Programming basics
(GKNB_INTA023)

Hatwagner F. Miklós, PhD.

Széchenyi István University, Győr, Hungary

https://github.com/sze-info/ProgrammingBasics

August 18, 2021

Sorting numbers

Task:

- Improve the existing bubble sort program! Let the user enter the numbers to be sorted! Finish reading the input by entering a negative number.
- Entering more numbers than the size of the array must be prevented.

Problems:

- The count of numbers should be known at compile time
- Undersized array $\rightarrow$ there will be no space for the data
- Oversized array $\rightarrow$ wasting the memory
- The oversized array causes the smaller problem.

Sorting numbers

Output1

```
Enter non-negative numbers
Number #1: 2
Number #2: 4
Number #3: 1
Number #4: 3
Number #5: -1
After sorting:
1       2       3       4
```

Output2

```
Enter non-negative numbers
Number #1: 5
Number #2: 4
Number #3: 3
Number #4: 2
Number #5: 1
After sorting:
1       2       3       4       5
```

Sorting numbers

bubble5.c

```
3  #define MAX 5

37 int main(void) {
38   int used; // Number of used array elements
39   int numbers[MAX];
40   printf("Enter non-negative numbers\n");
41   used = read(numbers);
42   bubble(numbers, used);
43   printf("After sorting:\n");
44   printArray(numbers, used);
45   return 0;
46 }
```

Sorting numbers

bubble5.c

```
5  int read(int* numbers) {
6    int current, used = 0;
7    do {
8      printf("Number #%d: ", used + 1);
9      scanf("%d", &current);
10     if(current>=0 and used<MAX) {
11       *(numbers + used) = current;
12       used++;
13     }
14   } while(current>=0 and used<MAX);
15   return used;
16 }
```

Dynamic memory allocation

- The programmer decides the lifetime of dynamic variables
- `stdlib.h` must be included
- Memory allocation:
  - `void *malloc(size_t size);`
    Allocates `size` bytes of memory and returns its address. The allocated area is *uninitialized*.
  - `void *calloc(size_t nmemb, size_t size);`
    Allocates and returns the address of a continuous memory area for an array containing `nmemb` elements, each of which requires `size` bytes of memory. The area is *initialized to zeros*.
  - `void *realloc(void *ptr, size_t size);`
    Resizing the already allocated memory area without modifying the stored content.
- The return value is NULL in case of an error $\rightarrow$ should be checked
- Freeing memory: `void free(void *ptr);`
- The same memory area cannot be freed several times
- Freeing NULL does not cause problems

Sorting numbers

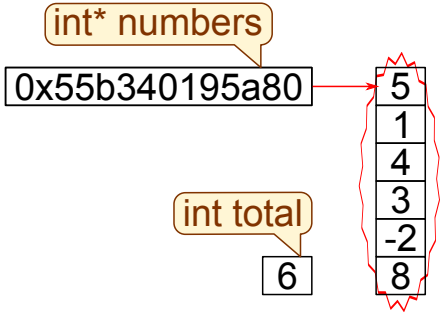
Tasks:

- Allocate memory dynamically for the array containing the numbers to be sorted
- Enter the count of numbers first, then allocate the required amount of memory and read the numbers
- Do not forget to free the allocated area as soon as possible

Sorting numbers

bubble6.c

```
34 int main(void) {
35   int total; // array size
36   int* numbers = read(&total);
37   bubble(numbers, total);
38   printf("After sorting:\n");
39   printArray(numbers, total);
40   free(numbers);
41   return 0;
42 }
```



Sorting numbers

bubble6.c

```
4  int* read(int* total) {
5    printf("How many numbers do you want to sort? ");
6    scanf("%d", total);
7    int* numbers = (int*)malloc(*total * sizeof(int));
8    for(int i=0; i<*total; i++) {
9      printf("Number #%d: ", i + 1);
10     scanf("%d", numbers + i); // &numbers[i]
11   }
12   return numbers;
13 }
```

Student register

Tasks:

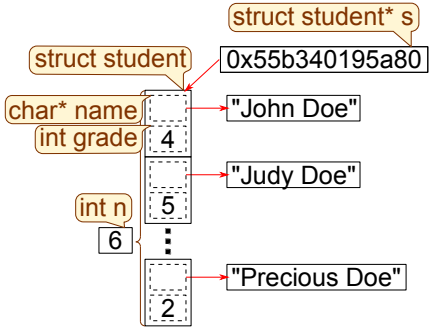
- Record the names and grades of students (one grade per student)
- Read the number of students first, then allocate the required amount of memory to store an array of name-grade structures
- Allocate memory dynamically even for the names!
- Sort the list according to the names and display it

