関数の再帰定義

自然数 n の階乗 $n!$ を計算する関数を定義してみる。引数は整数、返却値も整数

$n! = 1*2*3* \ldots * (n-1)*n$ である。ただし $0! = 1$ とする

```
int factorial(int n)
{
  int i, tmp=1;

  if( n>0 )
    for(i=1; i<=n; i++)
      tmp *= i;
  else
    if( n==0 )
      tmp = 1;
    else
      tmp = -1;

  return tmp;
}
```

$n > 0$ の時、for 文を使って $1*2* \ldots *(n-1)*n$ を計算

0! は 1 である

負の値に対してはエラーの意味で –1 を返す

再帰定義

n! = n*(n–1)! でもある。n! は、n に (n–1)! をかけたものに等しい
これを再帰定義という。C では関数の再帰定義が可能

```
int factorial(int n)
{
  int tmp;

  if( n > 0 )
    tmp = n*factorial(n-1);
  else
    if( n==0 )
      tmp = 1;
    else
      tmp = -1;

  return tmp;
}
```

← 再帰呼び出し。自分自身の定義に自分自身ををを呼び出している

factorial(n) の計算に factorial(n-1) を用いる。
実引数を 1 だけ減らして呼び出しているので、
最後の処理を適切に行わないと無限ループ
（実際は途中でメモリが不足してエラー）

正しい再帰関数定義はプログラマの責任。

```
int factorial(int);

main()
{
  int n=3, fact;
  fact = factorial(n);
  printf("%d ! is %d\n", n, fact);
}

int factorial(int n)
{
  int tmp;
  if( n > 0 )
    tmp = n*factorial(n-1);
  else
    if( n==0 )
      tmp = 1;
    else
      tmp = -1;

  return tmp;
}
```

実引数 $n = 3$ で関数 factorial を呼び出す

仮引数 n は 3 で初期化

$3 > 0$ なので、3*factorial(2)
$2 > 0$ なので、factorial(2) = 2*factorial(1)
$1 > 0$ なので、factorial(1) = 1*factorial(0)
factorial(0) は 1 である

結局、1*2*3 = 6 が返却される

再帰定義

取り組む問題によっては、関数の再帰定義が有用

いろいろなアルゴリズムが再帰的に定義可能

　階乗、組み合わせ数、ユークリッドの互除法、など

ただし、再帰呼び出しを正しく理解してコードすることが求められる

必ずしも計算効率が良くなるわけではない

「再帰」的思考に慣れてください！

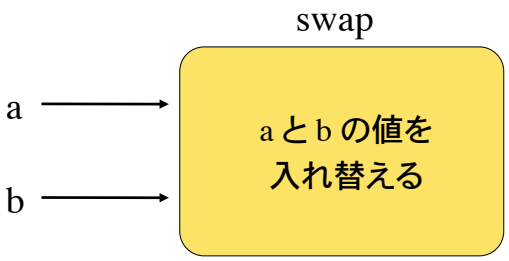
関数への値の受け渡し再考

関数へ値を受け渡すには、引数を用いる

関数を呼び出す側で指定する引数を、実引数（値が確定）、
関数定義部側の引数を、仮引数（値は未定）という

C 言語では、関数への値の受け渡しに際しては、仮引数は実引数で初期化される
これを値渡しという

受け取った 2 つの整数値を関数内部で
入れ替える関数 swap




```
void swap(int a, int b)
{
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
}
```


値渡しの例

```
void swap(int, int);

main()
{
  int x=1, y=2;
  printf("%d %d\n", x, y);
  swap(x, y);
  printf("%d %d\n", x, y);
}

void swap(int a, int b)
{
  int tmp;
  tmp = a;
  a = b;
  b = tmp;
}
```

```
% ./a.out
1 2     ←― 入力
1 2     ←― 出力
%
```

