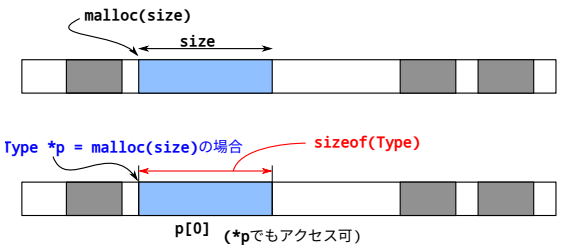
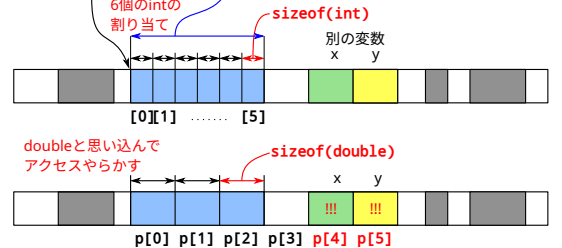


	もくじ	今回 (#11) の内容：シラバスでの該当部分
<div> <div>#11 動的メモリ割当て 2022 年度 / プログラミング及び実習 III</div> <div>角川裕次 龍谷大学 先端理工学部</div> </div> <div>1 / 45</div>	<div> <div>1 (独自) 動的メモリ割当てとは <ul style="list-style-type: none"> <li>■ ライブラリ関数 malloc</li> <li>■ ライブラリ関数 free</li> </ul> </div> <div> <div>2 (独自) プログラム例 <ul style="list-style-type: none"> <li>■ Sum and Average</li> <li>■ 構造体の配列</li> </ul> </div> </div> <div>2 / 45</div> </div>	<div>小テーマ：動的メモリ割当て</div> <div>第 18 回：構造体を利用したプログラミング</div> <div>第 19 回：動的メモリ割当て</div> <div>3 / 45</div>
重要概念リスト	今回の実習・課題 (manaba へ提出)	
<ul style="list-style-type: none"> <li>■ メモリの静的割当てと動的割当て</li> <li>■ malloc 関数によるメモリの動的割当て</li> <li>■ free 関数によるメモリの解放</li> <li>■ メモリリーク</li> </ul> <div>4 / 45</div>	<div>実習内容と課題内容は講義途中に提示します</div> <div>(作成したファイル類は manaba に提出)</div> <div>5 / 45</div>	<div>(独自) 動的メモリ割当てとは</div> <div>6 / 45</div>

メモリ割当ての2方式：静的 / 動的 <span>★重要★</span>	メモリ割り当て：静的/動的の比較 <span>★重要★</span>	
<p><b>静的メモリ割当て</b> (これまで)：事前に使用するメモリを決定する方法</p> <ul style="list-style-type: none"> <li>■ プログラミング時に配列の要素数を決定 (宣言)</li> <li>■ 実行時には要素数を変えられない</li> <li>■ 例：int a[100];</li> </ul> <p><b>動的メモリ割当て</b> (今回)：実行時にメモリ量を確保する方法</p> <ul style="list-style-type: none"> <li>■ 使用するメモリ量は実行時にきまる</li> <li>■ 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</li> </ul> <p>動的メモリ割当ての利点</p> <ul style="list-style-type: none"> <li>■ 実際に必要な量のメモリを必要な時だけ使用できる</li> <li>■ メモリの有効利用ができる</li> </ul> <p>今回の内容：malloc 関数と free 関数を紹介</p> <p>7 / 45</p>	<p>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</p> <p><b>静的割当：固定配列</b></p> <pre>#define MAX_N 10000 int d[MAX_N]; int n = 0; printf("データ数nは?\n"); scanf("%d", &amp;n);  if (n &gt; MAX_N) {     printf("データ数大杉\n");     exit(1); } ... d[i] を使用 ...</pre> <p>事前に多めの要素数で宣言 (n が大きい場合用)</p> <p><b>動的割当：malloc &amp; free</b></p> <pre>int *d = NULL; int n = 0; printf("データ数nは?\n"); scanf("%d", &amp;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);</pre> <p>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</p> <p>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</p> <p>8 / 45</p>	<p>ライブラリ関数 malloc</p> <p>9 / 45</p>
<p><b>malloc 関数：メモリの割当て</b><span>★重要★</span></p> <pre>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</pre> <p>例</p> <pre>int n = get_datasize(); int *a = malloc(n * sizeof(int)); if (a == NULL) {     fprintf(stderr, "NO MEMORY\n"); }</pre> <ul style="list-style-type: none"> <li>■ 要素数 n の int 配列用のメモリを要求</li> <li>■ n は実行時になって決まる (実行ごとでも違う)</li> </ul> <p>malloc の使用に際して必要なヘッダのインクルード #include &lt;stdlib.h&gt;</p> <p>10 / 45</p>	<p><b>malloc：使用例 1</b></p> <p>割当て (要素数 n の int 配列)</p> <pre>int n = get_datasize(); int *a = malloc(n * sizeof(int));</pre> <ul style="list-style-type: none"> <li>■ データ数を変数 n に得る</li> <li>■ その数ぶんのデータ型 int のメモリを割当て</li> <li>■ n * sizeof(int)：要求するメモリのバイト数の計算式</li> </ul> <p>使用例：割当後は配列のように使える</p> <pre>a[0] = 0; a[1] = 10;</pre> <ul style="list-style-type: none"> <li>■ 確保した要素数に注意</li> <li>■ 添字は 0 から (要素数 -1) まで</li> </ul> <p>11 / 45</p>	<p><b>malloc：使用例 2</b></p> <p>割当て (配列 x：要素数 n の構造体 struct xyz_s の配列)</p> <pre>struct xyz_s {     int x, y, z; }; int n = get_datasize(); struct xyz_s *x = malloc(n * sizeof(struct xyz_s));</pre> <ul style="list-style-type: none"> <li>■ データ数を変数 n に得る</li> <li>■ その数ぶんの構造体 struct xyz_s のメモリを割当て</li> </ul> <p>使用例：割当後はふつうの配列と同様に使える</p> <pre>x[0].x = 0; x[0].y = 1; x[1].z = 2;</pre> <ul style="list-style-type: none"> <li>■ 確保した要素数までしか使えない</li> </ul> <p>12 / 45</p>

