Programming basics
(GKNB_INTA023)

Hatwagner F. Miklós, PhD.

Széchenyi István University, Győr, Hungary

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

November 21, 2020

Macros with arguments

Problem:

If a function is very simple, the *time of invocation* is comparable with the *time of operation* $\rightarrow$ ineffectice, slow programs

Solution:

Change function calls to macro substitution! $\rightarrow$ text replacement with the preprocessor

Macros with arguments

operation1.c

```c
#include <stdio.h>

int square(int x) {
  return x * x;
}

int main(void) {
  printf("%d^2=%d\n", 3, square(3));      // 3*3 = 9
  printf("%d^2=%d\n", 3+1, square(3+1)); // 4*4 = 16
  return 0;
}
```

Task: substitute all function calls with macros!

`#define` *name(identifier-list) preprocessor-symbols newline*

Macros with arguments

Caution!
It is forbidden to put a whitespace character between SQUARE and (! → changes the meaning to simple macro substitution, without arguments

operation2.c

```
1  #include <stdio.h>
2
3  #define SQUARE(x) x*x
4
5  int main(void) {
6     printf("%d^2=%d\n", 3, SQUARE(3));      // 3*3 = 9
7     printf("%d^2=%d\n", 3+1, SQUARE(3+1)); // 3+1*3+1 = 7
8     return 0;
9  }
```

Problem: order of operations!
Solution: brackets

Macros with arguments

operation3.c

```c
#include <stdio.h>

#define SQUARE(x) (x)*(x)

int main(void) {
   printf("%d^2=%d\n", 3, SQUARE(3));      // (3)*(3) = 9
   printf("%d^2=%d\n", 3+1, SQUARE(3+1)); // (3+1)*(3+1)=16
   return 0;
}
```

Next task: write a macro for addition!

Macros with arguments

operation4.c

```
3  #define SQUARE(x) (x)*(x)
4  #define ADD(x,y) (x)+(y)
5
6  int add(int x, int y) {
7      return x + y;
8  }
9
10 int main(void) {
11    printf("%d^2=%d\n", 3, SQUARE(3));
12    printf("%d^2=%d\n", 3+1, SQUARE(3+1));
13    printf("1+2=%d\n", ADD(1, 2));          // (1)+(2)=3
14    printf("4*1+2=%d\n", 4*ADD(1, 2));      // 4*(1)+(2)=6
15    printf("4*(1+2)=%d\n", 4*add(1, 2));    // 4*(1+2)=12
16    printf("4*(1+2)=%d\n", 4*(ADD(1, 2)));  // 4*((1)+(2))=12
17    return 0;
18 }
```

Problem: actual parameters are evaluated before function invocation, but are used without any modifications in case of macros → different program behavior, misleading
Solution: further brackets

Macros with arguments

operation5.c

```
3  #define SQUARE(x) ((x)*(x))
4  #define ADD(x,y) ((x)+(y))
5
6  int add(int x, int y) {
7      return x + y;
8  }
9
10 int main(void) {
11    printf("%d^2=%d\n", 3, SQUARE(3));
12    printf("%d^2=%d\n", 3+1, SQUARE(3+1));
13    printf("1+2=%d\n", ADD(1, 2));          // ((1)+(2))=3
14    printf("4*(1+2)=%d\n", 4*ADD(1, 2)); // 4*((1)+(2))=12
15    printf("4*(1+2)=%d\n", 4*add(1, 2)); // 4*(1+2)=12
16    return 0;
17 }
```

Macros with arguments

Task: modify our earlier matrix addition program to calculate indexes with a macro!

mtxAdd5.c

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

Macros with arguments

Pros:

- Faster than a function call
- The code is less type-dependent

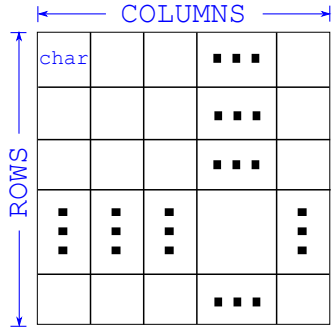
Cons:

- No type checks, hard to explore mistakes
- Code size increases, it is not suitable to replace complex functions
- The evaluation of passed expressions happens several times $\rightarrow$ time, side effects

Sorting names

Task:

- Read names and list them in alphabetical order!
- For the sake of simplicity, store the characters of names in a two-dimensional array



