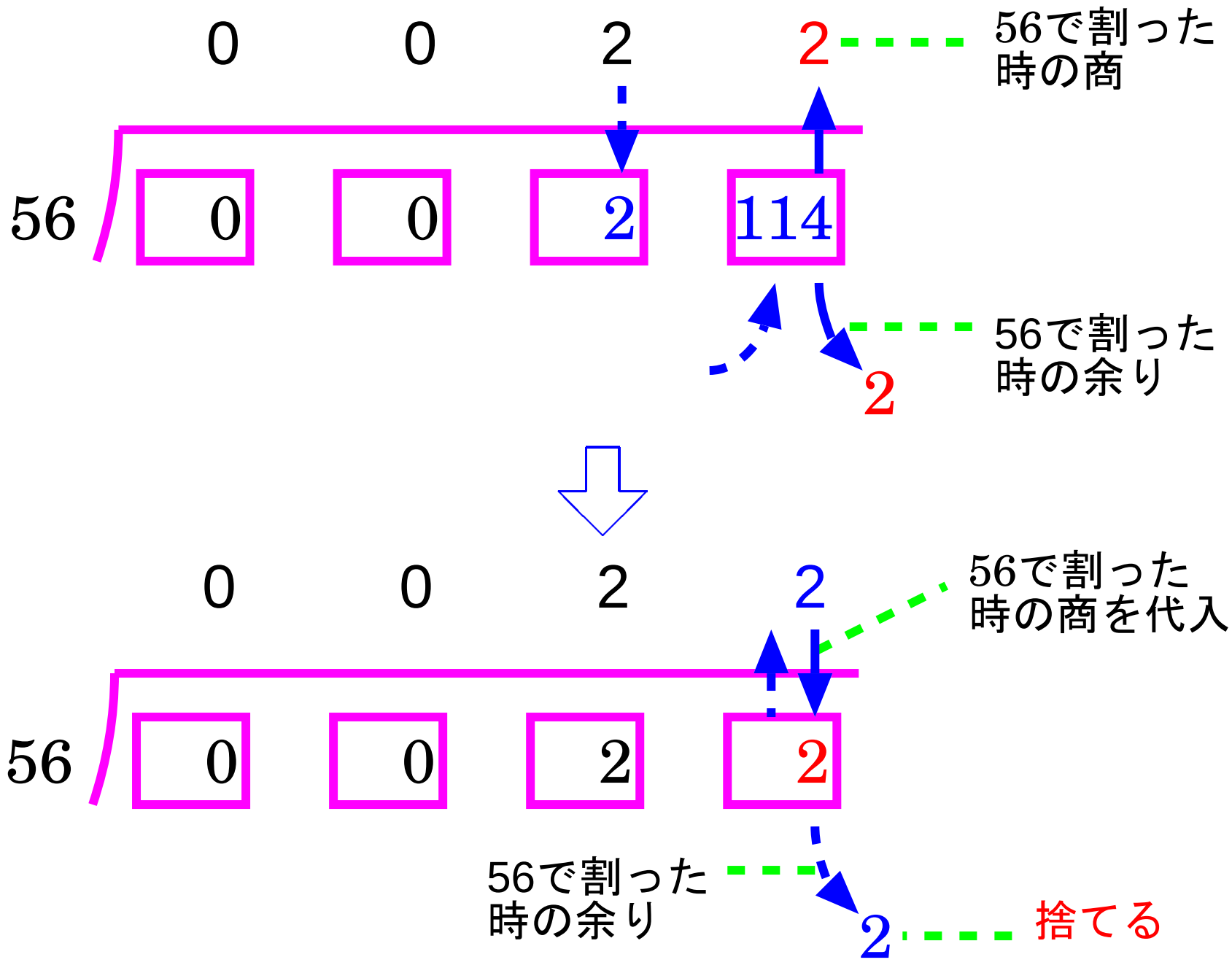
```
          2 2
56 ) 1 2 3 4
     1 1 2
       1 1 4
       1 1 2
           2
```

という場合は、⇨ 次の様に処理を...


0
56で割った
時の商
56
1
2
3
4
56で割った
時の余り
1
0
56で割った
時の商を代入
56
0
12
3
4
56で割った
時の余り
1
10を掛けて加算


0
0
56で割った
時の商
56
0
12
3
4
56で割った
時の余り
12
0
0
56で割った
時の商を代入
56
0
0
123
4
56で割った
時の余り
12
10を掛けて加算


0
0
2
56で割った
時の商
56
0
0
123
4
56で割った
時の余り
11
0
0
2
56で割った
時の商を代入
56
0
0
2
114
56で割った
時の余り
11
10を掛けて加算


0
0
2
2
56で割った
時の商
56
0
0
2
114
56で割った
時の余り
2
0
0
2
2
56で割った
時の商を代入
56
0
0
2
2
56で割った
時の余り
2
捨てる