実行結果は左のとおり。
変数 x と y の値は入れ替わっていない！

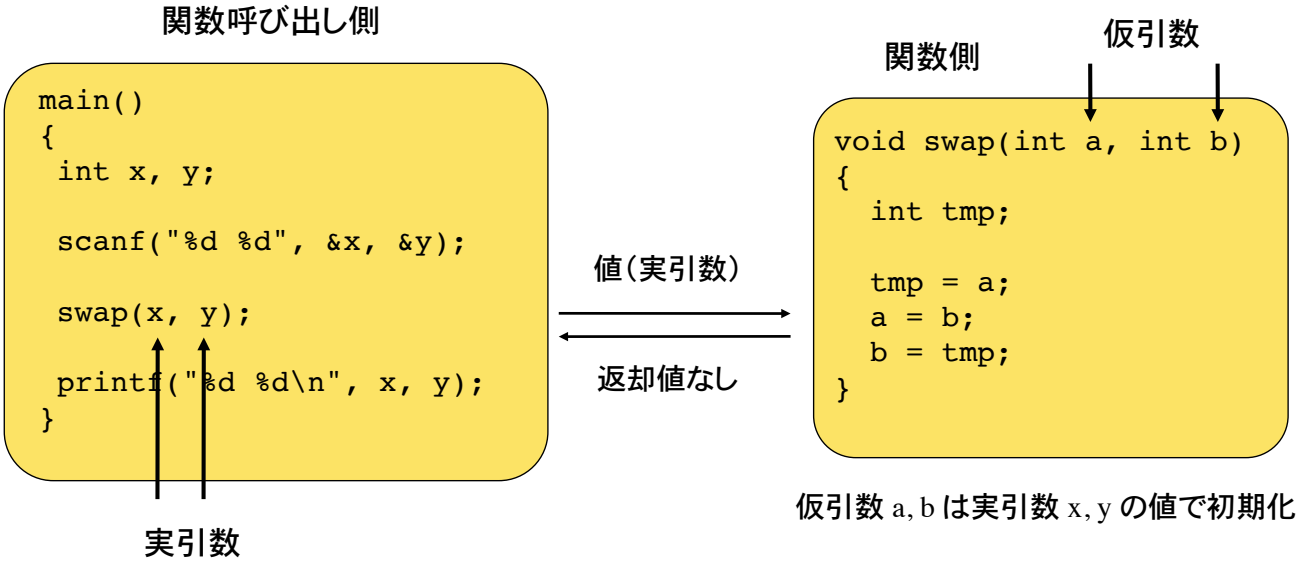
関数 swap には、x の値 1 と y の値 2 が受け渡され、局所変数 a は 1、b は 2 で初期化される

関数 swap 内部では変数 a と b の値は
入れ替わっているが main 文の変数 x, y は不変

関数 swap は実引数の値のコピーを
受け取り、コピーを入れ替えるだけ

値渡しでは、実引数の値は変化しない

値渡しのイメージ

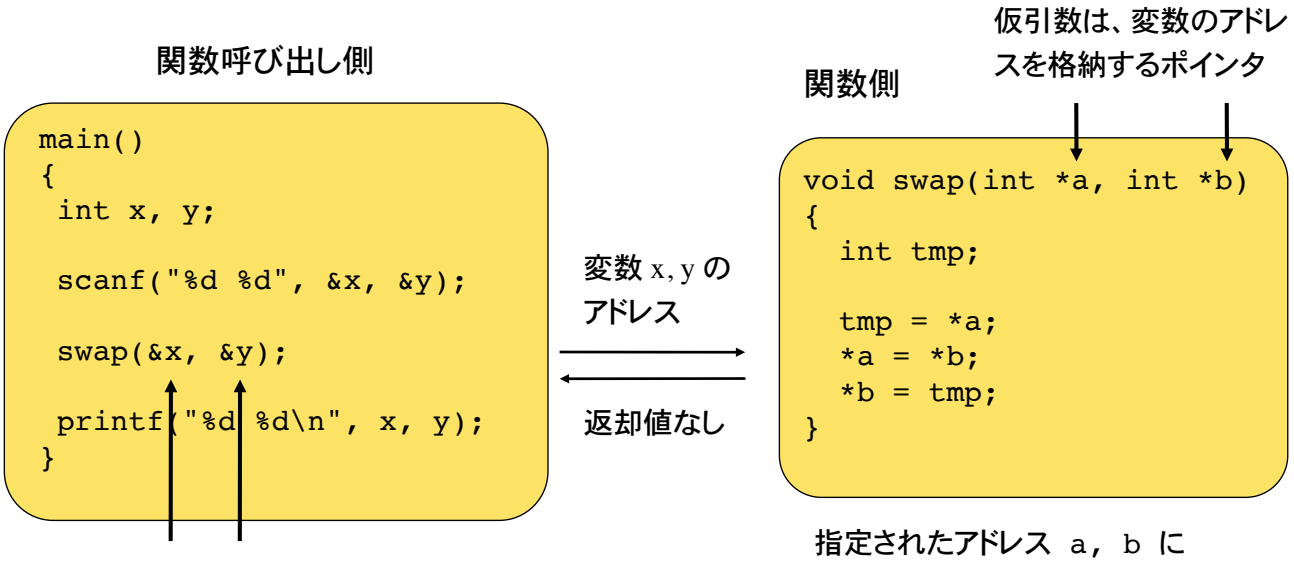


関数 swap は、実引数 x, y の値(x, y 自身ではない!)を仮引数 a, b を通じて受け取る。受け取った値を関数内で操作しても、関数呼び出し側の実引数は影響を受けない
値渡しでは、データの流れは一方通行