<div>★重要★</div> <div>malloc コードパターン：メモリ確保編</div> <div>コードパターン</div> <div>データ型名 *p = malloc( 個数 * sizeof(データ型名) );</div> <div>例 1: int を n 個</div> <div>int *a = malloc(n*sizeof(int));</div> <div>例 2: double を ndata 個</div> <div>double *f = malloc(ndata * sizeof(double));</div> <div>例 3: 構造体 struct Point を n 個</div> <div>struct Point *p = malloc(n * sizeof(struct Point));</div> <div>例 4: 構造体 struct student を 1 つ確保</div> <div>struct student *p = malloc(sizeof(struct student));</div> <div>13 / 45</div>	<div>★重要★</div> <div>malloc コードパターン：エラーチェック編 (1)</div> <div>メモリ割当てが失敗する場合あり：戻り値のチェックを忘れずに</div> <div>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; }</div> <div>例</div> <div>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); }</div> <div>14 / 45</div>	<div>★重要★</div> <div>malloc コードパターン：エラーチェック編 (2)</div> <div>別の書き方 (1)</div> <div>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; }</div> <div>別の書き方 (2)</div> <div>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; }</div> <div>15 / 45</div>
<div>malloc 関数：要求バイト数とデータ型の関係</div> <div>割り当て：p = malloc(size);</div> <div>引数：割当て要求するバイト数 (size) だけ</div> <div>戻り値：割当てられたメモリへのポインタ (void*)</div> <div>メモリ</div> <div>空きメモリ</div> <div>malloc(size)</div> <div>size</div> <div>空きメモリ</div> <div>Q. malloc の引数にデータ型 (int とか) の情報がないけど... ?</div> <div>A. 後でめ：戻り値を受け取るポインタ型で決める</div> <div>16 / 45</div>	<div>malloc 関数の戻り値の解釈：代入先のデータ型で決まる</div> <div>malloc(size)</div> <div>size</div> <div>int *p = malloc(size)の場合</div> <div>sizeof(int)</div> <div>p[0]p[1] ... p[5]</div> <div>double *q = malloc(size)の場合</div> <div>sizeof(double)</div> <div>q[0] q[1] q[2]</div> <div>17 / 45</div>	<div>malloc 関数の戻り値の解釈：配列</div> <div>要素数 n 個の Type 型データの配列のサイズ： size = n × sizeof(Type) バイト</div> <div>malloc(size)</div> <div>size</div> <div>Type *p = malloc(size)の場合</div> <div>sizeof(Type)</div> <div>p[0]p[1] ... p[n-1]</div> <div>18 / 45</div>

