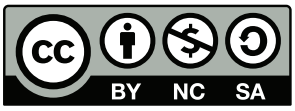
Learn programming by example Jos

Advanced C programming
Lab Exercises

v3.0 January 2022

Jos Onokiewicz



Lab Exercises

1

Arrays and Structs

1.1 Define a struct

The purpose of this command is to create input and output functions to fill and print an array of structs.

Defining an array with structs

Create a type definition (typedef) for a struct with at least 2 elements, for example:

```
1    typedef struct {
2       ???;
3       ???;
4    } data_t;
```

Define an array of at least 3 elements with this struct. (think about avoiding magic numbers, so use a symbolic name for the size of the array).

Create functions to fill and print a struct

Create a function to interactively fill a struct by a user, use the prototype:.

```
void getStruct(data_t *data);
```

Create a function to print the contents of a struct.

```
void printPoint(const point_t point);
```

Test the new functions

Create a main.c program that uses the functions of 4c and 4b to interactively fill the array with structs and then print this array with structs.

2

Pointers, Arrays and Functions

2.1 Hex dump function

Implement a function for displaying a hex dump of the bytes of a *long* value. Invoke this function several times with different input values in *main*(). Prototype:

```
void hexdumpLong(long data);
```

2.2 Calculate average of two doubles

Implement a function changing the two *double* input values to their average value. Invoke this function several times with different input values in *main*(). Prototype:

```
void average2D(double *pD1, double *pD2);
```

Do not implement the printing of the resulting values in this function because this will considerably limit the re-usability of this function. Print the resulting values in the updated arrays in *main*().

By using an empty body in the definition of the function, the code becomes compilable. The preprocessor *#warning* directive is added to the empty body of the function for showing at compile time that this function still needs to be implemented. This directive is not part of the C standard, but is supported by the *gcc* compiler. Using

this directive will limit the portability of the code.

Listing 2.1: average2D/main.c

```
1  #include <stdio.h>
2
3  void average2D(double *pD1, double *pD2);
4
5  int main(void)
6  {
7     double d1 = 2.0;
8     double d2 = 3.0;
9
10    average2D(&d1, &d2);
11    printf(" d1 = %lf   d2 = %lf\n", d1, d2);
12
13    return 0;
14 }
15
16 void average2D(double *pD1, double *pD2)
17 {
18 #warning function average2D() needs to be implemented!
19 }
```

The resulting value for $d1$ should be 2.5 and for $d2$ should also be 2.5.

2.3 Calculate average of a array

Implement a function changing the content of a *double* array for all elements to the average value of all the elements in the array. Invoke this function several times with different input values in *main*(). Prototype:

void averageDdata(*double data*[], *int size*);

· Why do we need to pass the size of the array to this function?

Listing 2.2: averageDdata/main.c

```
1  #include <stdio.h>
2
3  #define DATA_SIZE 4
4
5  void averageDdata(double data[], int size);
6
7  int main(void)
8  {
9     double sensorData[DATA_SIZE] = {1.0, 2.0, 3.0, 4.0};
10
11    averageDdata(sensorData, DATA_SIZE);
12
13    for (int i = 0; i < DATA_SIZE; i++)
14    {
15       printf(" %lf ", sensorData[i]);
16    }
17    puts("");
18
19    return 0;
20 }
21
22 void averageDdata(double data[], int size)
23 {
24 #warning function averageDdata() needs to be implemented!
25 }
```

The resulting values in *sensorData* should be: 2.5, 2.5, 2.5 and 2.5.

Print these resulting values in *main*(), not in the function *averageDdata()* because this will limit the re-usability of this function.

3

Command line arguments

3.1 Add command line arguments

Write a program that adds up all command line arguments representing a valid integer value without a preceding + or − sign.

Use *atoi*() to convert from an ASCII string to an integer value. If no arguments are given an appropriate error message needs to be given. Use the *stderr* stream for this error message.

Use the next function for checking if a string represents a valid positive integer value without a preceding + of − sign. Implementation:

```
1    bool isInt(const char str[])
2    {
3    int i = 0;
4    bool isInteger = true;
5
6    while (str[i] != '\0')
7    {
8    if (!isdigit(str[i]))
9    {
10   isInteger = false;
11   break;
12   }
13   i++;
14   }
15   return isInteger;
16   }
```

This function uses *isdigit*() for testing every character in the string *str* in a loop until a character is not a digit or the end of the string is reached (test for the null character). If this condition occurs the loop is terminated by a *break statement*.

