C/C++プログラミング

第7回

ポインタの知識と演習

ポインタ: C++

List7-1 ポインタ

```
-------------------------------------
// オブジェクトのアドレスを表示

#include <iostream>

using namespace std;

int main()
{
        int    n;
        double x;

        cout << "nのアドレス：" << &n << '\n';
        cout << "xのアドレス：" << &x << '\n';
}
-------------------------------------
```

ポインタ: C++

「

```
cout << "nのアドレス：" << &n << '\n';
cout << "xのアドレス：" << &x << '\n';
```

」

「&n」，「&x」は，変数n，xのメモリ上の格納場所のアドレスを表す．

以降，C++とCでは配列の扱いには大きな違いはないので，Cは省略．

ポインタ: C++

List7-2 ポインタ

```
----------------------------------------
// ポインタの基本（アドレス演算子&と間接演算子*）

#include <iostream>

using namespace std;

int main()
{
        int n = 135;
        cout << "n    : " <<  n << '\n';
        cout << "&n   : " << &n << "番地\n";

        int* ptr = &n;                          // ptrはnを指す
        cout << "ptr  : " <<  ptr << "番地\n";
        cout << "*ptr : " << *ptr << '\n';
}
----------------------------------------
```

ポインタ: C++

「
cout << "&n　: " << &n << "番地\n";
」
「&n」は変数nのメモリ上の格納場所のアドレスを表す．&は「アドレス演算子」という．

「 int* ptr = &n; 」
int* は整数型の変数のポインタを表す型．この場合 ptr は変数nのメモリ上のアドレス&n
が格納される．「ポインタ」という言葉の由来は「格納場所のアドレスを指し示す=ポイントする」
という意味から来ている．

「 cout << "ptr　: " << ptr << "番地\n"; 」
ptr にはnの番地が格納されているので，この行でその番地が表示される．
「 cout << "*ptr : " << *ptr << '\n'; 」
ポインタ型変数 ptr の前に * をつけて *ptr とすると，「ポインタptrが指し示す格納場所に格納されている値そのもの」を表す．この例の場合は変数 n の値，つまり 135 を表す．
* を「間接演算子」という．これにより「*p」と「n」の表すもの（値）は同一であり，n のエイリアスと呼ばれる．

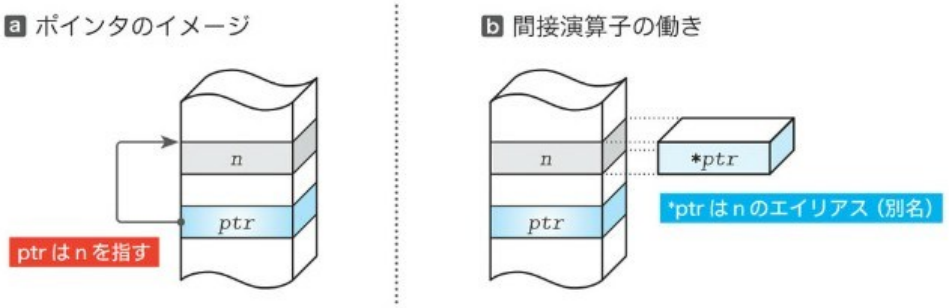

Fig.7-3　オブジェクトとポインタ

以降，C++とCでは配列の扱いには大きな違いはないので，Cは省略．

ポインタ: C++

List7-3 アドレス演算子と間接演算子

```
----------------------------------------
// アドレス演算子と間接参照演算子

#include <iostream>

using namespace std;

int main()
{
        int x = 123, y = 567, sw;

        cout << "x = " << x << '\n';
        cout << "y = " << y << '\n';

        cout << "値を変更する変数[0…x / 1…y] : ";
        cin >> sw;

        int* ptr;
        if (sw == 0)
                ptr = &x;       // ptrはxを指す
        else
                ptr = &y;       // ptrはyを指す

        *ptr = 999;

        cout << "x = " << x << '\n';
        cout << "y = " << y << '\n';
}
----------------------------------------
```

ポインタ: C++

「
```
cin >> sw;
int* ptr;
        if (sw == 0)
                ptr = &x;     // ptrはxを指す
        else
                ptr = &y;     // ptrはyを指す

        *ptr = 999;
```
」
キーボードから入力された sw の値によって場合分けする.
sw = 0 ならポインタ型変数 ptr には x のアドレス &x が格納され,
sw ≠ 0 ならポインタ型変数 ptr には y のアドレス &y が格納される.
その上で, *ptr = 999; でアドレス &x あるいは &y の格納場所に 999 という値が格納される. 同じ実行をするには, if文を次の様にする.

```
        if (sw == 0)
                x = 999;
        else
                y = 999;
```

しかしこの場合, 999という同じ数字を2か所に書く必要があり, これが誤りのリスクを高める.
このようなポインタ型変数の使い方は, アクセスしたい格納場所を状況に応じて「動的」に決定することを可能とする.

