配列の動的生成

アルゴリズムとデータ構造B

第03回

本日の予定

前回までの復習

- 構造体

今日の内容：**配列の動的生成**

- 静的 vs 動的
- 利用関数：`malloc()`
  - ✓ 教科書では `calloc()` を使用

前回の復習

確認レポート　解答例

- 随時，Teams にあげます

問題 2　C 言語の変数やポインタの値とアドレスの示し方をまとめよ。

- 通常の変数　int a;

値：a
アドレス：&a

- 配列の i 番目の要素　int a[N];

値：a[i]
アドレス：&a[i]

- 配列の先頭　int a[N];

値：あえて言えば a[0]
アドレス：a または &a[0]

- 関数　int func (int);

値：あえて言えば関数呼び出し後の戻り値 func(5)
アドレス：func または &func

- 配列 int a[N]; をポインタとしても利用可能

値：*(a+i) または a[i]
アドレス：a+i または &a[i]

- 通常のポインタ変数　int *p = &a;

（ポインタが指す変数の）値：*p
（ポインタが指す変数の）アドレス：p

- 配列として扱うポインタ変数　int *p = a;
  - 配列の先頭
    値：*p または p[0]
    アドレス：p または &p[0]
  - i 番目の要素
    値：*(p+i) または p[i]
    アドレス：p+i または &p[i]
- 関数ポインタ　int (*p_func) (int);
  p_func = func;

値：あえて言えば関数呼び出し後の戻り値 p_func(5)
アドレス：p_func

ex01.c


```c
// 通常のポインタ変数
{
    int a;
    a = 5;
    int *p;
    p = &a;  // 同時に書くと int *p=&a;
    printf("変数aのアドレス, 値をaを用いて出力\n");
    printf("&a : %p, a : %d\n", &a, a);

    printf("設問1 : 変数aのアドレス, 値をポインタpを用いて出力\n");
    // 設問1 : pが指す変数のアドレス, pが指す変数の値をpを用いて出力
    printf("&a : %p, a : %d\n", p, *p);
}


{
    int a[5] = {1, 2, 3, 4, 5};
    int *p1, *p2;
    p1 = &a[2];
    p2 = &a[4];
    printf("\n配列aの2番目, 4番目の要素のアドレス, 値をaを用いて出力\n");
    printf("&a[2] : %p, a[2] : %d\n", &a[2], a[2]);
    printf("&a[4] : %p, a[4] : %d\n", &a[4], a[4]);

    printf("設問2 : 配列aの2番目, 4番目の要素のアドレス, 値をp1, p2を用いて出力\n");
    // 設問2 :
    // 配列aの2番目の要素のアドレス, 値をp1を用いて出力
    printf("&a[2] : %p, a[2] : %d\n", p1, *p1);
    // 配列aの4番目の要素のアドレス, 値をp2を用いて出力
    printf("&a[4] : %p, a[4] : %d\n", p2, *p2);
}
```

ex01.c


```c
// 配列としてのポインタ
{
    int a[5] = {1, 2, 3, 4, 5};
    int *p;
    p = a;  // p = &a[0]; も同じ意味
    printf("\n配列aの先頭（0番目）, 2番目の要素のアドレス, 値をaを用いて出力\n");
    printf("&a[0] : %p, a[0] : %d\n", &a[0], a[0]);
    printf("&a[2] : %p, a[2] : %d\n", &a[2], a[2]);

    printf("設問3 : 配列aの先頭（0番目）のアドレス, 値をpを用いて出力\n");
    // 設問３：設問３：配列aの先頭（0番目）のアドレス，値をpを用いて出力
    printf("a : %p, a[0] : %d\n", p, *p);
    printf("設問4 : 配列aの2番目の要素のアドレス, 値をpを用いて出力\n");
    // 設問４：配列aの2番目の要素のアドレス，値をpを用いて出力
    printf("&a[2] : %p, a[2] : %d\n", p + 2, *(p + 2));
    printf("設問5 : 配列aの2番目の要素のアドレス, 値をpを用いて配列風の記述で出力\n");
    // 設問５：配列aの2番目の要素のアドレス，値をpを用いて配列風の記述で出力
    printf("&a[2] : %p, a[2] : %d\n", &p[2], p[2]);
    printf("設問6 : 配列aの2番目の要素のアドレス, 値をaを用いてポインタ風の記述で出力\n");
    // 設問６：配列aの2番目の要素のアドレス，値をaを用いてポインタ風の記述で出力
    printf("&a[2] : %p, a[2] : %d\n", a + 2, *(a + 2));
}
```