`char roster[ROWS][COLUMNS]`

Sorting names

roster1.c

```
4  #define ROWS 10
5  #define COLUMNS 30

47 int main(void) {
48    printf("Sorting names\n");
49    char roster[ROWS][COLUMNS];
50    int n = read(roster);
51    bubble(roster, n);
52    printf("Names in alphabetic order:\n");
53    print(roster, n);
54    return 0;
55 }
```

Sorting names

roster1.c

```
16 int read(char (*a)[COLUMNS]) {
17   int n;
18   printf("Number of names to be sorted (max. %d): ", ROWS);
19   scanf("%d%*c", &n);
20   if(n > ROWS) n = ROWS;
21   for(int i=0; i<n; i++) {
22     printf("Name #%d: ", i+1);
23     sze_getline(a[i], COLUMNS-1);
24   }
25   return n;
26 }
```

Sorting names

roster1.c

```
28 void bubble(char (*a)[COLUMNS], int n) {
29   for(int e=n-1; e>=1; e--) {
30     for(int b=0; b<e; b++) {
31       if(strcmp(a[b], a[b+1]) > 0) {
32         char swap[COLUMNS]; // moving whole strings
33         strcpy(swap, a[b]);
34         strcpy(a[b], a[b+1]);
35         strcpy(a[b+1], swap);
36       }
37     }
38   }
39 }
```

Sorting names

roster1.c

```
41 void print(char (*a)[COLUMNS], int n) {
42   for(int i=0; i<n; i++) {
43     printf("%s\n", a[i]);
44   }
45 }
```

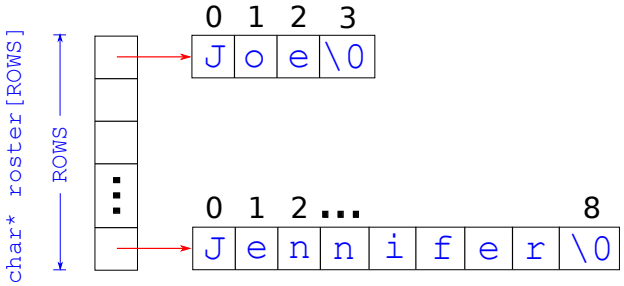
Sorting names

Problem:

The length of lines is the same → sometimes more memory would be needed, sometimes it is already too much

Task:

Modify the data structure, use an array of char pointers and allocate memory for the lines runtime!



Sorting names

roster2.c

```
5  #define ROWS 10

56 int main(void) {
57    printf("Sorting names\n");
58    char* roster[ROWS];
59    int n = read(roster);
60    bubble(roster, n);
61    printf("Names in alphabetic order:\n");
62    print(roster, n);
63    freeMem(roster, n);
64    return 0;
65 }
```

Sorting names

roster2.c

```
16 #define MAX 64
17 int read(char* a[ROWS]) {
18   int n;
19   char buf[MAX];
20   printf("Number of names to be sorted (max. %d): ", ROWS);
21   scanf("%d%*c", &n);
22   if(n > ROWS) n = ROWS;
23   for(int i=0; i<n; i++) {
24     printf("Name #%d: ", i+1);
25     int length = sze_getline(buf, MAX-1);
26     a[i] = (char*)malloc((length+1) * sizeof(char));
27     strcpy(a[i], buf);
28   }
29   return n;
30 }
```

Sorting names

roster2.c

```
32 void bubble(char* a[ROWS], int n) {
33   for(int e=n-1; e>=1; e--) {
34     for(int b=0; b<e; b++) {
35       if(strcmp(a[b], a[b+1]) > 0) {
36         char* swap = a[b]; // moving pointers only
37         a[b] = a[b+1];
38         a[b+1] = swap;
39       }
40     }
41   }
42 }

50 void freeMem(char* a[ROWS], int n) {
51   for(int i=0; i<n; i++) {
52     free(a[i]);
53   }
54 }
```

Sorting names

Problem:

The number of names is still limited

Task:

Allocate memory for the pointer array at runtime!

roster3.c

```
56 int main(void) {
57   printf("Sorting names\n");
58   char** roster; // Points to an array of pointers
59   int n = read(&roster);
60   bubble(roster, n);
61   printf("Names in alphabetic order:\n");
62   print(roster, n);
63   freeMem(roster, n);
64   return 0;
65 }
```