ポインタ: C++

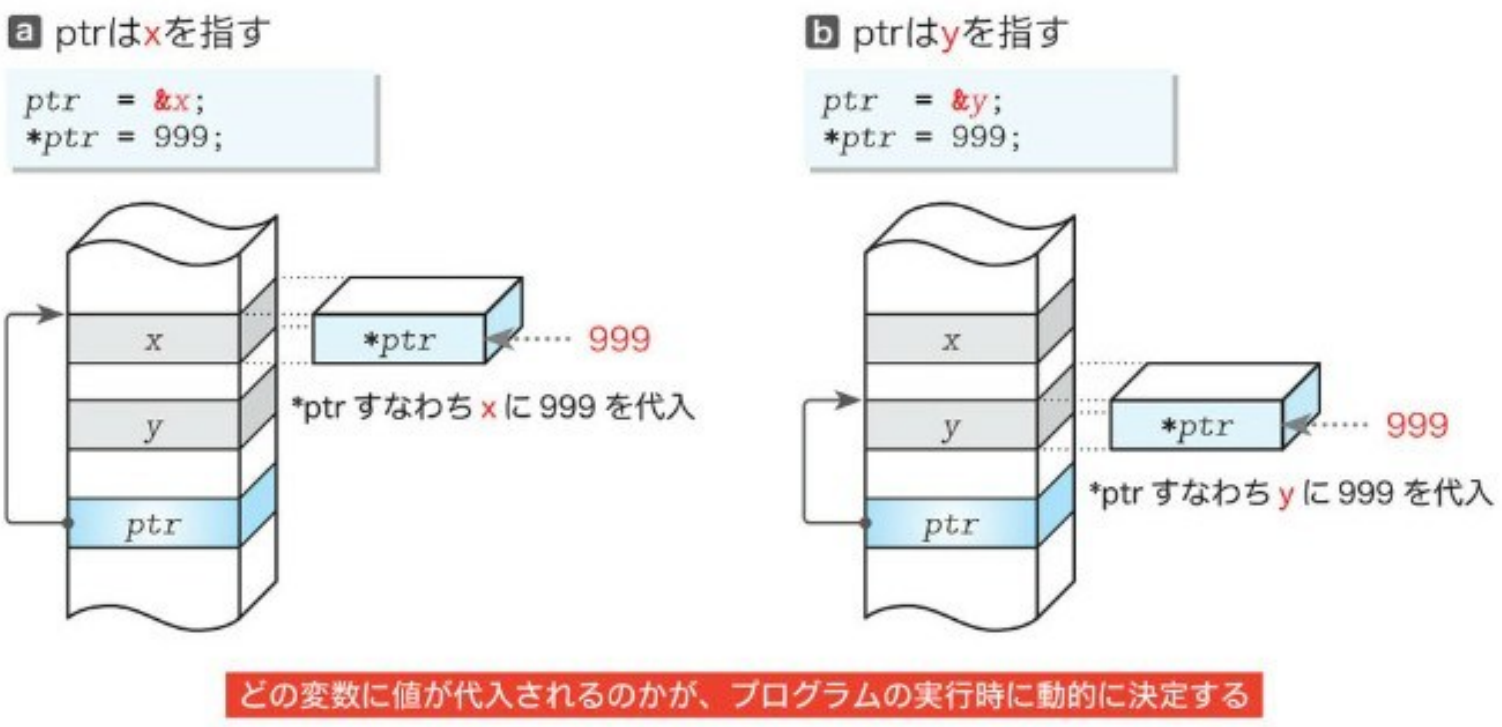

Fig.7-4 ポインタが指すオブジェクトへの値の代入

関数呼び出しとポインタ: C++

List7-5 関数呼び出しとポインタ

```
----------------------------------------
// 二つの整数値の和と積を関数によって求める

#include <iostream>

using namespace std;

//--- xとyの和と積を*sumと*mulに求める ---//
void sum_mul(int x, int y, int* sum, int* mul)
{
        *sum = x + y;
        *mul = x * y;
}

int main()
{
        int a, b;
        int wa = 0, seki = 0;

        cout << "整数a：";   cin >> a;
        cout << "整数b：";   cin >> b;

        sum_mul(a, b, &wa, &seki);                    // aとbの和と積を求める

        cout << "和は" << wa   << "です。\n";
        cout << "積は" << seki << "です。\n";
}
----------------------------------------
```

関数呼び出しとポインタ: C++

「
```
void sum_mul(int x, int y, int* sum, int* mul)
{
        *sum = x + y;
        *mul = x * y;
}
```
」
関数 sum_mul の定義．和と積の値の格納場所 sum と mul をポインタ型で宣言している．これには下で説明する理由がある．

「
```
sum_mul(a, b, &wa, &seki);
  cout << "和は" << wa   << "です。\n";
  cout << "積は" << seki << "です。\n";
```
」
1行目で関数 sum_mul を実行し和と積を計算する．この時点でメモリ上ではmain関数で用いられる変数 a, b の値の格納場所と，sum_mul で用いられる変数 x, y の値の格納場所が確保されている（教科書Fig.7-1参照）．そして関数同士の間（この場合 main と sum_mul の間）の値の受け渡しは「値渡し」であった．つまりここで関数 sum_mul が呼び出される際には，

x ← a
y ← b
sum ← &wa
mul ← &seki

関数呼び出しとポインタ: C++

次に計算結果の wa と seki の値を main 関数の中で利用したいのだが，もし関数 sum_mul の定義を誤って

void sum_mul(int x, int y, int sum, int mul)

と定義して，main 関数の中で

sum_mul(a, b, wa, seki);

などと呼び出しても，

sum ← wa
mul ← seki

といった値のコピーは実行されるが，sum_mul の中での計算結果である sum, mul について，逆方向の

wa ← sum
seki ← mul

といった値渡しは実行されない．また sum_mul の中だけで意味のある変数名 sum, mul は，main の中では意味を持たず，よってその値も参照できない．

関数呼び出しとポインタ: C++

そこでサンプルプログラムでは,
「
```
void sum_mul(int x, int y, int* sum, int* mul){….}
…
sum_mul(a, b, &wa, &seki);
```
」
というように wa と seki についてはそれらの変数のメモリ上の「アドレス」を sum_mul に引き渡す.
この時 sum_mul の中では，sum, mul には wa と seki の格納場所のアドレスがコピーされており,
「
```
*sum = x + y;
*mul = x * y;
```
」
の実行によって,
「
メモリ上の wa と seki の格納場所にそれぞれ x+y, x*y の計算結果を代入する.
」
という動作をする．これにより，main 関数の中で
「
```
cout << "和は" << wa   << "です。\n";
cout << "積は" << seki << "です。\n";
```
」
というように wa, seki のデータを参照すると，意図したとおりに和と積の計算結果が参照される.

以上の動作を教科書では次のように表現している.
"main 関数は関数 sum_mul に対して「wa と seki へのポインタを渡しますので，それが指すオブジェクトに対して処理をして下さい（値を変更して下さい）」と依頼する．"