（プログラミング）　例えば次のような...

```
[motoki@x205a]$  nl modules-napier.c
     1 /*****************************************************
     2 /* Napier数 e の1000桁計算
     3 /*-------------------------------------
     4 /*     大きさ251の配列 e に小数点以下1000桁の10進小数を
     5 /*         e[0]=整数部の数,
     6 /*         e[1]=小数点以下1~ 4桁目の数,
     7 /*         e[2]=小数点以下5~ 8桁目の数,
     8 /*             ......
     9 /*         e[250]=小数点以下996~ 1000桁目の数
    10 /*     という風に保存して高精度計算を行う
    11 /*****************************************************

    12 #include  <stdio.h>
```

```
13 #define   LIMIT   7

14 void set_array_to_keep_integer(
                        int a[], int size, int init_value);
15 void divide_array_value_by_integer(
                        int a[], int size, int divisor);
16 void add_integer_to_array(
                        int a[], int size, int value);
17 void print_array_value(
                        int a[], int size, char *padding);

18 int main(void)
19 {
20   int  e[251], k;

21   set_array_to_keep_integer(e, 251, 1); /* e <-- 1 */
```

```
22   for (k=450; k>0; --k) {
23     divide_array_value_by_integer(e, 251, k);
                                          /* e <-- e/k */
24     add_integer_to_array(e, 251, 1);    /* e <-- e+1 */
25   }

26   printf("e =");
27   print_array_value(e, 251, "   ");     /* 結果の出力 */
28   return 0;
29 }

30 /*-------------------------------------------------------
31 /* 配列全体で表す値(10進小数)を初期設定する
32 /*-----------------------------------------
33 /* (仮引数)   a     : int型配列
34 /*             size : int型配列 a の大きさ
35 /*       init_value : 初期設定したい値(int型)
```

```
36 /* (関数値)   : なし
37 /* (機能) :  配列 a 全体の表す値が init_value になる... *
38 /*-----------------------------------------------------
39 void set_array_to_keep_integer(
                        int a[], int size, int init_value)
40 {
41   int i;
42
43   a[0] = init_value;
44   for (i=1; i<size; ++i)
45     a[i] = 0;
46 }

47 /*-----------------------------------------------------
48 /* 配列全体で表す値(10進小数)を整数で割る
49 /*-----------------------------------------
50 /* (仮引数)   a     : int型配列
```

```
51 /*              size : int型配列 a の大きさ
52 /*           divisor : 除数
53 /* (関数値)    : なし
54 /* (機能) :  配列 a 全体の表す値を整数 divisor で割る
55 /*-----------------------------------------------------
56 void divide_array_value_by_integer(
                          int a[], int size, int divisor)
57 {
58   int i;
59
60   for (i=0; i<size-1; ++i) {
61     a[i+1] += (a[i] % divisor) * 10000;
62     a[i]   /= divisor;
63   }
64   a[size-1] /= divisor;
65 }
```

```
66 /*-----------------------------------------------------------
67 /* 配列全体で表す値(10進小数)に整数を加算する
68 /*-----------------------------------------
69 /* (仮引数)    a      : int型配列
70 /*              size : int型配列 a の大きさ
71 /*            addend : 加数
72 /* (関数値)    : なし
73 /* (機能) :  配列 a 全体の表す値に整数 addend を加算する
74 /*-----------------------------------------------------------
75 void add_integer_to_array(
                        int a[], int size, int addend)
76 {
77   a[0] += addend;
78 }

79 /*-----------------------------------------------------------
80 /* 配列全体で表す値(10進小数)を出力する
```

```
81 /*-----------------------------------------
82 /* (仮引数)    a      : int型配列
83 /*              size : int型配列 a の大きさ
84 /*           padding : 2行目以降の左端に必ず出力する文字列
85 /* (関数値)    : なし
86 /* (機能) :  配列 a 全体の表す10進小数値を出力する。その..
87 /*              *小数点以下は 8*LIMIT 桁毎に改行する,
88 /*              *2行目以降の出力においては左端に必ず文字..
89 /*                padding を埋める。
90 /*-----------------------------------------------------
91 void print_array_value(
                          int a[], int size, char *padding)
92 {
93   int i, count;
94
95   printf("%2d.", a[0]);
96   count = 1;
```

```
 97   for (i=1; i<size; i+=2, ++count) {
 98     printf("%04d%04d ", a[i], a[i+1]);
 99     if (count >= LIMIT) {
100       printf("\n%s   ", padding);
101       count = 0;
102     }
103   }
104   printf("\n");
105 }
[motoki@x205a]$  gcc modules-napier.c
[motoki@x205a]$  ./a.out
e = 2.71828182 84590452 35360287 47135266 24977572 47093699 95
      69676277 24076630 35354759 45713821 78525166 42742746 63
      03059921 81741359 66290435 72900334 29526059 56307381 32
      94349076 32338298 80753195 25101901 15738341 87930702 15
      99348841 67509244 76146066 80822648 00168477 41185374 23
```

(途中省略)

```
        93458072 73866738 58942287 92284998 92086805 82574927 96
        98444363 46324496 84875602 33624827 04197862 32090021 60
        30436994 18491463 14093431 73814364 05462531 52096183 69
        70167683 96424378 14059271 45635490 61303107 20851038 37
        15747704 17189861 06873969 65521267 15468895 70350354
[motoki@x205a]$
```

□演習８．２（階乗の表）　**1!~ 53!**を計算して、それらを桁を揃えて出力せよ。　[**53!** $\approx$ **4.27**$\times$**10**$^{\mathbf{69}}$ となります。]

8–2 モジュール化の方法 ---段階的詳細化---

モジュール化を実際に進めるための手法としては 、
段階的詳細化、すなわち、
処理手順を少しずつ詳細化していくことによってプログラム／アルゴリズムを作り上げてゆく、
という考え方がよく用いられている。

このプログラム設計方針に従えば 、
処理手順は大雑把なものから（十分に）細かなものへと少しずつ詳細化されていくことになるから、
詳細化の途中に現われる処理単位のうち
機能的にまとまりのあるものをモジュールに
すれば、元の大きな処理は比較的きれいに分割され、プログラムのモジュール化が自然に進むことになる。

例題8．2（自習 与えられた月のカレンダーを出力する） 2つの正整数 y と m (≤ 12) を入力して西暦 y 年 m 月のカレンダーを次の様な形で出力するCプログラムを作成せよ。

```
              2004 May
Sun Mon Tue Wed Thu Fri Sat
                            1
  2   3   4   5   6   7   8
  9  10  11  12  13  14  15
 16  17  18  19  20  21  22
 23  24  25  26  27  28  29
 30  31
```

⇨ この問題に対して、次のように段階的詳細化の考え方を適用できる。

第1段階 全体の処理の流れの記述

3つの関数

$$\mathtt{name(m)} = \begin{cases} \text{"January"} & \text{if } \mathtt{m=1} \\ \text{"February"} & \text{if } \mathtt{m=2} \\ \vdots & \\ \text{"December"} & \text{if } \mathtt{m=12} \end{cases}$$

number_of_days(y,m) = 西暦y年m月の日数

what_day(y,m) = 西暦y年m月1日の曜日を表す指標

$$= \begin{cases} 0 & \text{if 西暦y年m月1日が日曜日} \\ 1 & \text{if 西暦y年m月1日が月曜日} \\ 2 & \text{if 西暦y年m月1日が火曜日} \\ \vdots & \\ 6 & \text{if 西暦y年m月1日が土曜日} \end{cases}$$

の存在を仮定すれば、与えられた正整数 y, m に対して 西暦 y 年 m 月のカレンダーを求められた形に出力するアルゴリズムとしては次のようなものが考えられる。

```
年数、月数を表す正整数を入力して各々 year, month と名付ける ;
    { プロンプトの出力、入力データのチェックなどは適宜行なう。}
1行目の表題のために整数 year と月名 name(month) を出力 ;
曜日の見出し(" Sun Mon ...")を出力 ;
4 × what_day(year,month)個の空白を出力 ;
column ← what_day(year,month)+1 ;
for day ← 1 until number_of_days(year,month) do
  begin
    日付 day を出力 ;
    if column=7 then 改行
                else column ← 0 ;
    day ← day+1 ;
    column ← column+1
  end ;
if column>1 then 改行
```

第2段階 関数 name の詳細化

関数 name(m) は次の2次元char型配列 name[i][j] で表せる。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	'I'	'l'	'l'	'e'	'g'	'a'	'l'	'␣'	'm'	'o'	'n'	't'	'h'	'\0
1	'J'	'a'	'n'	'u'	'a'	'r'	'y'	'\0'						
2	'F'	'e'	'b'	'r'	'u'	'a'	'r'	'y'	'\0'					
3														
	.....													
10														
↓ 11	'N'	'o'	'v'	'e'	'm'	'b'	'e'	'r'	'\0'					
i 12	'D'	'e'	'c'	'e'	'm'	'b'	'e'	'r'	'\0'					

