7 関数（その2）

7–1 4つの記憶領域のクラス auto, extern, register, sta

C言語においては、変数(,配列)や関数はデータ型の他に、

- 割当てられる記憶域が局所的に使われるかどうか、
- 永続的に使われるかどうか、
- 高速性が要求されるかどうか、

といった利用区分を属性として持つ。(記憶域クラスという)

⇨ 次の4種類の記憶域クラスが用意されている。

```
auto
extern
register
static
```

記憶域クラスauto：

ブロック（関数本体も含む）に入ると自動的に新たに割り付けられ、ブロックから出ると割り付けが解除される記憶領域を指す。

⇨ この記憶域クラスの変数は自動変数と呼ばれ、ブロックの中で局所的なものとなる。

例えば 、関数本体の最初に宣言されている変数や配列は暗黙にautoとして処理される。

記憶域クラスextern：
全てのブロックの外で確保された記憶領域を指す。

⇨ この記憶域クラスの変数は外部変数と呼ばれ、
大域的なものとなる。

例えば 、関数や、関数の外で宣言された変数
(,配列)はこのクラスに属する。

変数宣言の際にデータ型の前に extern という修飾子を付けると 、
この変数が大域的で、ブロックの外、
あるいは別ファイルの中で確保されている
ことをコンパイラに知らせたことになる。

注意：
- 外部変数を使い過ぎると 、副作用のために関数同士の独立性がなくなり、分かりにくいプログラムになる恐れがある。

−−−

記憶域クラスregister：

変数宣言の際にデータ型の前に register という修飾子を付けると 、
この変数領域に高速レジスタを割り付けてほしい
ことをコンパイラに知らせたことになる。

レジスタの割り付けが不可能な場合はautoクラスとして扱われる。

記憶域クラスstatic：

ブロックの中で変数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 、この変数領域は
ブロックに付随した固有の変数
として永続的に生き続ける。

すなわち 、ブロックから出てもその記憶領域は保存され、その値は次にブロックに入った時にそのまま引き継がれる。

例7．1（extern宣言）次のような2つのソースファイル

func-extern-pt1-Kelley.c と func-extern-pt2-Kelley.c

を考える。

```
[motoki@x205a]$  cat -n func-extern-pt1-Kelley.c
     1 #include <stdio.h>
     2
     3 int  f(void);                /* 関数プロトタイプ */
     4
     5 int  a=1, b=2, c=3;          /* 外部変数 */
     6
     7 int main(void)
     8 {
     9   printf("%3d\n", f());
    10   printf("%3d%3d%3d\n", a, b, c);
    11   return 0;
    12 }
[motoki@x205a]$  cat -n func-extern-pt2-Kelley.c
     1 int f(void)
     2 {
     3   extern int  a;
     4   int  b, c;                 /* 自動変数 */
     5
     6   a = b = c = 4;
     7   return  a+b+c;
```

```
8 }
```

これらのソースファイルに関して、

コンパイル例(1)

func-extern-pt2-Kelley.cの3行目で宣言されている変数 a はextern宣言されているので、こちらの処理単位の中で記憶域は確保されない。

⇨ このブロックもしくはファイルの外で a という外部変数が宣言されていれば func-extern-pt2-Kelley.c のコンパイルは成功する。

```
[motoki@x205a]$ gcc func-extern-pt1-Kelley.c \
                    func-extern-pt2-Kelley.c
[motoki@x205a]$ ./a.out
 12
  4  2  3
[motoki@x205a]$
```

コンパイル例(2)

コンパイルのみ(-cオプション)のccコマンドが成功したとしても、
a という外部変数が外で宣言されていなければ、
リンクの時点でやはりエラーになる。

```
[motoki@x205a]$ gcc -c func-extern-pt2-Kelley.c
[motoki@x205a]$ gcc func-extern-pt2-Kelley.o
/usr/lib/crt1.o: In function `_start':
/usr/lib/crt1.o(.text+0x18): undefined reference to `main'
/tmp/ccnaNiDG.o: In function `f':
/tmp/ccnaNiDG.o(.text+0x16): undefined reference to `a'
/tmp/ccnaNiDG.o(.text+0x1f): undefined reference to `a'
collect2: ld returned 1 exit status
[motoki@x205a]$
```

コンパイル例(3)

func-extern-pt2-Kelley.c で、もし3行目の extern int の宣言が無いと外に a という外部変数が宣言されていたとしても3行目の a はその外部変数として認識されない。

```
[motoki@x205a]$  cat -n func-extern-pt2a-Kelley.c
     1 int f(void)
     2 {
     3   /* extern int  a; */
     4   int  b, c;                /* 自動変数 */
     5
     6   a = b = c = 4;
     7   return  a+b+c;
     8 }
[motoki@x205a]$  gcc func-extern-pt1-Kelley.c func-extern-pt2a

func-extern-pt2a-Kelley.c: In function ‘f’:
func-extern-pt2a-Kelley.c:6: ‘a’ undeclared (first use in this
func-extern-pt2a-Kelley.c:6: (Each undeclared identifier is re
func-extern-pt2a-Kelley.c:6: for each function it appears in.)
[motoki@x205a]$
```

7–2 自習 暗黙の初期化

auto変数
register変数
} ··· 暗黙の初期化は期待できない。

extern変数
static変数
} ··· C言語処理系によってゼロに初期化される。

7–3 自習 関数パラメータの受渡し方法

実引数と仮引数の対応付け：

関数呼び出しの際には、関数呼び出し側の実引数（または実パラメータ）と関数定義側の仮引数（または仮パラ メータ）の結合／対応付けが行われる。 引数結合の方式としては 次の2つが一般によく用いられている。

- 値呼出し (call by value)
  実引数として与えられた式が評価／計算され、その値が仮引数の変数の初期値として使われる。
- 参照呼出し (call by reference)
  実引数として与えられた変数の記憶領域と仮引数の変数領域を同一視する。従って、呼び出された関数が直接呼出し側の変数を操作することになる。

これらの内C言語で行えるのは値呼出しのみであるが、変数の主記憶内での番地（ポインタという）を関数に引き渡すことにより参照呼出しと同等のことも行える。（例7.2）

補足：

番地を値とする変数のことを
「ポインタ」と呼ぶ教科書もある。

関数実行のプロセス：

関数呼出しがあると、その処理は次のような順序で進む。

（1）各々の実引数を評価。

（2）（1）の結果を対応する仮引数のデータ型に変換

（3）（2）の結果を対応する仮引数（変数）に代入。

（(1)〜(3)：値呼出し）

（4）関数の本体を実行する。実行の途中に、

（場合1）`return;` という文に出会うと、
制御を呼出し元に戻す。（関数値なし）

（場合2）本体の実行が終了すると、
制御を呼出し元に戻す。（関数値なし）

（場合3）`return 式;` という文に出会うと、
式 の値を評価し、その値をその関数が本来返すべきデータ型に変換する。 そして、その結果を関数値として制御を呼出し元に戻す。