関数呼び出しとポインタ: C++

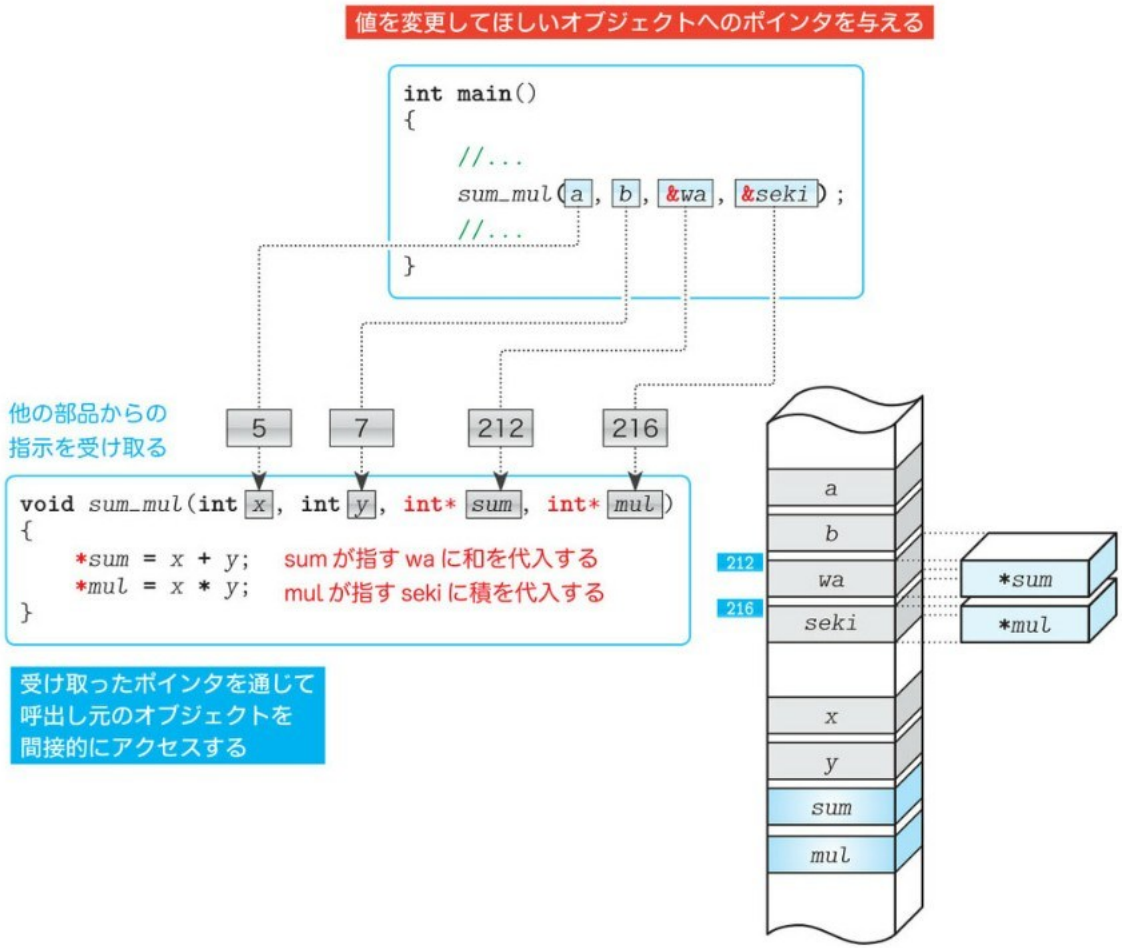

Fig.7-7 関数呼出しにおける引数の授受（値渡し）

ポインタと配列: C++

List7-7 間接演算子と添字演算子

```
----------------------------------------
// 配列の要素の値とアドレスを表示

#include <iostream>

using namespace std;

int main(void)
{
        int a[5] = {1, 2, 3, 4, 5};
        int* p = a;                     // pはa[0]を指す

        for (int i = 0; i < 5; i++)     // 要素の値を表示
                cout << "a[" << i << "] = " << a[i] << "  *(a+" << i << ") = " << *(a + i) << "  "
                     << "p[" << i << "] = " << p[i] << "  *(p+" << i << ") = " << *(p + i) << "\n";

        for (int i = 0; i < 5; i++)     // 要素へのポインタを表示
                cout << "&a[" << i << "] = " << &a[i] << "  a+" << i << " = " << a + i << "  "
                     << "&p[" << i << "] = " << &p[i] << "  p+" << i << " = " << p + i << "\n";
}
----------------------------------------
```

ポインタと配列: C++

「
```
int a[5] = {1, 2, 3, 4, 5};
int* p = a;
```
」
配列 a を定義すると，その名前のみの「a」はその配列の先頭のアドレスを表す．a[0]のアドレス &a[0] も配列 a の一番先頭のアドレスを表すので，a = &a[0] である．

同様にポインタ変数 p に p = a と代入すると，ポインタ変数 p に配列 a の先頭アドレスが代入される．これにより変数 p も配列として用いることができ，p[0], p[1], … は a[0], a[1], … と同じ値を表す．

「
```
cout << "a[" << i << "] = " << a[i] << "  *(a+" << i << ") = " << *(a + i) << "  "
     << "p[" << i << "] = " << p[i] << "  *(p+" << i << ") = " << *(p + i) << "\n";
```
」
a+1 は a が表すアドレスを基準にして，その配列の要素の1つ分先のアドレスを表すことになる．つまり a+1 = &a[1] である．次に間接演算子*を用いた *(a + i) は a+ i が表すアドレスの中身の値を表す．つまり *(a+ i) = a[i] である．ここで[ ] は添字演算子 とも呼ばれる2項演算子と解釈できる．つまり a[i] の場合，a と i の2つに作用し，配列 a の i 番目の中身を返す，という演算子である．p + i, *(p + i) も同様．

ポインタと配列: C++

a[i], *(a + i), p[i], *(p + i)
はそれぞれ同じであり， i 番目の要素の値を表す.

さらに,
a[i], i[a], *(a + i), *(i + a), p[i], i[p], *(p + i), *(i + p)
もそれぞれ同じであり，i 番目の要素の値を表す.

ポインタと配列: C++

List7-9 関数間の配列の受け渡し

```
----------------------------------------
// 配列の要素の並びを反転する（関数版）

#include <iostream>

using namespace std;

//--- 要素数nの配列aの並びを反転する ---//
void reverse(int a[], int n)
{
        for (int i = 0; i < n / 2; i++) {
                int t = a[i];
                a[i] = a[n - i - 1];
                a[n - i - 1] = t;
        }
}

int main()
{
        const int n = 5;                                        // 配列cの要素数
        int c[n];

        for (int i = 0; i < n; i++) {           // 各要素に値を読み込む
                cout << "c[" << i << "] : ";
                cin >> c[i];
        }
        reverse(c, n);                                          // 配列cの要素の並びを反転する

        cout << "要素の並びを反転しました。\n";
        for (int i = 0; i < n; i++)                             // 配列cを表示
                cout << "c[" << i << "] = " << c[i] << '\n';
}

----------------------------------------
```

ポインタと配列: C++

「
void reverse(int a[], int n)
」
関数 reverse の定義．1番目の引数 int a[] は配列へのポインタを表す．ポインタとして値を受ければよいので，上の定義の代わりに，

void reverse(int a[5], int n)

あるいは

void reverse(int* a, int n)

などとしても良い．ただし前者の a[5] の 5 は使われないので意味を持たない．一方，実際に引き渡される配列の要素数は5なので，その要素数を明示的に関数 reverse に引き渡す必要があり，そのために2番目の引数 int n が必要となる．

「
reverse(c, n);
」
この行の上で定義された配列 c に対して，「c」はその格納場所の先頭へのポインタを表すのだった．よってここでの引数 c は配列 c へのポインタを表し，関数 reverse の定義の1番目の引数 a[]（ポインタ）に
a ← c
とコピーされる．また2番目の引数 n (=5) は，reverse の2番目の引数 n に
n(関数reverseの中の変数) ← n (main関数の変数でその値は5)
とコピーされる．