値渡しでは、関数呼出側の実引数を関数の側で操作することができない
実引数を操作する為には、関数側でポインタを用いる必要がある

実引数の操作

関数による実引数(関数呼び出し側で指定する値)の操作をするには、関数に操作したい変数の格納場所(アドレス)を伝えれば良い
変数のアドレスを指定する変数をポインタ pointerと言う



関数呼び出し後、x, y の値は入れ替わっている

ポインタ

これまでは変数がメモリ上のどこに配置されるかは処理系に任せてきた

```
int a = 1;
double x = 1.234;
char c = 'A';
```

左で宣言したいずれの変数もメモリ上のどこかに
配置される（処理系が自動的に処理）

変数が格納されている場所（アドレス）を指定する変数をポインタ変数という
（単にポインタともいう）

ポインタを用いて、変数の値を取り出したり操作することを参照という

ポインタ変数の宣言は、参照する変数の型に引き続き、変数名の前に * を付ける

```
int *ptr;
```

整数型の変数へのポインタ変数 ptr を宣言
ポインタ変数 ptr は、int 値が格納されるアドレスを格納
宣言しただけでは、どの int 型変数を指すかは未定

ポインタの使い方の例

```
int a, b;
int *ptr;

a = 1;

ptr = &a;

b = *ptr;

printf("%d\n", b);
```

変数宣言により、変数 a, b はメモリ上の
どこかに配置される（値は不確定）

整数型変数へのポインタ変数 ptr を宣言

ポインタ変数 ptr に変数 a のアドレスを代入
& は変数のアドレス（格納場所）を返すアドレス演算子

ポインタ変数 ptr が指すアドレスに格納された
データを参照するには、ポインタ変数の前に * を
つける。参照した値を変数 b に代入

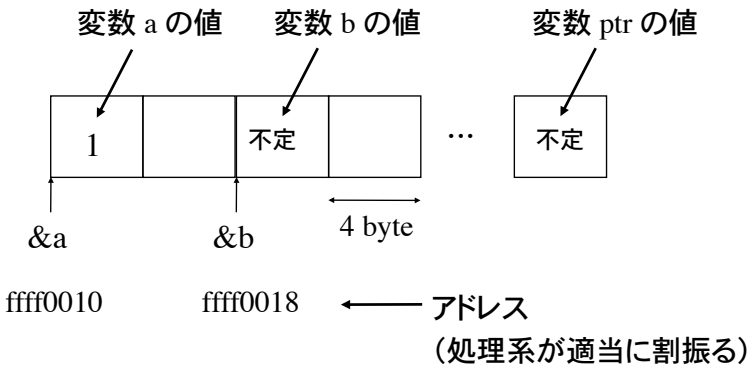
* は間接演算子。ポインタ変数が指す変数の内容を取り出す

この例では、b = a; とすれば済むことを、ポインタ変数 ptr を用いてわざわざ、
ptr = &a; b = *ptr; としている

処理内容

```
int a, b;
int *ptr;

a = 1;

ptr = &a;

b = *ptr;

printf("%d\n", b);
```

a = 1 を実行直後のメモリの状態



ptr = &a の実行でポインタ変数 ptr の値は変数 a のアドレス ffff0010 となる

b = *ptr の実行で変数 b の値は、ポインタ変数 ptr が指すアドレス ffff0010 に格納されている値 int 1 になる（参照）

ポインタの使い方の例 2

```
int a, b;
int *ptr;

ptr = &a;

*ptr = 100;

b = *ptr;

printf("%d %d\n", a, b);
```