次の例題は、

①C言語においては引数結合が値呼出しによって行われていること、そして

②値呼出しを用いて参照呼出しと同等のことも行えること

を説明している。

例題7．2（値呼出し，参照呼出し）次のCプログラムを実行するとどういう出力が得られるか？ 下の □ の部分に予想される出力文字列を入れよ。但し、ここでは空白は ␣ と明示せよ。

```
[motoki@x205a]$ nl func-binding-parameters.c  Enter
     1 #include <stdio.h>
     2 void call_by_value(int);
     3 void call_by_reference(int *);

     4 int main(void)
     5 {
```

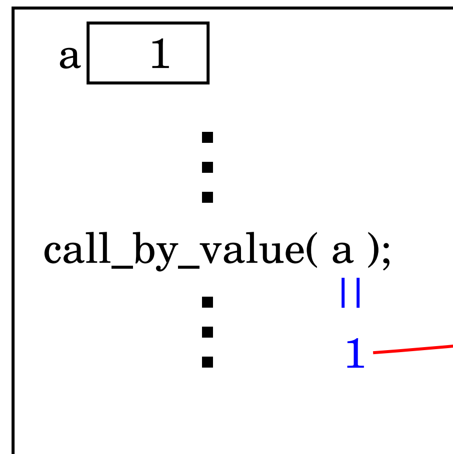
```
 6   int  a=1;

 7   printf("%d\n", a);
 8   call_by_value(a);       /* 値呼出し*/
 9   printf("%d\n", a);       /* aの値は不変！*/

10   call_by_reference(&a); /* 参照呼出し*/
11   printf("%d\n", a);       /* aの値は変わる！*/
12   return 0;
13 }

14 void call_by_value(int a)
15 {
16   a = 777;
17 }

18 void call_by_reference(int *a)
19 {
20   *a = 777;
21 }
```

```
[motoki@x205a]$ gcc func-binding-parameters.c  Enter
[motoki@x205a]$ ./a.out  Enter
```

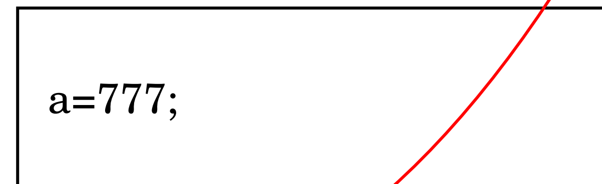
(考え方)　C言語では、
関数引数の結合が値呼出しによって行われる
から、

- もし実行が8行目に移り `call_by_value()` が次に実行されれば、
  次の様に実行が進む。
  (8行目, 引数結合)

int main(void)



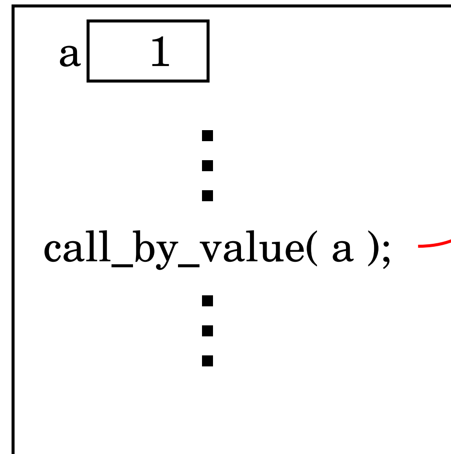
void call_by_value(int a)



引数結合
(代入)

(8行目，関数呼び出し)

int main(void)

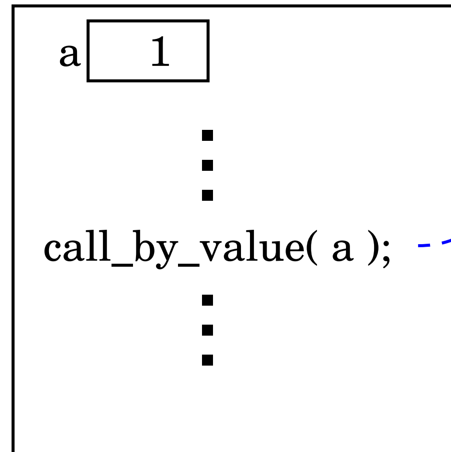


void call_by_value(int a 1)

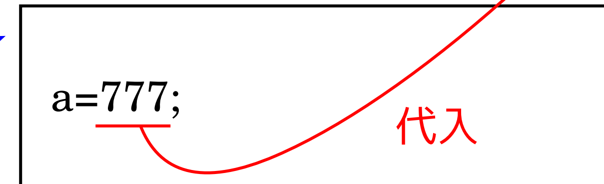
呼出し

(16行目，実行後)

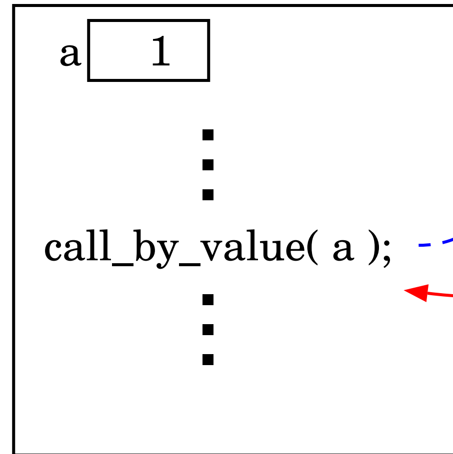
int main(void)

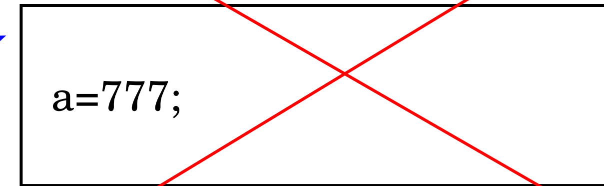


void call_by_value(int a 777)