ポインタと配列: C++

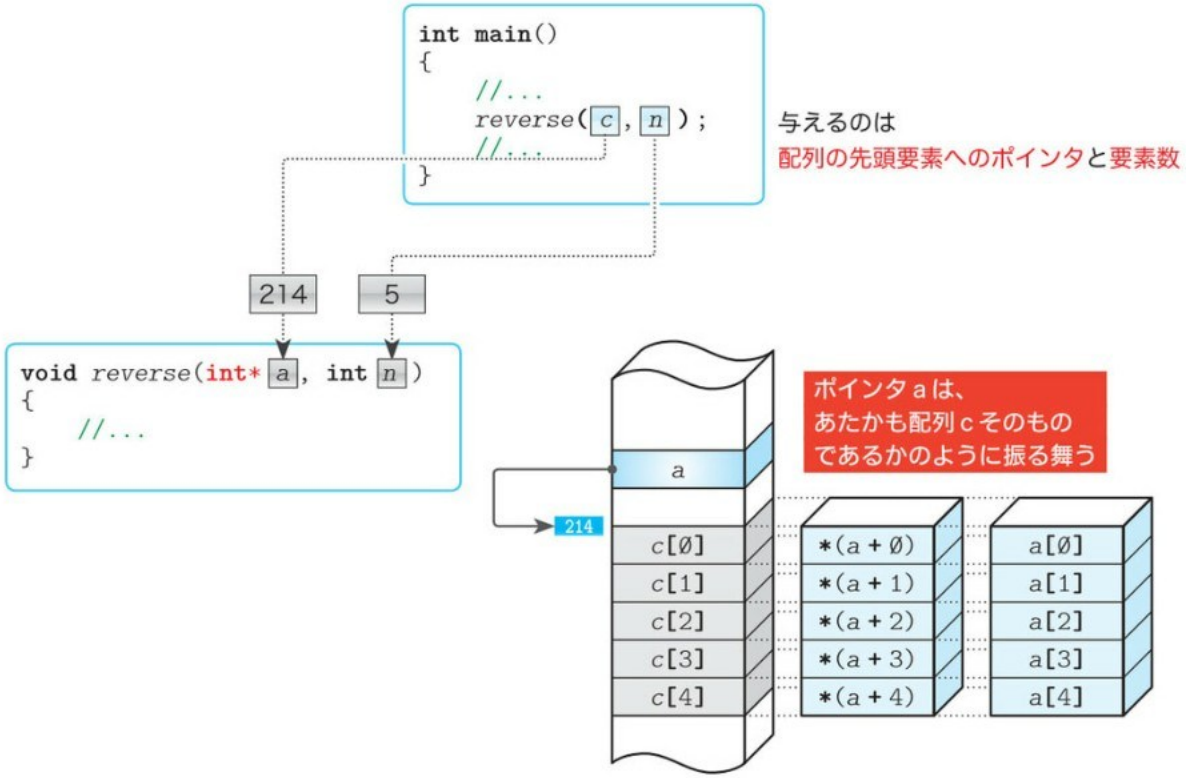

Fig.7-11　関数間の配列の受渡し

ポインタと配列: C++

List7-11 関数間の多次元配列の受け渡し

```
// n行3列の２次元配列の全構成要素に同一値を代入

#include <iomanip>
#include <iostream>

using namespace std;

//---“intを要素型とする要素数3の配列”を要素型とする要素数nの配列 ---//
//---　　（n行3列の２次元配列）の全構成要素にvを代入 ---//
void fill(int (*a)[3], int n, int v)
{
        for (int i = 0; i < n; i++)
                for (int j = 0; j < 3; j++)
                        a[i][j] = v;
}

int main(void)
{
        int no;
        int x[2][3] = {0};
        int y[4][3] = {0};

        cout << "全構成要素に代入する値：";
        cin >> no;

        fill(x, 2, no);                 // xの全構成要素にnoを代入
        fill(y, 4, no);                 // yの全構成要素にnoを代入

        cout << "--- x ---\n";
        for (int i = 0; i < 2; i++) {
                for (int j = 0; j < 3; j++)
                        cout << setw(3) << x[i][j];
                cout << '\n';
        }

        cout << "--- y ---\n";
        for (int i = 0; i < 4; i++) {
                for (int j = 0; j < 3; j++)
                        cout << setw(3) << y[i][j];
                cout << '\n';
        }
}
```

ポインタと配列: C++

「
void fill(int (*a)[3], int n, int v)
」
関数 fill の定義．fill は与えられた配列に対して操作する関数．
与えられる配列は x[2][3], y[4][3] といった2次元のものである．これらの配列は，
「要素型は int[3]，要素数は x[2][3] が2，y[4][3] が4」
である．より詳しく説明すると，
「「要素型が int で要素が3個からなる配列」を「要素型」とし，その要素数が x[2][3] なら2，y[4][3]なら4」
となる．

fill はこのような x[2][3]やy[4][3]を受けるもので，その引数の int (*a)[3] は，x[2][3], y[4][3] の要素（要素型が int[3]）へのポインタとしている．ここで a[3] はx[2][3]，y[4][3]の「要素」であり，その要素型は int型が3つまとまった配列 int[3]．ここではその「要素」へのポインタを引数としたいので，a に * を付けて (*a) としている．

次に引数 int n は，x[2][3] を扱う場合は2，y[4][3] を扱う場合は 4 が代入される．例えば x[2][3] は要素 x[3] が2個(要素数)集まったものであり，その要素数 2 が引き渡される．

最後の引数 int v は，fill によって操作される配列の末端(最小単位)の要素に格納される値を表す．

ポインタによる配列要素の走査: C++

List7-12 ポインタによる配列要素の走査

```
// 配列の全要素に0を代入（第１版）

#include <iostream>

using namespace std;

//--- 配列pの先頭n個の要素に0を代入（第１版）---//
void fill_zero(int* p, int n)
{
        while (n-- > 0) {
                *p = 0;                         // 着目要素に0を代入
                p++;                            // 次の要素に着目
        }
}

int main(void)
{
        int x[5] = {1, 2, 3, 4, 5};
        int x_size = sizeof(x) / sizeof(x[0]);                  // 配列xの要素数

        for (int i = 0; i < x_size; i++)
                cout << "x[" << i << "] = " << x[i] << '\n';     // x[i]の値を表示

        fill_zero(x, x_size);                                   // 配列xの全要素に0を代入

        cout << "全要素に0を代入しました。\n";

        for (int i = 0; i < x_size; i++)
                cout << "x[" << i << "] = " << x[i] << '\n';     // x[i]の値を表示
}
```

ポインタによる配列要素の走査: C++

「
```
void fill_zero(int* p, int n)
{
        while (n-- > 0) {
                *p = 0;                 // 着目要素に0を代入
                p++;                    // 次の要素に着目
        }
}
```
」
「
```
int x[5] = {1, 2, 3, 4, 5};
```
」
「
```
fill_zero(x, x_size);
```
」