Use *isInt*() in *main*() for checking in a loop if all command line arguments (*argv*[*i*]) are valid positive integers. This style of programming should prevent abnormal behaviour or crashing of the program if an input is not valid. If one of the arguments is not an integer, an appropriate usage message needs to be given before terminating the program with *exit*(). Use the *stderr* stream for this message.

Why shouldn't you implement the printing of result messages in the *isInt*() function?

3.2 Checking integers

Update the function *isInt*() to check if a string represents a valid integer value possibly preceded by only one + or − sign. Some examples:

```
Valid integer strings, isInt() returns true:

"0"
"123"
"-123"
"+123"
```

```
Invalid integer strings, isInt() returns false:

isInt("a123")
isInt("123a")
isInt("--123")
isInt("+-123")
isInt("-12+3")
isInt("123+")
isInt("1-23")
isInt("+++123")
isInt("+")
isInt("-")
isInt("")
```

Use the showed valid and invalid integer string value examples for testing the updated function *isInt*() in *main*().

4

Void pointers and Callback functions

4.1 Using function pointers

Study the code of the listing below and complete the code in such a way that the array operations is filled with the correct information to be able to perform the 4 operations (sum, sub, mult and divd) using function pointers:

- Complete the typedef of the struct (line 26) so that it contains a function pointer named ope.
- Make sure the correct function pointers are in the operations array (starting at line 50)

Listing 4.1: functionpointers/main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  /*!
6   * Multiple functions in an array
7   *
8   * This program uses an Array with functions to create a simple
9   * calculator
10  * complete the program:
11  *
12  * 1) add the functions and text ("sub" , "add" , "div" , "mul" ) to the array
```

```
13  * 2) test the program
14  *
15 */
16
17
18 /*!
19  *  definition of a struct that holds the name of the operation (name)
20  *  and a pointer (ope) that holds a pointer to the function that handles the
21  *  operation
22 */
23 /// ope should hold the pointer to a function tha belongs to an operation
24 typedef struct
25 {
26    int ope;      /// Change ope to the correct definition
27    char name[10];
28 } operation;
29
30 /*!
31  * \brief sum, sub, mul, div    The functions that handle the operations
32  * \param num1
33  * \param num2
34  * \return
35  */
36 int sum(int num1, int num2);
37 int sub(int num1, int num2);
38 int mult(int num1, int num2);
39 int divd(int num1, int num2);
40
41
42 int main()
43 {
44    int x, y, choice, result;
45    /// Array that contains the function and information for the operations
46    operation operations[4];
47    /// Replace the contents of the array below with the correct
48    /// assignments for each operation
49    ///
50    operations[0].ope = NULL;
51    strcpy(operations[0].name, "");
52    operations[1].ope = NULL;
53    strcpy(operations[1].name, "");
54    operations[2].ope = NULL;
55    strcpy(operations[2].name, "");
56    operations[3].ope = NULL;
57    strcpy(operations[3].name, "");
58
59    printf("Enter two integer numbers (on one line: ");
60    scanf("%d%d", &x, &y);
61
62    printf("Enter:\n0 to sum\n1 to subtract\n2 to multiply\n3 to divide\n");
63    scanf("%d", &choice);
64
65    if (operations[choice].ope != NULL && choice <= 3)
66      {
67        result = (operations[choice].ope)(x, y);
68        printf("Operation: %s  result: %d\n ", operations[choice].name, result);
69      }
70    else
71      {
72        printf("Function not available\n");
```

```
73          exit(1);
74        }
75
76     return 0;
77  }
78
79  int sum(int x, int y)
80  {
81     return(x + y);
82  }
83
84  int sub(int x, int y)
85  {
86     return(x - y);
87  }
88
89  int mult(int x, int y)
90  {
91     return(x * y);
92  }
93
94  int divd(int x, int y)
95  {
96     if (y != 0)
97     {
98        return (x / y);
99     }
100    else
101    {
102       return 0;
103    }
104 }
```

4.2 Create a callback function

Write a program for sorting an array of zip codes (postal codes) using *qsort*(). A zip code must be implemented in a struct. Use a *typedef* for the type name of this struct.

Example Dutch postal code:

```
1     /* Dutch postal code, example 1200AB */
2
3     typedef struct
4     {
5     int number;
6     char twoChars[3];  /* string, needs additional '\0' */
7     } zipcode_t;
```

Implement an appropriate necessary compare function for two of these structs containing zip codes. Prototype:

```
int compareZipCodes(const void *pZC1, const void *pZC2);
```

5

Singly Linked List

5.1 Create a function for a SLL

Implement the API function *sizeSLL*() for determine the size of an SLL (the SLL size equals the number of nodes). Prototype:

```
size_t  sizeSLL(const node_t *pHead);
```

Why should you use *const*?

The function *sizeSLL*() is very similar to *showSLL*(), instead of showing the data contents of a node it should increment the value of a local variable representing the size of the SLL. Draw a diagram for showing the counting of nodes in an SLL containing 3 nodes.

What happens to the execution time of the function if you double the number of nodes in the SLL? What is the *time complexity* of this function in *big O* notation?

5.2 Dynamic allocated memory in a SLL

Consider a struct containing a pointer to dynamic allocated memory. Structs can be assigned to each other for copying. What kinds of problems can occur when you copy such a struct to another one?

Draw diagrams with structs and pointers showing the runtime problem. Run the next program (in a Linux development environment) and do a memory check by running the next command:

```
valgrind ./deepcopy
```

Listing 5.1: deepcopy/main.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  typedef struct {
6     int count;
7     char *pText; /* must point to dynamic allocated memory! */
8  } problem_t;
9
10 int deepCopy(problem_t *pDestination, const problem_t *pSource);
11
12 int main(void)
13 {
14    problem_t s1 = {1, NULL}; /* do not do ... = {1, "ABC"} */
15    problem_t s2 = {2, NULL};
16
17    s1.pText = (char *)malloc(4 * sizeof(char)); /* array of 4 chars */
18    if (s1.pText != NULL)
19    {
20       strcpy(s1.pText, "111");
21       s2 = s1;           /* shallow copy, problem? */
22       s2.pText[0] = 'A'; /* problem? */
23
24       printf("s1 = %d %s\n", s1.count, s1.pText);
25       printf("s2 = %d %s\n", s2.count, s2.pText);
26
27       free(s1.pText); /* problem? */
28    }
29    s2.pText[1] = 'B'; /* problem? */
30
31    printf("s1 = %d %s\n", s1.count, s1.pText);
32    printf("s2 = %d %s\n", s2.count, s2.pText);
33
34    return 0;
35 }
36
37 int deepCopy(problem_t *pDestination, const problem_t *pSource)
38 {
```

```
39     /* TODO implementation, update main() code,
40      * call this function in main() to avoid the problems of
41      * a shallow copy.
42      */
43
44     /* TODO should return an error code if memory allocation fails */
45     return 0;
46 }
```

5.3 Runtime complexity check

Consider the next two implementations for determining if an SLL is empty.

First implementation:

```
1    bool isEmptySLL(const node_t *pHead)
2    {
3    return sizeSLL(pHead) == 0;
4    }
```

Second implementation:

```
1    bool isEmptySLL(const node_t *pHead)
2    {
3    return pHead == NULL;
4    }
```

Compare the run-time complexity of these two implementations. Which one is preferable?

5.4 Deep copy structs

Program a function for "deep" copying *problem_t* structs containing an *int* and a pointer *char** to dynamic allocated memory, and the contents of the dynamic allocated *char* array!

This function must solve the problems of a shallow copy causing an ownership problem because after the copy you will have two pointers pointing to the same allocated data in memory. If one of the pointers will deallocate the memory the second pointer becomes a *dangling pointer* (points to de-allocated memory).

Prototype:

*int deepCopy(problem_t * pDestination, const problem_t * pSource)*

The struct where *pDestination* is pointing to, must allocate its own external array of characters for avoiding an ownership problem.

Programming steps for implementing the function *deepCopy()* in *pseudo code* (a step-by-step written outline of your code that you can gradually transcribe into the C programming language):

- free the allocated memory pointed by the pointer in **pDestination* (for preventing memory leakage).
- do a shallow copy: **pDestination = *pSource;*
- determine the size of the allocated memory pointed by the string pointer in **pSource* (string length + 1).
- allocate dynamic memory with the determined size and overwrite the string pointer in **pDestination* with the value returned by *malloc*().
- copy the string pointed by the string pointer in **pSource* to the location pointed to by the string pointer in **pDestination*.

The returned *int* of *deepCopy*() should be an error value (not equal to 0) if the dynamic memory allocation fails.

Test this function in *main*() by replacing the shallow copy *s2 = s1;* by a deep copy calling the function *deepCopy*(&*s*2, &*s*1);.

Use in a Linux development environment *valgrind* for checking memory allocation problems.

6

Queues, exercises to the CSLL queue implementation.

6.1 Empty CSLL and CSLL with one node

Draw a diagram for an empty CSLL and a CSLL containing one node.

6.2 Ssize of a CSLL

Implement the queue API function: *sizeQueue*() for determining the number of nodes in a queue. Before implementing this function draw a diagram for visualising the steps of counting the number of nodes. Start with a queue containing 3 nodes and the necessary pointers. Test this function in *main*().