第3段階 関数 number_of_days の詳細化

m月の標準日数を表す関数

$$\texttt{n\_days(m)} = \begin{cases} 31 & \text{if } \texttt{m=1} \\ 28 & \text{if } \texttt{m=2} \\ 31 & \text{if } \texttt{m=3} \\ \vdots & \\ 31 & \text{if } \texttt{m=12} \end{cases}$$

と「西暦y年m月がうるう月かどうか」を表す関数

$$\texttt{leap(y,m)} = \begin{cases} 1 & \text{if 暦y年がうるう年でm=2} \\ 0 & \text{otherwise} \end{cases}$$

を用いて

number_of_days(y,m) = n_days(m) + leap(y,m)

と表せる。

⇨ 第1段階で示した全体の処理の流れの中で、
number_of_days(y,m) の代わりに
n_days(m) + leap(y,m) という式を用いれば良い。

第4段階 関数 n_days の詳細化

関数 n_days(m) はint型配列で表せる。

第5段階 関数 leap の詳細化

グレゴリオ歴によると

西暦年数が4で割り切れ100で割り切れない年、
または400で割り切れる年を閏年とする

ことになっている。

⇨ leap(y,m)の計算は 次のように行なうことが出来る。
[ここで、mod(y,i)はyをiで割った余りを表す。]

```
if m=2 ∧ ((mod(y,4)=0 ∧ mod(y,4) ≠ 0)
          ∨ mod(y,400)=0)
   then  return 1
   else  return 0
```

第6段階 関数 what_day の詳細化

1月1日から同年(m-1)月末日までの標準日数を表す関数

$$\text{total\_days}(m) = \sum_{i=1}^{m-1} \text{n\_days}(i)$$

を用いると、西暦1年1月1日から西暦y年m月1日までの日数は

(西暦1年1月1日から西暦(y-1)年12月31日までの日数)

+ (西暦y年1月1日から西暦y年m月1日までの日数)

$$= 365(y-1) + \lfloor \frac{y-1}{4} \rfloor - \lfloor \frac{y-1}{100} \rfloor + \lfloor \frac{y-1}{400} \rfloor + \text{total\_days}(m) + \text{leap}(y, \min\{m-1, 2\}) + 1$$

となる。 それゆえ、定数 c をうまく選べば、

what_day(y,m)

= mod(西暦1年1月1日から西暦y年m月1日までの日数+c, 7)

$$= \text{mod}((7 \times 53+1)(y-1) + \lfloor \frac{y-1}{4} \rfloor - \lfloor \frac{y-1}{100} \rfloor + \lfloor \frac{y-1}{400} \rfloor + \text{total\_days}(m) + \text{leap}(y, \min\{m-1, 2\}) + 1 + c,\ 7)$$

$$= \text{mod}((y-1) + \lfloor \frac{y-1}{4} \rfloor - \lfloor \frac{y-1}{100} \rfloor + \lfloor \frac{y-1}{400} \rfloor + \text{total\_days}(m) + \text{leap}(y, \min\{m-1, 2\}) + 1 + c,\ 7)$$

となるはずである。 ここで、what_day(1999,4)=4 であるから、

```
 what_day(1999,4)
= mod(1998+499-19+90+0+1+c,7)
= mod(4+c,7)
```

ということと合わせて、例えば c=0 と選べばよい。

⇨ 関数 what_day (y, m) の計算は 次のように詳細化できる。

$$\text{return}\ \ \mathrm{mod}\Big(\ (y-1) + \left\lfloor\frac{y-1}{4}\right\rfloor - \left\lfloor\frac{y-1}{100}\right\rfloor + \left\lfloor\frac{y-1}{400}\right\rfloor + \mathrm{total\_days}(m) + \mathrm{leap}(y,\min\{m-1,2\}) + 1,\ \ 7\ \Big)$$

第7段階 関数 total_days の詳細化

関数 total_days(m) はint型配列で表せる。

(プログラミング)

```
[motoki@x205a]$  cat modules-calendar.c
#include <stdio.h>
#include <stdlib.h>

#define  DUMMY     0
#define  min(A,B)  ((A) < (B) ? (A) : (B))

int what_day(int year, int month); /* ○年○月１日の曜日を計算 *
int leap(int year, int month);   /* ○年○月がうるう月かどうか *

/*****************************************/
/* ○月○日のカレンダーを表示する          */
/*----------------------------------------*/
/*    (入力) 正整数 y と m (<=12)         */
/*    (出力) 西暦 y 年 m 月のカレンダー */
/*****************************************/
```

```
int main(void)
{
  int year, month, day, column, k, limit;
  int n_days[13]={
          DUMMY,                          /* n_days[k]            */
          31, 28, 31, 30, 31, 30,         /* = k月の標準的な日数 */
          31, 31, 30, 31, 30, 31
      };
  char name[][15]={                       /* name[k]              */
          "Illegal month",                /* = k番目の月の名前 */
          "January", "February", "March",
          "April", "May", "June",
          "July", "August", "September",
          "October", "November", "December"
        };

  printf("Please type in year and month numbers.\n");
```

```
if ( scanf("%d %d", &year, &month) != 2
                    || year<1 || month <1 || month>12 ){
  printf("\nIllegal input!\n");
  printf("   ==> Input a positive integer and an integer in
  exit(EXIT_FAILURE);
}

printf("          %d %s\n", year, name[month]);
printf(" Sun Mon Tue Wed Thu Fri Sat\n");

limit = what_day(year,month);
for ( column=1 ; column <= limit ; column++ )
  printf("    ");

limit = n_days[month]+leap(year,month);
for ( day=1 ; day <= limit ; day++, column++ ){
  printf("%4d", day);
```

```
    if (column == 7 ){
      printf("\n");
      column=0;
    }
  }

  if (column>1)
    printf("\n");
  return 0;
}

/*****************************************/
/* ○年○月１日の曜日を計算して返す          */
/*-----------------------------------------*/
/*   (仮引数)  y : 西暦年 (正整数)          */
/*             m : 月の番号 (1以上12以下の整数)  */
/*   (関数値)  0  if 西暦 y 年 m 月１日が日曜日  */
```

```
/*                1  if          〃            月曜日    */
/*                     ......                    */
/*                7  if          〃            土曜日    */
/*****************************************************/
int what_day(int y, int m)
{
  int total_days[13]={
          DUMMY,                                /* total_days[k]
            0,  31,  59,  90, 120, 151,         /* = n_days[1]+n_da
          181, 212, 243, 273, 304, 334          /*      + ... +n_da
      };

  return (y+(y-1)/4-(y-1)/100+(y-1)/400
            +total_days[m]+leap(y,min(m-1,2))) % 7;
}

/*****************************************************/
```

```
/* ○年○月がうるう月かどうかを判定してして返す      */
/*-----------------------------------------------------*/
/*   (仮引数)  y : 西暦年 (正整数)                    */
/*             m : 月の番号 (1以上12以下の整数)        */
/*   (関数値)  0  if 西暦 y 年 m 月がうるう月でない */
/*             1  if          〃       うるう月       */
/*****************************************************/
int leap(int y, int m)
{
  if ( m==2 && ( (y%4==0 && y%100!=0) || y%400==0 ) )
    return 1;                    /* うるう月 */
  else
    return 0;
}
[motoki@x205a]$  gcc modules-calendar.c
[motoki@x205a]$  ./a.out
Please type in year and month numbers.
```