（17行目，関数実行終了）

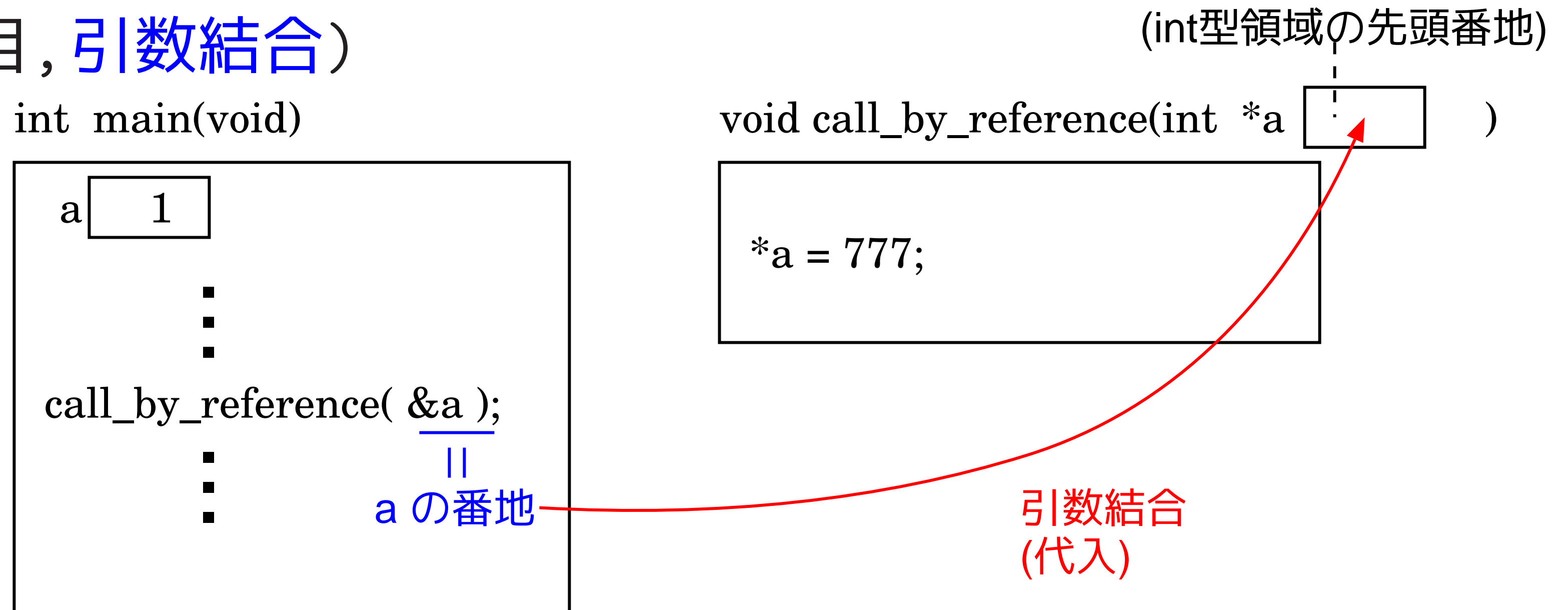




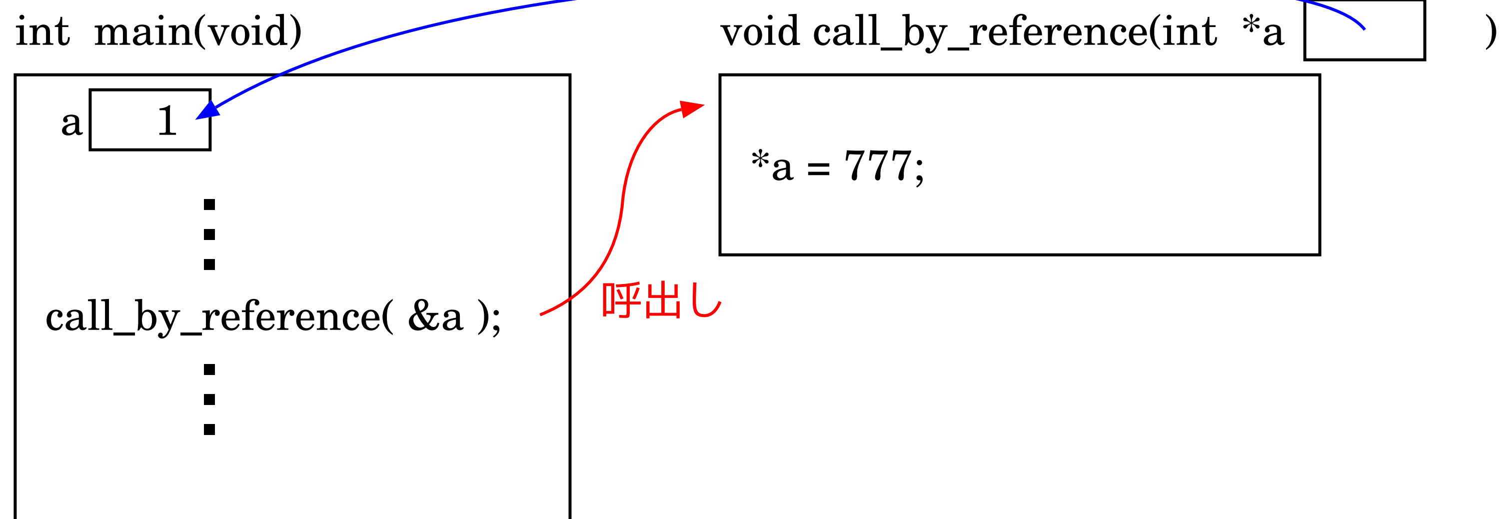
⟹ 8行目の関数実行終了後も6行目の a の値は 1 のまま変わらない。

- もし実行が10行目に移り call_by_reference() が次に実行されれば、次の様に実行が進む。

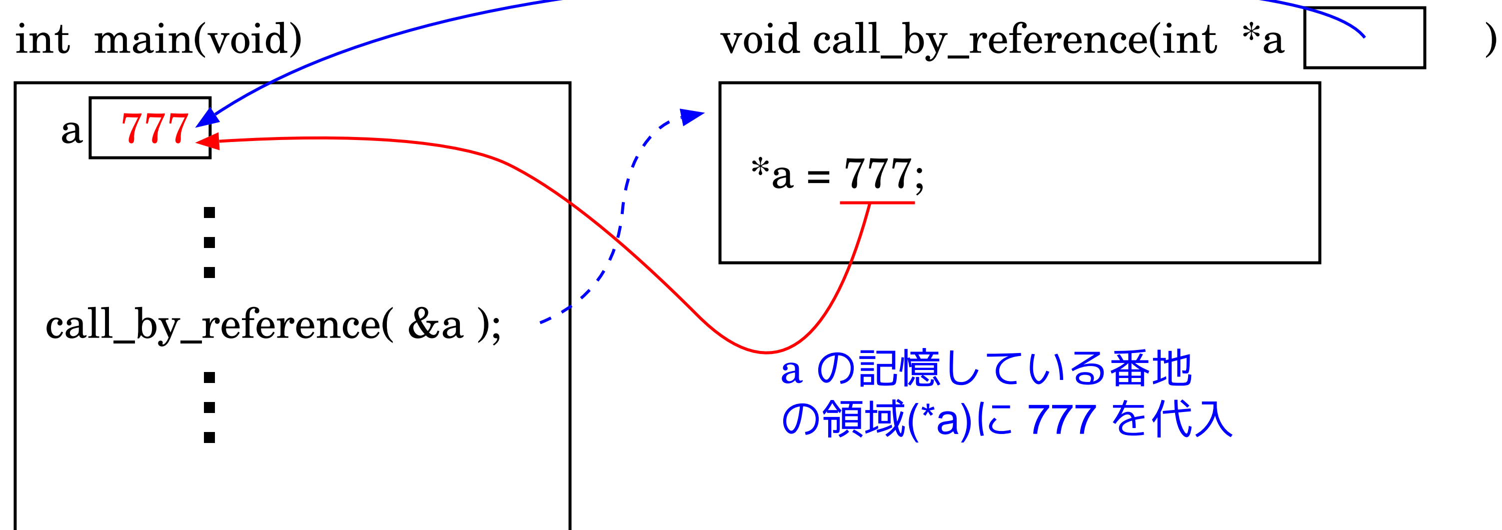
(10行目, 引数結合)



