Almog String Manipulation

Generated by Doxygen 1.9.1

Chapter 1

File Index

1.1 File List

Here is a list of all files with brief descriptions:

Chapter 2

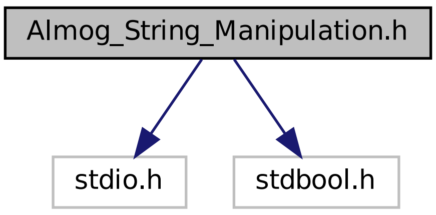
File Documentation

2.1 Almog_String_Manipulation.h File Reference

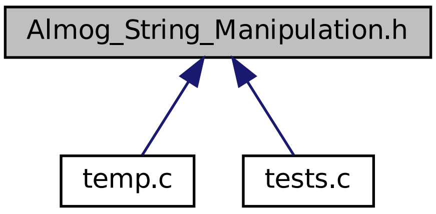
Lightweight string and line manipulation helpers.

```
#include <stdio.h>
#include <stdbool.h>
```

Include dependency graph for Almog_String_Manipulation.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define ASM_MAX_LEN (int)1e3

    Maximum number of characters processed in some string operations.
- #define asm_dprintSTRING(expr) printf(#expr " = %s\n", expr)

    Debug-print a C string expression as "expr = value\n".
- #define asm_dprintCHAR(expr) printf(#expr " = %c\n", expr)

    Debug-print a character expression as "expr = c\n".
- #define asm_dprintINT(expr) printf(#expr " = %d\n", expr)

    Debug-print an integer expression as "expr = n\n".
- #define asm_dprintFLOAT(expr) printf(#expr " = %#g\n", expr)

    Debug-print a float expression as "expr = n\n".
- #define asm_dprintDOUBLE(expr) printf(#expr " = %#g\n", expr)

    Debug-print a double expression as "expr = n\n".
- #define asm_dprintSIZE_T(expr) printf(#expr " = %zu\n", expr)

    Debug-print a size_t expression as "expr = n\n".
- #define asm_min(a, b) ((a) < (b) ? (a) : (b))

    Return the smaller of two values (macro).
- #define asm_max(a, b) ((a) > (b) ? (a) : (b))

    Return the larger of two values (macro).

Functions

- bool asm_check_char_belong_to_base (const char c, const size_t base)

    Check if a character is a valid digit in a given base.
- void asm_copy_array_by_indexes (char ∗const target, const int start, const int end, const char ∗const src)

    Copy a substring from `src` into `target` by indices and null-terminate.
- int asm_get_char_value_in_base (const char c, const size_t base)

    Convert a digit character to its numeric value in base-N.
- int asm_get_line (FILE ∗fp, char ∗const dst)

    Read a single line from a stream into a buffer.
- int asm_get_next_token_from_str (char ∗const dst, const char ∗const src, const char delimiter)

    Copy characters from the start of a string into a token buffer.
- int asm_get_token_and_cut (char ∗const dst, char ∗src, const char delimiter, const bool leave_delimiter)

    Get the next word and cut the source string at that point.
- bool asm_isalnum (char c)

    Test for an alphanumeric character (ASCII).
- bool asm_isalpha (char c)

    Test for an alphabetic character (ASCII).
- bool asm_iscntrl (char c)

    Test for a control character (ASCII).
- bool asm_isdigit (char c)

    Test for a decimal digit (ASCII).
- bool asm_isgraph (char c)

    Test for any printable character except space (ASCII).
- bool asm_islower (char c)

    Test for a lowercase letter (ASCII).
- bool asm_isprint (char c)

    Test for any printable character including space (ASCII).
- bool asm_ispunct (char c)

Test for a punctuation character (ASCII).
- bool asm_isspace (char c)

  Test for a whitespace character (ASCII).
- bool asm_isupper (char c)

  Test for an uppercase letter (ASCII).
- bool asm_isxdigit (char c)

  Test for a hexadecimal digit (lowercase or decimal).
- bool asm_isXdigit (char c)

  Test for a hexadecimal digit (uppercase or decimal).
- void asm_left_pad (char ∗const s, const size_t padding, const char pad)

  Left-pad a string in-place.
- void asm_left_shift (char ∗const s, const size_t shift)

  Shift a string left in-place by `shift` characters.
- size_t asm_length (const char ∗const str)

  Compute the length of a null-terminated C string.
- void ∗ asm_memset (void ∗const des, const unsigned char value, const size_t n)

  Set a block of memory to a repeated byte value.
- void asm_print_many_times (const char ∗const str, const size_t n)

  Print a string `n` times, then print a newline.
- void asm_remove_char_form_string (char ∗const s, const size_t index)

  Remove a single character from a string by index.
- int asm_str_in_str (const char ∗const src, const char ∗const word_to_search)

  Count occurrences of a substring within a string.
- double asm_str2double (const char ∗const s, const char ∗∗const end, const size_t base)

  Convert a string to double in the given base.
- float asm_str2float (const char ∗const s, const char ∗∗const end, const size_t base)

