#11
動的メモリ割当て
2022 年度 / プログラミング及び実習 III

角川裕次

龍谷大学　先端理工学部

もくじ

1. (独自) 動的メモリ割当てとは
   - ライブラリ関数 malloc
   - ライブラリ関数 free

2. (独自) プログラム例
   - Sum and Average
   - 構造体の配列

今回 (#11) の内容： シラバスでの該当部分

小テーマ: 動的メモリ割当て

第 18 回： 構造体を利用したプログラミング
第 19 回： 動的メモリ割り当て

重要概念リスト

- メモリの静的割当てと動的割当て
- malloc 関数によるメモリの動的割当て
- free 関数によるメモリの解放
- メモリリーク

今回の実習・課題 (manaba へ提出)

実習内容と課題内容は講義途中に提示します

(作成したファイル類は manaba に提出)

(独自) 動的メモリ割当てとは

メモリ割当ての 2 方式：静的 / 動的

★重要★

静的メモリ割当て (これまで)：事前に使用するメモリを決定する方法

- プログラミング時に配列の要素数を決定 (宣言)
- 実行時には要素数を変えられない
- 例：`int a[100];`

動的メモリ割当て (今回)：実行時にメモリ量を確保する方法

- 使用するメモリ量は実行時にきまる
- 例：実行時にデータ数をキーボードから入力して配列を確保

動的メモリ割当ての利点

- 実際に必要な量のメモリを必要な時だけ使用できる
- メモリの有効利用ができる

今回の内容：`malloc` 関数と `free` 関数を紹介

メモリ割り当て：静的/動的の比較

★重要★

整数配列で何かするコード断片例 (データ数 n はキーボードから入力)
いずれも配列の要素は d[i] の形でアクセスできる

静的割当：固定配列

```
#define MAX_N  10000
int d[MAX_N];
int n = 0;
printf("データ数nは?\n");
scanf("%d", &n);

if (n > MAX_N) {
  printf("データ数大杉\n");
  exit(1);
}
 ... d[i] を使用 ...
```

事前に多めの要素数で宣言
(n が大きい場合用)

動的割当：malloc & free

```
int *d = NULL;
int n = 0;
printf("データ数nは?\n");
scanf("%d", &n);
d = malloc(n*sizeof(int));
if (d == NULL) {
  printf("データ数多すぎ\n");
  exit(1);
}
 ... d[i] を使用 ...
free(d);
```

事前にメモリは用意しない
実行時の n に応じて必要量を確保

固定配列プログラムを動的メモリ割り当て方式へと変更できますよね

ライブラリ関数 malloc

malloc 関数：メモリの割当て

★重要★

`void *malloc(size_t size)`

- size バイト分のメモリを割り当てる
- 返り値は割り当てられたメモリへのポインタ
- ただしメモリ不足の場合には NULL が返る

例

```
int n = get_datasize();
int *a = malloc(n * sizeof(int));
if (a == NULL) {
  fprintf(stderr, "NO MEMORY\n");
}
```

- 要素数 n の int 配列用のメモリを要求
- n は実行時になって決まる (実行ごとでも違う)

malloc の使用に際して必要なヘッダのインクルード

```
#include <stdlib.h>
```

malloc：使用例 1

割当て (要素数の n の int 配列)

```
int n = get_datasize();
int *a = malloc(n * sizeof(int));
```

- データ数を変数 n に得る
- その数ぶんのデータ型 int のメモリを割当て
- n * sizeof(int)：要求するメモリのバイト数の計算式

使用例：割当後は配列のように使える

```
a[0] = 0;
a[1] = 10;
```

- 確保した要素数に注意
- 添字は 0 から (要素数 -1) まで

malloc：使用例 2

割当て (配列 x：要素数 n の構造体 struct xyz_s の配列)

```
struct xyz_s {
  int x, y, z;
};
int n = get_datasize();
struct xyz_s *x = malloc(n * sizeof(struct xyz_s));
```

