Informatik - AS19

Exercise 11: Classes & Pointers

Handout: 25. Nov. 2019 06:00

Due: 2. Dez. 2019 18:00

Task 1: Understanding struct & classes

Open Task

This task is a text based task. You do not need to write any program/C++ file: the answer should be written in main.md *(and might include code fragments if questions ask for them).*

Task

Consider the following definitions:

1. ```
   struct A {
       int a;
       double b;
       int c;
   };

   A str = {1, 1.5, 2};
   std::vector<int> vec = {1, 1, 3};
   int& a = vec[0];
   ```

   For each of the provided expressions state their *C++ type* and *value*:

   1. `str.a * str.b`
   2. `str.b == vec[1]`
   3. `str.a * str.b / str.c`
   4. `vec[str.a] / str.c`
   5. `a / 2 - str.b`

2. ```
   class B {
   public:
       B(): vec(128) {
           for (int i = 0; i < 128; ++i) {
   ```

```
                vec[i] = 0;
            }
        }

        // PRE: ...
        // POST: ...
        void add(const char c) {
            ++vec[c];
        }

        // PRE: ...
        // POST: ...
        int get(const char c) const {
            return vec[c];
        }

    private:
        std::vector<int> vec;
    };
```

Determine PRE- and POST-conditions for the methods `add` and `get`.

Task 2: Averager

Open Task

Task

Write a class `Averager` that computes averages of given values of type `double`.

Initially, an instance of class `Averager` does not contain any value. The class `Averager` must provide the following functionality:

```
// POST: Adds a value to the current average calculation.
void add_value(double value);

// POST: Returns the average of all added values,
//       or zero, if no value has been added.
double get_average();

// POST: Removes all values from the current average calculation.
void reset();
```

The declaration and implementation of class Averager **must** be split between

header file (`averager.h`, containing declaration) and implementation file (`averager.cpp`, containing implementation).

Task 3a: Understanding Pointers: - Lookup

Open Task

Task Description

Complete the definition of function `lookup` by editing the file `lookup.cpp` according to its specified pre- and post conditions.

We provide a test program using the implemented function to perform lookup on a vector.

Input

Vector length `l`.

`l` vector elements.

lookup index `i`.

Example:

```
3 9 8 7 1
```

means a 3-element vector `[9, 8, 7]` and the lookup index `1`.

Output

The value of the element at the lookup index.

Example: for the input given above, this value is `8`.

Task 3b: Understanding Pointers - Add

Open Task

Task Description

Complete the definition of function `add` by editing the file `add.cpp` according to its specified pre- and post conditions.

We provide a test program using the implemented function to add two values given by the user.

Input

Numbers a, b.

Example:

```
21 35
```

Output

The sum of a and b.

Example: for the input given above, this value is 56.

Task 3c: Understanding Pointers - Num_Elem

Open Task

Task Description

Complete the definition of function `num_elem` by editing the file `num_elem.cpp` according to its specified pre- and post conditions.

We provide a test program using the implemented function to find the number of elements in a specified interval of a vector.

Input

Vector length.

Start index a.

End index b.

Example:

```
10 3 10
```

means a 10-element vector and the interval between index `3` and `10`.

Output

The number of elements in the interval `[a,b)`.

Example: for the input given above, this value is `7`.

Task 3d: Understanding Pointers - First_Char

Open Task

Task Description

Complete the definition of function `first_char` by editing the file `first_char.cpp` according to its specified pre- and post conditions.

We provide a test program using the implemented function to find the index of the first character in a string.

Input

String `s` (no whitespaces).

Character `c`.

Example:

```
jabberwock b
```

Output

The index of the first occurence of character `c` in `s`, or "`Not found.`", if `c` is not present in `s`.

Example: for the input given above, this index is `2`.

Task 4: Quick Sort

Open Task

Task

Write a program that implements a naive sorting algorithm that sort the contents of an integer array in ascending order. The algorithm to be used is described in steps below; do not use a different algorithm. There is no need to write particularly efficient code. Implement the algorithm in `quicksort.cpp`.

Specific rules for this task:

1. The goal of this exercise is to exercise the usage of *pointers*: Instead of using a vector to manage the values to be sorted, you have to explicitly allocate the necessary memory yourself and to traverse the memory block using ranges.
2. In particular, usage of vectors is forbidden.
3. Dereferencing pointers with operator `[]`, or performing pointer addition/substraction (due to the equation `*(a+i) = a[i]`) is forbidden as well, with the **only** exception of function `input`. Note that pointer incrementation/decrementation *is* allowed, and is the expected method to traverse memory ranges.
4. Usage of library sorting function is of course not allowed.

