B3 - C++ Pool

B-CPP-300

Day 03

strings

1.1

{EPITECH.}

Day 03

binary name:	no binary
group size:	1
repository name:	cpp_d03
repository rights:	ramassage-tek
language:	C

- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).

All your exercises will be compiled with the `-W -Wall -Wextra -Werror` **flags**, unless specified otherwise.

All output goes to the standard output, and must be ended by a newline, unless specified otherwise.

None of your files must contain a `main` function, unless specified otherwise.
We will use our own `main` functions to compile and test your code.

For each exercise, the files must be turned-in in a separate directory called **exXX** where XX is the exercise number (for instance `ex01`), unless specified otherwise.

Read the examples CAREFULLY. They might require things that weren't mentioned in the subject…

If you do half the exercises because you have comprehension problems, it's okay, it happens. But if you do half the exercices because you're lazy, and leave at 2PM, you **WILL** have problems.
Do not tempt the devil.

Unit Tests

It is highly recommended to test your functions as you implement them. It is common practice to create and use what are called **unit tests**.

From now on, we expect you to write unit tests for your functions (when possible). To do so, please follow the instructions in the **"How to write Unit Tests"** document on the intranet, available here.

Create a directory named `tests`. For each of the functions you turn in, create a file in that directory named `tests-FUNCTION_NAME.c` containing all the tests needed to cover all of the exercise's possible cases (regular or irregular).

Here is a sample set of unit tests for the **my_strlen** function:

```
#include <criterion/criterion.h>

Test(my_strlen, positive_return_value)
{
        cr_assert_eq(my_strlen("toto"), 4);
}

Test(my_strlen, empty_string)
{
        cr_assert_eq(my_strlen(""), 0);
}
```

Exercise 0 - My String

Turn in: `String.h`, `String.c`

Create a `String` module.
The structure must hold a `char *str` member, and you module should contain an initialization function and a destruction function with the following prototypes:

```
void StringInit(String *this, const char *s);
void StringDestroy(String *this);
```

`StringInit` assigns `s` to the `str` member of the structure.

Exercise 1 - Assign

Turn in: `String.h`, `String.c`

Add the following member functions to your module:

```
void assign_s(String *this, const String *str);
```

sets the content of the current instance to that of `str`.

```
void assign_c(String *this, const char *s);
```

sets the content of the current instance to `s`.

Reminder: member functions can only be called from a `String` instance.

Remember to assign your function pointers.

Be careful with memory leaks.

Exercise 2 - Append

Turn in: `String.h`, `String.c`

Add the following member functions to your module:

```
void append_s(String *this, const String *ap);
```

appends the content of `ap` to that of the current instance.

```
void append_c(String *this, const char *ap);
```

appends `ap` to the content of the current instance.

Exercise 3 - At

Turn in: `String.h`, String.c‘

Add the following member function to your module:

```
char at(String *this, size_t pos);
```

returns the `char` at the `pos` position of the current instance, or -1 if the position is invalid.

Exercise 4 - Clear

Turn in: `String.h`, `String.c`

Add the following member function to your module:

```
void clear(String *this);
```

empties the content of the current instance.

Be careful with your pointers!

Exercise 5 - Size

Turn in: `String.h`, String.c'

Add the following member function to your module:

```
int size(String *this);
```

returns the size of the string, or -1 if the string pointer is `NULL`.

Exercise 6 - Compare

Turn in: `String.h`, `String.c`

Add the following member functions to your module:

```
int compare_s(String *this, const String *str);
```

compares the content of the current instance to that of `str`.
Results are the same as the `strcmp` function.

```
int compare_c(String *this, const char *str);
```

compares the content of the current instance to `str`.
Results are the same as the `strcmp` function.

Exercise 7 - Copy

Turn in: `String.h`, `String.c`

Add the following member function to your module:

```
size_t copy(String *this, char *s, size_t n, size_t pos);
```

copies `n` characters from the current instance's content, starting from the `pos` position, into `s`.
It returns the number of characters copied.

Exercise 8 - c_str

Turn in: `String.h`, `String.c`

Add the following member function to your module:

```
const char *c_str(String *this);
```

returns the buffer contained in the current instance.

Exercise 9 - empty

Turn in: `String.h`, `String.c`

Add the following member function to your module:

```
int empty(String *this);
```

returns 1 if the string is empty, -1 otherwise.

Exercise 10 - Find

Turn in: `String.h`, `String.c`

Add the following member functions to your module:

```
int find_s(String *this, const String *str, size_t pos);
```

searches for the first occurence of `str`'s content in the current instance, starting from the `pos` position.

```
int find_c(String *this, const char *str, size_t pos);
```

searches for the first occurence of `str` in the current instance, starting from the `pos` position.

These functions return the position where the occurence was found, or -1 if `str` wasn't found, if `str` is too long or if the position is invalid.

Exercise 11 - Insert

Turn in: `String.h`, `String.c`

Add the following member functions to your module:

```
void insert_c(String *this, size_t pos, const char *str);
```

copies `str` into the current instance, at the `pos` position.

```
void insert_s(String *this, size_t pos, const String *str);
```

copies the content of `str` into the current instance, at the `pos` position.

These functions enlarge the current instance.
If `pos` is greater than the size of the current instance, `str` should be appended to its content.

Be careful with null-terminating bytes…

Exercise 12 - to_int

Turn in: `String.h`, `String.c`

Add the following member function to your module:

```
int to_int(String *this);
```

returns the content of the current instance converted into an int.
It behaves like the `atoi(3)` function.

Exercise 13 - Split

Turn in: `String.h`, `String.c`

Add the following member functions to your module:

```
String *split_s(String *this, char separator);
```

returns an array of strings filled with the content of the current instance split using the `separator` delimiter.

```
char **split_c(String *this, char separator);
```

returns an array of C-style strings filled with the content of the current instance split using the `separator` delimiter.

Exercise 14 - Aff

Turn in: `String.h`, `String.c`

Add the following member function to your module:

```
void aff(String *this);
```

displays the content of the current instance to the standard output.

Be careful, I never said anything about carriage returns!

Exercise 15 - Join

Turn in: `String.h`, `String.c`

Add the following member functions to your module:

```
void join_c(String *this, char delim, const char **tab);
```

assigns a string of characters created by joining all the C-style strings in `tab`, separated by the `delim` delimiter, to the current instance.
`tab` will always be null-terminated.

```
void join_s(String *this, char delim, String *tab);
```

assigns a string of characters created by joining all the strings in `tab`, separated by the `delim` delimiter, to the current instance.
`tab` will always be terminated by an empty `String`.

Exercise 16 - Substr

Turn in: `String.h`, `String.c`

Add the following member function to your module:

```
String *substr(String *this, int offset, int length);
```

xtracts a substring of `length` characters from the current instance, starting from the `offset` position.
It returns the substring as a new `String` instance.

If `offset` is negative, it must be interpreted as the number of characters to skip starting from the end.
If `length` is negative, it represents the number of characters to be copied from the left of the offset.
If these specifications are in part out of bounds, the generated substring must be truncated to only contain parts of the current instance.