Informatics II, Spring 2024, Solution Exercise 1

Publication of exercise: February 19, 2024
Publication of solution: February 26, 2024
Exercise classes: February 26 - March 1, 2024

Learning Goal

- Gain an initial understanding of the C programming language. This should include initial experience with the syntax, getting to know some peculiarities of C and being able to program simple algorithms yourself.
- Understand and implement simple sorting algorithms.
- Gain some basic intuition for the runtime of an algorithm.

Task 1 [Easy/Medium]

In the following task you will find six code snippets, each of which should show you a characteristics of C or draw your attention to possible dangers in this language. Run all of the code snippets and answer the following questions for each:

- What do you observe?
- How do you explain what you observe?

You can also find all snippets on OLAT.

1.1: Danger when programming in C, possible source of bugs

```
1 #include "stdio.h"
2 int main () {
3     int a = 2147483647;
4     int b = 2147483648;
5     int c = 2147483649;
6     printf("%d, %d, %d", a, b, c);
7     return 0;
8 }
```

Observation: First, as expected, the number 2147483647 is output, but then there is a jump to -2147483648. After that, counting continues at -2147483647.

Explanation: In C, an integer is limited to a length of 4 bytes, or 32 bits. Normally the most significant bit is used for the sign, which is why we can cover a range from -2'147'483'648 to 2'147'483'647 with integers. If the number stored in the integer exceeds this range, a so-called integer overflow occurs. In this case, on a binary level, counting simply continues after the maximum, whereby it starts again at the minimum. C does not check for this danger and also does not warn you if such an overflow occurs. Consequently, you must be careful when programming, otherwise unpredictable bugs may occur. If you need to store larger numbers, you can use the "long" data type (8 bytes) or a "short" (2 bytes) for shorter numbers.

1.2: Danger when programming in C, possible source of bugs

```
1 #include "stdio.h"
2 int main() {
3     int myArray[20];
4     for(int i = 0; i < 20; i++) {
5         printf("%d \n", myArray[i]);
6     }
7     return 0;
8 }
```

Observation: The code snipped outputs random looking numbers. If the program is executed several times, it may result in different outputs.

Explanation: In C, variables are not initialized with a default value. The value that a variable has after declaration is the binary data that was previously stored in memory at that location. When programming, you should pay close attention to this, as it can lead to unpredictable bugs.

1.3: Danger when programming in C, try to avoid this at all costs

```
1 #include "stdio.h"
2 int main() {
3     int myArray[1];
4     myArray[0] = 0;
5     myArray[1] = 1;
6     myArray[2] = 2;
7     printf("%d, %d, %d", myArray[0], myArray[1], myArray[2]);
8     return 0;
9 }
```

Observation: Even if the array only has a length of 1, you can exceed the end of the array without an error occurring.

Explanation: C does not ensure that you do not exceed the end of an array. Therefore, in this example, we can exceed the limit of the array to store and retrieve values there. However, these values may overwrite other data which is stored there or your values will be overwritten by another part of your program at a later time. In general, this leads to some of the most unpredictable bugs. Therefore, when programming with C, you should make sure that you always have the boundaries of your arrays under control.

1.4: Danger when programming in C, possible source of bugs and leads to code that is difficult to understand

```
1  #include "stdio.h"
2  int main() {
3      int myArray[] = {72, 101, 108, 108, 111, 32,
4                       87, 111, 114, 108, 100, 33};
5      for(int i = 0; i < 12; i++) {
6          printf("%d", myArray[i]);
7      }
8      printf("\n");
9      for(int i = 0; i < 12; i++) {
10         printf("%c", myArray[i]);
11     }
12     return 0;
13 }
```

Observation: As you would expect, the program first outputs the numbers one after the other but on the second line the message "Hello World!" appears.

Explanation: Even if a variable has been declared as a type in C, you can also read its binary content as other types. In this example, the integers in myArray are read as characters and therefore the numbers are interpreted as ASCII characters. For example the number 72 becomes a capital "H". However, such an implicit data type change can lead to bugs and poorly understandable code. If you want to do something like this, you should do it explicitly. In the example here, on line 10 the following code would be better: (char)myArray[i]

1.5: Special property of C

```
1  #include "stdio.h"
2  int main() {
3      int myArray[5];
4      int size1 = sizeof(myArray);
5      int size2 = sizeof(myArray[0]);
6      int size3 = size1 / size2;
7      printf("%d, %d, %d", size1, size2, size3);
8      return 0;
9  }
```

Observation: The code snipped first outputs the value 20, 4 and then 5.