ex01.c


```c
// swap関数の作成
{
    printf("\n設問８～１０：配列の要素の値を入れ替えるswap関数を作成\n");
    // 設問８：正しくプロトタイプ宣言する
    void swap(int *, int *);
    int a[2] = {5, 3};
    printf("a[0] = %d, a[1] = %d\n", a[0], a[1]);
    // 設問９：正しく swap関数呼び出し，a[0]とa[1]の値を交換する
    swap(&a[0], &a[1]);
    // 表示して確認
    printf("swap後\n");
    printf("a[0] = %d, a[1] = %d\n", a[0], a[1]);
}
```

```c
// 設問１０：swap関数を作成する
void swap(int *a, int *b)
{
    int tmp;
    tmp = *a;
    *a = *b;
    *b = tmp;
}
```

ex01.c: swap 関数

二つの変数の値を交換する関数

- 関数ではない場合、単純に書くと

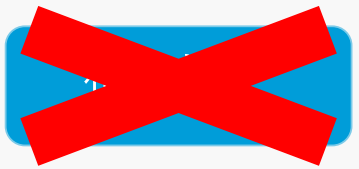
```
int a, b, tmp;
tmp = a; a = b; b = tmp;
```

- これを関数化すると

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

- 関数呼び出しは

```
int a[2] = {5, 3};
swap( a[0],  a[1]);
```



参照渡し

ex01.c: swap 関数

二つの変数の値を交換する関数

- 関数ではない場合、単純に書くと

```
int a, b, tmp;
tmp = a; a = b; b = tmp;
```

- これを関数化すると

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

- 関数呼び出しは

```
int a[2] = {5, 3};
swap(&a[0], &a[1]);
```

ex02.c


```c
// 設問１：氏名(name)，身長(height)，体重(weight)を管理するための構造体を宣言
struct body {
    char name[20];
    int height;
    float weight;
};

// 設問２：構造体変数 a を宣言し，各メンバに "高専太郎", 174, 64.2 を代入
struct body a;
strcpy(a.name, "高専太郎");
a.height = 174;
a.weight = 64.2;
// struct body a = {"高専太郎", 174, 64.2};

printf("設問３：aを用いて「氏名：高専太郎，身長：174 cm 体重：64.2 kg」と出力\n");
// 設問３：aを用いて「氏名：高専太郎，身長：174 cm 体重：64.2 kg」と出力
printf("氏名：%s，身長：%d cm 体重：%.1f kg\n", a.name, a.height, a.weight);

printf("設問４：構造体ポインタ p を用いて，構造体変数 a の情報を設問３と同様に出力\n");
// 設問４：構造体ポインタ p を用いて，構造体変数 a の情報を設問３と同様に出力
struct body *p;
p = &a;
printf("氏名：%s，身長：%d cm 体重：%.1f kg\n", (*p).name, (*p).height, (*p).weight);
// printf("氏名：%s，身長：%d cm 体重：%.1f kg\n", p->name, p->height, p->weight);
```

ex02.c


```c
/*
設問５：struct body 型の構造体配列 b（要素数3）を宣言し，
0番目の要素に "高専太郎", 174, 64.2
1番目の要素に "高専次郎", 163, 58.1
2番目の要素に "高専三郎", 168, 70.3
を代入
*/
struct body b[3];
strcpy(b[0].name, "高専太郎");
b[0].height = 174;
b[0].weight = 64.2;
strcpy(b[1].name, "高専次郎");
b[1].height = 163;
b[1].weight = 58.1;
strcpy(b[2].name, "高専三郎");
b[2].height = 168;
b[2].weight = 70.3;
// struct body b[3] = {
//     {"高専太郎", 174, 64.2},
//     {"高専次郎", 163, 58.1},
//     {"高専三郎", 168, 70.3}
//     };
```

ex02.c


```c
printf("設問６：構造体配列の情報を設問３と同様に，for文を用いて出力\n");
// 設問６：構造体配列の情報を設問３と同様に，for文を用いて出力
int i;
for(i = 0; i < 3; i++) {
    printf("氏名：%s，身長：%d cm 体重：%.1f kg\n", b[i].name, b[i].height, b[i].weight);
}
```