Student register

students1.c

```
6  struct student {
7    char* name;
8    int grade;
9  };

72 int main(void) {
73   int n;
74   struct student *s = read(&n);
75   if(s) {
76     bubble(s, n);
77     print(s, n);
78     freeMem(s, n);
79   }
80   return 0;
81 }
```



Student register

students1.c

```
27 #define MAX 128
28 struct student* read(int* n) {
29   printf("How many students are on the course? ");
30   scanf("%d%*c", n); // delete the new line character from the input buffer
31   struct student* s = (struct student*)malloc(
32     *n * sizeof(struct student));
33   if(!s) return NULL;
34   for(int i=0; i<*n; i++) {
35     printf("Name of student #%d: ", i+1);
36     char name[MAX];
37     int length = sze_getline(name, MAX-1);
38     s[i].name = (char*)malloc(length + 1);
39     if(!s[i].name) { freeMem(s, i-1); return NULL; }
40     strcpy(s[i].name, name);
41     printf("Grade: ");
42     scanf("%d%*c", &s[i].grade);
43   }
44   return s;
45 }
```

Student register

students1.c

```
47 void bubble(struct student* s, int n) {
48   for(int e=n-1; e>=1; e--) {
49     for(int b=0; b<e; b++) {
50       if(strcmp(s[b].name, s[b+1].name) > 0) {
51         struct student swap = s[b];
52         s[b] = s[b+1];
53         s[b+1] = swap;
54       }
55     }
56   }
57 }
```

Student register

students1.c

```
59 void print(struct student* s, int n) {
60   int longest = 0;
61   for(int i=0; i<n; i++) {
62     int length = strlen(s[i].name);
63     if(length > longest) longest = length;
64   }
65   for(int i=0; i<n; i++) {
66   // Column width specified at runtime
67     printf("%*s %d\n",
68            longest, s[i].name, s[i].grade);
69   }
70 }
```

```
    Baby Doe 3
   James Doe 4
    Jane Doe 4
    John Doe 4
    Judy Doe 5
Precious Doe 2
```

Student register

students1.c

```
20 void freeMem(struct student* s, int n) {
21   for(int i=0; i<n; i++) {
22     free(s[i].name);
23   }
24   free(s);
25 }
```

Sorting numbers

Task:

- Make the usage of our earlier program more comfortable!
- Instead of entering the quantity of numbers in advance, let a special value signal the end of input $\rightarrow$ negative value

Problem:

How much memory should we allocate if we do not know the number of numbers?

Solution #1:

Allocate a small memory block initially and as soon as it is full double its size $\rightarrow$ minimizes the number of memory re-allocations (=fast) at the cost of at most half of the allocated area remains unused

Remark:

Due to simplicity and brevity we assume that memory allocations are always successful

Sorting numbers

bubble7.c

```
45 int main(void) {
46   int used; // Number of currently used array elements
47   int* numbers; // Address of the array
48   printf("Enter non-negative numbers\n");
49   numbers = read(&used);
50   bubble(numbers, used);
51   printf("After sorting:\n");
52   printArray(numbers, used);
53   free(numbers);
54   return 0;
55 }
```

Sorting numbers

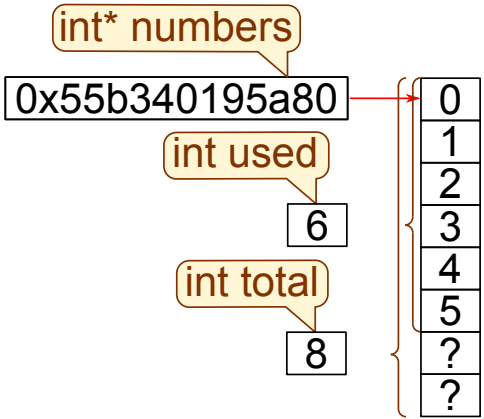
bubble7.c

```
4  int* read(int* used) {
5    int current, total = 1;
6    printf("\t[Initial memory allocation]\n");
7    int* numbers = (int*)malloc(total * sizeof(int));
8    *used = 0;
9    do {
10     printf("\t[Used: %d, Array size: %d]\n", *used, total);
11     printf("Number #%d: ", *used + 1);
12     scanf("%d", &current);
13     if(current >= 0) {
14       if(*used == total) {
15         printf("\t[Memory re-allocation]\n");
16         total *= 2;
17         numbers = (int*)realloc(numbers, total*sizeof(int));
18       }
19       numbers[*used] = current;
20       (*used)++;
21     }
22   } while(current >= 0);
23   return numbers;
24 }
```