Explanation: In C, the sizeof() function returns the size in number of bytes. As an integer has a size of 4 bytes, the value 4 is output as the second number. The first value is 20 because the array contains 5 integers of length 4. The last value is the length of the array, since the total number of bytes divided by the bytes per entry gives us the number of elements.

1.6: Special property of C

```
1  #include "stdio.h"
2  int main() {
3      char myString[] = "hello";
4      int stringSize = sizeof(myString) / sizeof(myString[0]);
5      printf("%d, ", stringSize);
6      for (int i = 0; i < stringSize; i++) {
7          printf("%c", myString[i]);
8      }
9      return 0;
10 }
```

Observation: The string is 6 characters long and not 5 as expected. At the end of the string, an unusual character is output which was not part of the original string. Depending on the IDE used, this character may also be filtered out and not displayed in the terminal.

Explanation: In C, strings are arrays of characters. To mark the end of the string, a '\0' is inserted as the last character. This makes it relatively easy to find out what the end of a string is. However, you have to be careful as the string from the example has length 6 and not 5 as expected.

Task 2 [Medium]

One of the simplest ways to compress a string is the so-called Run-Length-Encoding (RLE). In this method, consecutive identical characters in a string are replaced by the amount and the character itself.

An example of this would be "AAABBAAAA" is transformed by RLE to "3A2B4A", where "3A" stands for three consecutive "A"s, "2B" for two consecutive "B"s and "4A" for four consecutive "A"s.

Write the function `rleCompression(char string[], int length)` in C that takes a string and its length as input and outputs the compressed version on the terminal. To make it easier for you, you will find a code skeleton on OLAT, so that you only have to insert the relevant parts of the code.

```
1  #include <stdio.h>
2
3  void rleCompression(char string[], int length) {
4      if (length == 0) {
5          return;
6      }
7      int charCount = 1;
8      char mostRecentChar = string[0];
9      for (int i = 1; i < length; i++) {
10         if (mostRecentChar == string[i]) {
11             charCount++;
12         } else {
13             printf("%d%c", charCount, mostRecentChar);
14             charCount = 1;
15             mostRecentChar = string[i];
16         }
17     }
18     printf("%d%c", charCount, mostRecentChar);
19 }
20
```

```
21 int main() {
22     char string[] = "AAABBAAAA";
23     int length = (sizeof(string) / sizeof(string[0])) - 1;
24     rleCompression(string, length);
25     return 0;
26 }
```

Task 3 [Medium]

3.1: Write the function `bubbleSort(int array[], int length)` in C, which takes an array and its length as input and sorts it in ascending order using the bubble sort algorithm.

```
1  #include <stdio.h>
2
3  void swap(int array[], int index1, int index2) {
4      int tmp = array[index2];
5      array[index2] = array[index1];
6      array[index1] = tmp;
7  }
8
9  void bubbleSort(int array[], int length) {
10     int counter = 0;
11     for (int i = length - 1; i > 0; i--) {
12         for (int j = 1; j <=i; j++) {
13             counter++;
14             if (array[j] < array[j - 1]) {
15                 swap(array, j, j - 1);
16             }
17         }
18     }
19     printf("Counter: %d \n", counter);
20 }
21
22 int main() {
23     int array[] = {0, 1, 3, 4, 2, 8, 9, 5, 6, 7};
24     int length = sizeof(array) / sizeof(array[0]);
25     bubbleSort(array, length);
26     //Output sorted array
27     for (int i = 0; i < length; i++) {
28         printf("%d, ", array[i]);
29     }
30     return 0;
31 }
```

3.2: Also write the function `insertionSort(int array[], int length)` in C which takes an array and its length as input and sorts it using the insertion sort algorithm.

```c
#include <stdio.h>

void insertionSort(int array[], int length) {
    int counter = 0;
    for (int i = 1; i < length; i++) {
        int j = i - 1;
        int current = array[i];
        while (j >=0 && array[j] > current) {
            counter++;
            array[j + 1] = array[j];
            j--;
        }
        array[j + 1] = current;
    }
    printf("Counter: %d \n", counter);
}

int main() {
    int array[] = {0, 1, 3, 4, 2, 8, 9, 5, 6, 7};
    int length = sizeof(array) / sizeof(array[0]);
    insertionSort(array, length);
    //Output sorted array
    for (int i = 0; i < length; i++) {
        printf("%d, ", array[i]);
    }
    return 0;
}
```

3.3: Modify the two previously programmed sorting algorithms so that they now count how often the innermost loop has been run through. After running the algorithm, output the number of runs on the terminal.