関数 fill_zero(int* p, int n) の1つ目の引数はint型のポインタ．main 関数で配列 x[5] が定義されるが，「x」は配列x[5]の先頭のアドレスを表す．よって fill_zero(x, x_size) で配列 x[5] の先頭へのアドレス（=x[0]のアドレス）が p にコピーされる．
次に関数 fill_zero での処理内容で，「*p = 0;」の「*p」は p が指し示すアドレスの格納場所を表し，そこへ数値「0」を代入する(ループの1巡目ではx[0]=0となる)．
「p++;」ではポインタ p に格納されているアドレスの番地を1つインクリメントし，配列の次の格納場所のアドレス(ループの1巡目ではx[1]のアドレス)が格納される．

ポインタによる配列要素の走査: C++

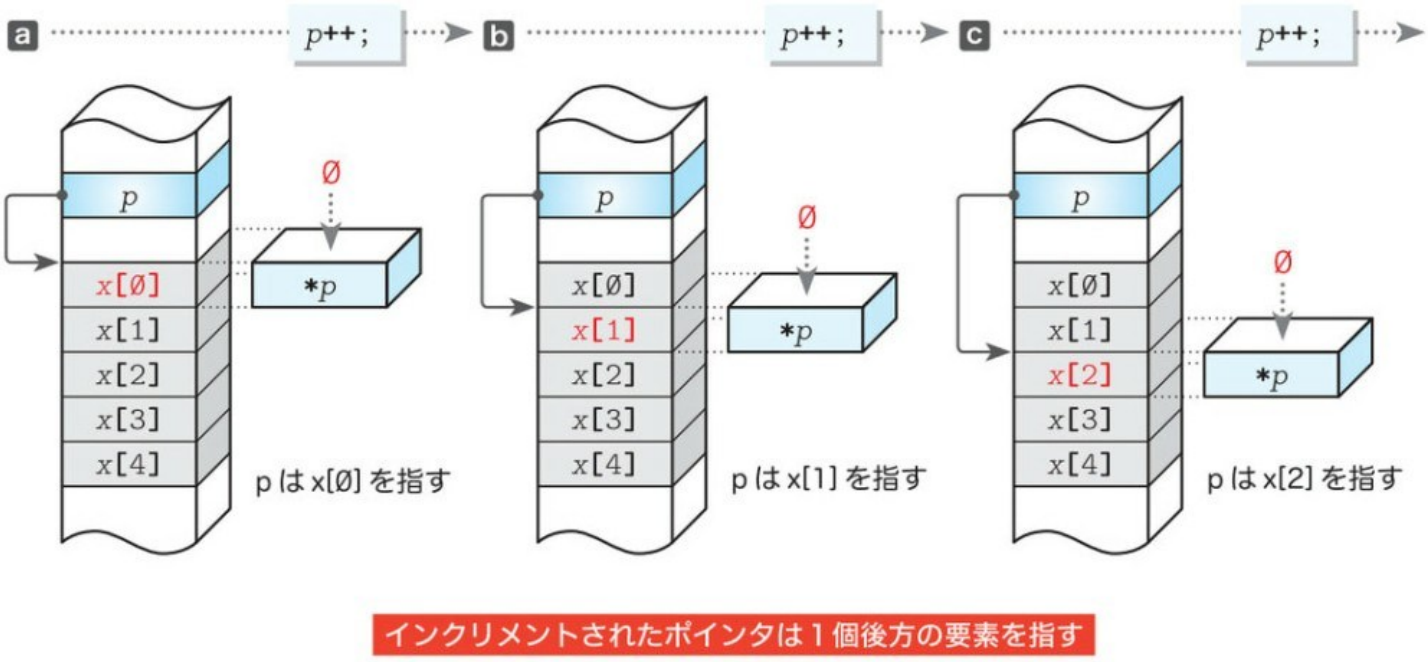

Fig.7-14 ポインタをインクリメントしながらの配列の走査

ポインタによる配列要素の走査: C++

List7-14 線形探索 1

```
// 線形探索（第 1 版）

#include <iostream>

using namespace std;

//--- 配列aの先頭n個の要素から値keyを線形探索（第 1 版）---//
int seq_search(int* a, int n, int key)
{
        for (int i = 0; i < n; i++)
                if (*a++ == key)                        // 探索成功
                        return i;
        return -1;                                      // 探索失敗
}

int main(void)
{
        int key, idx;
        int x[8];
        int x_size = sizeof(x) / sizeof(x[0]);          // 配列xの要素数

        for (int i = 0; i < x_size; i++) {
                cout << "x[" << i << "] : ";
                cin >> x[i];
        }
        cout << "探す値は：";
        cin >> key;

        if ((idx = seq_search(x, x_size, key)) != -1)
                cout << "その値をもつ要素はx[" << idx << "]です。\n";
        else
                cout << "見つかりませんでした。\n";
}
```

ポインタによる配列要素の走査: C++

「

```
int seq_search(int* a, int n, int key)
{
        for (int i = 0; i < n; i++)
                if (*a++ == key)                // 探索成功
                        return i;
                return -1;                      // 探索失敗
}
```

」

「

```
if ((idx = seq_search(x, x_size, key)) != -1)
```

」

関数 seq_search の「a*++ == key」においてaはポインタ変数．よってa*はそのポインタが指し示す格納場所．main関数では配列x[7]の先頭の番地がseq_searchに引き渡される．よって上のfor文のループの1巡目ではa*はx[0]を表す．よって「*a == key」ではx[0]の中のデータとkeyが一致しているかどうかを調べる．さらに「*a++ == key」では，「*a == key」の確認が完了後，aのポインタの番地をインクリメントし，for文のループの2巡目では*aはx[1]の格納場所を表す．

