Informatik für Mathematiker und Physiker - AS18

Exercise 11: Classes & Pointers

Handout: 27. Nov. 2018 06:00

Due: 3. Dez. 2018 23:59

Task 1: Understanding struct & classes

Open Task (/solve/hyo7A3wK62frPRakR)

This task is a text based task. You do not need to write any program/C++ file: the answer should be written in main.txt *(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 () {
    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[128];
};
```

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

Task 2: Averager

Open Task (/solve/GhqA7odga7DqomeNd)

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 3: Understanding Pointers

Open Task (/solve/jYnWaphWFoTtxvg7K)

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

Task

Complete the following function definitions according to the specified pre- and post conditions:

1.
   ```
   // PRE:  0 <= i < vec.size()
   // POST: Returns the address of the i-th element of vec.
   int* lookup(const std::vector<int> vec &, const int i) {

   }
   ```

2.
   ```
   // PRE:  a, b, and res are valid pointers to integer values.
   // POST: integer at location res contains the result of adding t
   void add(int* res, const int* a, const int* b) {

   }
   ```

3.
   ```
   // PRE: a <= b are valid pointers to elements of the same contig
   // POST: Returns the number of elements in between those pointer
   int num_elem(const int* a, const int* b) {

   }
   ```

4.
   ```
   // PRE: str point within an allocated memory block containing a
   //      after str.
   // POST: Returns the pointer to first element after str (inclusi
   //       that is equal to ch, otherwise return 0.
   const char* first_char(const char* str, const char ch) {
       // hint: use natural iteration over str
   }
   ```

Task 4: Quick Sort

Open Task (/solve/fkzhxfJJ295moaqA7)

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. There is no need to invent a sorting algorithm, nor to write particularly efficient code.

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

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` .

1. Write a function `void` **`input`**`(std::istream& is, int*& begin, int*& end)` that read a sequence of integer values from stream `is` , and store 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) by writing a main that input a sequence and output it immediately.

3. Write a function `void` **`swap`**`(int* a,int* b)` that exchange the content of location `a` and `b` . You can test it with a program that declares two integers variables with chosen values, swap them, then output their content.

4. Write a function `int*` **`pivot`**`(int* begin,int* end)` that re-order 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 at location before `res`
   3. All elements greater or equal than the pivot are moved at location after

`res`

This is done by repeatedly

1. picking any leftover non-pivot element (like the one located at `begin+1`),
2. swapping with either the first or 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.

5. Write a recursive function `void` **`quicksort`**`(int* begin,int* end)` that sort a range by pivoting, then recursively sorting the halves on each side of the pivot result. Make sure to correctly handle empty range, as well as to ensures that the range size decrease on each recursive call, as otherwise your function may not terminate.
6. Write a program that use functions `input` , `quicksort` and `output` to input a sequence of integer value, sort it and output the sorted sequence.