UNIVERSITÄT BASEL

Thorsten Möller
Maarten Schenk - maarten.schenk@unibas.ch
Michael Plüss - m.pluess@unibas.ch

Programming Paradigms – C++ FS 2019

Exercise 2 Due: 14.04.2019 23:59:59

Upload answers to the questions **and source code** before the delivery deadline via courses.cs.unibas.ch. In addition, running programs have to be demonstrated during the exercise slots until **Friday, 26.04.2019** the latest. Also note, of all mandatory exercises given throughout the course, you must score at least 2/3 of their total sum of points to get accepted in the final exam.

Modalities of work: The exercise can be completed in groups of at the most 2 people. Do not forget to provide the full name of all group members together with the submitted solution.

Question 1: Pointers (6 points)

In this exercise you are asked to analyze the following, admittedly contrived, C++ code. What is its output? Explain what is happening in each line. Also, the last line contains the expression `((p + 5) - p)`. Is there a difference to using `(p + 5 - p)`? On the other hand, what would be the problem if we would write `(p + (5 - p))` instead?

Hint: If the output is not uniquely determinable describe of what kind it would be.

```
int a = 2;
int b = 3;
int *p = &a, **q = &p;
*p *= 2**&b***q; // Never write such minified lines in your code!
                 // This is just for the sake of it.
p = &b;
(*p)++;
p -= 5;
cout << a << " " << b << " " << p << " " << ((p + 5) - p) << endl;
```

Question 2: Pointers (8 points)

a) The following C++ code is supposed to retrieve an address value created within the function `foo`. Does that make sense? Explain your answer in any case.

```
int& foo() {
   int x = 3;
   return x;
}

int main() {
   int print_out = foo();
   cout << &print_out << endl;
}
```

(2 points)

b) Write a C++ function that takes two ordered Integer arrays of arbitrary size, computes the ordered array that contains the elements of both arrays up to the point where they are still the same and returns the pointer to the beginning of the new array. Write a main function to test your implementation. You are not allowed to use square brackets to access array values. You can use this function declaration:
`int* firstCommonSequence(int* a, int alen, int* b, int blen, int& len);`

Example: Let's say that we get the arrays $[1, 2, 3]$ and $[1, 2, 4]$ as input. Then we get the pointer to the first element of the array $[1, 2]$ as output. Another example: $[1, 2, 3]$ and $[2, 3, 4]$ should return a `nullptr` that shall represent the empty array $[]$.

(3 points)

c) Consider your answer to b). Is it possible to write a reliable function fulfilling the requirements of b) such that the pointer to the first element of the new array is the same as the pointer to the first element of the first given array? If yes, implement a function that guarantees this and explain your implementation. If not, explain why.

Please provide a detailed explanation.

(2 points)

d) The following C++ fragment contains some errors. Find, explain and fix them.

```
int a = 1, int b = 1;
int* p1, p2;
p1 = &a, p2 = &b;

if (*p1 == &p2) {
  cout << "a != b" << endl;
} else {
  cout << "a == b" << endl;
}
```

(1 points)

Question 3: Function Pointers (8 points)

This exercise is about function pointers. In practice you often need to define an interface that can use different functions within an algorithm. Design an algorithm that can *compare* two arrays of the same length according to different *comparator* functions. More precisely, the algorithm takes two `double` arrays of size 3 and a *comparator* function as input and returns whether one array is larger than the other in regard to the given comparator; that is, return either of $-1, 0, 1$ if the first is smaller, equal, or larger than the second one.

Implement two comparators: One that compares the two arrays according to the cosinus value of the 2nd element (see `math.h`). The other comparator considers an array to represent a point in the Euclidian space $\mathbb{R}^3$ and compares the points (arrays) according to the Euclidean distance (see `https://en.wikipedia.org/wiki/Euclidean_distance`) from the point of origin $(0, 0, 0)$; i.e., the first point is larger than the second if it is more distant from $(0, 0, 0)$ regarding the Euclidian norm.

Use function pointers to pass the comparator function into the function.

Question 4: Structures and Pointer Arithmetic (15 points)

a) In this subtask you are asked to implement a structure for a doubly linked list of arbitrary size where elements are `int` numbers. Your doubly linked list should have a head and a tail and also keep memory management in mind. In addition, there are at least the following functions to be implemented for a doubly linked list:

```
doublyLinkedList& addFirst(doublyLinkedList& list, int& value);
//adds the value to the end of the given doubly linked list.

doublyLinkedList& addLast(doublyLinkedList& list, int& value);
//adds the value to the front of the given doubly linked list.

int size(DoublyLinkedList& list);
//returns the size of the given doubly linked list.

remove(DoublyLinkedList& list, int value);
//removes all occurrence of the given value from the given doubly linked list.

unique(DoublyLinkedList& list);
//removes all duplicates inside the given doubly linked list.

reverse(DoublyLinkedList& list);
//reverses the given doubly linked list. E.g.: [1, 2, 3] -> [3, 2, 1].
```

(9 points)

b) In this exercise you will implement the *shellsort* algorithm for our newly created doubly linked list data structure. Shellsort is a sorting algorithm which can be seen as either a generalization of sorting by exchange (bubble sort) or sorting by insertion (insertion sort) (see `https://en.wikipedia.org/wiki/Shellsort`). Conceptually it relies on gaps, at the beginning a larger gap, so that a value which is totally out of place moves faster back or forward. The gap gets smaller and smaller, in the end (gap=1) this algorithm behaves like a normal insertion sort. For the size of the gaps use the length of the doubly linked list divided by 2.

Important: The implementation should be as memory-efficient as possible. This means we always work on the same structure.

(6 points)

Question 5: Match Three in Python (10 points)

In this exercise you will implement the well known game *Match Three* on the terminal. The rules are as follows:

- A single player tries to score points by matching three of a kind
- The player specifies which two adjacent blocks he/she wants to swap
- When the swap results in three of a kind in a straigt line (horizontal or vertical) the blocks are destroyed and the player receives points
- Normally, the player can only swap blocks when the swap leads to a match. In our case you are allowed to disregard this constraint
- Blocks fall from above into the now empty slots
- If this results in more matches of three of a kind or more, these matching blocks are again destroyed and new blocks fall into their place
- Normally, the game runs until the player cannot create any more matches. In our case you are allowed to disregard this constraint

Important: You can find an incomplete implementation in the file `matchThree.py`.