変数 a のアドレスをポインタ変数 ptr に代入
ptr には変数 a の格納場所が格納されている

ptr が指す変数の内容に 100 を代入
a=100 とするのと同じ

変数 b に ptr が参照する値 100 を代入

実行結果は、

```
% ./a.out
100 100
%
```

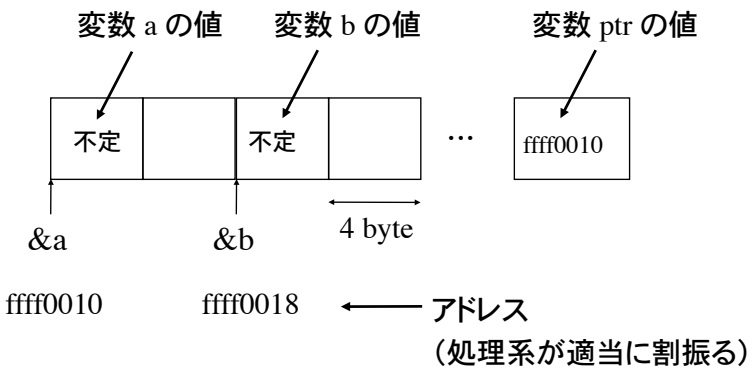
変数 a, b 共に 100 という値が格納される
a=100; b=100; とするのと同じ

ポインタは、変数のアドレスを格納する変数

処理内容 2

```
int a, b;
int *ptr;

ptr = &a;

*ptr = 100;

b = *ptr;

printf("%d\n", a);
printf("%d\n", b);
```

ptr = &a を実行直後のメモリの状態



*ptr = 100 の実行でポインタ変数 ptr が指すアドレス ffff0010 に int 100 が格納される（参照）。（変数 a に 100 を代入するのと同じ）

b = *ptr の実行で変数 b の値は、ポインタ変数 ptr が指すアドレス ffff0010 に格納されている値 int 100 になる（参照）

なぜポインタを使うのか？

1) データの柔軟な取り扱いが容易になる

2) 値渡しでは不可能な実引数の操作が可能になる

```
int a=1, b=2;

printf(“a = %d, b = %d\n”, a, b);

swap(&a, &b);

printf(“a = %d, b = %d\n”, a, b);
```

関数 swap を呼び出すと、実引数 a, b の値が入れ替わっているようにしたい！

ポインタを用いた参照渡し call by reference をすると実引数の操作が可能になる

参照渡し

```
void swap(int*, int*);

main()
{
  int x=1, y=2;
  printf("%d %d\n", x, y);

  swap(&x, &y);

  printf("%d %d\n", x, y);
}

void swap(int *a, int *b)
{
  int tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
}
```