Algorithm

The algorithm which you are going to implement is called quicksort. It takes a range of values on its input and picks a pivot among these values. Then, it partitions the range: it swaps the elements within the range so that all elements less than pivot are on its "left side", and all elements greater on equal than the pivot are on its "right side". Then, it repeats the entire procedure to the "left" and "right" sides separately, picking new pivots for them and so on. The way to choose a pivot is arbitrary: in this task, you should pick the first element of the range.

For example, for range

```
[4, 6, 1, 8, 3, 2]
```

we pick the pivot at 4:

```
[__4__, 6, 1, 8, 3, 2]
```

the result of the partition is

```
[2, 1, 3, __4__, 8, 6].
```

Then, we repeat the algorithm for ranges `[2, 1, 3]` and `[8, 6]`.

Ranges of a memory block are given as interval $[begin, end)$. $begin$ points to the first element of the range and end points just behind the last element of the range. E.g., for a chunk of memory of size `N` beginning at pointer `ptr`, `int* begin = ptr` and `int* end = ptr + N`.

The code in function `main` allows you to run the program in four different modes which you can use for testing. It accepts input of form

```
mode arguments
```

where `mode` is a character descibing the mode of execution:

- `i` - test input / output.
- `s` - test swap.
- `p` - test partition.
- `q` - run quicksort (final test).

and `arguments` are a sequence of relevant arguments.

More details will follow in the next section.

The task

Your task consists of the following parts:

1. Write a function `void input(std::istream& is, int*& begin, int*& end)` that reads a sequence of integer values from stream `is`, and stores them in a freshly allocated memory range. The bounds of the range must be stored in `begin` and `end` at the end of the function execution.

   The sequence of integer values is given in the following format:

   1. an unsigned integer N giving the length of the sequence.
   2. N successive (signed) integer values giving the content of the sequence
2. Write a function `void output(std::ostream& os,const int* begin,const int* end)` that displays the values in range from `begin` to `end`, in order, separated by single spaces. You can test it (and input) with the function `test_input_output()`. It inputs a sequence and outputs it immediately. It is run when the first input character is `i`, for example

   ```
   i 6 4 6 1 8 3 2
   ```

   The expected output is:

   ```
   4 6 1 8 3 2
   ```

3. Write a function `void swap(int* a,int* b)` that exchanges the content of location `a` and `b`. You can test it with the function `test_swap()`. It declares two integers variables with chosen values, swaps them, then outputs their content. It is run when the first input character is `s`, for example

   ```
   s 1 2
   ```

   Where numbers to be swapped are given. The expected output is:

   ```
   2 1
   ```

   Where the numbers have been swapped.

4. Write a function `int* pivot(int* begin,int* end)` that re-orders the content of a non-empty range $[begin; end)$ such that:

    1. The element initially at `begin` (called the pivot) is at the returned location `res`.
    2. All elements strictly lower than the pivot are moved to a location before `res`
    3. All elements greater or equal than the pivot are moved to a location after `res`

    For this task, this *must be done* by repeatedly:

    1. picking any leftover non-pivot element (like the one located at `begin+1`),
    2. swapping with either the pivot or the last element of the range, depending on whether the element is lower than the pivot or not, so that the chosen element is now at a correct position,
    3. then shrinking the range to exclude the now well-placed chosen element until only the pivot is left in the range. In other words, the method is to eject elements on the expected side of the range until the range is reduced to the pivot.

    In our example (markers `|` represent the range):

    `[|__4__, 6, 1, 8, 3, 2|]`

    `[|__4__, 2, 1, 8, 3|, 6]`

    `[2, |__4__, 1, 8, 3|, 6]`

    `[2, 1, |__4__, 8, 3|, 6]`

    `[2, 1, |__4__, 3|, 8, 6]`

    `[2, 1, 3, |__4__|, 8, 6]`

    You can test this function with the function `test_pivot()`. It inputs a sequence, calls the pivoting function once and outputs the result. It is run when the first input character is `p`, for example

    ```
    p 6 4 6 1 8 3 2
    ```

    The expected output is:

    ```
    4
    2 1 3 4 8
    ```

    where the first line contains the value of the pivot and the second line shows the partitioned range.