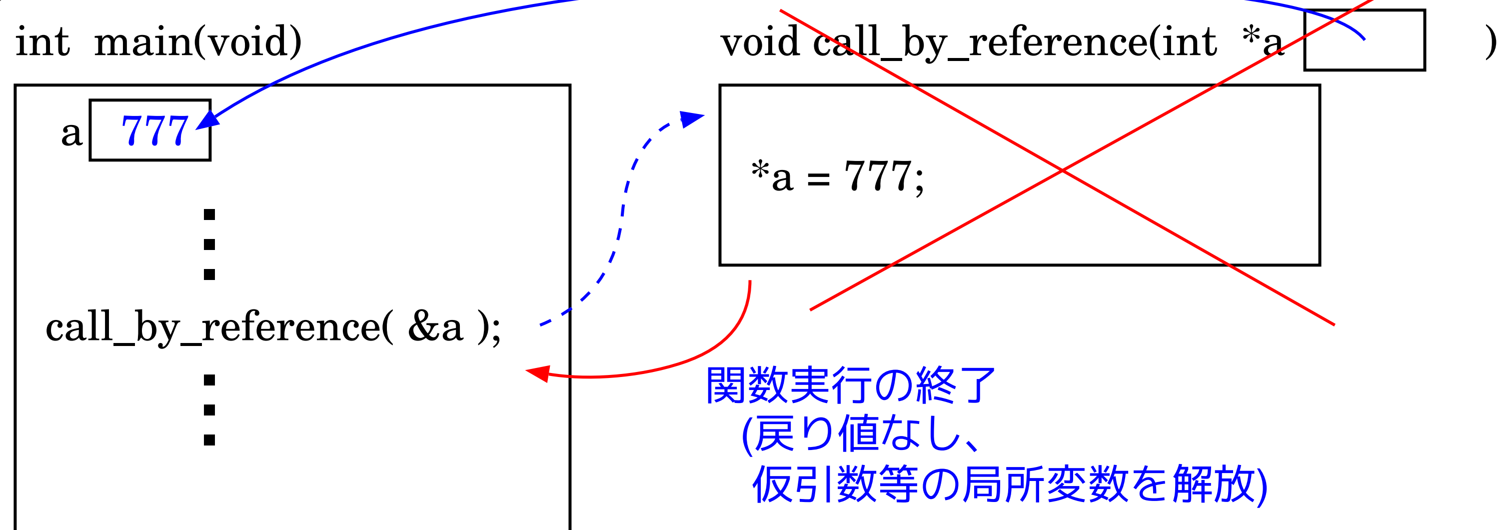
(10行目, 関数呼び出し)



(20行目, 実行後)



(21行目, 関数実行終了)



$\Longrightarrow$ 10行目の関数実行によって6行目の a の値は 777 に変わる。

(実行結果)　結局、プログラムの

- 7行目では a の値は 1,
- 9行目では a の値は 1,
- 11行目では a の値は 777

になるから、実行結果は次の様になる。

```
[motoki@x205a]$  ./a.out  Enter
1
1
777
[motoki@x205a]
```

番地演算子 & と間接演算子 * :

&v ··· 変数 v へのポインタ(≈番地)。

*p ··· ポインタ p の指す記憶領域、
すなわち、p番地の記憶領域。

参照呼出しと同等のことを行なう方法:

- 参照呼出しの仮引数は、ポインタとして宣言する。
- 関数の本体部では、参照呼出しの仮引数は間接演算子 * を付けて使う。
- 関数を呼ぶ時、参照呼出しの実引数として変数等へのポインタ(i.e. 番地)を与える。

7–4 一次元配列を関数パラメータとして受渡しする方法

配列データの受渡しを行いたい場合、配列要素毎に値呼出しによる引数結合を行っていたのでは引数結合に相当の時間がかかってしまう。

⇨ C言語では、配列データの受渡しを行いたい場合には、その配列（の先頭要素）へのポインタを呼び出し先の関数に引き渡す様にする。

例題7．3（Quicksort;外部配列を使わない版） 例題 4.5 で **Quicksort** のプログラムを提示した時は、並び替える要素の入った配列 a[SIZE] を外部配列としてこの配列を 5 つの関数 main, set_an_array_random, pretty_print, quicksort, partition に共有させていたが、これではこれら 5 つの関数はこの配列についてのサービスを提供する特殊な関数としての役割しか持たない。 そこで、各々の関数が独立なモジュール(i.e. プログラム部品) として働く様に、例題4.5のプログラムを修正せよ。

(考え方)　5つの関数が共有して使うことになる配列 a[SIZE] はmain関数の中で確保し、残りの関数を呼び出す際にはその配列領域の先頭番地とその配列の大きさを関数パラメータとして受け渡す様にすれば良い。