今日はここから

配列の動的生成（記憶域の動的な確保）

- 教科書 **pp. 46-49**
- これまで：配列の記憶域は静的に確保
- 利用関数：**`malloc()`**
  - ✓ 同様の機能を持つ関数として `calloc()` もあり，教科書ではそちらを使用
  - ✓ ポインタが関係

配列の動的生成

静的な確保：`int a[N];` ※Nは十分大きな値

- コンパイル時にそのサイズが固定
- プログラムの実行開始から終了まで確保される
  - ✓ 無駄が多い（かもしれない）

動的な確保：

- プログラム実行中に必要な分だけを変数で指定して確保
- 不要になったら記憶域を解放（`free`）

必要なときに必要な分だけ確保するので効率的

動的生成の手順

1. **配列に相当するポインタを用意する**
   - ✓ 1次元配列ならば、1連鎖（通常の）ポインタ
     - `int a[N];` 相当は `int *p;`
   - ✓ 予習：2次元配列ならば、2連鎖ポインタ
     - `int a[N][N];` 相当は `int **p;`
2. **`malloc()` 関数の戻り値をポインタに代入する**
   - ✓ `p = malloc(sizeof(int)*n);`
   - ✓ ここで `n` は要素数を示す**変数**とする
     - 定数での指定ではあまり意味がない
3. **ポインタ `p` を配列とみなして処理する**
   - ✓ 値：`p[i]`（配列風），`*(p+i)`（ポインタ風）等．配列とポインタの関係を思い出そう
4. **不要になったら `free()`で確保した領域を解放する**

malloc() の使い方

- malloc()（マロック，エムアロック）
- #include <stdlib.h>
  - ✓ このヘッダファイルの中に定義が記述されている
- 型 *ポインタ名; に対して、
  ポインタ名 = malloc(sizeof(型)*要素数);
  - ✓ malloc() の引数は、確保する**バイト数**
  - ✓ sizeof(型) でその型１つ分のバイト数を計算
  - ✓ 要素数は変数で示すようにする
  - ✓ 確保できた場合はポインタにその領域の先頭アドレスのポインタが，
    確保できない場合はNULLポインタが返される
- 使い終わったら領域を解放：free(ポインタ名);

参考：旧・教科書 p.49 Column 2-1

void へのポインタ

calloc 関数，malloc 関数，free 関数は，char 型オブジェクト，int 型オブジェクト，さらには配列や構造体オブジェクトなど，あらゆる型の確保・解放に利用されます．したがって，特定の型のポインタをやりとりする仕様となっていては不都合です．

そこで，融通のきく万能なポインタである void へのポインタを返却したり，受け取ったりする仕様となっています．

void へのポインタは，任意の型のオブジェクトを指すことのできる，特殊な型のポインタです．void へのポインタの値は，任意の型 Type へのポインタに代入することができますし，その逆の代入も可能です．