Sorting names

roster3.c

```
15 #define MAX 64
16 int read(char*** a) { // Points to a location where a pointer to an
17   int n;              // array of pointers is placed
18   char buf[MAX];
19   printf("Number of names to be sorted: ");
20   scanf("%d%*c", &n);
21   *a = (char**)malloc(n * sizeof(char*)); // Allocate memory for the
22   for(int i=0; i<n; i++) {                // array of pointers
23     printf("Name #%d: ", i+1);
24     int length = sze_getline(buf, MAX-1);
25     (*a)[i] = (char*)malloc((length+1) * sizeof(char));
26     strcpy((*a)[i], buf);
27   }
28   return n;
29 }
```

Sorting names

roster3.c

```
43 void print(char** a, int n) {
44   for(int i=0; i<n; i++) {
45     printf("%s\n", a[i]);
46   }
47 }
48
49 void freeMem(char** a, int n) {
50   for(int i=0; i<n; i++) {
51     free(a[i]);
52   }
53   free(a); // freeing the array
54 }
```

Distances of cities

Task:

- Reading the names of two cities
- Printing the distance of these cities
- Exit if the same city is entered twice

Solution:

- Storing the names of cities in a vector
- and the distances of cities in a matrix
- The same index can be used to access the name of the city and it's distance from another city

Distances of cities

Output

```
Determining the distances of cities
Exit by entering the same city twice
Departure from: Budapest
Arrival at: Eldorado
Non-existent city!
Gyor
Distance: 121km
Departure from: Gyor
Arrival at: Gyor
```

Distances of cities

cities1.c

```
48 int main(void) {
49   int from, to;
50   printf("Determining the distances of cities\n"
51          "Exit by entering the same city twice\n");
52   do {
53     printf("Departure from: "); from = getCityIdx();
54     printf("Arrival at: "); to = getCityIdx();
55     if(from != to) {
56       printf("Distance: %dkm\n", getDistance(from, to));
57     }
58   } while(from != to);
59   return 0;
60 }
```

Distances of cities

cities1.c

```
15 #define CITIES 7
16 #define MAX 32
17 int getCityIdx() {
18   char* cityList[CITIES] = {
19     "Budapest", "Gyor", "Szeged",
20     "Debrecen", "Veszprem",
21     "Dunaujvaros", "Eger"
22   };
23   char cityName[MAX+1];
24   do {
25     sze_getline(cityName, MAX);
26     for(int i=0; i<CITIES; i++) {
27       if(not strcmp(cityList[i], cityName)) {
28         return i;
29       }
30     }
31     printf("Non-existent city!\n");
32   } while(true);
33 }
```

Distances of cities

cities1.c

```
35 int getDistance(int from, int to) {
36   int distanceMtx[CITIES][CITIES] = {
37     {   0, 121, 174, 231, 115,  83, 139 },
38     { 121,   0, 287, 377,  82, 176, 285 },
39     { 174, 287,   0, 218, 278, 161, 298 },
40     { 231, 377, 218,   0, 368, 320, 131 },
41     { 115,  82, 278, 368,   0, 103, 275 },
42     {  83, 176, 161, 320, 103,   0, 228 },
43     { 139, 285, 298, 131, 275, 228,   0 }
44   };
45   return distanceMtx[from][to];
46 }
```

"Triangular" matrices

Remark: more than the half of the data in the matrix can be omitted ($m_{i,j} = m_{j,i}, m_{i,i} = 0$)

	Budapest	Győr	Szeged	Debrecen	Veszprém	Dunaújváros	Eger
Budapest	0	121	174	231	115	83	139
Győr	121	0	287	377	82	176	285
Szeged	174	287	0	218	278	161	298
Debrecen	231	377	218	0	368	320	131
Veszprém	115	82	278	368	0	103	275
Dunaújváros	83	176	161	320	103	0	228
Eger	139	285	298	131	275	228	0

Problem: all rows of a matrix have the same length

Solution: creating a vector addressing vectors of different lengths (array of pointers, $\approx$ lower triangular matrix)

“Triangular” matrices

cities2.c

```
24 int getDistance(int from, int to) {
25   if(from == to) return 0;
26   int a[] = { 121 };
27   int b[] = { 174, 287 };
28   int c[] = { 231, 377, 218 };
29   int d[] = { 115,  82, 278, 368 };
30   int e[] = {  83, 176, 161, 320, 103 };
31   int f[] = { 139, 285, 298, 131, 275, 228 };
32   int* distanceMtx[CITIES-1] = { a, b, c, d, e, f };
33   if(from < to) {
34     int swap = from;
35     from = to;
36     to = swap;
37   }
38   return distanceMtx[from-1][to]; // -1 -> no need to store the 1st row
39 }
```

