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

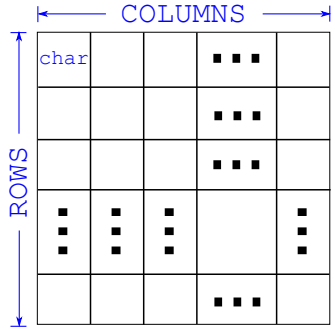
Cons:

- 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];
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 }
```