ポインタによる配列要素の走査: C++

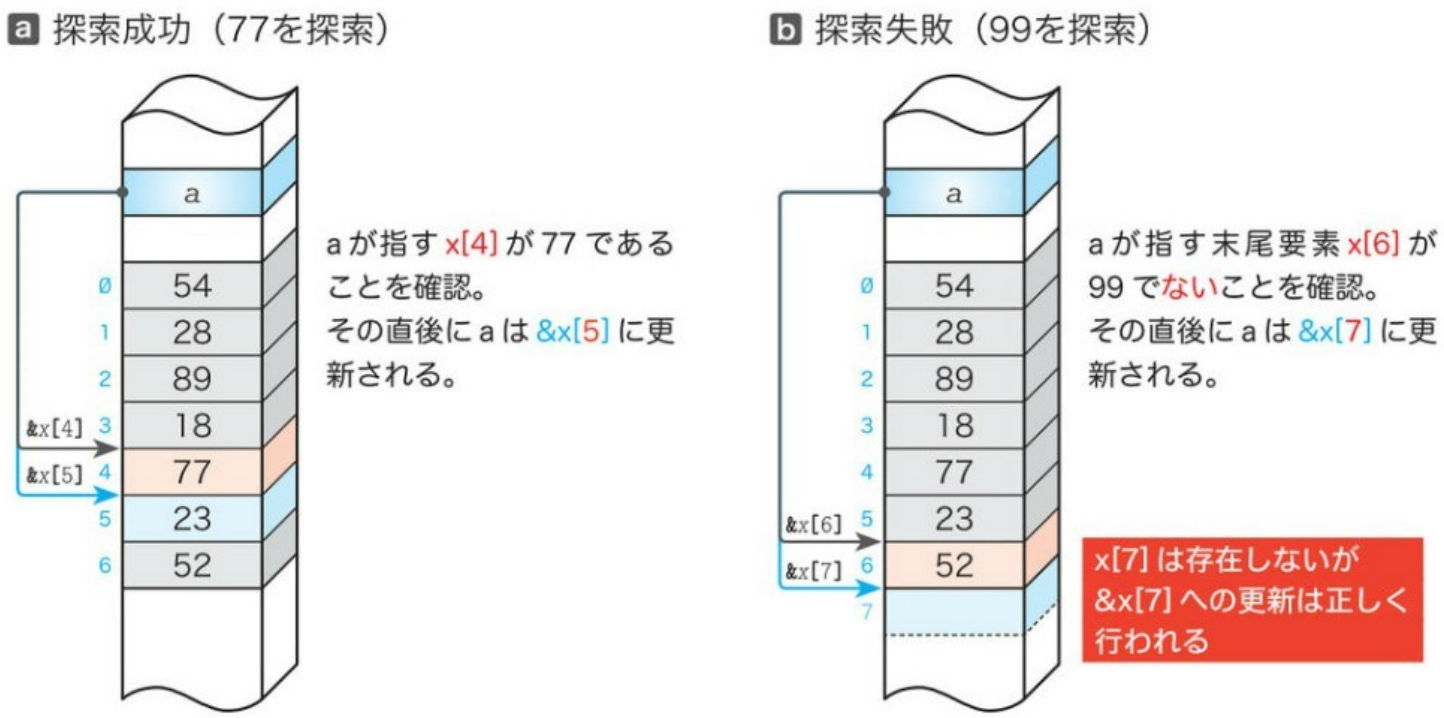

Fig.7-15　探索成功時・失敗時のポインタ

ポインタによる配列要素の走査: C++

List7-15　線形探索2

```
// 線形探索（第２版）

#include <iostream>

using namespace std;

//--- 配列aの先頭n個の要素から値keyを線形探索（第２版）---//
int seq_search(int* a, int n, int key)
{
        int* p = a;

        while (n-- > 0) {
                if (*p == key)                          // 探索成功
                        return p - a;
                else
                        p++;
        }
        return -1;                                      // 探索失敗
}

int main(void)
{
        int key, idx;
        int x[8];
        int x_size = sizeof(x) / sizeof(x[0]);          // 配列xの要素数

        for (int i = 0; i < x_size; i++) {
                cout << "x[" << i << "] : ";
                cin >> x[i];
        }
        cout << "探す値は：";
        cin >> key;

        if ((idx = seq_search(x, x_size, key)) != -1)
                cout << "その値をもつ要素はx[" << idx << "]です。\n";
        else
                cout << "見つかりませんでした。\n";
}
```

ポインタによる配列要素の走査: C++

「

```
int seq_search(int* a, int n, int key)
{
        int* p = a;
        while (n-- > 0) {
                if (*p == key)                  // 探索成功
                        return p - a;
                else
                        p++;
        }
        return -1;                              // 探索失敗
}
```

」

「

```
if ((idx = seq_search(x, x_size, key)) != -1)
```

」

list7-14 と同様であるが，「return p – a;」が異なる．list7-14 では for文で i を引数にして「return i;」で配列の前から「i」番目を返していたが，同じことをポインタ p が表す現在の番地から，配列の先頭のアドレス a との差を返すことにより，まえから数えて「i」番目であることを返す．

