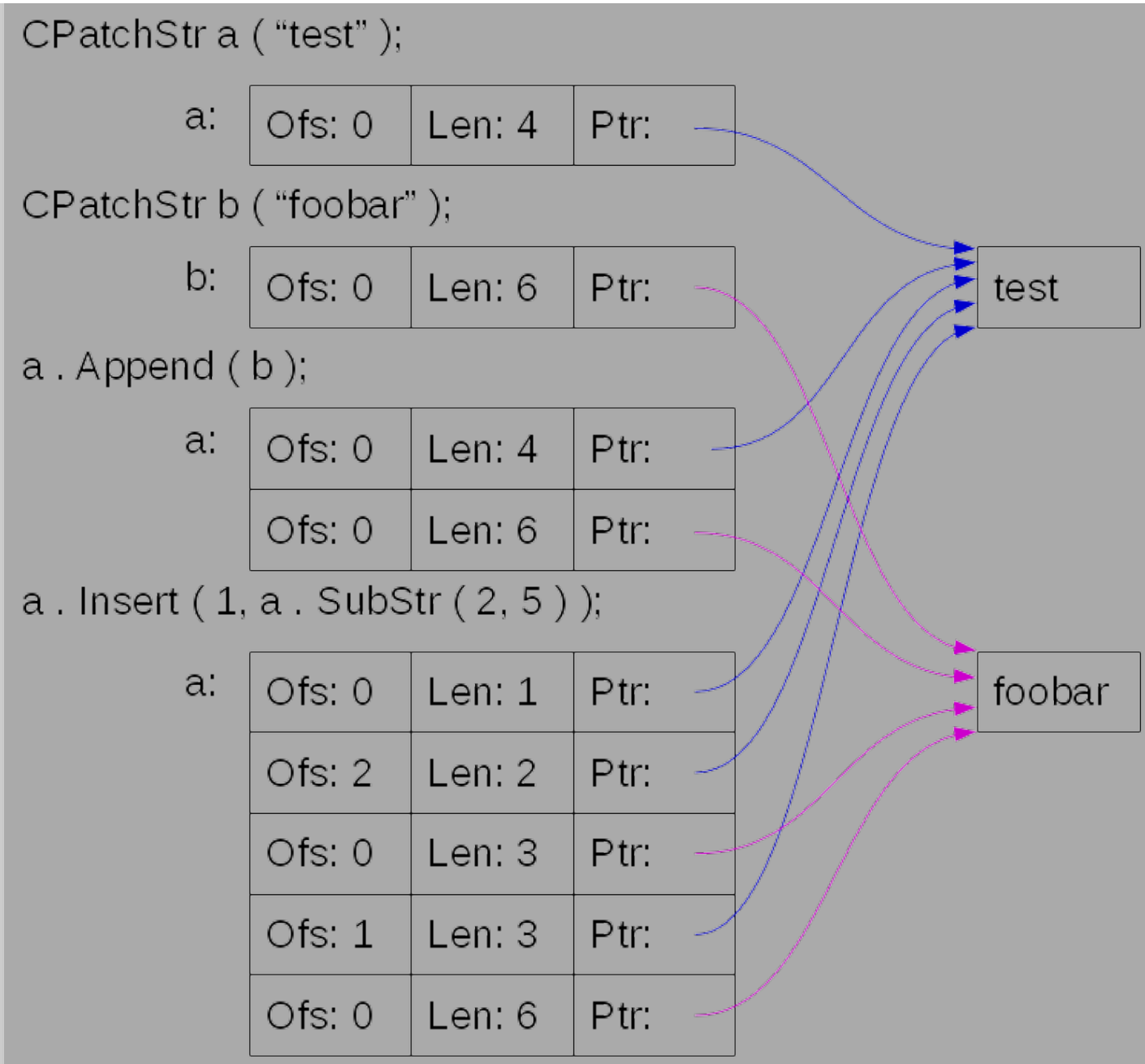
Patched strings

Submission deadline:	**2019-04-14 23:59:59**
Late submission with malus:	**2019-06-30 23:59:59** (Late submission malus: 100.0000 %)
Evaluation:	**5.0000**
Max. assessment:	**5.0000** (Without bonus points)
Submissions:	20 / 20 Free retries + 20 Penalized retries (-2 % penalty each retry)
Advices:	2 / 2 Advices for free + 2 Advices with a penalty (-10 % penalty each advice)

The task is to implement class `CPatchStr`, which simulates a string that consists of many string "patches".

Standard `std::string` represents a string as a dynamically allocated array filled with the string characters. The class is responsible for the allocations, moreover, it provides an interface with the basis string operations (concatenate, insert, delete, substring). A similar interface is required from `CPatchStr`. The difference is in the storage - the `CPatchStr` class will prefer not to store the string as an array of chars. Instead, there will exist string "patches"; an instance of `CPatchStr` will maintain a list of references to such string patches. Edit operations on `CPatchStr` will update the list of references. This may speed up the operation and conserve memory, especially with long strings.

A basic operation is an initialization where a `CPatchStr` instance is initialized with a C string. This initialization, of course, needs to store the string in some dynamically allocated array (much like `std::string`). On the other hand, suppose there are two instances of `CPatchStr`: variables `x` and `y`. We need to append `y` to the end of `x`. In the standard case, this may require to reallocate the array in `x` and subsequently copy the characters from `y` into the target array. Instead, our implementation just updates `x` with some references to the contents of `y`. No extensive copying/reallocation is needed. This implementation is possible for all edit operations (Append/Insert/Delete). If the edit operations are frequent, or if the strings are long, such implementation saves a lot of CPU power and/or RAM. The characters will be physically copied only if the instance is asked to provide its contents (there is a method in the interface).



The following interface is required in `CPatchStr` class:

`default constructor`
: initializes an empty instance (represents an empty string).

`constructor (const char *)`
: initializes a trivial instance that represents the string from parameter.

`destructor, operator =, copy constructor`
: provide the standard behavior. The automatically generated variants are not likely to be useful here.

`Append ( x )`
: add string `x` from the parameter to the end of the string `this`. Return value is a reference to itself (this), i.e., the operations may be chained ("fluent interface").

`Insert ( pos, x )`
: inserts string `x` from the parameter into the string `this`. String `x` will be inserted at position `pos`. The value of `pos` must fit into the length of string `this`, i.e., from 0 to `this` length (if `pos = this length`, then the method appends string `x` just like `Append`). If the value of `pos` exceeds the length of string `this`, the method throws `InvalidIndexException` exception. Return value is a reference to itself (this), i.e., the operations may be chained ("fluent interface").

`Delete ( from, len )`
: the method removes `len` characters from string `this` starting from index `from`. The removed characters must fit into the length of string `this`, i.e., `from + len` must be less or equal to the length of string `this`. If `from + len` is greater, the method throws `InvalidIndexException` exception. Parameter `len` = 0 is accepted, however, the method does not modify the string in this case. Return value is a reference to itself (this), i.e., the operations may be chained ("fluent interface").

`SubStr ( from, len )`
: the method creates a new instance of `CPatchStr`, the returned instance will represent `len` of the string starting from index `from`.