  Convert a string to float in the given base.
- int asm_str2int (const char ∗const s, const char ∗∗const end, const size_t base)

  Convert a string to int in the given base.
- size_t asm_str2size_t (const char ∗const s, const char ∗∗const end, const size_t base)

  Convert a string to size_t in the given base.
- void asm_strip_whitespace (char ∗const s)

  Remove all ASCII whitespace characters from a string in-place.
- bool asm_str_is_whitespace (const char ∗const s)

  Check whether a string contains only ASCII whitespace characters.
- int asm_strncat (char ∗const s1, const char ∗const s2, const int N)

  Append up to `N` characters from `s2` to the end of `s1`.
- int asm_strncmp (const char ∗s1, const char ∗s2, const int N)

  Compare up to N characters for equality (boolean result).
- void asm_tolower (char ∗const s)

  Convert all ASCII letters in a string to lowercase in-place.
- void asm_toupper (char ∗const s)

  Convert all ASCII letters in a string to uppercase in-place.

2.1.1 Detailed Description

Lightweight string and line manipulation helpers.

This single-header module provides small utilities for working with C strings:

- Reading a single line from a FILE stream
- Measuring string length
- Extracting the next token from a string using a delimiter (skipping leading ASCII whitespace)
- Cutting the extracted token (and leading whitespace) from the source buffer
- Copying a substring by indices
- Counting occurrences of a substring
- A boolean-style strncmp (returns 1 on equality, 0 otherwise)
- ASCII-only character classification helpers (isalnum, isalpha, ...)
- ASCII case conversion (toupper / tolower)
- In-place whitespace stripping and left padding
- Base-N string-to-number conversion for int, size_t, float, and double

Usage

- In exactly one translation unit, define ALMOG_STRING_MANIPULATION_IMPLEMENTATION before including this header to compile the implementation.
- In all other files, include the header without the macro to get declarations only.

Notes and limitations

- All destination buffers must be large enough; functions do not grow or allocate buffers.
- asm_get_line and asm_length enforce ASM_MAX_LEN characters (not counting the terminating '\0'). Longer lines cause an early return with an error message.
- asm_strncmp differs from the standard C strncmp: this version returns 1 if equal and 0 otherwise.
- Character classification and case-conversion helpers are ASCII-only and not locale aware.

Definition in file Almog_String_Manipulation.h.

2.1.2 Macro Definition Documentation

2.1.2.1 asm_dprintCHAR

```
#define asm_dprintCHAR(
              expr ) printf(#expr " = %c\n", expr)
```

Debug-print a character expression as "expr = c\n".

Parameters

expr	An expression that yields a character (or an int promoted from a character). The expression is evaluated exactly once.

Definition at line 82 of file Almog_String_Manipulation.h.

2.1.2.2 asm_dprintDOUBLE

```
#define asm_dprintDOUBLE(
              expr ) printf(#expr " = %#g\n", expr)
```

Debug-print a double expression as "expr = n\n".

Parameters

expr	An expression that yields a double. The expression is evaluated exactly once.

Definition at line 109 of file Almog_String_Manipulation.h.

2.1.2.3 asm_dprintFLOAT

```
#define asm_dprintFLOAT(
              expr ) printf(#expr " = %#g\n", expr)
```

Debug-print a float expression as "expr = n\n".

Parameters

expr	An expression that yields a float. The expression is evaluated exactly once.

Definition at line 100 of file Almog_String_Manipulation.h.

2.1.2.4 asm_dprintINT

```
#define asm_dprintINT(
              expr ) printf(#expr " = %d\n", expr)
```

Debug-print an integer expression as "expr = n\n".

Parameters

expr	An expression that yields an int. The expression is evaluated exactly once.

Definition at line 91 of file Almog_String_Manipulation.h.

2.1.2.5 asm_dprintSIZE_T

```
#define asm_dprintSIZE_T(
              expr ) printf(#expr " = %zu\n", expr)
```

Debug-print a size_t expression as "expr = n\n".

Parameters

expr	An expression that yields a size_t. The expression is evaluated exactly once.

Definition at line 118 of file Almog_String_Manipulation.h.

2.1.2.6 asm_dprintSTRING

```
#define asm_dprintSTRING(
              expr ) printf(#expr " = %s\n", expr)
```

Debug-print a C string expression as "expr = value\n".

Parameters

expr	An expression that yields a pointer to char (const or non-const). The expression is evaluated exactly once.

Definition at line 73 of file Almog_String_Manipulation.h.

2.1.2.7 asm_max

```
#define asm_max(
              a,
              b ) ((a) > (b) ? (a) : (b))
```

Return the larger of two values (macro).

Parameters

a	First value.
b	Second value.

Returns

The larger of `a` and `b`.

Note

Each parameter is evaluated exactly once.

Definition at line 142 of file Almog_String_Manipulation.h.

2.1.2.8 ASM_MAX_LEN

```
#define ASM_MAX_LEN (int)1e3
```

Maximum number of characters processed in some string operations.