Execute the algorithms with the following array: [0, 1, 3, 4, 2, 8, 9, 5, 6, 7]

What do you notice? How do you explain what you observed?

The Insertion Sort runs through its innermost loop significantly less than the Bubble Sort. This is because the insertion sort breaks off the innermost loop early as soon as the subarray currently being processed is sorted. With the bubble sort, on the other hand, the inner loop is always completed regardless of the input.

3.4: Change the input array such that your algorithm outputs the highest possible counter. You are allowed change the order of the elements but you need to keep the same numbers.

The bubble sort always runs through the innermost loop the same number of times, regardless of the input, so you cannot increase the counter by changing the arrays order. With the insertion sort, however, you can increase the number of passes through the innermost loop. This reaches its maximum when the input array is present in reverse order. In our case, this is [9, 8, 7, 6, 5, 4, 3, 2, 1, 0].

3.5: On OLAT you will find a C program with an array of 100,000 elements. Sort this list with your bubble sort. How many seconds does the algorithm take? How often is the innermost loop run through for this list length? Try to calculate this number and then check with your counter.

Note: If this large list causes difficulties for your computer, you can also just do the calculation.

This task should show that already an array of the size of 100,000 has a relatively long runtime. This should hardly be less than 15 seconds for any student. This long runtime comes about because there are almost 5 billion iterations (i.e. 4'999'950'000,). If we look closely at the algorithm, we can see that 99'999 iterations have to be made in the first run in order to sort the first element correctly. In the next run, however, it only needs 99'998 because an element has already been placed correctly. This continues until we only need one iteration of the inner loop in the last run of the outer loop. To calculate the total sum we can use the sum of an arithmetic series.

We want the sum of the first 99999 natural numbers. This can be represented as the series:

$$S = 1 + 2 + 3 + \ldots + 99998 + 99999$$

The sum of an arithmetic series can be calculated using the formula:

$$S = \frac{n}{2} \times (a_1 + a_n)$$

where:

- n is the number of terms in the series,
- a_1 is the first term in the series,
- a_n is the last term in the series.

In our case:

- $n = 99999$ (the number of terms),
- $a_1 = 1$ (the first term),
- $a_n = 99999$ (the last term).

Substituting these values into the formula gives:

$$S = \frac{99999}{2} \times (1 + 99999)$$

Which equals:

$$S = 4,999,950,000$$

Task 4 [Medium/Hard]

In computer science, a subarray is a contiguous sequence of elements within an array. For example, for the array [1, 2, 3], the subarrays are [1], [2], [3], [1, 2], [2, 3], and [1, 2, 3].

Write a C function `zeroSubarray(int array[], int length)` that takes an array of integers and its length as parameters and returns 1 if it is possible to find a subarray where the sum of all integers adds up to zero. Otherwise, the function should return 0.

Examples:

- $[3, -2, 4, 2, 1, -5]$ should return 1, since [-2, 4, 2, 1, -5] is a contiguous subarray that adds up to zero.
- $[1, 2, 3, 4, 5, 0]$ should return 1, since [0] is a contiguous subarray that adds up to zero.
- $[-2, 4, -1, 5, 9, -12]$ should return 0, since there is no possibility to find such a subarray that adds up to zero.

```c
#include <stdio.h>

int zeroSubarray(int const array[], int length) {
    for (int i = 0; i < length; i++) {
        int sum = 0;
        for (int j = i; j < length; j++) {
            sum += array[j];
            if (sum == 0) {
                return 1;
            }
        }
    }
    return 0;
}

int main() {
    int array[] = {3, −2, 4, 2, 1, −5};
    int arrayLength = sizeof(array) / sizeof(array[0]);
    printf("Solution: %d", zeroSubarray(array, arrayLength));
    return 0;
}
```