- データ数を変数 n に得る
- その数ぶんの構造体 struct xyz_s のメモリを割当て

使用例：割当後はふつうの配列と同様に使える

```
x[0].x = 0;
x[0].y = 1;
x[1].z = 2;
```

- 確保した要素数までしか使えない

malloc コードパターン：メモリ確保編

★重要★

コードパターン

```
データ型名 *p = malloc( 個数 * sizeof(データ型名) );
```

例 1：int を n 個

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

例 2：double を ndata 個

```
double *f = malloc(ndata * sizeof(double));
```

例 3：構造体 struct Point を n 個

```
struct Point *p = malloc(n * sizeof(struct Point));
```

例 4：構造体 struct student を 1 つ確保

```
struct student *p = malloc(sizeof(struct student));
```

malloc コードパターン：エラーチェック編 (1)

★重要★

メモリ割当てが失敗する場合あり：返り値のチェックを忘れずに

```
データ型名 *p = malloc( 個数 * sizeof(データ型名) );
if (p == NULL) {    /* 空きメモリがもう無い */
  エラー処理;
}
```

例

```
int *a = malloc( ndata * sizeof(int) );
if (p == NULL) {    /* 空きメモリがもう無い */
  fprintf(stderr, "NO MEMORY\n");
  exit(1);
}
```

★重要★

malloc コードパターン：エラーチェック編 (2)

別の書き方 (1)

```
データ型名 *p = NULL;
p = malloc( 個数 * sizeof(データ型名) );
if (p == NULL) {    /* 空きメモリがもう無い */
  エラー処理;
}
```

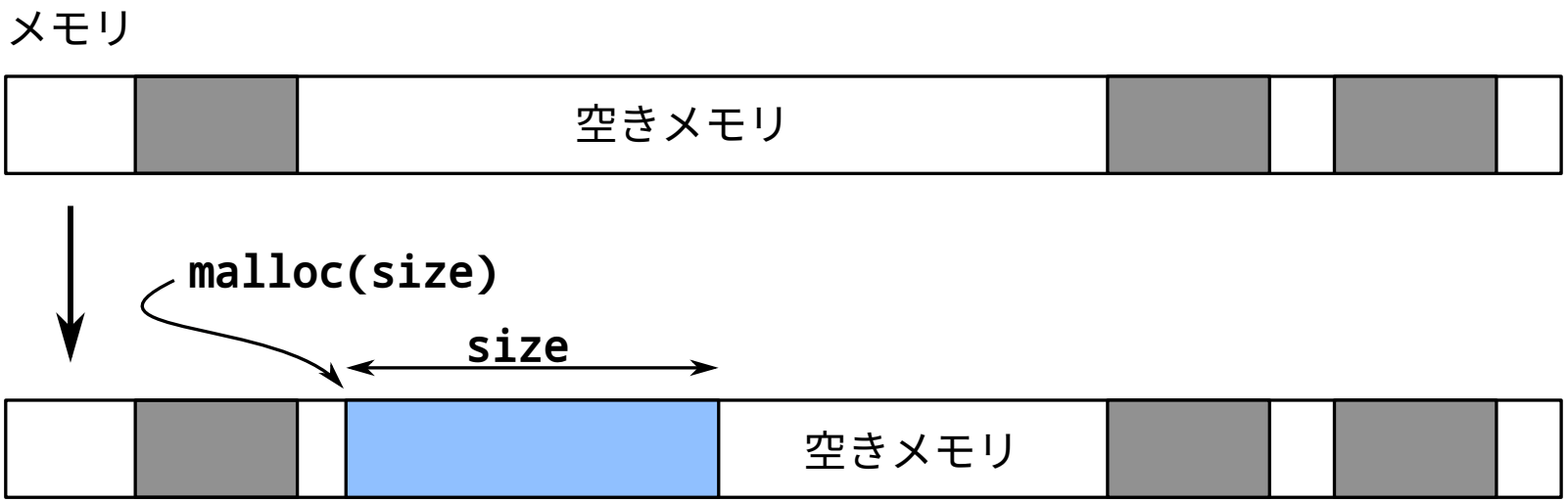
別の書き方 (2)

```
データ型名 *p = NULL;
if ((p = malloc(個数 * sizeof(データ型名))) == NULL) {
  /* 空きメモリがもう無い */
  エラー処理;
}
```