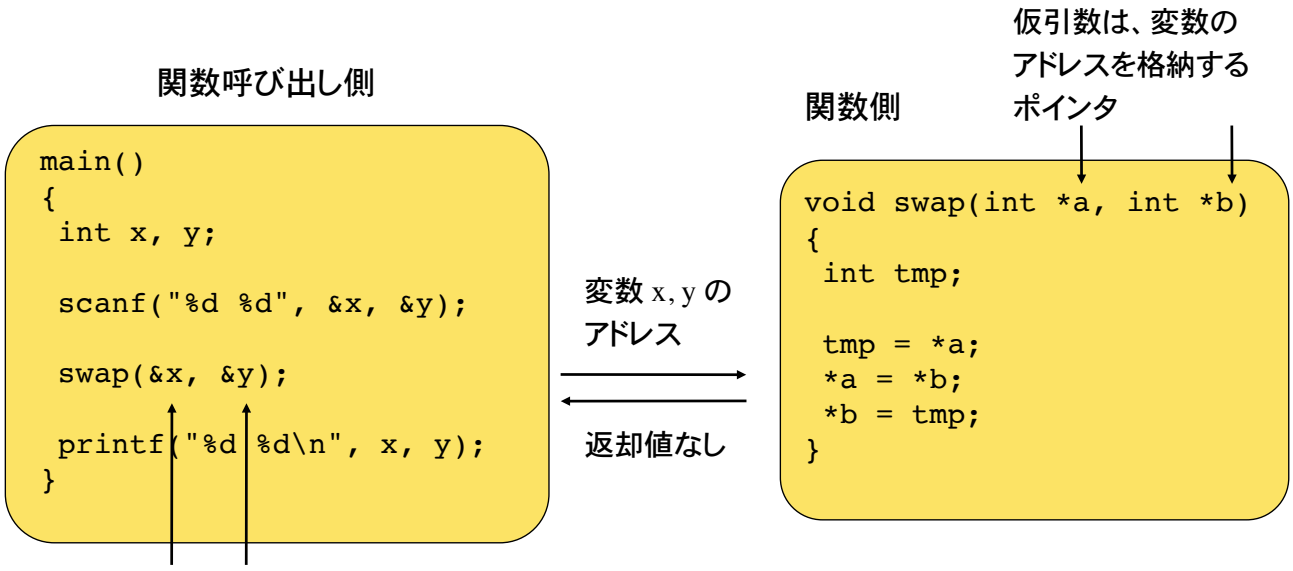
関数 swap のプロトタイプ宣言
引数は整数型へのポインタ 2 つ

関数 swap に変数 x, y のアドレスを実引数として受け渡す。これを参照渡しという。関数呼び出し後は、x, y の値は入れ替わっている

仮引数 a, b を整数値へのポインタとして宣言

アドレス a に格納されている値を tmp に代入
アドレス a に、アドレス b に格納されている値を代入
アドレス b に tmp の値を代入

参照渡しのイメージ



操作したい変数のアドレスを実引数として関数呼び出し

関数 swap は、変数の格納場所（アドレス）を引数として受け取り、そのアドレスに格納されている値を操作する

関数呼び出し後、x, y の値は入れ替わっている

値渡しと参照渡し

値渡し　　値のコピーを受け渡す
あとは関数に任せた

関数呼び出し側 ⟶ 関数側

実引数の値をコピーして、関数に引き渡す。
関数側で実引数の値を操作することはできない。
関数呼び出しの結果を引数を介して得ることはできない。

参照渡し　　値の格納場所（アドレス）を受け渡す
あとは関数に任せた

関数呼び出し側 ⟶ 関数側

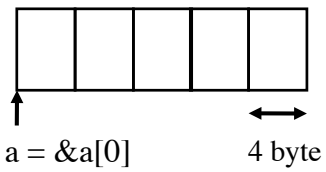
実引数のアドレスをポインタとして、関数に引き渡す。
関数側で実引数の値を操作することが可能。
関数呼び出しの結果を引数を介して得ることが可能。

関数への配列の受け渡し

配列名は、その配列が格納されている場所（アドレス）に等しい。

```
int a[5];
```

配列 a



配列を関数に受け渡すには、配列名および要素数、の2つの情報が必要

1 次元配列を受け取り、各要素の和を求める関数の例

```
double fn(double x[], int size)
{
  int i;
  double sum=0;
  for(i=0; i<size; i++)
    sum += x[i];
  return sum;
}
```

関数頭部の仮引数宣言は、配列である事
を示すために [] を付ける

配列添え字の範囲はプログラマの責任
この例では、引数 size のチェックをしていない

関数への配列の受け渡し

```
double fn(double[], int);

main()
{
  double vector[]={0.0, 1.1, 3.2}, sum;
  int size = sizeof(vector)/sizeof(double);

  sum = fn(vector, size);
  printf("%f\n", sum);
}

double fn(double x[], int size)
{
  int i;
  double sum=0;
  for(i=0; i<size; i++)
    sum += x[i];
  return sum;
}
```

プロトタイプ宣言

配列要素の合計を返却する関数を定義

参照渡しの例

2 つの 1 次元配列（ベクトル）の和を計算し、その結果を実引き数を介して関数呼び出し側に返す関数を考える

配列名は、その配列へのポインタに等しい
参照渡しにより、実引数（配列の内容）を操作することが可能

```
double x[] = {1,2,3,4,5}, y[] = {5,4,3,2,1};
int i, size = sizeof(x)/sizeof(double);

for(i=0; i<size; i++)
  printf("%f ", x[i]);
printf("\n");

vector_add(x, y, size);

for(i=0; i<size; i++)
  printf("%f ", x[i]);
printf("\n");
```

2 つのベクトルの和を計算する関数 vector_add を呼び出す

関数の返却値として 1 次元配列を返すことはできない
1 次元配列の先頭要素へのポインタを返却する関数は定義可能（本講義ではやらない）

ここでは、参照渡しにより実引数を操作することを考える

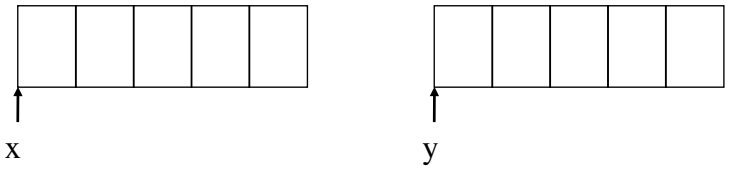
```
void vector_add(double x[], double y[], int size)
{
  int i;

  for(i=0; i<size; i++)
    x[i] += y[i];
}
```