5. Write a recursive function `void quicksort(int* begin,int* end)` that sorts a range by pivoting, then recursively sorting the halves on each side of the pivot result. Make sure to correctly handle empty ranges, as well as to ensures that the range size decreases on each recursive call, as otherwise your function may not terminate.

6. Use the function `test_quicksort()` to test your implementation. The final test mode uses the functions `input`, `quicksort` and `output` to input a sequence of integer values, sort it and output the sorted sequence. It is run when the first input character is `q`, for example

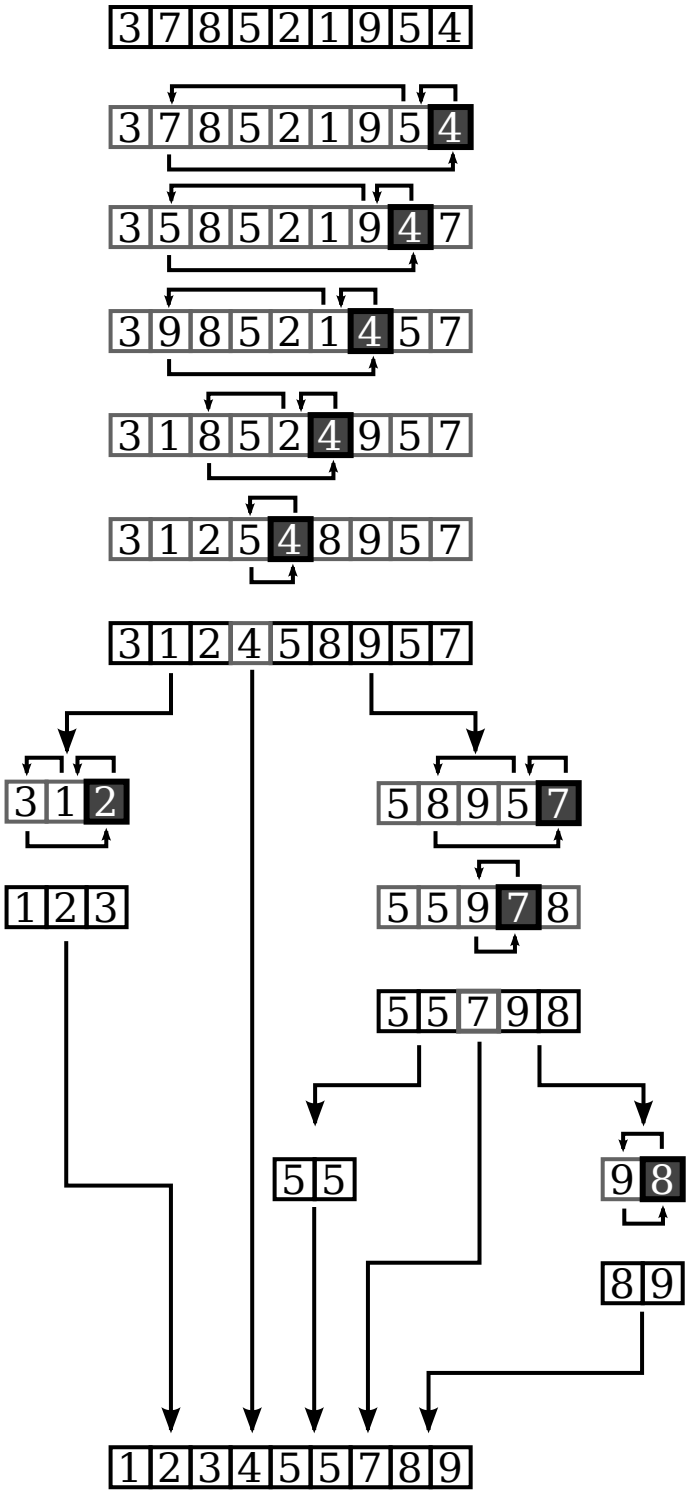
   ```
   q 6 4 6 1 8 3 2
   ```

   The expected output is:

   ```
   1 2 3 4 6 8
   ```

Full example

The following picture shows an execution of quicksort which picks the last element of the range as a pivot.


3 7 8 5 2 1 9 5 4
3 7 8 5 2 1 9 5 4
3 5 8 5 2 1 9 4 7
3 9 8 5 2 1 4 5 7
3 1 8 5 2 4 9 5 7
3 1 2 5 4 8 9 5 7
3 1 2 4 5 8 9 5 7
3 1 2
1 2 3
5 8 9 5 7
5 5 9 7 8
5 5 7 9 8
5 5
9 8
8 9
1 2 3 4 5 5 7 8 9