malloc 関数：要求バイト数とデータ型の関係

割り当て：`p = malloc(size);`

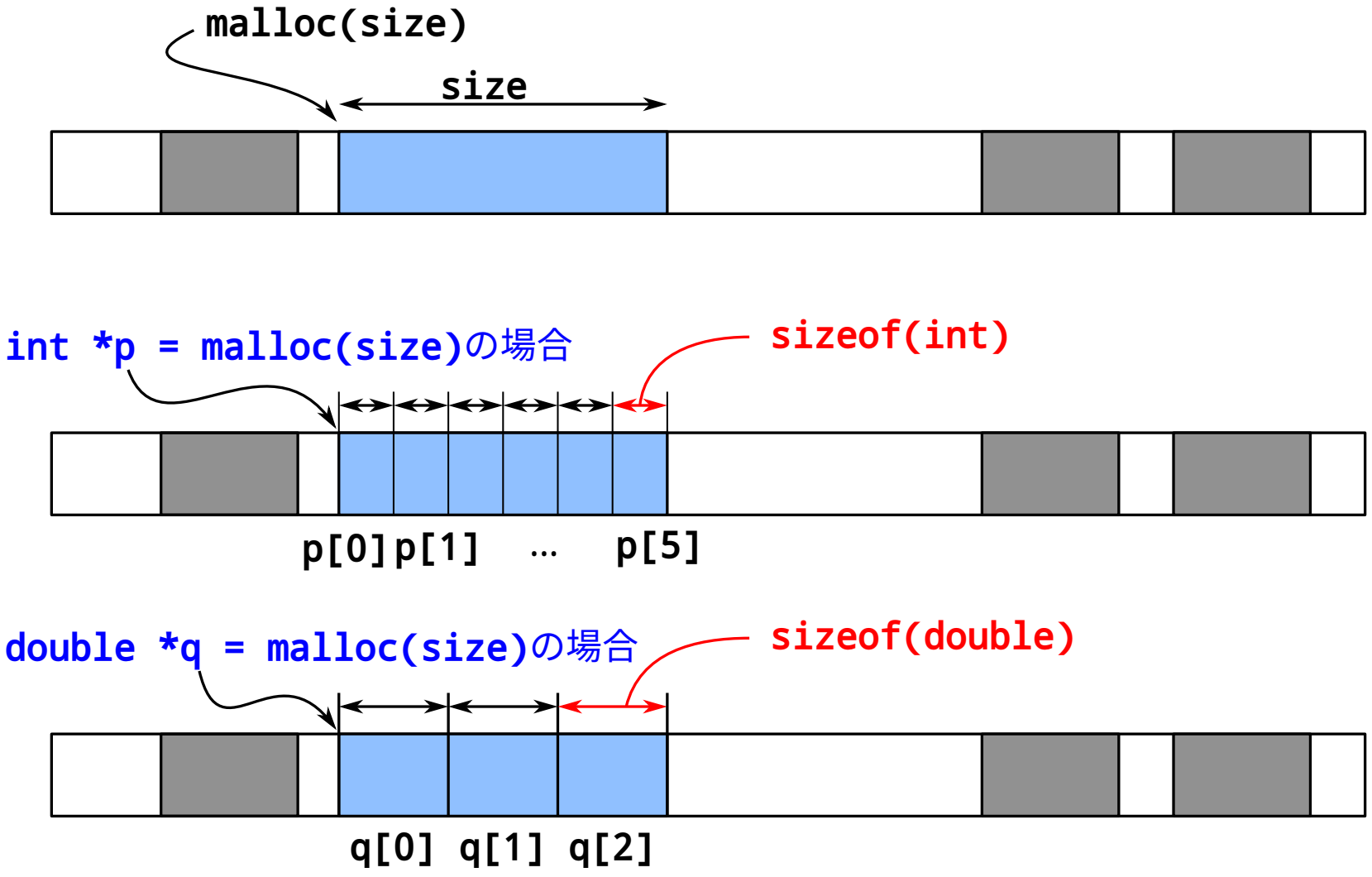
引数：割当て要求するバイト数 (`size`) だけ
返り値：割当てられたメモリへのポインタ (`void*`型)