Alternative solution

Remark: even the pointers to vectors can be saved if the values under the main diagonal are stored continuously in a single vector

	Bp. [0]	Győr [1]	Szeged [2]	Debr. [3]	Veszp. [4]	Duv. [5]
Győr [1]	121					
Szeged [2]	174	287				
Debrecen [3]	231	377	218			
Veszprém [4]	115	82	278	368		
Dunaújváros [5]	83	176	161	320	103	
Eger [6]	139	285	298	131	275	228

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
121	174	287	231	377	218	115	82	278	...

The number of distances in rows above the searched data forms an arithmetical progression $\rightarrow S_n = \frac{[2a_1+(n-1)d]\cdot n}{2}$, specifically $a_1 = 1, d = 1 \rightarrow S_n = \frac{(n+1)\cdot n}{2}$

Alternative solution

Watch out! The indexing of rows in the program starts with 0, not with 1!

cities3.c

```
35 int getDistance(int from, int to) {
36   if(from == to) return 0;
37   int distanceMtx[(CITIES*(CITIES-1))/2] = {
38     121,
39     174, 287,
40     231, 377, 218,
41     115,  82, 278, 368,
42      83, 176, 161, 320, 103,
43     139, 285, 298, 131, 275, 228
44   };
45   if(from < to) {
46     int swap = from;
47     from = to;
48     to = swap;
49   }
50   return distanceMtx[from*(from-1)/2 + to];
51 }
```

Static variables

Problem:

- local variables have *automatic* lifetime (`auto`): from the moment of definition to the moment when execution leaves their block
- re-allocation of vectors and matrices are time consuming

Solution: usage of the `static` modifier

- lifetime: from the start of the program to its end $\rightarrow$ keep their values between function calls
- scope: unchanged (from declaration to the end of the block)
- initialized implicitely (filling with zero bits)

Static variables

cities4.c

```
35 int getDistance(int from, int to) {
36   if(from == to) return 0;
37   static int distanceMtx[(CITIES*(CITIES-1))/2] = {
38     121,
39     174, 287,
40     231, 377, 218,
41     115,  82, 278, 368,
42      83, 176, 161, 320, 103,
43     139, 285, 298, 131, 275, 228
44   };
45   if(from < to) {
46     int swap = from;
47     from = to;
48     to = swap;
49   }
50   return distanceMtx[from*(from-1)/2 + to];
51 }
```

Dynamic "Triangle matrices"

Task:

- Create the vector and the triangle matrix at runtime
- Fill them with values given by the user

cities5.c – main

```
74  int main(void) {
75    char** cityList;
76    int numCities;
77    int** cityDistances;
78    int from, to;
79    printf("Determining the distances of cities\n"
80           "Number of cities: ");
81    scanf("%d%*c", &numCities);
82    cityList = readCities(numCities);
83    printf("Enter distances between cities\n");
84    cityDistances = readDistances(cityList, numCities);
85    printf("Exit by entering the same city twice.\n");
```

Dynamic "Triangle matrices"

cities5.c – main

```
86    do {
87      printf("Departure from: ");
88      from = getCityIdx(cityList, numCities);
89      printf("Arrival at: ");
90      to = getCityIdx(cityList, numCities);
91      if(from != to) {
92        printf("Distance: %dkm\n",
93          getDistance(cityDistances, from, to));
94      }
95    } while(from != to);
96    freeMem(cityDistances, cityList, numCities);
97    return 0;
98  }
```

Dynamic "Triangle matrices"

cities5.c

```
16 #define MAX 32
17 char** readCities(int n) {
18   char** cityList = (char**)malloc(n*sizeof(char*));
19   for(int i=0; i<n; i++) {
20     printf("Name of city #%d: ", i+1);
21     char buf[MAX+1];
22     int length = sze_getline(buf, MAX);
23     cityList[i] = (char*)malloc(length+1);
24     strcpy(cityList[i], buf);
25   }
26   return cityList;
27 }
```