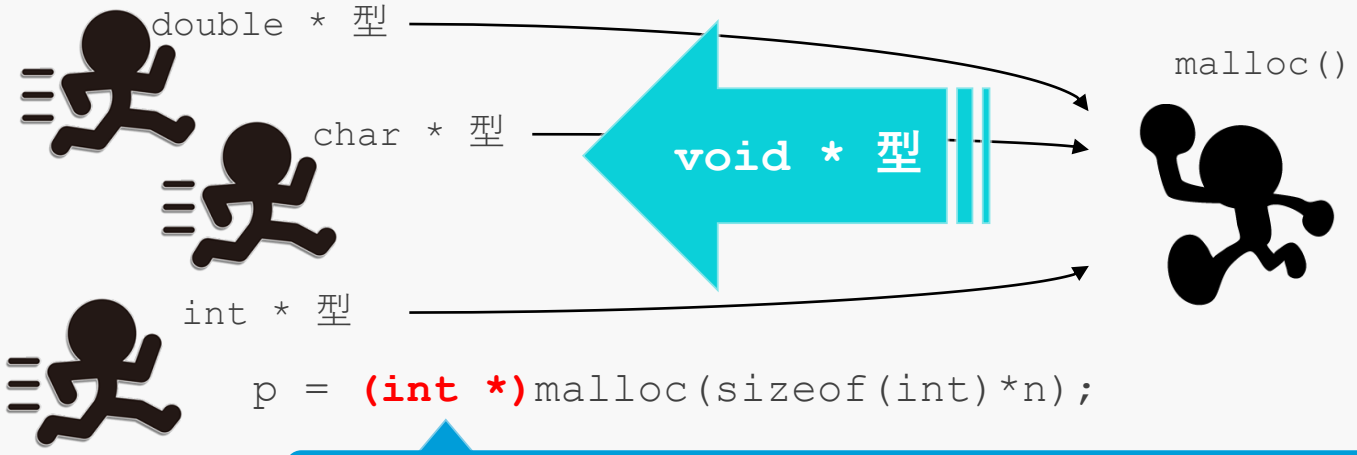
参考：malloc()の戻り値

malloc() は関数 → 戻り値が存在

- int* 型、char* 型、double* 型など…
  様々な型の領域の確保を行う必要

特定の型を戻り値とすると、
異なる型の領域確保の際に不都合が生じる

万能なポインタである void型のポインタを戻り値とする




```
p = (int *)malloc(sizeof(int)*n);
```

型キャストによって型を明示すると丁寧な記述になる

sizeof 演算子

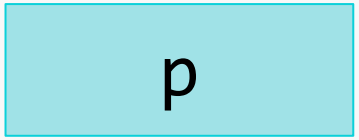
渡された型や変数のメモリサイズを調べる演算子

- 例１．`sizeof(long);`
  - ✓ `long` 型が利用するメモリサイズ
- 例２．`int a[10]; sizeof(a);` や `sizeof(a[0]);`
  - ✓ 配列全体が利用するメモリサイズや先頭要素が利用するメモリサイズ

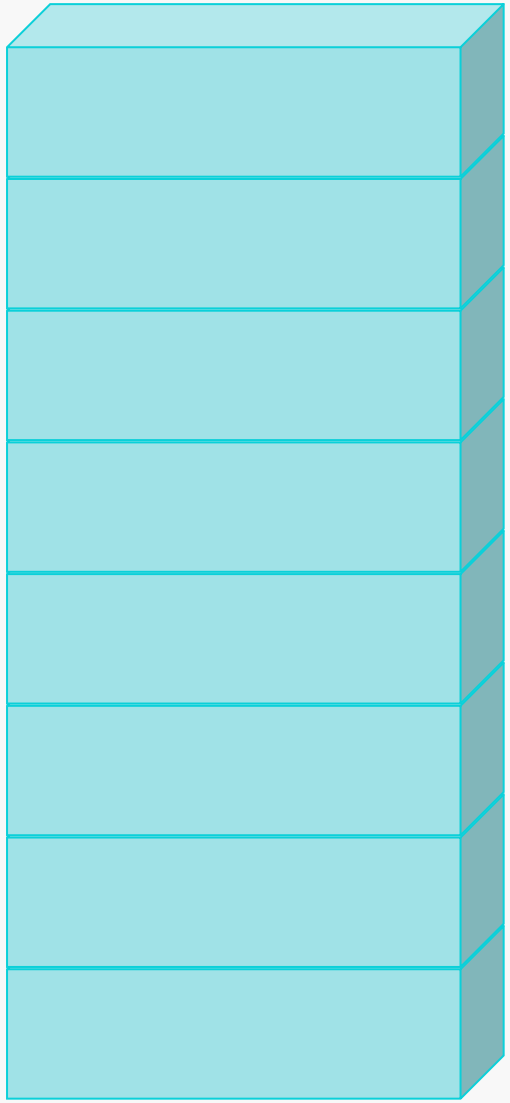
環境によって、ある型が利用するメモリサイズが異なる可能性があるため、sizeof によって取得する