Q. malloc の引数にデータ型 (`int` とか) の情報がないけど... ？

A. 後ぎめ：返り値を受け取るポインタ型で決める

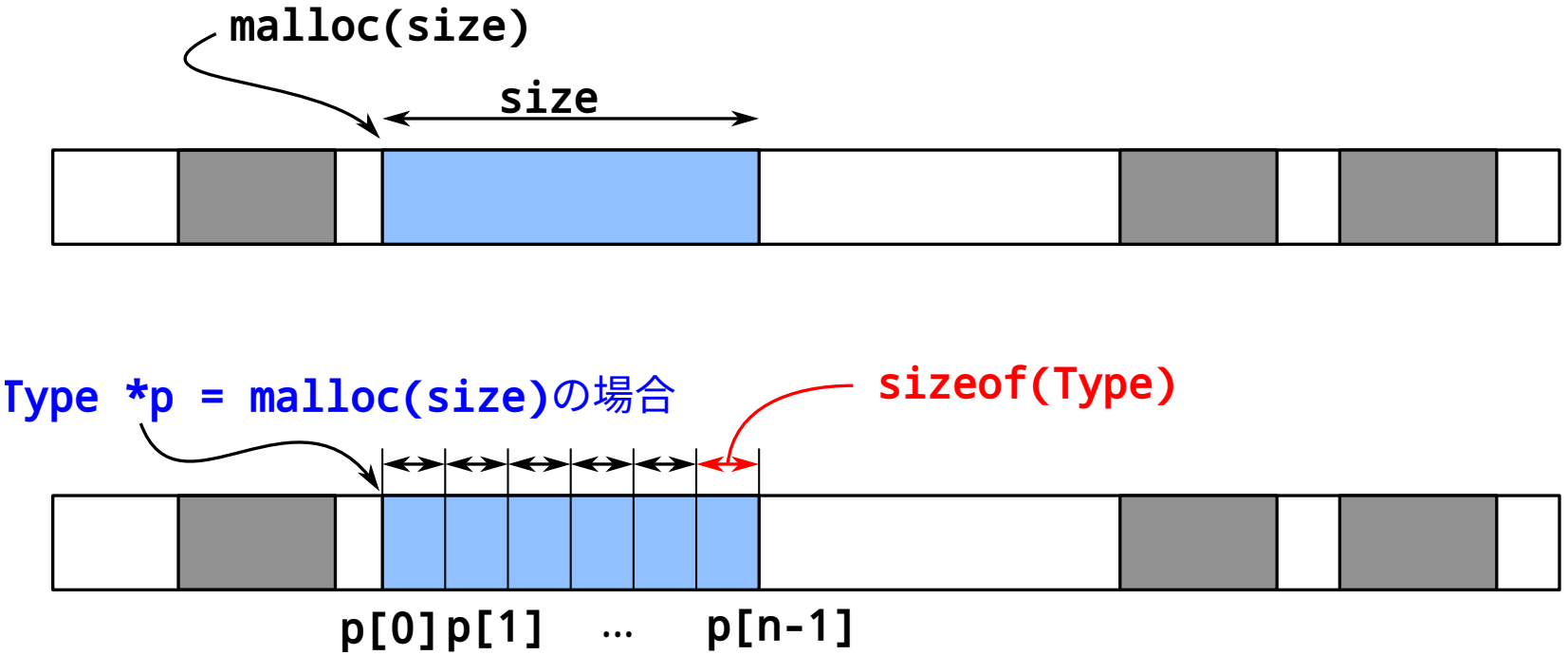
malloc 関数の返り値の解釈：代入先のデータ型で決まる



malloc 関数の返り値の解釈：配列