2004 5

```
          2004 May
 Sun Mon Tue Wed Thu Fri Sat
                           1
   2   3   4   5   6   7   8
   9  10  11  12  13  14  15
  16  17  18  19  20  21  22
  23  24  25  26  27  28  29
  30  31
[motoki@x205a]$
```

8–3 静的外部変数, 静的関数---関数以外のモジュー

静的外部変数, 静的関数：

外部変数を宣言する際にデータ型の前に static という修飾子を付けると、その変数の有効範囲が同一ファイル内のその宣言以降に制限される。

また、関数定義の前に static という修飾子を付けると、その関数の有効範囲が同一ファイル内に制限される。

こういった変数, 関数を各々静的外部変数, 静的関数という。

関数以外のモジュール：

静的外部変数はそのファイル内の関数が共有する局所的なデータを保有することになる。

⇨ そのファイル内の関数は独立なものではなく、この共通のデータを協調して管理する関数群と見なすことが出来る。

[この考え方は「オブジェクト指向」へと発展する。実際、こういったモジュールが「オブジェクト指向」における「オブジェクト」に相当する。]

⇨ このソースファイル内の静的外部変数群、関数群を合わせたものも1つのモジュールと見ることが出来る。

情報隠蔽：

プログラムを作る上での設計指針の1つに、

> 各モジュールの外部インターフェースとして必要なものを用意し、モジュール内部の実装に関わる細部は外部からは見えない様にする

というものがある。（ソフトウェアの信頼性のため）

C言語においては、関数以外のモジュールに対してこの情報隠蔽を進めるために静的外部変数や静的関数を導入することが出来る。

例8．3（静的外部変数，静的関数）

```
static int  i;          /* 静的外部変数 */
                        /* 別のファイルからは参照できない */

static void f(...);     /* 静的関数のプロトタイプ */

void g1(...)            /* 外部とのインターフェース */
{
  .........             /*ここでは配列aを参照できない*/
}

void g2(...)            /* 外部とのインターフェース */
{
  .........             /*ここでは配列aを参照できない*/
}

static int  a[100];     /* 静的外部配列 */
                        /*別のファイルからは参照できない*/
                        /*上の関数 g1, g2 からも参照できない*/

static void f(...)      /* 静的関数 */
{                       /*別のファイルからは呼び出せない*/
  .........
}
```

例題8．4（eの1000桁計算） 個々の関数をモジュールと見るのではなく、関連するデータとそれらを管理する関数群の集まりをモジュールと見る観点に立てば、例題8.1で挙げたプログラムは、

① 小数点以下1000桁の10進小数を記憶するint型配列 e[] とこのデータ領域を管理／操作する関数

set_array_to_keep_integer,
divide_array_value_by_integer,
add_integer_to_array,
print_array_value

の集まり、

② 外部インターフェース用に提供されている 4 つの関数 (set_array_to_keep_integer, ...) を然るべき順序に呼び出すことによって①のモジュール内のデータ領域 e にNapier数が記録されるようにしている部分、

の2つのモジュールに分けることが出来る。この考えの下で例題8.1のプログラムを構成し直してみよ。

(考え方)　ここで言う「モジュール」の実体は1つのソースファイルに他ならない。

例題8.1では

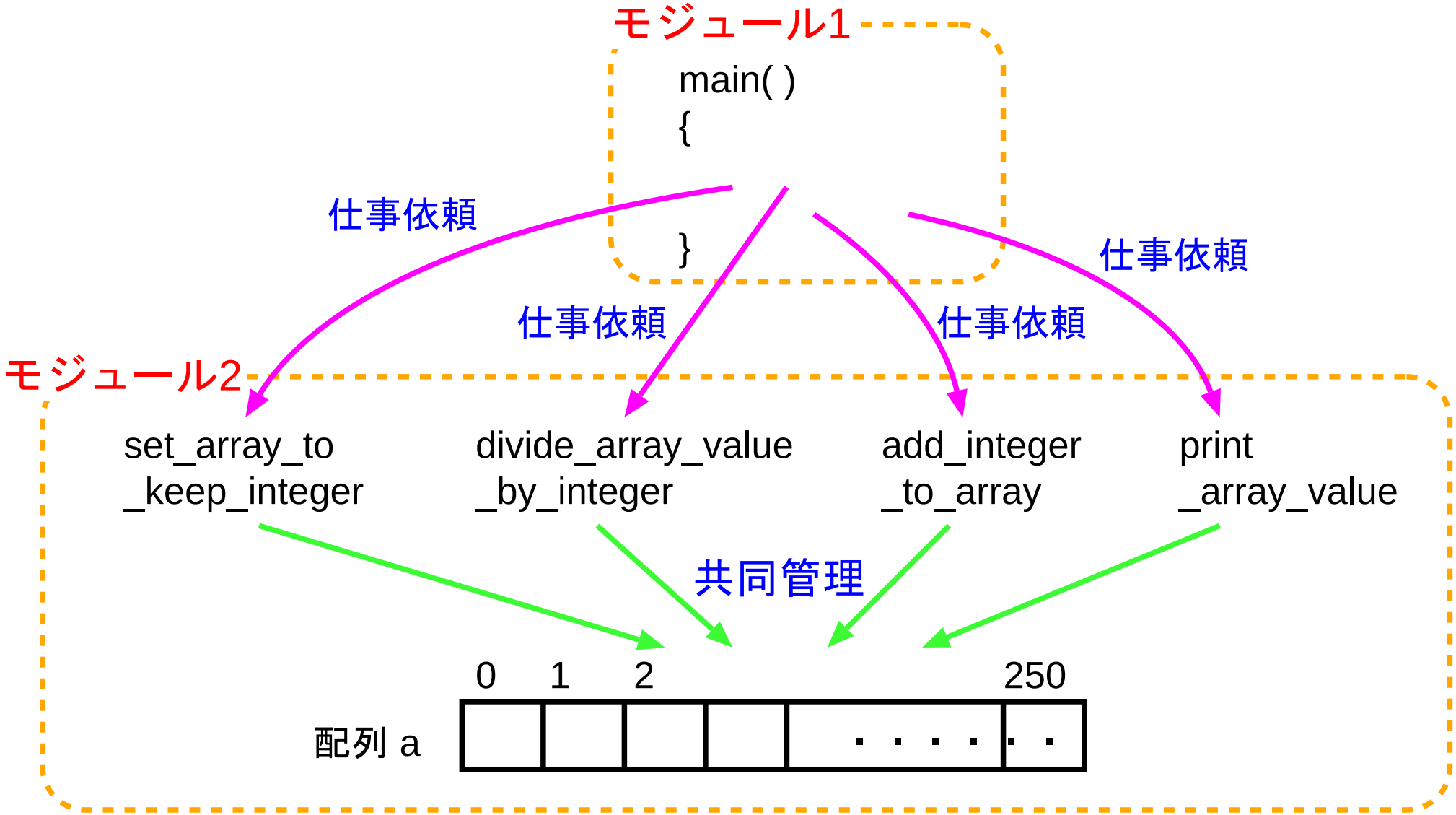
int型配列　e[] はmain関数内で確保 し、main内で他の関数を呼び出す際に その配列のアクセス権を呼び出し先の関数にパラメータとして引き渡し ていた。