配列の動的生成のイメージ図

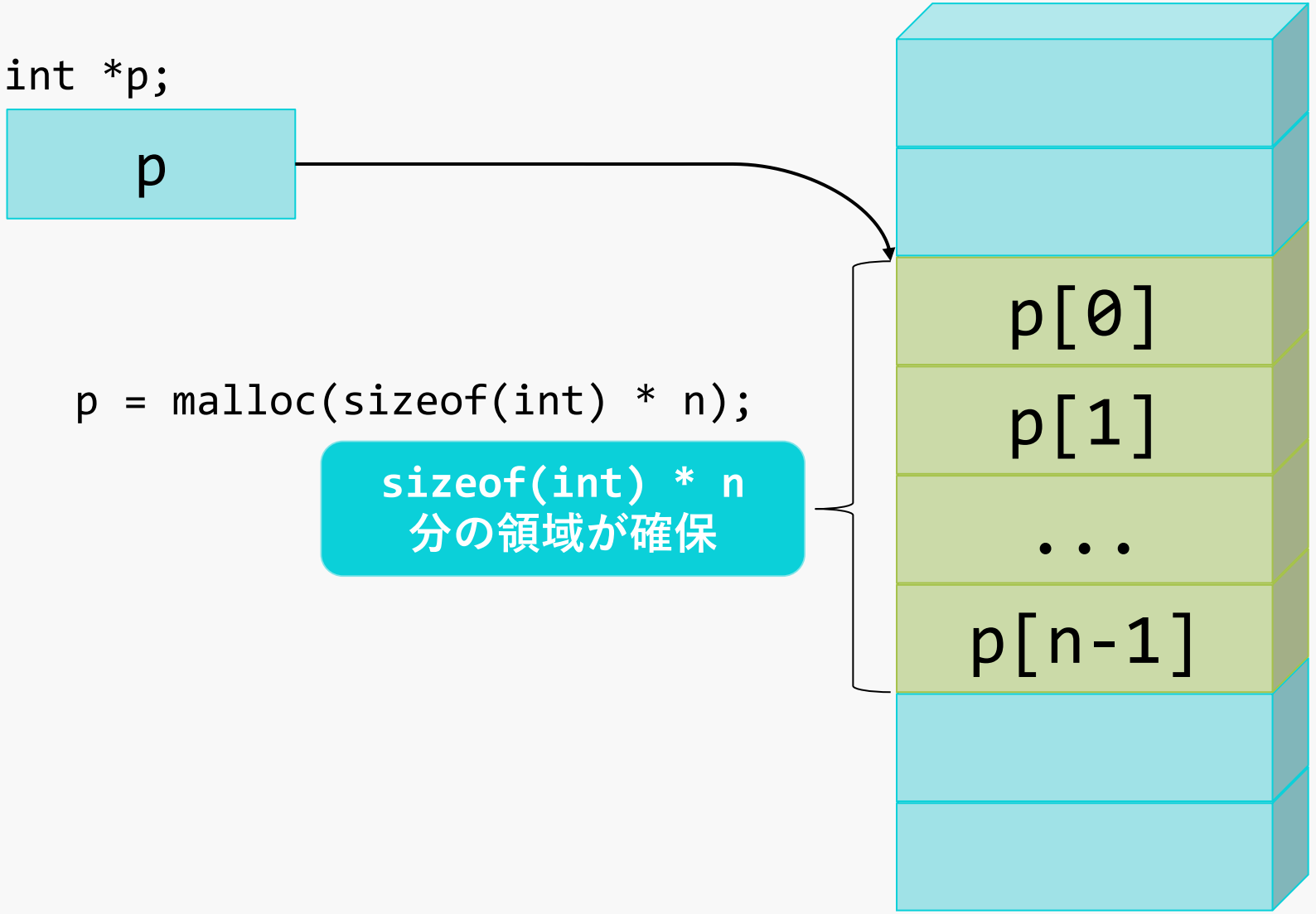

```
int *p;
```




```
p = malloc(sizeof(int) * n);
```