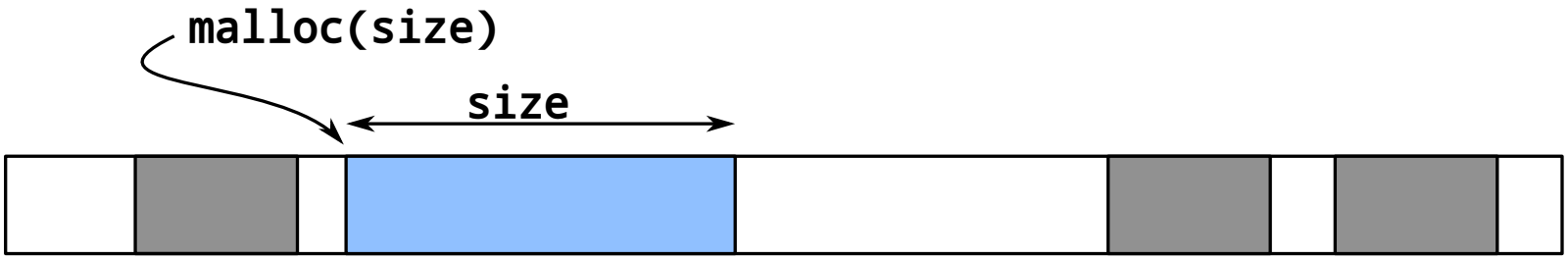
要素数 n 個の *Type* 型データの配列のサイズ：

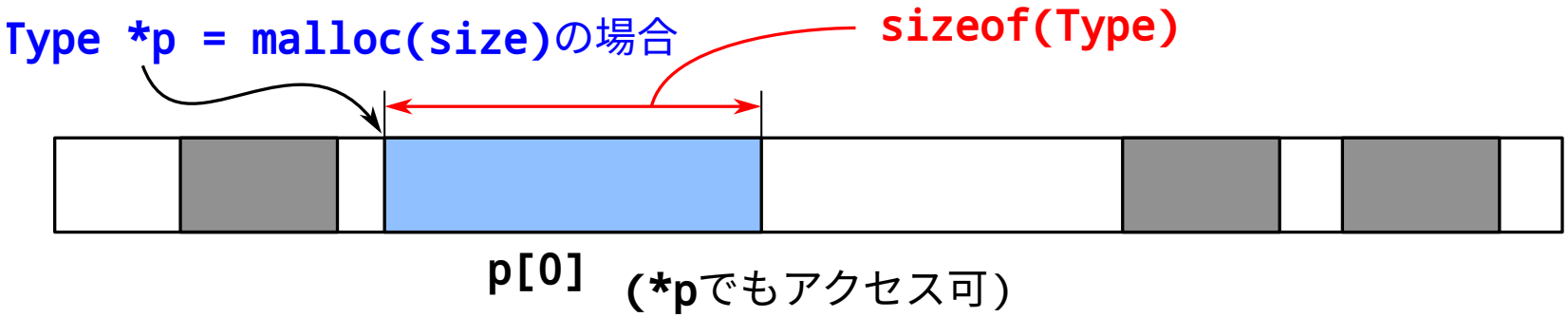
size $= n \times$ `sizeof(`*Type*`)` バイト