これに対して、指示された観点に立てば、
配列 e[] はこれを直接使う関数群の共有するものなので、

- e[] は静的外部配列として確保され、この配列を共有する関数群と合わせて1つのソースファイルを作り、これを1つのモジュールと見る。

そして、

- 残ったmain関数(とmainから呼び出す関数のプロトタイプ)を基に別のソースファイルを作り、これを2つ目のモジュールと見る。



モジュール1の中では小数点以下1000桁の10進小数がどういう風に表されているかについては全く関知しない、従って小数点以下1000桁の10進小数の表し方はモジュール2の内部に隠蔽されている。

(再構成例)

```
[motoki@x205a]$  nl modules-napier-sub1.c
     1  /*****************************************************
     2  /* Napier数 e の1000桁計算
     3  /*-------------------------------------
     4  /*     別途用意された、
     5  /*        (1)小数点以下1000桁の10進小数を記憶するための...
     6  /*        (2)そのデータ領域を操作するための関数群
     7  /*     から成るモジュールに然るべき順序に然るべき指令を出..
     8  /*     ことによってNapier数 e の値を小数点以下1000桁ま...
     9  /*     するモジュール
    10  /*     (このモジュールの中では、小数点以下1000桁の10進...
    11  /*       どういう風に表されているかについては全く関知しな...
    12  /*****************************************************

    13  void set_array_to_keep_integer(int init_value);
    14  void divide_array_value_by_integer(int divisor);
```

```
15  void add_integer_to_array(int value);
16  void print_array_value(char *padding);

17  int main(void)
18  {
19    int  k;

20    set_array_to_keep_integer(1);        /* e <-- 1 */

21    for (k=450; k>0; --k) {
22      divide_array_value_by_integer(k);  /* e <-- e/k */
23      add_integer_to_array(1);           /* e <-- e+1 */
24    }

25    printf("e =");
26    print_array_value("   ");            /* 結果の出力 */
27    return 0;
```

```
    28  }
[motoki@x205a]$  nl modules-napier-sub2.c
     1  /*****************************************************
     2  /* 小数点以下1000桁の10進小数を
     3  /*        e[0]=整数部の数,
     4  /*        e[1]=小数点以下1~ 4桁目の数,
     5  /*        e[2]=小数点以下5~ 8桁目の数,
     6  /*            ......
     7  /*        e[250]=小数点以下996~ 1000桁目の数
     8  /* という風に保存する配列 e と、
     9  /* このデータ領域を管理する関数群のモジュール
    10  /*****************************************************

    11  #include  <stdio.h>
    12  #define   LIMIT    7
    13  #define   SIZE   251
```

```
14  static  int  e[SIZE];

15  /*-------------------------------------------------------
16  /* 10進小数を表す静的外部配列 e の値を初期設定する
17  /*-----------------------------------------
18  /* (仮引数) init_value : 初期設定したい値(int型)
19  /* (関数値)    : なし
20  /* (機能) :  静的外部配列 e の表す値が init_value に
21  /*            なる様にする
22  /*-------------------------------------------------------
23  void set_array_to_keep_integer(int init_value)
24  {
25    int i;
26
27    e[0] = init_value;
28    for (i=1; i<SIZE; ++i)
29      e[i] = 0;
```

```
30  }

31  /*-----------------------------------------------------------
32  /* 10進小数を表す静的外部配列 e の値を整数で割る
33  /*-----------------------------------------
34  /* (仮引数) divisor : 除数
35  /* (関数値)    : なし
36  /* (機能) :  静的外部配列 e の表す値を整数 divisor で割.
37  /*-----------------------------------------------------------
38  void divide_array_value_by_integer(int divisor)
39  {
40    int i;
41
42    for (i=0; i<SIZE-1; ++i) {
43      e[i+1] += (e[i] % divisor) * 10000;
44      e[i]   /= divisor;
45    }
```

```
46    e[SIZE-1] /= divisor;
47  }

48  /*-----------------------------------------------------------
49  /* 10進小数を表す静的外部配列 e の値に整数を加算する
50  /*-----------------------------------------
51  /* (仮引数) addend : 加数
52  /* (関数値)    : なし
53  /* (機能) :  静的外部配列 e の表す値に整数 addend を加..
54  /*-----------------------------------------------------------
55  void add_integer_to_array(int addend)
56  {
57    e[0] += addend;
58  }

59  /*-----------------------------------------------------------
60  /* 10進小数を表す静的外部配列 e の値を出力する
```

```
61  /*-----------------------------------------
62  /* (仮引数) padding : 2行目以降の左端に必ず出力する文...
63  /* (関数値)    : なし
64  /* (機能) :  静的外部配列 e の表す10進小数値を出力する.
65  /*            その際、
66  /*              *小数点以下は 8*LIMIT 桁毎に改行する,
67  /*              *2行目以降の出力においては左端に必ず文..
68  /*                padding を埋める。
69  /*-----------------------------------------------------
70  void print_array_value(char *padding)
71  {
72    int i, count;
73
74    printf("%2d.", e[0]);
75    count = 1;
76    for (i=1; i<SIZE; i+=2, ++count) {
77      printf("%04d%04d ", e[i], e[i+1]);
```

```
78        if (count >= LIMIT) {
79          printf("\n%s   ", padding);
80          count = 0;
81        }
82      }
83      printf("\n");
84    }
```

