String Handling Practice Guide (C)

---

1. Core Concepts & Building Blocks

Students should implement the functionality themselves.

1.1. Null-terminated strings

- C strings are arrays of `char` ending with `'\0'`.
- Always ensure you don't run past the terminator.

1.2. Basic operations to implement from scratch

Provide signatures and description. No full implementations.

```c
size_t my_strlen(const char *s); // return length before '\0'
int my_strcmp(const char *a, const char *b); // lexicographical compare
char *my_strcpy(char *dest, const char *src); // copy src into dest
char *my_strcat(char *dest, const char *src); // append src to dest
char *my_strchr(const char *s, int c); // find first occurrence of char
char *my_strstr(const char *haystack, const char *needle); // substring search
```

Hints:

- Loop over characters until `\0`.
- For `strcmp`, return negative/zero/positive like standard behavior.
- For `strstr`, use nested loops to test prefix matches.

---

2. Simple Examples (Skeletons)

These are small scenarios. Students should fill in the missing logic.

2.1. Reverse a string in place

Prototype:

```c
void reverse_inplace(char *s);
```

Hint: Swap characters from ends moving inward.

2.2. Check palindrome (ignore case and non-alphanumeric)

Prototype:

```c
int is_palindrome(const char *s);
```

Hints:

- Normalize characters: lowercase, skip non-alphanumeric.
- Use two pointers from front and back.

2.3. Count words in a string (words separated by whitespace)

Prototype:

```c
int count_words(const char *s);
```

Hint: Detect transitions from delimiters to non-delimiters.

2.4. Find most frequent word excluding stopwords

Description: Given a buffer of text, tokenize by spaces/punctuation, ignore a small manual list of stopwords (e.g., "the", "and", "is", "of"), and find the word with highest frequency. Prototype:

```c
char *most_frequent_word(const char *text, const char **stopwords, int
stopcount);
```

Hint: Use your own hashtable or linked list of (word, count) pairs; normalize case.

---

3. Practice Exercises (Increasing Difficulty)

Each exercise should be done with minimal helper code; students write their own utility functions.

Exercise 1: Implement all basic functions (`my_strlen`, `my_strcmp`, etc.) and write a small test harness that compares their output against the standard library equivalents.

Exercise 2: Given a CSV line like `"apple,banana,,cherry"`, write a tokenizer that returns the individual fields, handling empty fields correctly. Do not use `strtok`.

Exercise 3: Implement a simple templating replacement: given a template string like `"Hello {{name}}, today is {{day}}"` and a list of (key, value) pairs, produce the filled string. Assume placeholders are of form `{{key}}`.

Exercise 4: Case-insensitive substring search. Implement `my_strcasestr` that finds a substring ignoring case.

Exercise 5: Normalize whitespace: collapse multiple spaces/tabs/newlines into a single space and trim leading/trailing spaces.

Exercise 6: Extract all URLs from a block of text. A naive rule: look for sequences starting with `http://` or `https://` and ending at whitespace. Return them in a linked list (you will design the node structure).

Bonus Tips for Students

- Always null-terminate buffers you build manually.
- Watch out for off-by-one when indexing strings.
- Write small functions and test them before composing bigger logic.
- Use sentinel values (like a special terminator) when parsing to simplify loops.