<p>malloc 関数の返り値の解釈：1 つのオブジェクト</p>	<p>malloc コードパターン：間違い編 <span>★重要★</span></p>	<p>malloc コードパターン：これ覚えておけば OK <span>★重要★</span></p>
<p>1 つの <i>Type</i> 型データのサイズ： size = sizeof(<i>Type</i>) バイト</p>  <p>19 / 45</p>	<p>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</p> <p><code>double *p = malloc(6*sizeof(int))</code></p>  <p>でも p[5] = 1.0; を実行した時にきつと変なことが起きる</p> <ul style="list-style-type: none"><li>■ sizeof(int) と sizeof(double) が違う時</li></ul> <p>20 / 45</p>	<p>パターン</p> <div><pre>データ型 *変数 = malloc( 個数 * sizeof(データ型)); if (変数 == NULL) {     エラー処理; }</pre></div> <p>具体例</p> <div><pre>int *a = malloc(n * sizeof(int)); if (a == NULL) {     printf("NO MEMORY\n");     exit(1); }</pre></div> <p>21 / 45</p>
<p>ライブラリ関数 free</p> <p>22 / 45</p>	<p>free 関数 <span>★重要★</span></p> <pre>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 &lt;stdlib.h&gt; 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 <pre>int *p = malloc(n * sizeof(int)); /*割当*/ ... 略 ... free(p); /*解放*/</pre></pre> <p>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</p> <ul style="list-style-type: none"><li>■ 解放後のメモリは再利用されている場合があるため</li></ul> <p>23 / 45</p>	<p>コードパターン (1)：main 関数にて</p> <div><pre>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; }</pre></div> <p>機能単位に分解してみた</p> <ul style="list-style-type: none"><li>■ データ個数 <i>n</i> を得る</li><li>■ 作業用のメモリを割り当てる (malloc)</li><li>■ データを読み込んで、計算し、結果を出力</li><li>■ メモリを解放 (free)</li></ul> <p>24 / 45</p>

コードパターン (2) : 繰り返し	メモリリーク (memory leak) とは <span>★重要★</span>	メモリリーク (もうひとつ) : 関数内ローカルで <span>★重要★</span>
<div data-bbox="159 403 736 520"><pre>for (;;) {     int n = ...;     int *p = malloc(n * sizeof(int)); /*割当*/     配列 p を使った処理;     free(p); /*解放*/ }</pre></div> <p>処理の繰り返し (int 型データの配列を想定した場合)</p> <ul style="list-style-type: none"><li>■ int データの個数 n を得る</li><li>■ データのメモリサイズは n * sizeof(int) に注意</li><li>■ 作業用のメモリを割り当てる (malloc)</li><li>■ 実際に処理をする</li><li>■ メモリを解放 (free) して次の処理へ</li></ul> <div data-bbox="714 775 768 794">25 / 45</div>	<div data-bbox="831 392 1408 491"><pre>for (;;) {     int n = ...;     int *p = malloc(n * sizeof(int)); /*割当*/     配列 p を使った処理; }</pre></div> <p>ここでやっていること</p> <ul style="list-style-type: none"><li>■ 作業用のメモリを割り当てる</li><li>■ メモリを解放せずに次の処理を繰り返す</li></ul> <p>やがてメモリを全て使い果たす</p> <ul style="list-style-type: none"><li>■ メモリ不足により malloc がメモリ割り当てに失敗</li><li>■ プログラムが強制終了されてしまう</li></ul> <p>開放されないメモリがどんどん増えてゆく</p> <ul style="list-style-type: none"><li>■ <b>メモリリークと呼んでいる (バグの一種)</b></li></ul> <div data-bbox="1386 775 1440 794">26 / 45</div>	<p>関数 foo 内で局所変数に割当ポインタを代入</p> <div data-bbox="1503 416 2080 515"><pre>int foo(int n) {     int *p = malloc(n * sizeof(int)); /*割当*/     配列 p を使った処理;     return ...; }</pre></div> <p>関数から戻るときにメモリリーク発生</p> <p>このように書きましょう</p> <div data-bbox="1503 596 2080 713"><pre>int foo(int n) {     int *p = malloc(n * sizeof(int)); /*割当*/     配列 p を使った処理;     free(p);     return ...; }</pre></div> <div data-bbox="2058 775 2112 794">27 / 45</div>
<p>Q. これメモリリークしないですか?</p> <div data-bbox="159 903 736 1032"><pre>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; }</pre></div> <p>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</p> <ul style="list-style-type: none"><li>■ そのような仕組みになっているのが普通の OS</li><li>■ 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)</li></ul> <div data-bbox="714 1283 768 1302">28 / 45</div>	<p>(独自) プログラム例</p> <div data-bbox="1386 1283 1440 1302">29 / 45</div>	<p>Sum and Average</p> <div data-bbox="2058 1283 2112 1302">30 / 45</div>