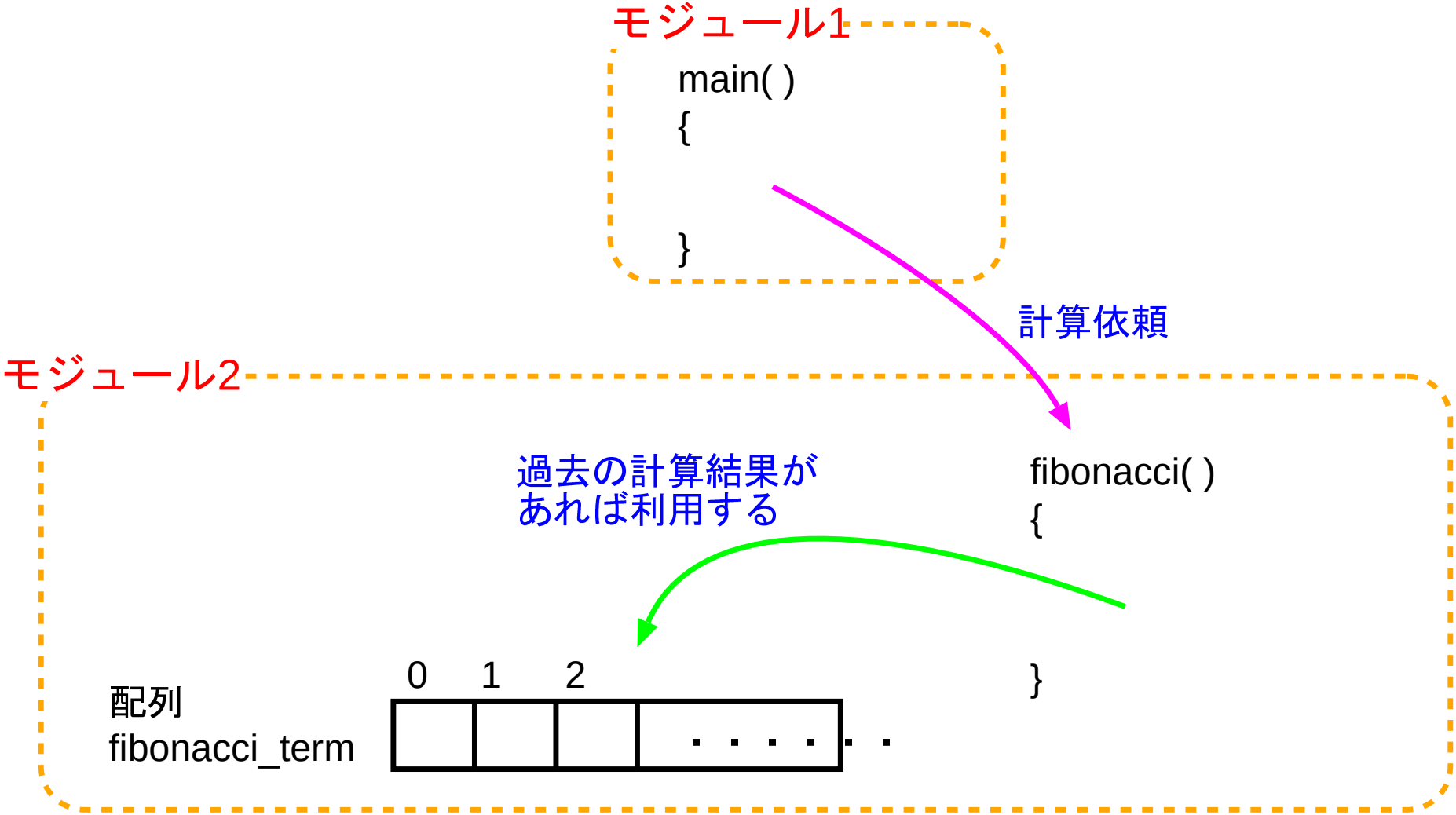
```
[motoki@x205a]$  gcc modules-napier-sub1.c modules-napier-sub2

[motoki@x205a]$  ./a.out
e = 2.71828182 84590452 35360287 47135266 24977572 47093699 95
      69676277 24076630 35354759 45713821 78525166 42742746 63
                (途中省略)
      15747704 17189861 06873969 65521267 15468895 70350354
[motoki@x205a]$
```

□演習８．４（階乗の表）演習8.2で作ったプログラムを例題8.4に倣って2つのモジュールに分けてみよ。

例題8．5（連想計算;Fibonacci数列の再帰計算を効率的にする） 例7.6において、フィボナッチ数列を定義した漸化式に基づいて数列の項を再帰計算すると異常に非効率な計算になることを示した。これに対して、漸化式に(ほぼ)忠実に従った計算方法を保ったまま計算の非効率性を避けることが出来るかどうか考えよ。

(考え方)　Fibonacci数列を計算するモジュール内に静的外部配列を用意して、この配列に過去に計算したことのある計算結果を保存する。そして、漸化式に従った計算を続ける前に過去に計算結果が保存されているかどうかをチェックする様にすれば、同じ計算を何度も繰り返す無駄を省くことが出来る。



(プログラミング)

```
[motoki@x205a]$  nl modules-fibonacci-sub1.c
     1  #include <stdio.h>

     2  int  fibonacci(int n);

     3  int main(void)
     4  {
     5    int  i;

     6    scanf("%d", &i);
     7    printf("fibonacci(%d) = %d\n", i, fibonacci(i));
     8    return 0;
     9  }
[motoki@x205a]$  nl modules-fibonacci-sub2.c
     1  #define  SIZE  100
     2  #define  TRUE  1
```

```
 3  typedef  int  Boolean;

 4  static Boolean Already_computed[SIZE];
                                /* 過去に計算したことがあるか */
 5                              /* どうかを記憶する静的外部変数; */
 6                              /* ゼロ(false)に初期化される。    */
 7  static int     fibonacci_term[SIZE];
                                /* 過去の計算結果をここに保存 */
 8  static int record(int num, int value);

 9  int fibonacci(int n)
10  {
11    printf("called fibonacci(%d)\n", n);
12          /* どういう計算が行われるかを観察するために */
13    if (Already_computed[n])
14      return fibonacci_term[n];
```

```
15
16    if (n <= 1)
17      return record(n, n);
18    else
19      return record(n, fibonacci(n-1)+fibonacci(n-2));
20  }

21  static int record(int n, int value)
22  {
23    Already_computed[n] = TRUE;
24    fibonacci_term[n]   = value;

25    return value;
26  }
```

```
[motoki@x205a]$  gcc modules-fibonacci-sub1.c modules-fibonacc

[motoki@x205a]$  ./a.out
```

```
6
called fibonacci(6)
called fibonacci(5)
called fibonacci(4)
called fibonacci(3)
called fibonacci(2)
called fibonacci(1)
called fibonacci(0)
called fibonacci(1)
called fibonacci(2)
called fibonacci(3)
called fibonacci(4)
fibonacci(6) = 8
[motoki@x205a]$
```