(プログラミング)　修正例を次に示す。

```
[motoki@x205a]$ nl function-quicksort-2.c
     1  /*****************************************************
     2  /* Quicksort   :  一次元配列を関数パラメータとして受渡... *
     3  /*-------------------------------------------
     4  /*     大きさ100の配列にランダムに整数を生成し、その配... *
     5  /*     Quicksortアルゴリズムを使って昇順に並べ替えて出力..
     6  /*****************************************************

     7  #include <stdio.h>
     8  #include <stdlib.h>      /* 乱数発生のライブラリ関数を使.
```

```
 9  #define  SIZE   100
10  #define  WIDTH  10
11  #define  TRUE    1

12  void set_an_array_random( int x[], int size );
13  void pretty_print( int x[], int size );
14  void quicksort( int x[],  int from, int to);
15  int  partition( int x[],  int from, int to);

16  int main(void)
17  {
18    int   a[SIZE],  seed;    /* a[SIZE] を外部配列には...

19    printf("Input a random seed (0 - %d):  ", RAND_MAX);
20    scanf("%d", &seed);
21    srand(seed);
```

```
22      set_an_array_random( a, SIZE );
23      printf("\nbefore sorting:\n");
24      pretty_print( a, SIZE );

25      quicksort( a,  0, SIZE-1);
26      printf("\nafter sorting:\n");
27      pretty_print( a, SIZE );
28      return 0;
29    }

30    /*----------------------------------------
31    /* 引数で与えられた配列の各要素をランダムに設定
32    /*--------------------------------------
33    /* (仮引数)  x     : int 型配列
34    /*            size : int 型配列 x の大きさ
35    /* (機能) :   配列要素  x [0]〜 x [ size -1] に 0〜999 の
36    /*            設定する。
```

```
37  /*------------------------------------------
38  void set_an_array_random( int x[], int size )
39  {
40    int i;

41    for (i=0; i< size ; ++i)
42      x [i] = rand() % 1000;
43  }

44  /*------------------------------------------
45  /* 引数で与えられた配列の要素を順番に全て出力する
46  /*------------------------------------
47  /* (仮引数)  x      : int型配列
48  /*            size : int型配列 x の大きさ
49  /* (機能) :   配列要素  x [0]~ x [ size -1] の値を順番.
50  /*            する。但し、各々の値は横幅7カラムのフィー...
51  /*            に出力することにし、また、1行にWIDTH個の...
```

```
52  /*                を出力する。
53  /*---------------------------------------------
54  void pretty_print( int x[], int size )
55  {
56    int i, count=1;

57    for (i=0; i< size ; ++i, ++count) {
58      printf("%7d",  x [i]);
59      if (count >= WIDTH) {
60        printf("\n");
61        count = 0;
62      }
63    }
64    if (count > 1)
65      printf("\n");
66  }
```

```
67  /*-----------------------------------------
68  /* 引数で与えられた配列要素を小さい順に並べ替える
69  /*--------------------------------
70  /* (仮引数)  x    : int 型配列
71  /*           from : int 型配列  x  の添字
72  /*           to   : int 型配列  x  の添字
73  /* (関数値)   :  なし
74  /* (機能) :  quicksort アルゴリズムを使って、配列要素
75  /*              x [from], x [from+1], ..., x [to]
76  /*            を値の小さい順に並べ替える。
77  /*-----------------------------------------
78  void quicksort( int x[], int from, int to)
79  {
80    int  pivot_sub;               /* pivot subscript の意

81    if (from < to) {
82      pivot_sub = partition( x , from, to); /*分割操作*/
```

```
83        quicksort( x , from, pivot_sub - 1);
84        quicksort( x , pivot_sub + 1, to);
85      }
86    }


87    /*-----------------------------------------------------
88    /* 引数で与えられた配列の部分列にquicksortの分割操作 */
89    /*                                        (quicksortの関数) */
90    /*------------------------------------------
91    /* (仮引数)  x      : int型配列
92    /*           from : int型配列  x  の添字
93    /*           to   : int型配列  x  の添字
94    /* (関数値)    :  分割操作によって得られた枢軸要素の添字番..
95    /*              (以下の「(機能)」の項で出て来る pivot_sub
96    /* (機能) :   x [from]~ x [to]を並べ替えて
97    /*    max{ x [from],..., x [pivot_sub-1]} <=  x [pivot_
98    /*     x [pivot_sub] < min{ x [pivot_sub+1],..., x [to]
```

```
 99  /*                となるようにする。
100  /*-----------------------------------------------
101  int  partition( int x[],  int from, int to)
102  {
103    int pivot;

104    pivot =  x [from];    /* 最初の要素を枢軸要素に選ぶ。 */
105    while (TRUE) {        /* 工夫の余地あり。              *
106      for ( ; from<to &&  x [to]>pivot; --to)
107        ;
108      if (from == to) {
109         x [from] = pivot;
110        return from;
111      }
112       x [from++] =  x [to];

113      for ( ; from<to &&  x [from]<=pivot; ++from)
```

```
114           ;
115         if (from == to) {
116            x [to] = pivot;
117           return to;
118         }
119          x [to--] =  x [from];
120       }
121     }
```

一次元配列 a を関数の引数として受渡しする方法：

- 仮引数側では、次のいずれかの書き方をする。

  データ型 配列名 []

  データ型 *配列名

  データ型 配列名 [大きさ]

> 補足：
> 配列の大きさを明示する必要はない。
> 明示したとしても捨てられる。

- 関数本体の中では、仮引数の配列要素の参照はこれまでと全く同じ様な書き方をする。

- 実引数側では、次のいずれかの書き方をする。

  a

  &a[0]

> 補足：
> 配列名は、計算機内部では
> 先頭要素を指す定数ポインタ
> として扱われている。

例題7．4（一次元配列の一部を関数パラメータとして受け渡す）

double型一次元配列の部分要素列についての情報を引数として受け取り、その部分配列内の要素の総和を計算して返す関数 sum() を定義せよ。そして、

v[k] ＝ $\mathbf{2^{49\text{-}k}}$ **(k=0~ 49)**

という風に値の設定された大きさ**50**の**double**型一次元配列について、この関数を用いて、

v[0]+v[1]+v[2]+···+v[49],

v[40]+v[41]+v[42]+···+v[49],

v[20]+v[21]+v[22]+···+v[39]

の値を計算して出力する**C**プログラムを作成せよ。

(考え方)　素直に考えるなら、部分配列内の要素の総和を計算する関数 sum() には

{ ①配列の名前(i.e. 先頭要素の番地),
②総和の始めとなる配列要素の添字番号,
③総和を締めくくる配列要素の添字番号

の3つを引数として引き渡すことが頭に浮かぶ。もちろん、これは妥当な考えで、関数 sum() も使い易くなる。

しかし、呼ばれる関数側としては、受け渡されるポインタが指す領域以降に然るべき型のデータ領域が十分に長く確保されていれば良いだけである。 従って、逆に、これさえ守れば良い訳で、もし

double型配列の名前 a と大きさ size を引数として受け取り、
a[0]+a[1]+···+a[size-1] を計算して返す関数
double sum(double a[], int size)

が定義できているなら、この関数を sum(&v[from], size) という風に使うことも許されるはずで、この呼び出しによって部分配列の総和 v[from]+v[from+1]+···+v[from+size-1] が計算されることになる。

確認：

配列要素 v[from]～v[from+size-1] が sum() を呼び出す側で確保されていれば、確かに、

◇ &v[from] は配列要素を指すポインタで、

◇ &v[from] 番地以降にも同じ型のデータが十分長く続いている。

呼び出された側の関数が実際にメモリ確保された領域だけを使うようにするのはプログラマの責任である。

(プログラミング)　(部分)配列の要素の総和を計算する関数 sum() は、引数として配列の名前(先頭要素の番地) a と大きさ size を受け取り、a[0]+···+a[size-1] を計算して返すものとする。

また、main() 関数の中では、配列要素 v[0]~v[49] に対する値の設定は数学関数 pow() を使うのではなく

v[49] ⟵ 1,
v[48] ⟵ v[49]×2,
v[47] ⟵ v[48]×2,
....