Sorting numbers

Output

```
Enter non-negative numbers
      [Initial memory allocation]
      [Used: 0, Array size: 1]
Number #1: 6
      [Used: 1, Array size: 1]
Number #2: 5
      [Memory re-allocation]
      [Used: 2, Array size: 2]
Number #3: 4
      [Memory re-allocation]
      [Used: 3, Array size: 4]
Number #4: 3
      [Used: 4, Array size: 4]
Number #5: 2
      [Memory re-allocation]
      [Used: 5, Array size: 8]
Number #6: 1
      [Used: 6, Array size: 8]
Number #7: 0
      [Used: 7, Array size: 8]
Number #8: -1
After sorting:
0       1       2       3       4       5       6
```



Drawing rectangles

Task:

- Modify the rectangle drawing program similarly, too
- Memory allocation strategy #2: if the allocated area is full, increase its size always with the same amount → minimizes the unused memory area at the cost of much re-allocations (=slow)
- The function allocating memory for the array returns the number of elements and writes the address of the array where its parameter points to

rectangle3.c

```
91 int main(void) {
92   struct rectangle* ar; int used;
93   printf("Please enter the data of rectangles!\n");
94   used = readAll(&ar);
95   draw(ar, used);
96   free(ar);
97   return 0;
98 }
```

Drawing rectangles

rectangle3.c

```
60 int readAll(struct rectangle** ar) {
61   int used=0, total = 2, tlx;
62   bool goon;
63   printf("\t[Initial memory allocation]\n");
64   *ar = (struct rectangle*)malloc(
65     total * sizeof(struct rectangle));
66   do {
67     printf("\t[Used: %d, Array size: %d]\n", used, total);
68     goon = readTLX(used+1, MINX, MAXX-1, &tlx);
69     if(goon) {
70       if(used == total) {
71         printf("\t[Memory re-allocation]\n");
72         total += 2;
73         *ar = (struct rectangle*)realloc(
74           *ar, total*sizeof(struct rectangle));
75       }
```

Drawing rectangles

rectangle3.c

```
76        (*ar)[used].tl.x = tlx;
77        (*ar)[used].tl.y = read(used+1,
78          "Y coordinate of the top left corner", MINY, MAXY-1);
79        (*ar)[used].br.x = read(used+1,
80          "X coordinate of the bottom right corner",
81          (*ar)[used].tl.x+1, MAXX);
82        (*ar)[used].br.y = read(used+1,
83          "Y coordinate of the bottom right corner",
84          (*ar)[used].tl.y+1, MAXY);
85        printf("Drawing character of rectangle #%d: ", used+1);
86        scanf(" %c", &(*ar)[used].c);
87        used++;
88      }
89    } while(goon);
90    return used;
91  }
```

Drawing rectangles

Output 1/2

```
Please enter the data of rectangles!
        [Initial memory allocation]
        [Used: 0, Array size: 2]
X coordinate of the top left corner of rectangle #1 [0, 78] (exits to a negative value) 3
Y coordinate of the top left corner rectangle #1 [0, 23] 3
X coordinate of the bottom right corner rectangle #1 [4, 79] 6
Y coordinate of the bottom right corner rectangle #1 [4, 24] 6
Drawing character of rectangle #1: +
        [Used: 1, Array size: 2]
X coordinate of the top left corner of rectangle #2 [0, 78] (exits to a negative value) 5
Y coordinate of the top left corner rectangle #2 [0, 23] 5
X coordinate of the bottom right corner rectangle #2 [6, 79] 8
Y coordinate of the bottom right corner rectangle #2 [6, 24] 8
Drawing character of rectangle #2: -
        [Used: 2, Array size: 2]
X coordinate of the top left corner of rectangle #3 [0, 78] (exits to a negative value) 7
        [Memory re-allocation]
Y coordinate of the top left corner rectangle #3 [0, 23] 7
X coordinate of the bottom right corner rectangle #3 [8, 79] 10
Y coordinate of the bottom right corner rectangle #3 [8, 24] 10
Drawing character of rectangle #3: *
        [Used: 3, Array size: 4]
X coordinate of the top left corner of rectangle #4 [0, 78] (exits to a negative value) -1
```

Output 2/2

```
   ++++
   ++++
   ++----
   ++----
     --****
     --****
       ****
       ****
```