例題8．6（疑似乱数発生） 標準ライブラリ関数 `int rand()` を用いずに、疑似乱数発生のための汎用のモジュールを作成せよ。

(考え方)　疑似乱数を発生させる最も単純な方法は線形合同法と呼ばれる方法で、この方法では定数 a,b,N を定めて、

$$r_{k+1} \longleftarrow \mathrm{mod}(a \times r_k + b,\ N)$$

という式に基づいて現在の疑似乱数値 r_k から次の疑似乱数値 r_{k+1} を求める。

結果的に、初期値 r_0 を与えてやれば 0~ N−1 の間の整数列

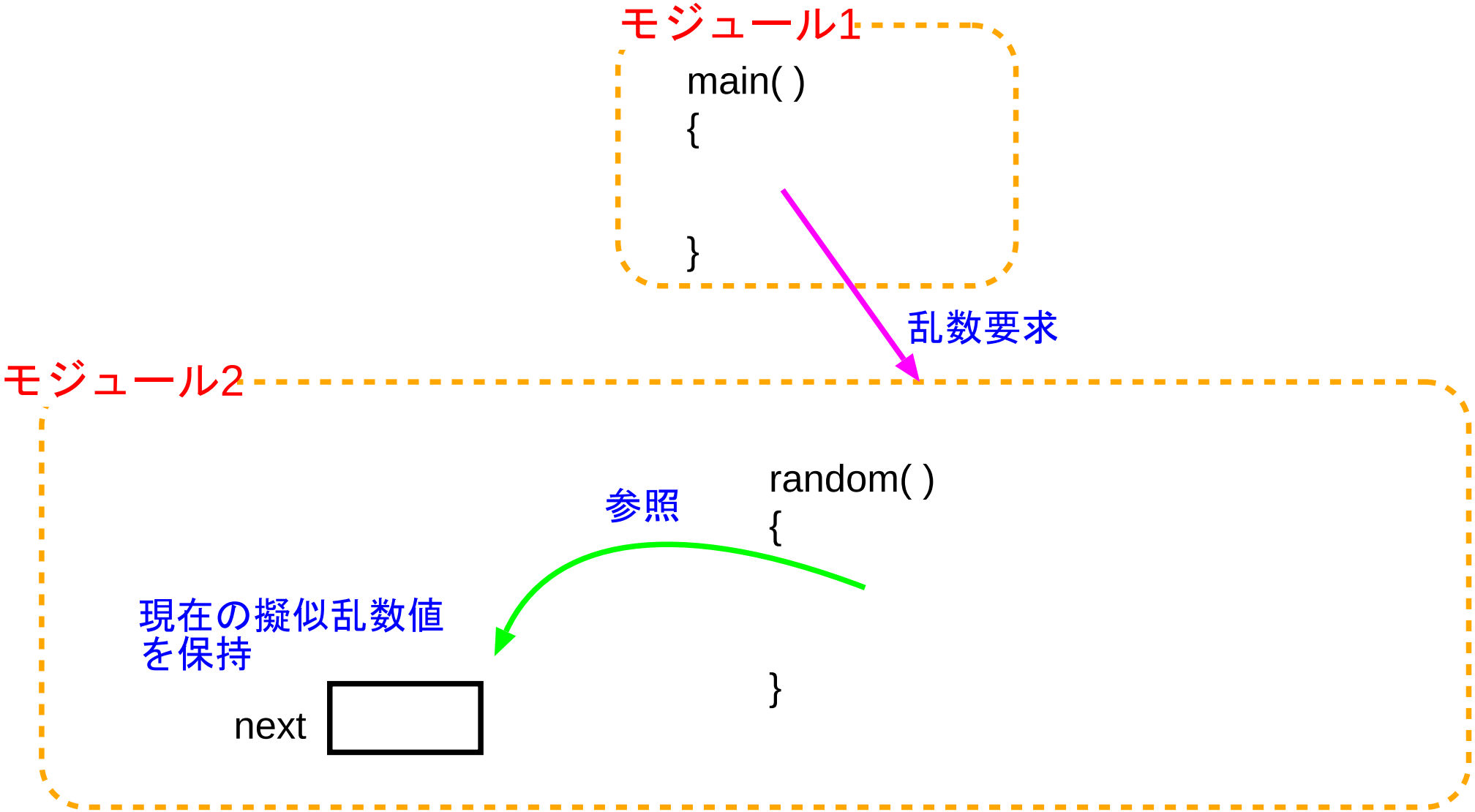
$$r_0,\ r_1,\ r_2,\ r_3,\ \ldots$$

が決まるので、これを乱数の列と見做そうというのである。

この線形合同法で疑似乱数列を発生させる際は、

- 現在の疑似乱数値を保持する変数,
- 現在の疑似乱数値を更新してその結果を返す関数

の2つから成るモジュールを考えれば良い。


モジュール1
main( )
{
}
乱数要求
モジュール2
random( )
{
}
参照
現在の擬似乱数値
を保持
next

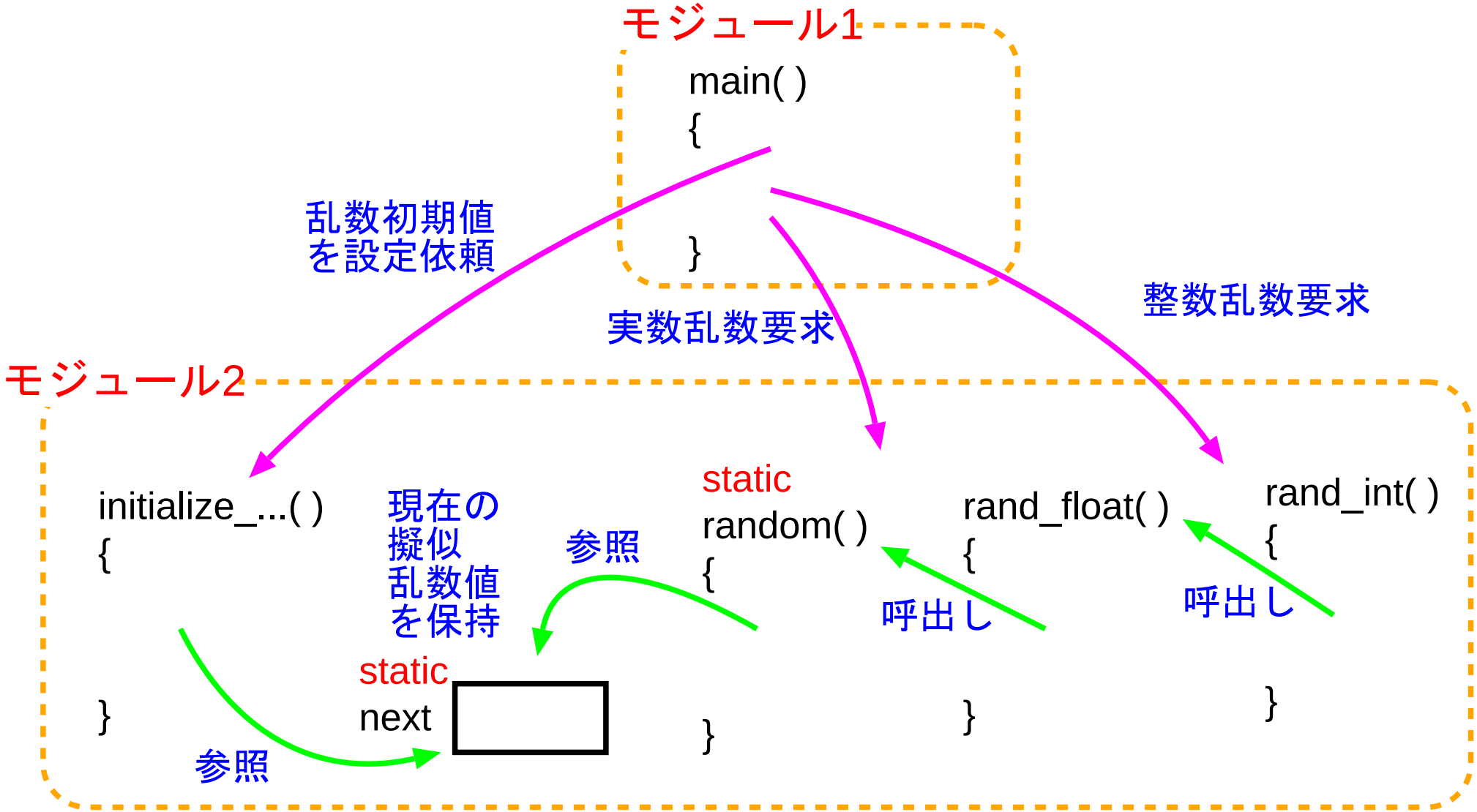
(プログラミング)　a=1103515245, b=12345, N=2^{32} としてプログラムを構成した。

また、生成する疑似乱数列の明白な周期性を避けるために現在の疑似乱数値を構成する32ビットの下位から 31~ 17ビット目を取り出し、

使い易さのために、その取り出した値を基に

- 指定された数未満の非負整数を生成する関数,
- 0以上1未満の実数値を生成する関数

の2つもモジュールに追加した。


モジュール1
main( )
{
}
乱数初期値
を設定依頼
実数乱数要求
整数乱数要求
モジュール2
initialize_...( )
{
}
現在の
擬似
乱数値
を保持
static
next
参照
参照
static
random( )
{
}
呼出し
rand_float( )
{
}
呼出し
rand_int( )
{
}

```
[motoki@x205a]$  nl modules-random.c
     1  /*****************************************************
     2  /* (1) 指定された数以下の非負整数をランダムに割り当てる...
     3  /* および (2) [0,1) 内の実数をランダムに割り当てる関数...
     4  /* を提供するモジュール
     5  /*-----------------------------------------------------
     6  /*      ここでは、標準ライブラリで提供された疑似乱数生成...
     7  /*      とほぼ同等のものを示す。(あまり良くない。)
     8  /*****************************************************

     9  static unsigned long next=1;

    10  double     rand_float(void);
    11  static int random(void);

    12  /*-----------------------------------------------------
    13  /* random seed を初期設定する関数
```

```
14  /*-----------------------------------------
15  /* (仮引数) seed : random seed の初期値を記憶した変数
16  /* (関数値)  :  なし
17  /*-----------------------------------------------------
18  void initialize_randomizer(unsigned long seed)
19  {
20    next = seed;
21  }

22  /*-----------------------------------------------------
23  /* 指定された数未満の非負整数をランダムに割り当てる関数
24  /*-----------------------------------------
25  /* (仮引数) to : 生成すべき乱数値の上限(+1)を記憶した...
26  /* (関数値)  :  区間 [0, to) 内の整数疑似乱数値