malloc 関数の返り値の解釈：1 つのオブジェクト

1 つの *Type* 型データのサイズ：
size = `sizeof(`*Type*`)` バイト

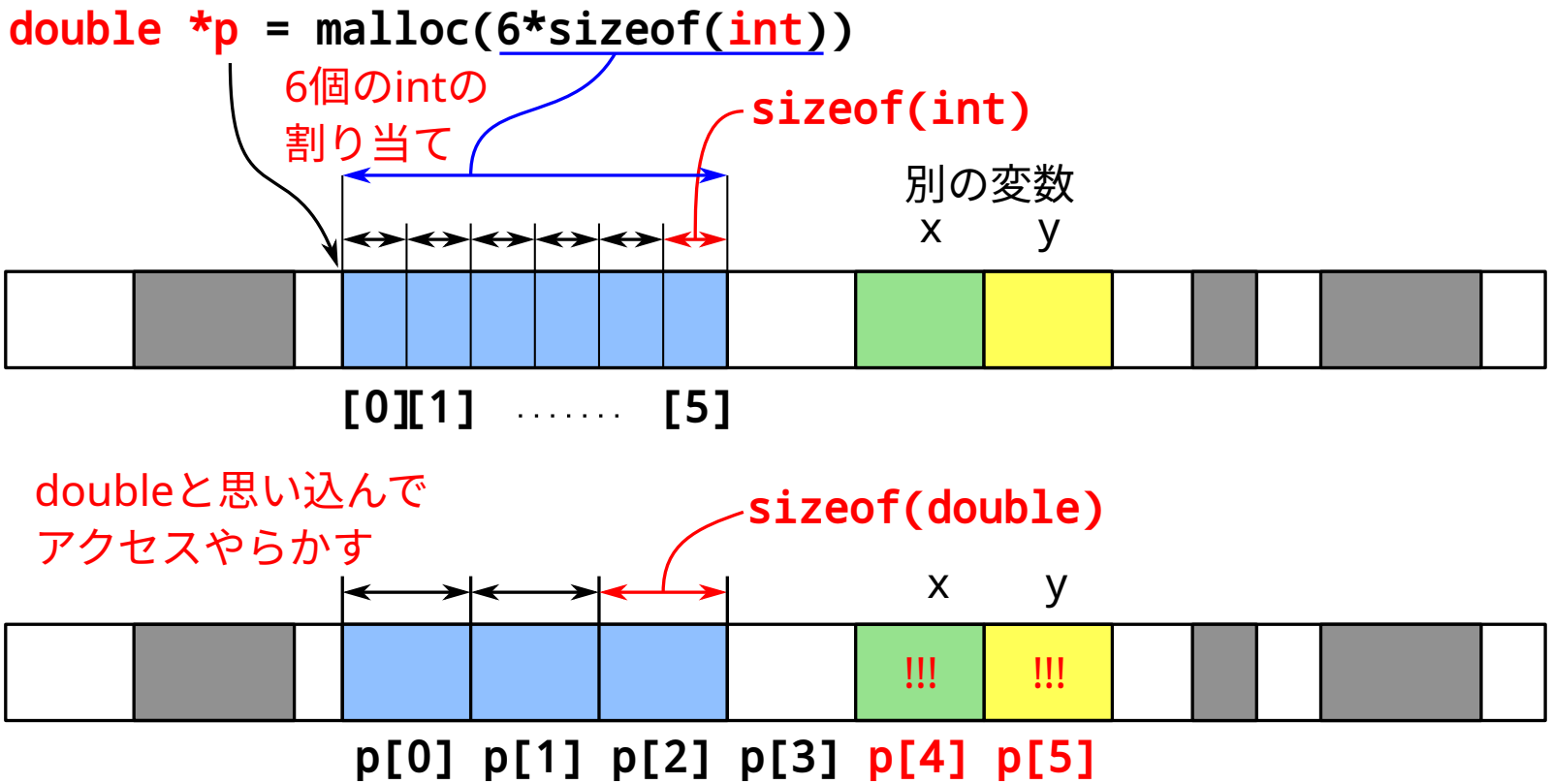




★重要★

malloc コードパターン：間違い編

データ型名が不一致 (double と int)：でもこの割り当て自体はできる



でも `p[5] = 1.0;` を実行した時にきっと変なことが起きる

- `sizeof(int)` と `sizeof(double)` が違う時

malloc コードパターン：これ覚えておけば OK

★重要★

パターン

```
データ型 *変数 = malloc( 個数 * sizeof(データ型));
if (変数 == NULL) {
  エラー処理;
}
```

具体例

```
int *a = malloc(n * sizeof(int));
if (a == NULL) {
  printf("NO MEMORY\n");
  exit(1);
}
```

ライブラリ関数 free

free 関数

★重要★

`void free(void *ptr)`

- ポインタ ptr が指すメモリを解放する
- ただし malloc で割り当てられたメモリに限る

注意：#include <stdlib.h> でヘッダをインクルードすること

解放したメモリは後の malloc 呼び出しで再利用される場合あり

使用例

```
int *p = malloc(n * sizeof(int));   /*割当*/
  ... 略 ...
free(p);   /*解放*/
```

禁止事項１：メモリを解放した後のアクセス
禁止事項２：解放済みメモリを再び解放

- 解放後のメモリは再利用されている場合があるため

コードパターン (1)：main 関数にて

```
int main (void) {
  int n = ...; /*データ数*/
  int *p = malloc(n * sizeof(int));   /*割当*/
  read_data(n, p); /*データ読み込み*/
  compute(n, p);    /*計算*/
  output(n, p);     /*結果を出力*/
  free(p);  /*解放*/
  return 0;
}
```