Dynamic "Triangle matrices"

cities5.c

```
41 int getCityIdx(char** cityList, int n) {
42   char cityName[MAX+1];
43   do {
44     sze_getline(cityName, MAX);
45     for(int i=0; i<n; i++) {
46       if(not strcmp(cityList[i], cityName)) {
47         return i;
48       }
49     }
50     printf("Non-existent city!\n");
51   } while(true);
52 }
```

Dynamic "Triangle matrices"

cities5.c

```
29 int** readDistances(char** cityList, int n) {
30   int** distanceMtx = (int**)malloc((n-1)*sizeof(int*));
31   for(int from=1; from<n; from++) {
32     distanceMtx[from-1] = (int*)malloc(from * sizeof(int));
33     for(int to=0; to<from; to++) {
34       printf("%s --> %s: ", cityList[from], cityList[to]);
35       scanf("%d%*c", &distanceMtx[from-1][to]);
36     }
37   }
38   return distanceMtx;
39 }
```

Dynamic "Triangle matrices"

cities5.c

```
54 int getDistance(int** distanceMtx, int from, int to) {
55   if(from == to) return 0;
56   if(from < to) {
57     int swap = from;
58     from = to;
59     to = swap;
60   }
61   return distanceMtx[from-1][to];
62 }
```

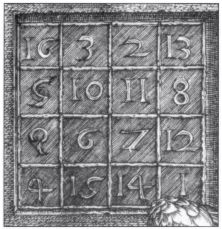
Dynamic "Triangle matrices"

cities5.c

```
64 void freeMem(int** distanceMtx, char** cityList, int n) {
65   for(int i=0; i<n-1; i++) {
66     free(distanceMtx[i]);
67     free(cityList[i]);
68   }
69   free(distanceMtx);
70   free(cityList[n-1]);
71   free(cityList);
72 }
```

Magic square

- A square matrix containing (generally between 1 and n^2) integers in which
  - the sum of all rows,
  - the sum of all columns,
  - the sum of numbers on the main diagonal and
  - the sum of numbers on the side diagonal are the same.
- Further curiosities



Albrecht Dürer: Melencolia I (detail)


Sagrada Família, Barcelona

Magic square

Constructing a magic square of odd order

1. Start in the central column of the first row with the number 1
2. Write increasing numbers in the matrix during the consecutive steps (2, 3, ..., n^2)!
3. The place of the next value is in general one step upwards and to the right
   - If the determined element is already filled the operation must be continued below the previously filled element
   - If the determined element lies outside the matrix move to the opposite side (eg. instead of the element "above the topmost one" use the element at the bottom).

Step 1		
	1	

Step 2		
	1	
		2

Step 3		
	1	
3		
		2

Step 4		
	1	
3		
4		2

Step 5		
	1	
3	5	
4		2

Step 6		
	1	6
3	5	
4		2

Step 7		
	1	6
3	5	7
4		2

Step 8		
8	1	6
3	5	7
4		2

Step 9		
8	1	6
3	5	7
4	9	2

Magic square

magic.c

```
43 int main(void) {
44   int size;
45   do {
46     printf("Size of the magic square: ");
47     scanf("%d", &size);
48   } while(size%2 == 0);
49   int** magic = generate(size);
50   print(magic, size);
51   freeMem(magic, size);
52   return 0;
53 }
```

Magic square

magic.c – generate

```
4  // It works only with square matrices of odd order!
5  int** generate(int size) {
6    // Allocating memory
7    int** mtx = (int**)malloc(size*sizeof(int*));
8    for(int r=0; r<size; r++) {
9      mtx[r] = (int*)calloc(size, sizeof(int));
10   }
```

Magic square

magic.c – generate

```
11    // Filling
12    int r=0, c=size/2;
13    for(int n=1; n<=size*size; n++) {
14      mtx[r][c] = n;
15      int i = r-1; if(i==-1) i=size-1;
16      int j = c+1; if(j==size) j=0;
17      if(mtx[i][j] != 0) {
18        r++;
19      } else {
20        r = i;
21        c = j;
22      }
23    }
24    return mtx;
25  }
```

Magic square

magic.c

```
27 void print(int** mtx, int size) {
28   for(int r=0; r<size; r++) {
29     for(int c=0; c<size; c++) {
30       printf("%d\t", mtx[r][c]);
31     }
32     putchar('\n');
33   }
34 }
35
36 void freeMem(int** mtx, int size) {
37   for(int r=0; r<size; r++) {
38     free(mtx[r]);
39   }
40   free(mtx);
41 }
```