Sum and Average : 問題設定	実行例	実装の方針 (機能分割の方針)
<p>n 個の実数値データの総和と平均値を求める</p> <p>要求仕様</p> <ul style="list-style-type: none"> <li>■ データは実数値</li> <li>■ データ数 n はキーボードより入力</li> <li>■ データを記憶する配列は malloc で動的に割り当てる</li> <li>■ n 個のデータは順次キーボードより入力</li> <li>■ 総和と平均値を計算して表示</li> </ul> <p>プログラム名 : avr</p>	<pre>\$ ./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</pre> <ol style="list-style-type: none"> <li>1 データ数は 3 と入力</li> <li>2 3 つの実数データを順次入力</li> <li>3 入力データに対する総和と平均値を計算・表示</li> </ol>	<p>ソースファイル名 : avr.c</p> <p>データは実数値 → double 型のデータとする</p> <p>データ数 n はキーボードより入力 → 関数 int get_n(void) で実現</p> <p>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</p> <p>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</p> <p>総和と平均値を計算して表示 → 関数 void compute(int n, double *x) で実現</p>
実装 : ソースファイル全体像	実装 : main 関数	実装 : 関数 get_n
<pre>#include &lt;stdio.h&gt; #include &lt;stdlib.h&gt;  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... }</pre>	<pre>#include &lt;stdio.h&gt; #include &lt;stdlib.h&gt;  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; }</pre>	<pre>int get_n(void) {     int n = 0;     while (n &lt;= 0) {         char s[1024];         printf("データ数 n : ");         if (fgets(s, sizeof(s), stdin) == NULL) {             exit(0);         }         n = atoi(s);     }     return n; }</pre> <p>scanf を使わず fgets 使用してみた例</p>

実装：関数 alloc_array	実装：関数 read_data	実装：関数 compute
<pre data-bbox="159 403 734 569">double *alloc_array(int n) {     double *x = NULL;     x = malloc(n * sizeof(double));     if (x == NULL) {         printf("NO MEMORY\n");         exit(1);     }     return x; }</pre> <div data-bbox="712 772 768 793">37 / 45</div>	<pre data-bbox="831 403 1406 585">void read_data(int n, double *x) {     for (int i = 0; i &lt; 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", &amp;x[i]);     } }</pre> <div data-bbox="1384 772 1440 793">38 / 45</div>	<pre data-bbox="1503 403 2078 585">void compute(int n, double *x) {     double sum = 0;     double avr = 0;     for (int i = 0; i &lt; n; i++) {         sum = sum + x[i];     }     avr = sum / n;     printf("総和=%f\n", sum);     printf("平均=%f\n", avr); }</pre> <div data-bbox="2056 772 2112 793">39 / 45</div>
	個人の情報を表す構造体 struct student	割当
<p data-bbox="241 1034 367 1054">構造体の配列</p> <div data-bbox="712 1278 768 1299">40 / 45</div>	<pre data-bbox="831 912 1406 1026">struct student {     char    name[64]; /*名前*/     int     height;  /*身長*/     float   weight;  /*体重*/     long    schols;  /*奨学金*/ };</pre> <p data-bbox="835 1038 1088 1059">構造体の回を思い出しましょう...</p> <p data-bbox="835 1080 1088 1101">任意の人数の情報を取り扱いたい</p> <ul data-bbox="835 1107 1075 1128" style="list-style-type: none"> <li>■ 実行時に人数が指定される</li> </ul> <p data-bbox="835 1134 1180 1155">→ この構造体の配列を動的メモリ割当てで実現</p> <div data-bbox="1384 1278 1440 1299">41 / 45</div>	<p data-bbox="1507 903 1868 924">構造体 student の n 個ぶんのメモリを割当てる</p> <pre data-bbox="1503 936 2078 1042">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); }</pre> <p data-bbox="1507 1062 1686 1083">i 番目の要素にアクセス</p> <pre data-bbox="1503 1096 2078 1137">p[i].height = 162; p[i].weight = 58;</pre> <div data-bbox="2056 1278 2112 1299">42 / 45</div>

## おわり

43 / 45

### 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"

44 / 45

### 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

45 / 45