What is the *time complexity* of this function in *big O* notation?

6.3 Delete a CSLL

Implement the queue API function: *deleteQueue*() for deleting all nodes in a queue. Before implementing this function draw several diagrams for visualising the steps for

deleting all nodes. Start with a queue containing 3 nodes and the necessary pointers. Test this function in *main*().

What is the *time complexity* of this function in *big O* notation?

6.4 Create a CSLL, checking *pBack*

The queue API function: *createQueue*() is not fully implemented. It lacks checking if the pointer *pBack* is already pointing to an existing queue. This queue must be deleted otherwise this will cause a leaking memory problem. Why? Test this function in *main*().

6.5 Checking the memory allocation creating a CSLL

The following queue API functions *createQueue*() and *pushQueue*() return no error code indicating if the allocation of memory has succeeded or not. Which type and values should be used for the error code? Implement this feature. Why is this important? Are these functions easy to test in *main*()? Why?

7

Software engineering, debugging and testing

7.1 Testing your own software

Discussion

7.1.1 Testing

During the Introduction Software Engineering course you developed the simulation of a machine based on a statemachien design. Discussion about testing

- What kind of tests can you distinguish before submitting the program?
- How did you perform these tests?
- After submitting, the teacher has checked your work, what type of tests do you think the teacher has performed?

7.1.2 Debugging

- How did you check out and fix any errors in your program?
- What resources did you use?

7.2 Test cases

The book show's an example about white box testing

```
1     if (a && b \|\| c)
2     {
3
4     }
```

Show all options if you want to test this feature thoroughly.

8

Unit testing

8.1 Unit testing with Unity

Review the code *checkStrings* of the *Unity* unit tests. Add more appropriate tests for the functions *trim* and *isInt*(). Always apply boundary value analysis.

8.2 More unit testing with Unity

Review the code of the *Unity* unit tests. Always check the test code first, otherwise you will loose a lot of time in trying to solve non-existing bugs because the test code is buggy.

Add some new appropriate tests based on the internals of the functions (white box testing). Always apply boundary value analysis.

8.3 Another testing framework, Catch2

Review the code of the *Catch*2 unit tests. Check this code if necessary and add some new appropriate tests. Always apply boundary value analysis.

9

Complex numbers

9.1 Calculation with complex numbers

Calculate and print the complex result of the following mathematical formulas:

- $result1 = (-3 - i)(5 - 2i)$
- $result2 = \frac{1 - i}{i + 3}$

9.2 Using structs to simulate complex numbers

Let's say there is no library for Complex numbers then you can simulate a complex number by defining a struct representing the real and the imaginair part of a complex number. This can look like:

```
1  /*!
2  * Definition of the struct-type complexInt\_t
3  */
4
5  typedef struct ComplexInt\_t\{
6     int real;
7     int img;
8  \} complexInt\_t;
```

Create a function to add two of these simulated complex numbers.

```
complexInt_t addComplex(const complexInt_t a, const complexInt_t b);
```

Create a function to multiply two of these simulated complex numbers.

```
*complexInt_t mulComplex(const complexInt_t a, const complexInt_t b);
```

10

Compare floating point numbers

10.1 Using fp variables in a for-loop

Why should you never use floating-point variables as a loop counter in a for-loop? Code example for-loop:

```
1 for (float x = 0.1f; x <=1.0f; x += 0.1f)
2 {
3 /* do something, use x */
4 }
```

How many times do you expect that this loop will be iterated? Why is the next code block preferred?

```
1 for (int i = 1; i <= 10; i++)
2 {
3 float x = i * 0.1;
4 /* do something, use x */
5 }
```

See 'Rule 05. Floating Point' in [**?**].

10.2 Compare two structs

How can you compare two structs?

11

Bool type

11.1 Comparing two arays with booltype result

Implement a function returning a *bool* typed result for comparing the content of two same sized *int* arrays: *arr*1 and *arr*2. Use a for-loop and a break statement (see code example *bool stringIsAllUpperCase*(*const char str*[])).

Test this function in *main*(). Prototype:

bool compareIntArrays(*const int arr*1[], *const int arr*2[], *int size*);

Example of calling this function and printing the boolean value as *true* or *false* text:

```
1       #define DATA_SIZE 4
2
3       int data1[DATA_SIZE] = {1, 2, 3, 4};
4       int data2[DATA_SIZE] = {1, 2, 3, 4};
5
6       bool arraysAreEqual = compareIntArrays(data1, data2, DATA_SIZE);
7
8       printf("Arrays are equal = %s\n", arraysAreEqual ? "true" : "false");
```