機能単位に分解してみた

- データ個数 n を得る
- 作業用のメモリを割り当てる (malloc)
- データを読み込んで, 計算し, 結果を出力
- メモリを解放 (free)

コードパターン (2) : 繰り返し

```
for (;;) {
  int n = ...;
  int *p = malloc(n * sizeof(int));  /*割当*/
  配列 p を使った処理;
  free(p);  /*解放*/
}
```

処理の繰り返し (int 型データの配列を想定した場合)

- int データの個数 n を得る
- データのメモリサイズは n * sizeof(int) に注意
- 作業用のメモリを割り当てる (malloc)
- 実際に処理をする
- メモリを解放 (free) して次回の処理へ

★重要★

メモリリーク (memory leak) とは

```
for (;;) {
  int n = ...;
  int *p = malloc(n * sizeof(int));   /*割当*/
  配列pを使った処理;
}
```

ここでやっていること

- 作業用のメモリを割り当てる
- メモリを解放せずに次回の処理を繰り返す

やがてメモリを全て使い果たす

- メモリ不足により malloc がメモリ割り当てに失敗
- プログラムが強制終了されてしまう

開放されないメモリがどんどん増えてゆく

- メモリリークと呼んでいる (バグの一種)

メモリリーク (もうひとつ) : 関数内ローカルで

★重要★

関数 foo 内で局所変数に割当ポインタを代入

```
int foo(int n) {
  int *p = malloc(n * sizeof(int));   /*割当*/
  配列 p を使った処理;
  return ...;
}
```

関数から戻るときにメモリリーク発生

このように書きましょう

```
int foo(int n) {
  int *p = malloc(n * sizeof(int));   /*割当*/
  配列 p を使った処理;
  free(p);
  return ...;
}
```

Q. これメモリリークしないですか?

free せずにプログラムを終了... 大丈夫?

```
int main(void) {
  int n = 100;
  int *p = malloc(n * sizeof(int));   /*割当*/
  return 0;
}
```

A. リークしてるけど大丈夫
プログラム終了時に自動的に解放される

- そのような仕組みになっているのが普通の OS
- よっぽど特殊な OS でない限り...
  (まだ見たことがありません; ぜひ見てみたい)

(独自) プログラム例

Sum and Average

Sum and Average : 問題設定

n 個の実数値データの総和と平均値を求める

要求仕様

- データは実数値
- データ数 n はキーボードより入力
- データを記憶する配列は malloc で動的に割り当てる
- n 個のデータは順次キーボードより入力
- 総和と平均値を計算して表示

プログラム名 : `avr`

実行例

```
$ ./avr
データ数n : 3
データ0の値: 12.4
データ1の値: 11.4
データ2の値: 16.2
総和=40.000000
平均=13.333333
```

1. データ数は 3 と入力
2. 3 つの実数データを順次入力
3. 入力データに対する総和と平均値を計算・表示

実装の方針 (機能分割の方針)

ソースファイル名：`avr.c`

データは実数値
　→ `double` 型のデータとする

データ数 `n` はキーボードより入力
　→ 関数 `int get_n(void)` で実現

データを記憶する配列は動的に割り当てる
　→ 関数 `double *alloc_array(int n)` で実現
　　“alloc” ... allocate (割当) の略でつかってます

`n` 個のデータは順次キーボードより入力
　→ 関数 `void read_data(int n, double *x)` で実現

総和と平均値を計算して表示
　→ 関数 `void compute(int n, double *x)` で実現

実装：ソースファイル全体像

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
  ...下記の関数を順次呼び出す...
}
int get_n(void) {
  ...データ数nをキーボードより入力...
}
double *alloc_array(int n) {
  ...データを記憶する配列を動的に割り当てる...
}
void read_data(int n, double *x) {
  ...n個のデータをキーボードより入力...
}
void compute(int n, double *x) {
  ...総和と平均値を計算して表示...
}
```

実装：main 関数