ポインタによる配列要素の走査: C++

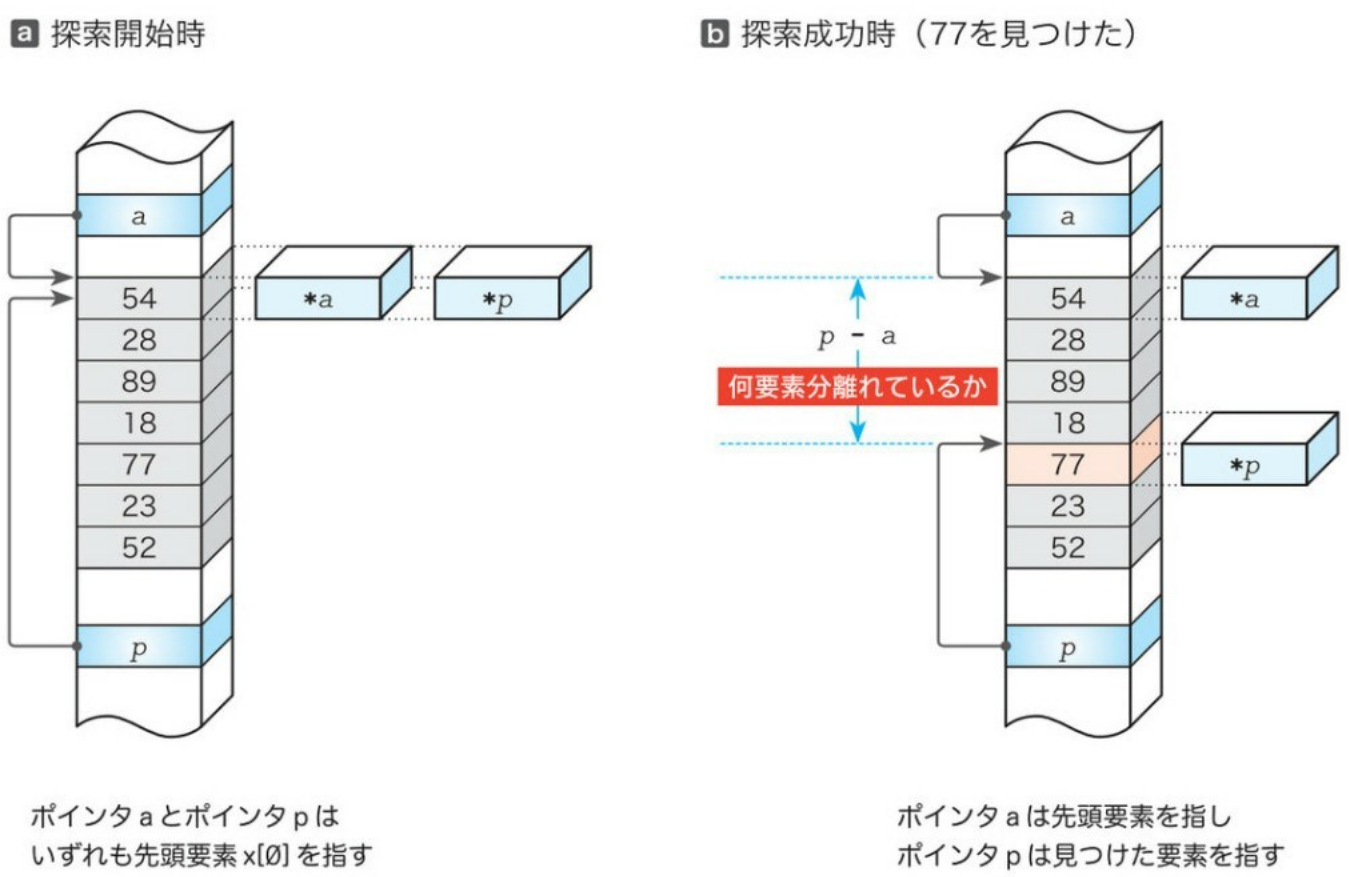

Fig.7-17　線形探索におけるポインタ

オブジェクトの動的な生成: C++

List7-16 整数オブジェクトの動的生成

```
----------------------------------------
// 整数オブジェクトを動的に生成

#include <iostream>

using namespace std;

int main()
{
        int* x = new int;                       // 生成（記憶域の確保）

        cout << "整数 : ";
        cin >> *x;

        cout << "*x = " << *x << '\n';

        delete x;                               // 破棄（記憶域の解放）
}

----------------------------------------
```

オブジェクトの動的な生成: C++

「
int* x = new int;
」
「
delete x;
」
変数やオブジェクトの自動記憶域と静的記憶域の「寿命」(メモリ上に確保される期間)は，プログラムの流れとC/C++の基本的なルールに従って定まる．しかしプログラムによっては，プログラマの意図に沿うように，その記憶域を確保するタイミングや解放するタイミングを強制的に定めたい場合がある．そのように制御される記憶域の確保の期間を「動的記憶域期間」と呼ぶ．そのための命令が「new *type*」と「delete *x*」である．前者を「new 演算子」，後者を「delete 演算子」と呼ぶ.

上の例で「new int;」とすると，メモリ上に int 型のオブジェクトのための記憶域を確保する．さらに「int* x = new int;」によりその確保された記憶域のアドレスが，ポインタ型変数 x に保存される.

次に「delete x;」とすることにより，ポインタ x が指し示すオブジェクトのメモリ上の記憶域が破棄され解放される.

オブジェクトの動的な生成: C++

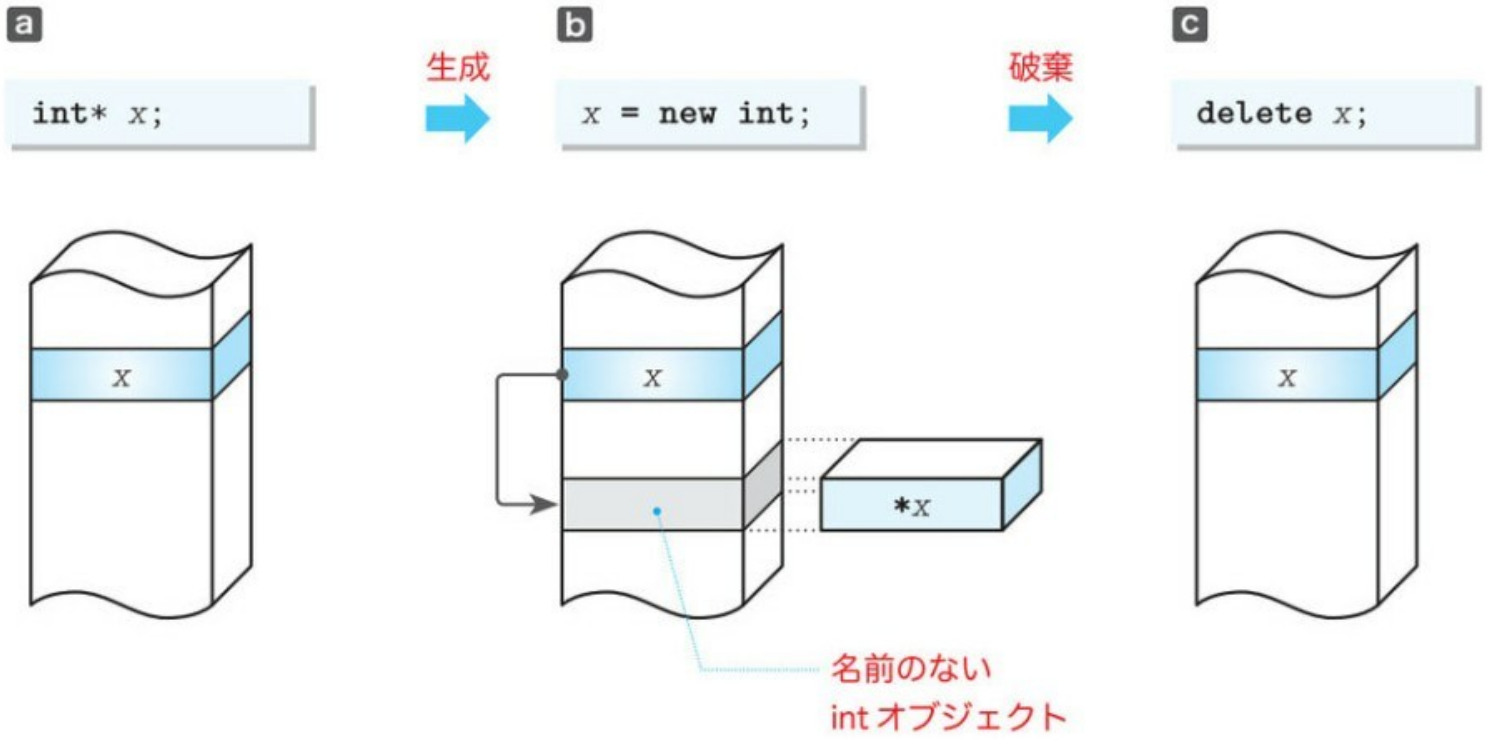

Fig.7-18 動的なオブジェクトの生成と破棄

オブジェクトの動的な生成: C++

List7-18 整数配列オブジェクトの動的生成

```
----------------------------------------
// 整数配列オブジェクトを動的に生成

#include <iostream>

using namespace std;

int main()
{
        int asize;                                              // 配列の要素数
        cout << "要素数：";
        cin >> asize;

        int* a = new int[asize];  // 生成

        for (int i = 0; i < asize; i++)
                a[i] = i;

        for (int i = 0; i < asize; i++)
                cout << "a[" << i << "] = " << a[i] << '\n';

        delete[] a;                                             // 破棄
}
----------------------------------------
```