関数 vector_add は、2 つの配列の先頭要素のアドレスと、配列サイズを受け取り、配列要素の内容を操作する

関数呼び出し側

関数 vector_add に、配列 x と y のアドレスを引き渡す。

配列名は、配列の第 0 要素へのポインタである。



関数側

アドレス x から格納されている配列要素それぞれにアドレス y から始まる配列要素を足す。

関数呼び出し後

配列 x には、配列 x と y の和が格納されている。

問題 1

フィボナッチ数列 x_n は次で定義される。

$$x_n = x_{n-1} + x_{n-2} \quad (n = 2, 3, 4, \cdots)$$
$$x_1 = 1, x_2 = 1 \qquad \{x_n\} = \{1, 1, 2, 3, 5, 8, 13, 21, \cdots\}$$

1000 以下のフィボナッチ数を全て表示するプログラムを作れ。n 番目のフィボナッチ数を返す関数 int fibonacci(int n) を再帰定義して用いよ。

```
int fibonacci(int);

main()
{
  int i=1, f;

  while( (f=fibonacci(i)) < 1000 ){
    printf("%d\n", f);
    i++;
  }
}
```

余力があれば、fibonacciを再帰定義しないプログラムを作成して実行時間を比較せよ。

問題 2

n 個の中から r 個を取り出す組み合わせの数 ${}_nC_r$ を計算する関数を作れ。

$${}_nC_r = \binom{n}{r} = \frac{n!}{r!(n-r)!} = {}_{n-1}C_{r-1} + {}_{n-1}C_r \qquad (n \geq r)$$
$${}_nC_r = 1 \quad (r = 0 \text{ or } n = r) \qquad {}_nC_r = 0 \quad (n < r)$$

組合わせ数 ${}_nC_r$ を階乗を用いて定義すると、階乗を計算する際、桁あふれが起こりうる
int combinatorial(int, int) を再帰定義してプログラムを作れ

```
int combinatorial(int, int);

main()
{
  int n, r, c;

  scanf("%d %d", &n, &r);
  c = combinatorial(n, r);
  printf("%d\n", c);
}
```

main 文は完成している。
関数 combinatorial を定義せよ。

次の関係を用いた再帰定義を試みよ

$${}_nC_r = {}_nC_{r-1}\frac{n-r+1}{r} \quad (n, r \geq 1)$$
$${}_nC_r = 1 \quad (n = 0 \text{ or } r = 0)$$

問題 3

10 名分の成績（100 点満点の整数値）を配列に収め、この配列を受け取って
平均点を計算するユーザ関数 heikin を定義せよ。

main 文の骨格はすでに完成している。

```
double heikin(int[10], int);

main()
{
  int score[10];
  double average;

  /* 成績の入力部分 */

  average = heikin(score, 10);

  printf("平均点は %f \n", average);
}
```


問題 4

2 × 2 行列の行列式を計算する関数 det を作れ。

```
double det(double[2][2]);

main()
{
  double a[2][2]={{1.0,2.0},{3.0,4.0}}, x;

  x = det(a);

  printf("行列式は %f\n", x);
}

double det(double x[2][2])
{
  double tmp;
  .....
}
```

プロトタイプ宣言

引数は配列名のみ

配列名 x は仮引数

問題 5

2 つの行列の積を計算する関数を作れ。main 文は既に完成している。

```
main()
{
  double A[4][4], B[4][4];

  display_matrix(A, 4);
  display_matrix(B, 4);
  multiply_matrix(A, B, 4);
  display_matrix(A, 4);
}
```

行列は 4 × 4 行列とする

← 関数呼びだし後、実引数 A は行列の積 AB で上書きされる。

関数 display_matrix は 2 次元配列と行数を受け取りその内容を表示する関数

```
void display_matrix(double x[4][4], int dim);
```

関数 multiply_matrix は 2 次元配列を 2 つ受け取り、積を第一引数として返す関数

```
void multiply_matrix(double x[4][4], double y[4][4], int dim);
```


問題 6

次のプログラムを実行せよ。　　変換指定 %x は整数値の 16 進数表記。アドレス表記に用いる。

```
main()
{
  int a,b;
  int *ptr,


  a=1;
  b=7;


  ptr = &a;
  printf("a = %d, address of a = %x\n", a, ptr);
  printf("a = %d, address of a = %x\n", *ptr, ptr);

  ptr = &b;
  printf("b = %d, address of b = %x\n", b, ptr);
  printf("b = %d, address of b = %x\n", *prt, ptr);
}
```

変数 a と b が格納されているアドレスはどうなっているか?

それぞれの変数を指すポインタを参照せよ。