27  /*-----------------------------------------------------
28  int rand_int(int to)
29  {
```

```
30    int  k;

31    k = (int) (rand_float() * to);
32    return (k==to ? to-1 : k);
33  }

34  /*-----------------------------------------------------
35  /* [0,1) 内の実数をランダムに割り当てる関数
36  /*-----------------------------------------
37  /* (仮引数)  : なし
38  /* (関数値)  :  区間 [0, 1) 内の実数疑似乱数値
39  /*-----------------------------------------------------
40  double rand_float(void)
41  {
42    return (double)random() / 32768;
43  }
```

```
44  /*-----------------------------------------------------
45  /* 次の疑似乱数に移るために用意された局所的な関数
46  /*-----------------------------------------
47  /* (仮引数)  : なし
48  /* (関数値)  :  0 以上 32767 以下の疑似整数乱数値
49  /* (機能) :  疑似乱数生成のためのに用意した静的外部変数
50  /*            next の値を線形合同法で更新し、結果のnext の
51  /*            を基に 0 以上 32767 以下の疑似整数乱数値を.
52  /*-----------------------------------------------------
53  static int random(void)
54  {
55    next = next*1103515245 + 12345;
56    return (unsigned int)  ((next/65536) % 32768) ;
57  }
```

変数 next の下から17~ 31ビット目の部分を取り出す操作

このモジュールは色々なモジュールと組み合わせて使うことが出来ます。

例えば次の通り。

```
[motoki@x205a]$  nl modules-random-main.c
     1  #include <stdio.h>

     2  void initialize_randomizer(unsigned long seed);
     3  int rand_int(int to);
     4  double rand_float(void);

     5  int main(void)                /* こちら側からは                *
     6  {                             /* modules-random.c 内の静的
     7    int i;                      /* 外部変数 next は見えない  *
     8    unsigned long seed;

     9    printf("Input a random seed:  ");
    10    scanf("%d", &seed);
    11    initialize_randomizer(seed);
```

```
12    for (i=0; i<7; ++i)
13      printf("%6d", rand_int(100));
14    printf("\n");
15    for (i=0; i<5; ++i)
16      printf("%f ", rand_float());
17    printf("\n");
18    return 0;
19  }
```

```
[motoki@x205a]$  gcc modules-random-main.c modules-random.c
[motoki@x205a]$  ./a.out
Input a random seed:   333
    11     1    58    91    11    68    27
0.380646 0.934570 0.318390 0.709808 0.658264
[motoki@x205a]$
```
