Computer Science 352 Summer 2023
Programming Assignment 6
Due 7/14/2023 by 7pm

This PA for CS 352 will require you to implement a library for string creation and several string functions. For this, you should name the files **`zstr.h`** and **`zstr.c`**. You are not required to submit an additional program that uses the library, though you definitely should write some additional code in separate file(s) to call and test your functions. You may not use and C standard library functions other than:

`printf   fprintf   malloc   calloc   free`

For the project, you should end up with the following directory / file structure:

```
pa6
├── test_zstr.c (optional)
├── makefile
├── zstr.h
└── zstr.c
```

The **`makefile`** should have at least two rules: one for **`zstr.o`** and one for **`clean`**. The **`zstr.o`** rule should compile **`zstr.c`** with the required gcc flags and produce a library file named **`zstr.o`** in the current working directory. The **`clean`** rule should delete the **`zstr.o`** file from the current working directory.

The ZStr Library

For this assignment, you must implement a C library named zstr (**`zstr.h`** for the header information, **`zstr.c`** for the implementation). The purpose of this library is to have a collection of functions that can create a zstr string, delete a cstr string, and do various operations with these strings such as concatenate, search, etc. A description of the functions / types / variables you should create for this library follows. You are welcome to additional helper functions that are "private" to the **`zstr.c`** file.

Custom Types

You should use the **`typedef`** keyword to create two custom types in the header file of this library. The first should be a **`zstr`** which should be defined as a **`char*`**. You should also define a **`zstr_code`** which is defined as an int. The **`zstr`** type will be what we use to represent a zstr when using this library, and a **`zstr_code`** will be used to specify various types of errors.

You should also define several global constants either using the **`static const`** keywords or using a preprocessor **`#define`**. These will be used as various options for a **`zstr_code`**:

- **`ZSTR_OK`** should equal **0**
- **`ZSTR_ERROR`** should equal **100**
- **`ZSTR_GREATER`** should equal **1**
- **`ZSTR_LESS`** should equal **-1**
- **`ZSTR_EQUAL`** should equal **0**

These should be put in the header file for zstr. You may use these throughout your code if / when needed. You are also welcome to add additional global consts that you deem necessary for a good implementation of your library. Though, you should only make global constants / #defines if actually necessary. You should also create a global variable named **`zstr_status`** of type **`zstr_code`**.

zstr_create

One of the most important functions for this library is **`zstr zstr_create(char* initial_data);`**. This function should allocate enough memory to hold the char array initial_data using the malloc function, store the string length and allocated size, and then return a **`zstr`**. Basically, the way that a **`zstr`** 's memory should be organized is as follows:

ALLOCATED_SPACE

`char array length (int)`	`Allocated size (int)`	`Char array data . . .`	`\0 terminator`	`Possible extra / unused space`

For any given zstr, the total **ALLOCATED_SPACE** must be either **16**, **32**, **256**, **1024**, or **2048**. Thus, for the purpose of this assignment, a **zstr** has a hard limit on the max number of characters it can contain **(2048 - sizeof(int) * 2 - 1)**. so

The “char array length” should be an int representing the number of characters actually included in the string, and the “allocated size” should be the size of the allocation (not of the valid ALLOCATED_SIZE values). Then should come the actual string. When this function is called, it should choose the smallest possible valid ALLOCATED_SIZE that can fit the string correctly.

For example, say that you called this functions like so:

```
zstr sentence;
sentence = zstr_create("abcdefghijklmnopqrstuvwxyz");
```

In this case, zstr create would have to choose size 256 because 26 characters + 1 null terminator + 4 + 4 = 35. Thus, the layout of the memory for this zstr would be:

ALLOCATED_SPACE = 256 bytes

`26`	**`256`**	**`abcdefghijklmnopqrstuvwxyz`**	**`\0`**	`221 extra bytes`

The storage space for every zstr should be created with **`malloc`**, and the function should return this zstr (char*) but what it returns should specifically point to the beginning of the actual char* data. Thus, in the example shown above, a pointer to the 'a' should be returned. This is helpful so that zstrs will still work fine with standard library functions such as printf. If there is any issue, such as the string being too big to fit in a **`zstr`**, a **`malloc`** failure, etc, it should set the **`zstr_code`** global variable to **`ZSTR_ERROR`**.

zstr_destroy

The function **`void zstr_destroy (zstr to_destroy);`** should destroy the zstr. It should do some pointer math to get a pointer to the true beginning of the allocated space (the length integer) and then call the **`free`** function.

zstr_append

The function **`void zstr_append (zstr * base, zstr to_append);`** is (arguably) the most difficult to implement correctly. This function should take a pointer to a zstr (thus, a **`char**`**) which will act as the base zstr, and a zstr containing the string content to append to the base zstr. The function should append the char array content from **`to_append`** into **`base`** (resizing **`base`** if necessary) and update the length and allocated size values as needed. This section shows two examples of how this should work, one without a needed resize, and one with.