という風に行うことにして、プログラムを構成した。

[motoki@x205a]$ nl func-bind-part-of-array-Kelley.c Enter

```
     1 #include <stdio.h>

     2 double sum(double a[], int size);
```

```
 3 int main(void)
 4 {
 5   int    i;
 6   double v[50];

 7   v[49] = 1.0;
 8   for (i=48; i>=0; --i)
 9     v[i] = v[i+1] * 2.0;

10   printf("v[0] +v[1] + ... +v[49] = %16.0f\n",
            sum(v, 50));
11   printf("v[40]+v[41]+ ... +v[49] = %16.0f\n",
            sum(&v[40], 10));
12   printf("v[40]+v[41]+ ... +v[49] = %16.0f\n",
            sum(v+40, 10));
13   printf("v[20]+v[21]+ ... +v[39] = %16.0f\n",
            sum(v+20, 20));
```

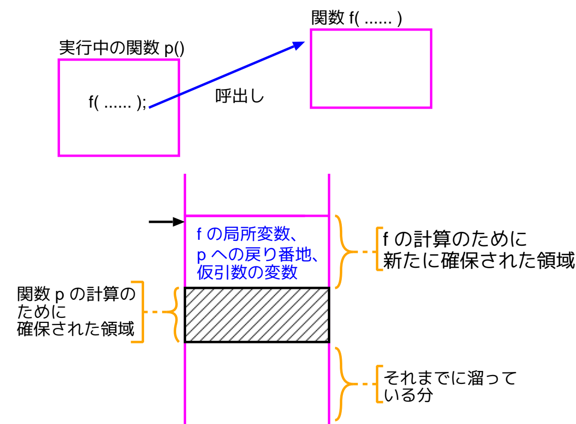
```
14   return 0;
15 }

16 /*-----------------------------------------------
17 /* double型配列(もしくは配列の断片)の要素の総和を計算し...
18 /*-----------------------------------------------
19 /*   (仮引数)     a : double型配列
20 /*             size : double型配列 a の大きさ
21 /*   (関数値) : a[0]+a[1]+a[2]+...+a[size-1]
22 /*-----------------------------------------------
23 double sum(double a[], int size)
24 {
25   int    i;
26   double sum=0.0;

27   for (i=0; i < size; ++i)
28     sum += a[i];
```

```
    29   return sum;
    30 }
[motoki@x205a]$ gcc func-bind-part-of-array-Kelley.c  Enter
[motoki@x205a]$ ./a.out  Enter
v[0] +v[1] + ... +v[49] = 1125899906842623
v[40]+v[41]+ ... +v[49] =             1023
v[40]+v[41]+ ... +v[49] =             1023
v[20]+v[21]+ ... +v[39] =       1073740800
[motoki@x205a]$
```