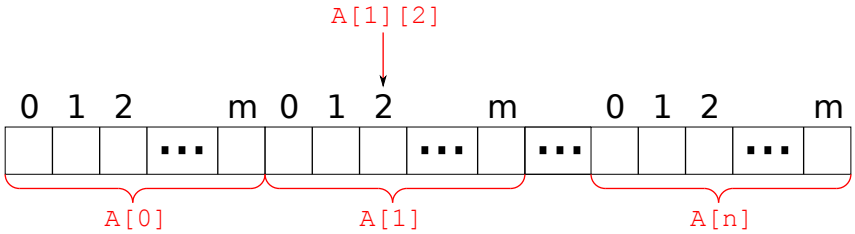
Matrices

Matrix: two dimensional array of data of the same type

Only one dimensional arrays exist in C, but these can be embedded into each other → matrix = vector of vectors

$$A = \begin{bmatrix} 11 & 12 & 13 & 14 \\ 21 & 22 & 23 & 24 \\ 31 & 32 & 33 & 34 \end{bmatrix}$$

```
int A[3][4] = {
    {11, 12, 13, 14},
    {21, 22, 23, 24},
    {31, 32, 33, 34} };
```



Adding up two matrices: $(A + B)[i, j] = A[i, j] + B[i, j]$, where A and B are $n \times m$ sized matrices.

mtxAdd1.c

```
4  #define ROWS 3
5  #define COLS 4
6
7  int main(void) {
8    // declaration, initialization
9    int ma[ROWS][COLS] = {
10     { 11, 12, 13, 14 },
11     { 21, 22, 23, 24 },
12     { 31, 32, 33, 34 }
13   };
14   int mb[ROWS][COLS], mc[ROWS][COLS];
15   srand(time(NULL));
16   for(int r=0; r<ROWS; r++) { // filling the mtx.
17     for(int c=0; c<COLS; c++) {
18       mb[r][c] = 10 + rand()%40;
19     }
20   }
```

Matrices

mtxAdd1.c

```
21    for(int r=0; r<ROWS; r++) { // add up mtxs.
22      for(int c=0; c<COLS; c++) {
23        mc[r][c] = ma[r][c] + mb[r][c];
24      }
25    }
26    for(int r=0; r<ROWS; r++) { // printing
27      for(int c=0; c<COLS; c++) {
28        printf("%d ", ma[r][c]);
29      }
30      printf("%c ", r==ROWS/2?'+':' ');
31      for(int c=0; c<COLS; c++) {
32        printf("%d ", mb[r][c]);
33      }
34      printf("%c ", r==ROWS/2?'=':' ');
35      for(int c=0; c<COLS; c++) {
36        printf("%d ", mc[r][c]);
37      }
38      putchar('\n');
39    }
40    return 0; }
```

Matrices

(A possible) output

```
11 12 13 14   33 49 36 12   44 61 49 26
21 22 23 24 + 20 45 24 18 = 41 67 47 42
31 32 33 34   19 10 11 42   50 42 44 76
```

How can a matrix be passed to a function?

OK ✓

```
void fn(int m[ROWS][COLS]) { //...
void fn(int m[][COLS]) { //...
void fn(int (*m)[COLS]) { //...
```

Error X – It is an array of pointers, not a matrix!

```
void fn(int *m[COLS]) { //...
```

mtxAdd2.c

```
44 int main(void) {
45   int ma[ROWS][COLS], mb[ROWS][COLS],
46       mc[ROWS][COLS];
47   srand(time(NULL));
48   generate(ma);
49   generate(mb);
50   add((const int(*)[COLS])ma,
51       (const int(*)[COLS])mb, mc);
52   print((const int(*)[COLS])ma,
53         (const int(*)[COLS])mb,
54         (const int(*)[COLS])mc);
55   return 0;
56 }
```

Matrices

mtxAdd2.c

```
4  #define ROWS 3
5  #define COLS 4
6
7  void generate(int m[][COLS]) {
8    for(int r=0; r<ROWS; r++) {
9      for(int c=0; c<COLS; c++) {
10       m[r][c] = 10 + rand()%40;
11     }
12   }
13 }
14
15 void add(const int (*ma)[COLS],
16          const int (*mb)[COLS],
17                int (*mc)[COLS]) {
18   for(int r=0; r<ROWS; r++) {
19     for(int c=0; c<COLS; c++) {
20       mc[r][c] = *(ma[r] + c) + *(*(mb+r) + c);
21     }
22   }
23 }
```