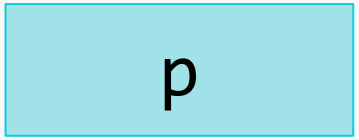
配列の動的生成のイメージ図



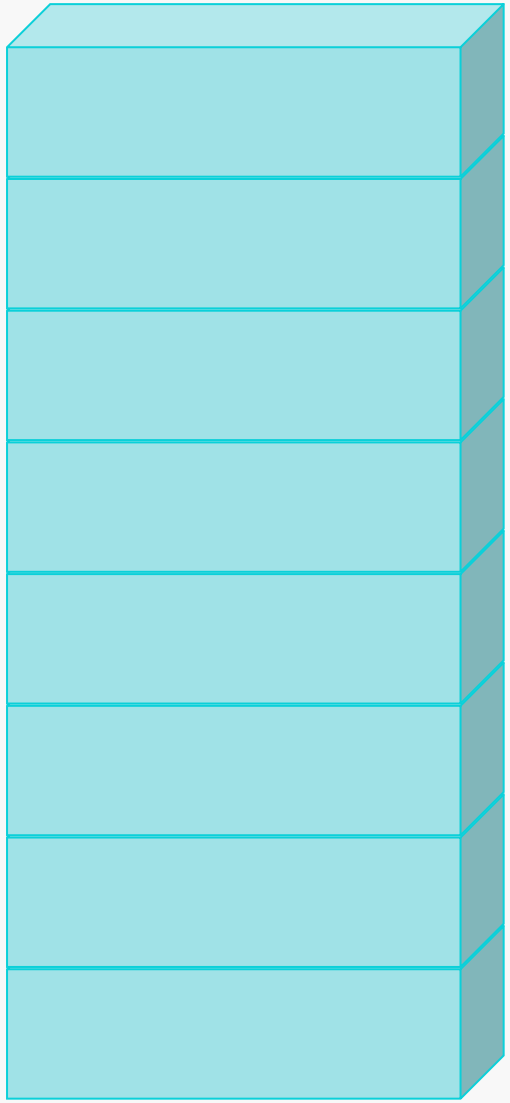
※箱が int 一つ分を
表している
（イメージ）

配列の動的生成のイメージ図


```
int *p;
```




```
free(p);
```



関連：calloc()関数

- #include <stdlib.h>
  - ✓ このヘッダファイルの中に定義が記述されている
- ポインタを宣言：型 *ポインタ名;
- 宣言したポインタに対して、
  ポインタ名 = calloc(要素数, sizeof(型));
  - ✓ sizeof(型) 分のバイト数を要素数分確保する
  - ✓ 確保した領域は 0 クリア（すべてのビットが 0 で初期化）される
  - ✓ 確保できない場合はNULLポインタが代入される
- 使い終わったら領域を解放：free(ポインタ名);

確認レポート

問題 1　以下の文章を完成させよ。

　配列等のための記憶域を［　　］に確保する方法がある．最も簡単な方法は［　　］関数の利用である．これにより，プログラム中の［　　］値によって配列の要素数を指定して領域を確保できる．定数値での確保はあまり意味がない．不要になったら［　　］関数により領域を解放する．

問題 2　`malloc()` 関数によって配列を動的に生成するソースコードを記述せよ．（ポインタ `p` の宣言，要素数 `n` の取得，動的生成，領域解放）

確認レポート：練習問題

練習問題：
以下のポインタを配列として扱うサンプルプログラムについて，実行結果の予想，および，17〜19 行目のポインタ利用法の説明をせよ．

```
1      #include <stdio.h>
2      #define N 5
3      int main() {
4          int a[N] = {0,1,2,3,4};
5          int *p; int i, n;
6
7          // 配列の要素数の算出
8          n = sizeof(a) / sizeof(int);
9          printf("n=%d¥n", n);
10
11         // 配列の要素に対する処理
12         for (i=0; i<n; i++)
13             printf("%d ", a[i]);
14         printf("¥n");
15
16         // ポインタを利用して配列を扱う処理
17         for (i=0, p=a; i<n; i++, p++)
18             printf("%d ", *p);
19         printf("¥n");
20
21     return 0;
22     }
```

演習　ex03.c

バブルソートのプログラム

- 配列を動的に生成し，バブルソートを実行
  - ✓ `malloc()`の引数で変数 `n`(要素数)を利用して記憶域を動的に確保
    - `n` は `scanf()`で入力
  - ✓ バブルソートの関数を呼び出して配列をソート
    - 配列はランダムな値で初期化する
  - ✓ 配列の利用が終わったら領域を解放

出力例


```
❯ ./ex3
設問３：scanfを用いて要素数を取得し　n へ代入
要素数を入力：20
20の要素数の領域を確保しました
設問６：ソート前の配列の中身をfor文を用いて出力
183  86 177 115 193 135 186  92  49  21 162  27  90  59 163 126 140  26 172 136
設問８：ソート後の配列の中身を出力
 21  26  27  49  59  86  90  92 115 126 135 136 140 162 163 172 177 183 186 193
確保した領域を解放しました
```