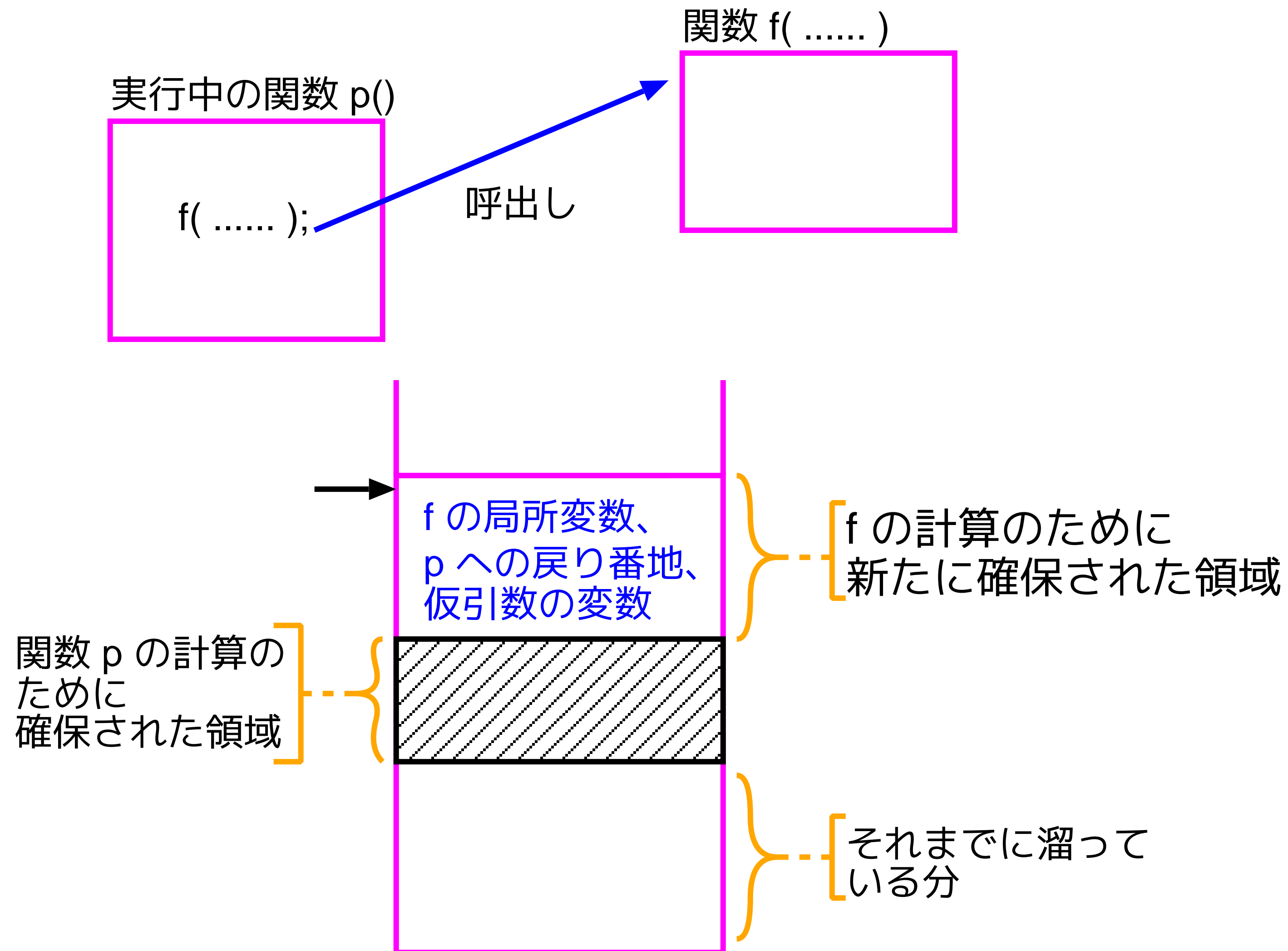
7–5 関数呼出しの実装

計算機内部でどの様に関数呼出しが実現されているか：

- それぞれの関数の計算に必要な作業用領域を確保するために、push-down スタックを一つ用意する。
- 関数が呼び出されるたびに 、
  ① 呼び出し元の戻り番地を入れておく領域、仮引数の変数、新しい自動変数のための領域を必要なだけスタック上に確保し、そこに、呼び出し元の戻り番地、実引数の値等を初期設定する。
  ② 呼び出し先の関数に制御を渡す。



−−−



- 関数実行が終了するたびに 、
  ① 関数値と呼び出し元への戻り番地をどこかに保持した上で、呼び出し先の関数の計算のために確保した領域を解放する。
  ② 呼び出し元の関数の計算を再開する。

例7．5（バッファ・オーバーフロー）次のプログラムとその実行結果を考える。

```
[motoki@x205a]$  cat -n buffer-overflow.c
     1 /*------------------------------------------------------
     2 /* 関数呼出しの際に引数の値や局所変数等が                    */
     3 /*      共通のスタックに積まれることを理解するための例     */
     4 /*------------------------------------------------------
     5 /* これらのことが悪用されてコンピュータが不正侵入される */
     6 /* こともある。(バッファオーバーフロー攻撃)             */
     7 /*------------------------------------------------------
     8 #include <stdio.h>
     9 #include <stdlib.h>
    10
    11 void test(int a, int b, int c);
    12
    13 int main(void)
    14 {
```

```
15     test(1, 2, 3);
16     exit(EXIT_SUCCESS);
17
18     printf("\nここには来ないはずだが、、、。\n");
19     return 0;
20 }
21
22 void test(int a, int b, int c)
23 {
24     int buf[256];
25
26     printf("(関数test内) a= %d\n", a);
27     buf[258] = 999;        <-- 確保メモリ外への書き込み
28     printf("(関数test内) a= %d\n", a);
29
30     buf[257] += 16;        <-- 確保メモリ外への書き込み
31 }
```

```
[motoki@x205a]$  gcc buffer-overflow.c
[motoki@x205a]$  ./a.out
(関数test内) a= 1
(関数test内) a= 999
```