オブジェクトの動的な生成: C++

「
int* a = new int[asize];
」
これにより整数型配列のオブジェクトの領域を動的に確保し，その先頭アドレスを a に代入している．これによりポインタ変数 a は，あたかも配列 a[assize]と同等の扱いが可能となる．

「
a[i] = i;
」
配列と同等の扱いが可能となった a[ ] を用いて，i番目のa[i]に値 i を代入．

「
delete[] a;
」
配列型オブジェクトの記憶域の破棄の命令は「delete[]」である．ここではポインタ a が指し示す配列型オブジェクトの記憶域を破棄し解放する．

オブジェクトの動的な生成: C++

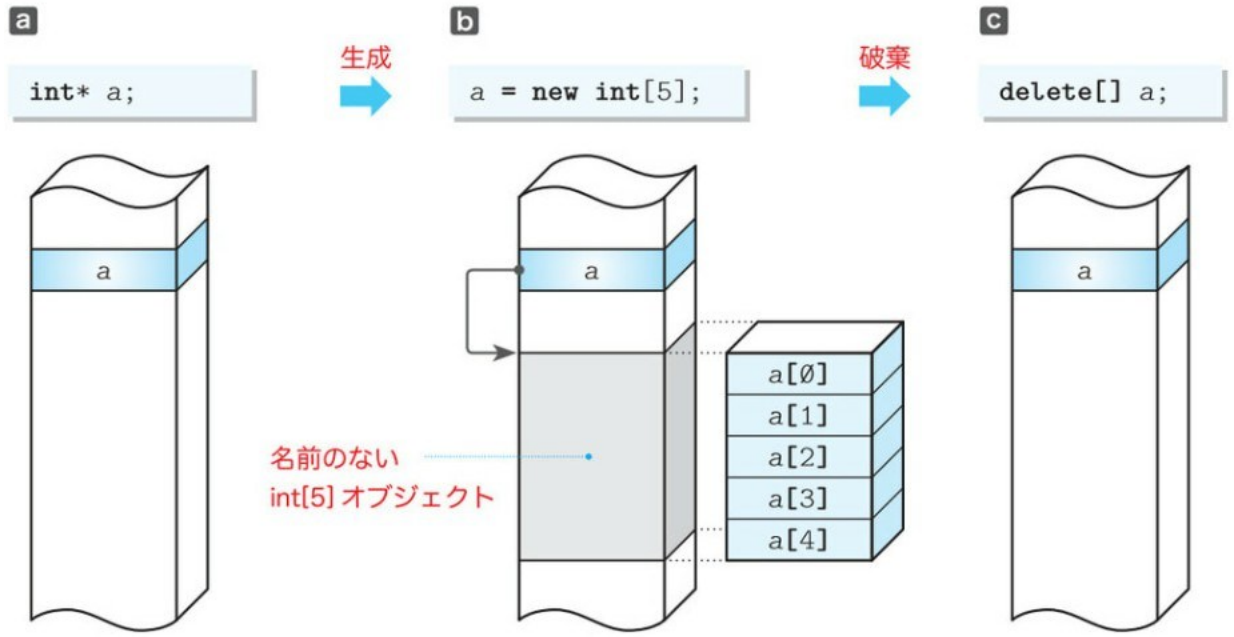

Fig.7-19 動的に生成した配列オブジェクト

課題

課題7-1 （**プログラム名：ex07_1.cpp**）

ポインタ引数で2つの整数を交換する関数を実装せよ。

- 関数プロトタイプ

  void swap(int *a, int *b);

- main 関数で整数を2つ読み取り，swap を呼び出して値を入れ替え，入れ替え後の2値を出力すること。

【実行例】

例1
入力：
10⏎
5⏎

出力：
5 10

例2
入力：
3⏎
-1⏎

出力：
-1 3

⏎ : enter

課題

課題7-2 （**プログラム名：ex07_2.cpp**）

ポインタを用いて，実数配列の最小値・最大値・平均値を計算する関数 calc_stat を実装せよ。

- 関数プロトタイプ

  void calc_stat(float* arr, int n, float* min_val, float* max_val, float* mean);

  arr：入力配列（読み取り専用）

  n：要素数

  min_val，max_val，mean：結果を書き込むためのポインタ（呼び出し側に返す）
- main 関数で 5 個の実数値を読み取り，配列に格納する。
- calc_stat を呼び出して，最小値・最大値・平均値（いずれも float）を計算する。
- 計算結果を出力する（最小値・最大値・平均値の順に，各値の間を半角スペース1つで区切って出力すること。）。

【実行例】

例1
入力：
1 2 3 4 6⏎

出力：
1 6 3.2

例2
入力：
3 5 8 9 3.8⏎

出力：
3 9 5.76

⏎ : enter

課題

課題7-3 （**プログラム名：ex07_3.cpp**）

以下のC++コード（次のページ）を基に、2つの昇順にソートされたint型配列を結合し、結合後の配列も昇順に保つ関数を実装してください。

要件

- merge_sorted関数の完成:この関数は、すでに昇順にソートされている2つの配列a（サイズna）とb（サイズnb）を受け取ります。これらをマージし、結果をout配列（サイズna + nb）に昇順で格納してください。

- main関数内のコメント部分で、完成させたmerge_sorted関数を呼び出してください。その他のmain関数の記述（配列の初期化、出力処理など）は一切変更しないでください。

【実行例】

入力：
なし

出力：

```
PS C:\Users\sxwin\Documents\Lecture\cpp\ex> .\ex07-3.exe
1 3 5 5 8 9 12 13 18
```

課題

課題7-3 （**プログラム名：ex07_3.cpp**）

```
#include <iostream>
using namespace std;


void merge_sorted(int* a, int na, int* b, int nb, int* out)
{
    // この関数を完成させること
}


int main()
{
    // 初期化部分は変更しないこと
    int x[] = {3, 5, 8, 9};
    int y[] = {1, 5, 12, 13, 18};
    int out[9] = {0};


    // merge_sorted関数をここで呼び出すこと


    // 出力部分は変更しないこと
    for (int i = 0; i < sizeof(out) / sizeof(out[0]); i++)
        cout << out[i] << ' ';
}
```