```
#include <stdio.h>
#include <stdlib.h>

int get_n(void);
double *alloc_array(int n);
void read_data(int n, double *x);
void compute(int n, double *x);

int main(void) {
  int n = 0;          /*データ数*/
  double *x = NULL; /*データ記憶用の配列(動 的 割 当)*/
  /*データ数を得る*/
  n = get_n();
  /*配列のメモリを確保*/
  x = alloc_array(n);
  /*データを配列に読み込む*/
  read_data(n, x);
  /*計算処理を行う*/
  compute(n, x);
  return 0;
}
```

実装：関数 get_n

```
int get_n(void) {
  int n = 0;
  while (n <= 0) {
    char s[1024];
    printf("データ数n : ");
    if (fgets(s, sizeof(s), stdin) == NULL) {
      exit(0);
    }
    n = atoi(s);
  }
  return n;
}
```

scanf を使わず fgets 使用してみた例

実装：関数 alloc_array

```
double *alloc_array(int n) {
  double *x = NULL;
  x = malloc(n * sizeof(double));
  if (x == NULL) {
    printf("NO MEMORY\n");
    exit(1);
  }
  return x;
}
```

実装：関数 read_data

```
void read_data(int n, double *x) {
  for (int i = 0; i < n; i++) {
    char s[1024];
    printf("データ%dの値： ", i);
    if (fgets(s, sizeof(s), stdin) == NULL) {
      exit(0);
    }
    sscanf(s, "%lf", &x[i]);
  }
}
```

実装：関数 compute

```
void compute(int n, double *x) {
  double sum = 0;
  double avr = 0;
  for (int i = 0; i < n; i++) {
    sum = sum + x[i];
  }
  avr = sum / n;
  printf("総和=%f\n", sum);
  printf("平均=%f\n", avr);
}
```

構造体の配列

個人の情報を表す構造体 struct student

```
struct student {
  char    name[64]; /*名前*/
  int     height;   /*身長*/
  float   weight;   /*体重*/
  long    schols;   /*奨学金*/
};
```

構造体の回を思い出しましょう...

任意の人数の情報を取り扱いたい

- 実行時に人数が指定される

→ この構造体の配列を動的メモリ割当で実現

割当

構造体 student の n 個ぶんのメモリを割当てる

```
struct student *p = NULL;
p = malloc(n * sizeof(struct student));   /*割当*/
if (p == NULL) {
  fprintf(stderr, "NO MEMOEY\n"); /*メモリ割当て失敗*/
  exit(1);
}
```

i 番目の要素にアクセス

```
p[i].height = 162;
p[i].weight =  58;
```

おわり

番外編の課題 1

文字列の配列を連結して 1 つの文字列にして返す関数の作成

- `char *concat_str(char **s);`
- 新たな文字列をメモリ確保して作成
- 文字列の長さは任意: 事前に最大長は分からない

例 1

```
char *sx1[] = { "Hello", " ", "world", NULL };
```

`concat_str(sx1)` の結果 : `"Hello world"`

例 2

```
char *sx2[] = { "Jugemu", "Jugemu", "Gogouno", NULL };
```

`concat_str(sx2)` の結果 : `"JugemuJugemuGogouno"`

番外編の課題 2

メモリの許す限り任意の桁数の整数値を取り扱えるようにしたい (1 万桁でも取り扱い可能にする)。以下の関数を作成せよ。

- 数を記憶するデータ型 `struct bignum` を以下の通りとする

```
struct bignum {
  int    n;   /* 桁数 */
  char *d;   /* d[i]は10進数表現で第i桁目の数 */
};
typedef struct bignum bn;
```

- 数は 10 進数 n 桁で表現し、配列 d の 1 要素で 1 桁を記憶する
- `bn *bn_new(int v);` — 値 v の数オブジェクトの生成.
- `bn *bn_print(bn *bn1);` — 数の表示
- `bn *bn_add(bn *bn1, bn *bn2);`
  — 2 つの数を加算した結果を新たな数オブジェクトとして返す
- `bn *bn_mul(bn *bn1, bn *bn2);`
  — 2 つの数を乗じた結果を新たな数オブジェクトとして返す
- 1000 の階乗を計算し表示するプログラムを作成せよ