ここには来ないはずだが、、、。

```
[motoki@x205a]$
```

27行目と30行目を見当外れの代入文と見て無視すると、
プログラムは次の様な実行結果を出してくれるはず である。

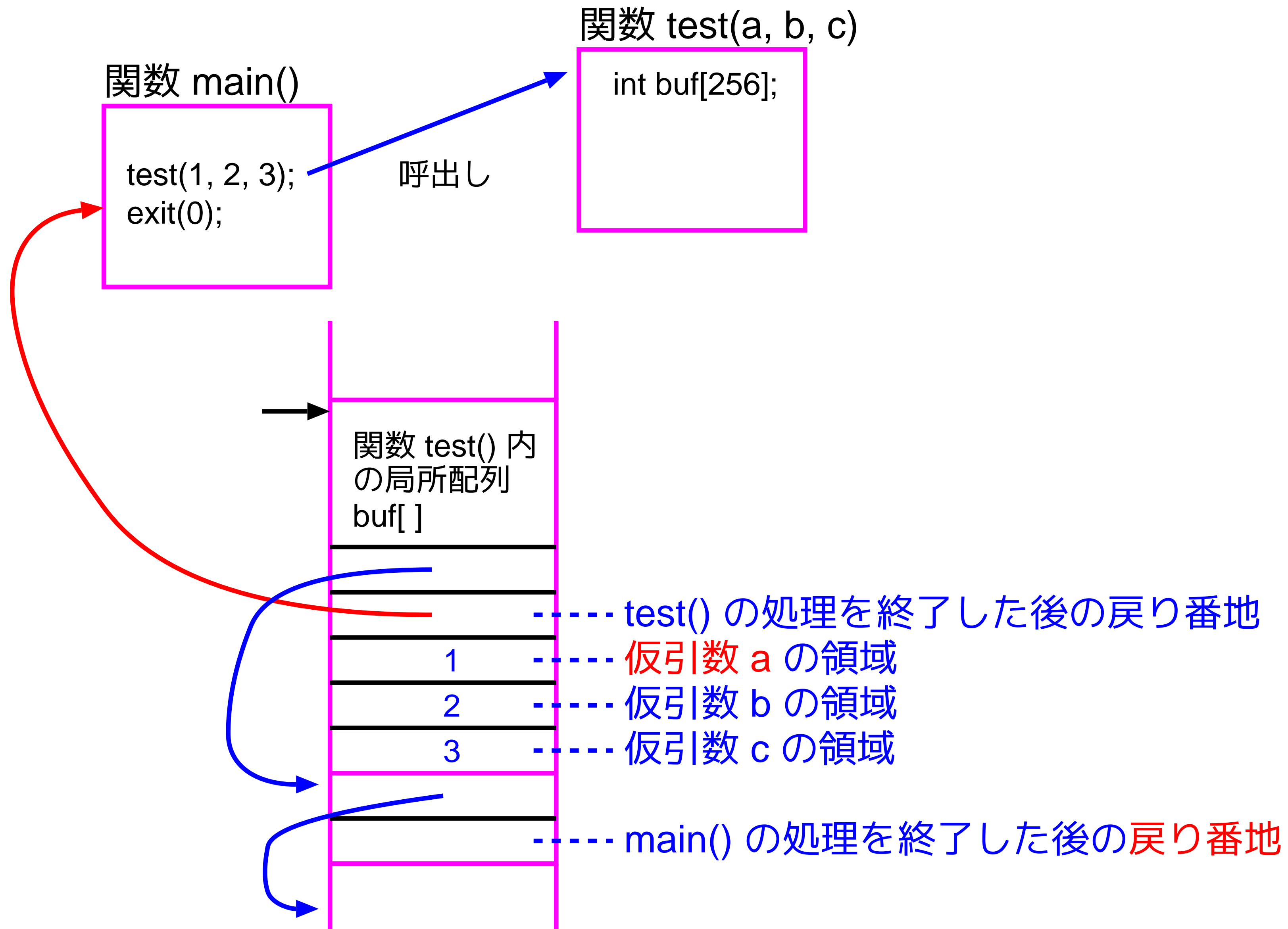
```
(関数test内) a= 1
(関数test内) a= 1
```

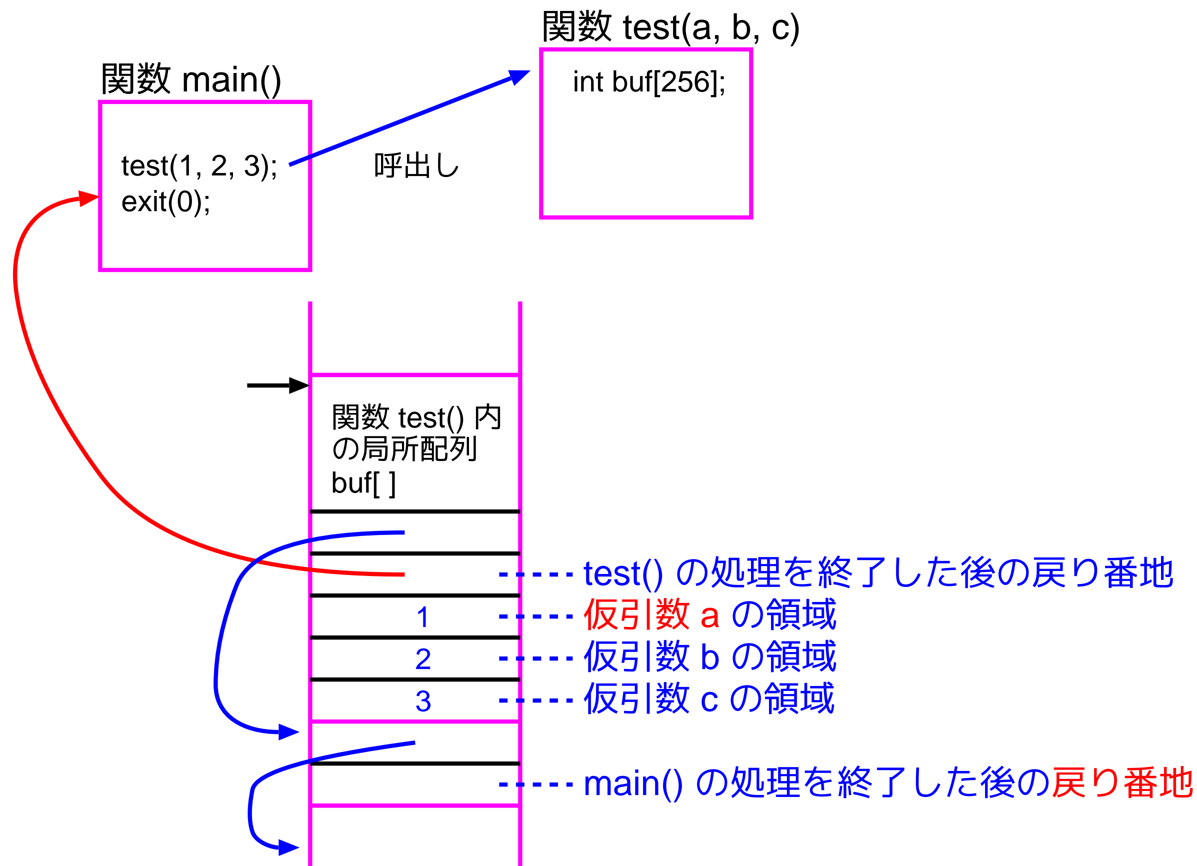
しかし 、……

では何故こういう実行結果になったのかというと 、

⇨ 関数呼び出しの実装方法に由来

実は、上のプログラムで関数test() が呼び出された直後には、
関数呼び出しを実装するためのpush-downスタックの状態は ...





この図を見ると、

buf[258] は 関数 test() の第1仮引数 a の領域と重なり、

buf[257] は関数 main() への戻り番地を保持する領域と重なる

ことが分かる。

⇨ …… それゆえ、

バッファ・オーバーフロー攻撃：

もし、外部へのサービスを行っているサーバプログラムにユーザ名入力の場面があり、そこからサーバプログラムの局所変数領域内に実行コードが組み込まれ、「関数処理終了後の戻り番地」を保持する領域にこの実行コードの番地がセットされてしまうと、関数処理終了と同時に組み込まれたコードが(**root 権限で**)実行されてしまう。

⇨ 関数呼び出しのための上記の機構はインターネット等を通じたコンピュータ/サーバへの攻撃の足掛かりにされることもある。一旦不正侵入されると、**root** 権限で任意のプログラムが実行され得るので甚大な被害を被る危険性も高い。

Buffer Overflow のセキュリティホールに繋がる関数としてはC言語では次の様なものを挙げることが出来る。

`gets()`, `strcpy()`, `strcat()`, `sprintf()`, `vsprintf()`, `scanf()`, `vscanf()`, ...

7–6 再帰計算vs.反復計算

- 再帰計算と同等のことは、無理なく反復計算で記述できることが多い。
- 再帰計算の方が、変数も少なくて済み、分かり易いプログラムになることが多い。
- 再帰計算を行うと、関数呼出しが多くなるので、その分計算時間も記憶領域も多く必要になる。
- 再帰計算によって、異常に非効率な計算が起こることもある。

例7．6（Fibonacci数列の再帰計算）次のプログラムの場合、再帰計算によって異常に非効率な計算になる。

```
[motoki@x205a]$  nl func-bad-recursion-fibo-Kelley.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 }

    10 int fibonacci(int n)
    11 {
    12   printf("called fibonacci(%d)\n", n);
    13          /* どういう計算が行われるかを観察するために */
    14   if (n <= 1)
    15     return n;
    16   else
    17     return fibonacci(n-1)+fibonacci(n-2);
```

```
    18 }
[motoki@x205a]$  gcc func-bad-recursion-fibo-Kelley.c
[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(1)
called fibonacci(0)
called fibonacci(3)
called fibonacci(2)
called fibonacci(1)
called fibonacci(0)
called fibonacci(1)
called fibonacci(4)
called fibonacci(3)
called fibonacci(2)
called fibonacci(1)
called fibonacci(0)
```

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