For the first example, say that we have two very short zstrs that we want to append

```
zstr az = zstr_create("az");
zstr wa = zstr_create("wa");
zstr_append(&az, wa);
printf("%s\n", az);
```

This should print **`"azwa"`**. Since the strings are so short, no resize should be needed. After az and wa are first created, the memory should look like so, assuming that an int takes up 4 bytes as it does using gcc on lectura:

az **ALLOCATED_SPACE = 16 bytes**

2	**16**	**az**	**\0**	5 extra bytes

wa **ALLOCATED_SPACE = 16 bytes**

2	**16**	**wa**	**\0**	5 extra bytes

After concatenating, the wa variable should be unchanged, but the az zstr should be modified to be:

az **ALLOCATED_SPACE = 16 bytes**

4	**16**	**azwa**	**\0**	3 extra bytes

Now for another example say that we had these two zstrs:

```
zstr y = zstr_create("yote");
zstr g = zstr_create("giraffe");
zstr_append(&y, g);
printf("%s\n", e);
```

This code should print **`"yotegiraffe"`**. However, due to the size of the concatenated data, the function will end up having to free the *old* **`b`** zstr, and malloc a new one with size 32 instead. After **`y`** and **`g`** are first created, the memory should look like so, assuming that an int takes up 4 bytes as it does using gcc on lectura:

y **ALLOCATED_SPACE = 16 bytes**

4	**16**	**yote**	**\0**	3 extra bytes

g **ALLOCATED_SPACE = 16 bytes**

7	**16**	**giraffe**	**\0**	0 extra bytes

After concatenating, the y variable should point to a new zstr or the next size up, 32.

y **ALLOCATED_SPACE = 32 bytes**

11	**32**	**yotegiraffe**	**\0**	12 extra bytes

When creating and concatenating zstrs, you should always use the smallest size possible to fit the data, and only grow it when needed. If there is any issue, such as the string being too big to fit in a **`zstr`**, a **`malloc`** failure, etc, it should set the **`zstr_code`** global variable to **`ZSTR_ERROR`**.

zstr_index

The function **`int zstr_index(zstr base, zstr to_search);`** should search for the first occurrence of **`to_search`** within **`base`**. It should return the index if found, or -1 if not found. It should return the index based on the beginning of the actual char array.

zstr_count

The function **`int zstr_count (zstr base, zstr to_search);`** should count how many times **`to_search`** appears within **`base`**. It should return 0 if no match is found.

zstr_compare

The function **int zstr_compare (zstr x, zstr y);** should compare return **ZSTR_GREATER** if x > y, **ZSTR_EQUAL** if x == y, and **ZSTR_LESS** if x < y. The function should compare based on ascii character values, in the same way that **strcmp** does.

zstr_print_detailed

The function **void zstr_print_detailed(zstr data);** should print out a zstr, with the size and allocated space values included. For example, if this code were run:

```
zstr dna = zstr_create("DeoxyribonucleicAcid");
zstr_print_detailed(dna);
```

It should print:

```
STRLENGTH: 20
 DATASIZE: 32
   STRING: >DeoxyribonucleicAcid<
```

zstr_substring (extra credit)

The function **zstr zstr_substring (zstr base, int begin, int end);** should create a new zstr with the contents that are contained within the substring of **base** between **begin** (inclusive) and **end** (exclusive). The function should ensure that the new zstr created uses the smallest zstr size in order to fit the substring. For example, say that you have this code:

```
zstr dna = zstr_create("DeoxyribonucleicAcid");
zstr n = zstr_substring(dna, 16, 20);
```

After this code runs, the **dna** and **n** zstrs should be:

dna **ALLOCATED_SPACE = 32 bytes**

20	32	DeoxyribonucleicAcid	\0	3 extra bytes

n **ALLOCATED_SPACE = 16 bytes**

4	16	Acid	\0	3 extra bytes

If there is any issue, such as the string being too big to fit in a **zstr**, a **malloc** failure, etc, it should set the **zstr_code** global variable to **ZSTR_ERROR**.

You can receive extra credit for implementing this function.

Compiling

Since you are to write a library in this PA, your makefile should compile your .c file to a .o file. As with PA 5, you should compile the **zstr.c** file with a special flag, -c. This option tells the compiler to compile and assemble the program, but to skip the linking step. So, you should run:

```
$ gcc -Wall -Werror -std=c11 -c zstr.c
```

This should produce a file named **zstr.o**. After this file has been generated, you can compile one of your test programs using a command such as

```
$ gcc -Wall -Werror -std=c11 -o test_zstr test_zstr.c zstr.o
```

You can then run **./test_zstr** to test it out.

Memory Management

As long as the user of the library correctly calls the destroy function for the strings that are created, the code you submit should have no memory leaks. You must ensure that any allocated heap memory is correctly freed.

Submitting Your PA

After you have completed PA, double check that the file structure and file / directory names match what is shown in the project overview on the first page, ensure that your program is thoroughly tested with lectura, and that you follow the rules from the style guide. Remove all files from the pa6 directory and subdirectories other than the ones shown on the first page of this spec. This includes test files, executable files, and other file types. If you are on a mac, you should avoid zipping the file on the mac because it might include one or more hidden / extra files. Instead, zip on lectura.

Once you are ready to submit, zip up your project directory using this command:

```
$ zip -r pa6.zip ./pa6
```

Then, turn this file to the PA 6 dropbox on gradescope. There will be some test cases that will not be visible until after the grades get published.