mtxAdd2.c

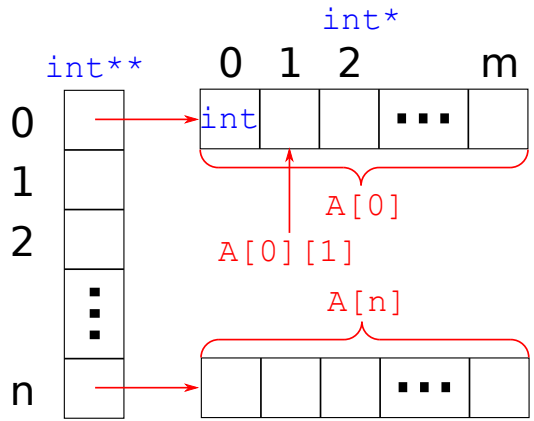
```
25 void print(const int ma[][COLS],
26            const int mb[][COLS],
27            const int mc[][COLS]) {
28   for(int r=0; r<ROWS; r++) {
29     for(int c=0; c<COLS; c++) {
30       printf("%d ", ma[r][c]);
31     }
32     printf("%c ", r==ROWS/2?'+':' ');
33     for(int c=0; c<COLS; c++) {
34       printf("%d ", mb[r][c]);
35     }
36     printf("%c ", r==ROWS/2?'=':' ');
37     for(int c=0; c<COLS; c++) {
38       printf("%d ", mc[r][c]);
39     }
40     putchar('\n');
41   }
42 }
```

Problem:

inflexible functions, the number of columns is fixed

Solution:

- create vectors dynamically (can be addressed by eg. `int*`), then
- store their addresses in another dynamic vector (`int**`, array of pointers)!



mtxAdd3.c

```
55 int main(void) {
56   srand(time(NULL));
57   int rows = 1 + rand()%4;
58   int cols = 1 + rand()%4;
59   int** a = allocate(rows, cols);
60   int** b = allocate(rows, cols);
61   int** c = allocate(rows, cols);
62   generate(a, rows, cols);
63   generate(b, rows, cols);
64   add(a, b, c, rows, cols);
65   print(a, b, c, rows, cols);
66   freeMem(a, rows);
67   freeMem(b, rows);
68   freeMem(c, rows);
69   return 0;
70 }
```

mtxAdd3.c

```
5  int** allocate(int rows, int cols) {
6    int** m = (int**)malloc(rows * sizeof(int*));
7    for(int r=0; r<rows; r++) {
8      m[r] = (int*)malloc(cols * sizeof(int));
9    }
10   return m;
11 }
12
13 void generate(int** m, int rows, int cols) {
14   for(int r=0; r<rows; r++) {
15     for(int c=0; c<cols; c++) {
16       m[r][c] = 10 + rand()%40;
17     }
18   }
19 }
```

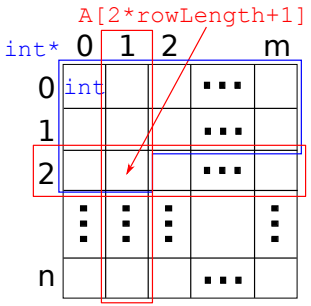
mtxAdd3.c

```
21 void add(int** ma, int** mb, int** mc,
22          int rows, int cols) {
23   for(int r=0; r<rows; r++) {
24     for(int c=0; c<cols; c++) {
25       mc[r][c] = *(ma[r] + c) + *(*(mb+r) + c);
26     }
27   }
28 }

49 void freeMem(int** m, int rows) {
50   for(int r=0; r<rows; r++) {
51     free(m[r]);
52   }
53   free(m);
54 }
```

Alternative solution:

- we can mimic the stucture of „static” arrays in memory, thus
- we allocate memory for a vector and map the elements of the matrix to this area



mtxAdd4.c

```
44 int main(void) {
45   srand(time(NULL));
46   int rows = 1 + rand()%4;
47   int cols = 1 + rand()%4;
48   int* ma = allocate(rows, cols);
49   int* mb = allocate(rows, cols);
50   int* mc = allocate(rows, cols);
51   generate(ma, rows, cols);
52   generate(mb, rows, cols);
53   add(ma, mb, mc, rows, cols);
54   print(ma, mb, mc, rows, cols);
55   free(ma);
56   free(mb);
57   free(mc);
58   return 0;
59 }
```

mtxAdd4.c

```
5  int* allocate(int rows, int cols) {
6    return (int*)malloc(rows * cols * sizeof(int));
7  }
8
9  void generate(int* m, int rows, int cols) {
10   for(int r=0; r<rows; r++) {
11     for(int c=0; c<cols; c++) {
12       m[r*cols + c] = 10 + rand()%40;
13     }
14   }
15 }
16
17 void add(int* ma, int* mb, int* mc,
18          int rows, int cols) {
19   for(int r=0; r<rows; r++) {
20     for(int c=0; c<cols; c++) {
21       mc[r*cols+c] = ma[r*cols+c] + mb[r*cols+c];
22     }
23   }
24 }
```