This constant limits:

- The number of characters read by asm_get_line from a stream (excluding the terminating null byte).
- The maximum number of characters inspected by asm_length.

If asm_get_line reads more than ASM_MAX_LEN characters before encountering '
' or EOF, it prints an error to stderr and returns -1. In that error case, the contents of the destination buffer are not guaranteed to be null-terminated.

Definition at line 63 of file Almog_String_Manipulation.h.

2.1.2.9 asm_min

```
#define asm_min(
              a,
              b ) ((a) < (b) ? (a) : (b))
```

Return the smaller of two values (macro).

Parameters

a	First value.
b	Second value.

Returns

The smaller of `a` and `b`.

Note

Each parameter is evaluated exactly once.

Definition at line 130 of file Almog_String_Manipulation.h.

2.1.3 Function Documentation

2.1.3.1 asm_check_char_belong_to_base()

```
bool asm_check_char_belong_to_base (
            const char c,
            const size_t base )
```

Check if a character is a valid digit in a given base.

Parameters

c	Character to test (e.g., '0'–'9', 'a'–'z', 'A'–'Z').
base	Numeric base in the range [2, 36].

Returns

true if `c` is a valid digit for `base`, false otherwise.

Note

If `base` is outside [2, 36], an error is printed to stderr and false is returned.

Definition at line 195 of file Almog_String_Manipulation.h.

References asm_isdigit().

Referenced by asm_get_char_value_in_base(), asm_str2double(), asm_str2float(), asm_str2int(), asm_str2size_t(), and test_base_digit_helpers().

2.1.3.2 asm_copy_array_by_indexes()

```
void asm_copy_array_by_indexes (
            char *const target,
            const int start,
            const int end,
            const char *const src )
```

Copy a substring from `src` into `target` by indices and null-terminate.

Copies characters with indices i = start, start + 1, ..., end from `src` into `target` (note: `end` is inclusive in this implementation), then ensures `target` is null-terminated.

Parameters

target	Destination buffer. Must be large enough to hold (end - start + 1) characters plus the null terminator.
start	Inclusive start index within `src` (0-based).
end	Inclusive end index within `src` (must satisfy end >= start).
src	Source string buffer.

Warning

No bounds checking is performed. The caller must ensure valid indices and sufficient target capacity.

Definition at line 230 of file Almog_String_Manipulation.h.

Referenced by main(), and test_copy_array_by_indexes_behavior_and_bounds().

2.1.3.3 asm_get_char_value_in_base()

```
int asm_get_char_value_in_base (
            const char c,
            const size_t base )
```

Convert a digit character to its numeric value in base-N.

Parameters

c	Digit character ('0'–'9', 'a'–'z', 'A'–'Z').
base	Numeric base in the range [2, 36] (used for validation).

Returns

The numeric value of `c` in the range [0, 35].

Note

This function assumes `c` is a valid digit character. Call asm_check_char_belong_to_base() first if validation is needed.

Definition at line 253 of file Almog_String_Manipulation.h.

References asm_check_char_belong_to_base(), asm_isdigit(), and asm_isupper().

Referenced by asm_str2double(), asm_str2float(), asm_str2int(), asm_str2size_t(), and test_base_digit_helpers().

2.1.3.4 asm_get_line()

```
int asm_get_line (
            FILE * fp,
            char *const dst )
```

Read a single line from a stream into a buffer.

Reads characters from the FILE stream until a newline ('
') or EOF is encountered. The newline, if present, is not copied. The result is always null-terminated on normal (non-error) completion.

Parameters

fp	Input stream (must be non-NULL).
dst	Destination buffer. Must have capacity of at least ASM_MAX_LEN + 1 bytes.

Returns

Number of characters stored in `dst` (excluding the terminating null byte).

Return values

-1	EOF was encountered before any character was read, or the line exceeded ASM_MAX_LEN characters (error).

Note

If the line exceeds ASM_MAX_LEN characters before a newline or EOF is seen, the function prints an error message to stderr and returns -1. In that case, `dst` is not guaranteed to be null-terminated.

An empty line (just '
') returns 0 (not -1).

Definition at line 285 of file Almog_String_Manipulation.h.

References ASM_MAX_LEN.

Referenced by test_get_line_tmpfile(), and test_get_line_too_long().

2.1.3.5 asm_get_next_token_from_str()

```
int asm_get_next_token_from_str (
            char *const dst,
            const char *const src,
            const char delimiter )
```

Copy characters from the start of a string into a token buffer.

Copies characters from `src` into `dst` until one of the following is encountered in `src`:

- the delimiter character,
- or the string terminator ('\0').

The delimiter (if present) is not copied into `dst`. The resulting token in `dst` is always null-terminated.

Parameters

dst	Destination buffer for the extracted token. Must be large enough to hold the token plus the null terminator.
src	Source C string to parse (not modified by this function).
delimiter	Delimiter character to stop at.

Returns

The number of characters copied into `dst` (excluding the null terminator). This is also the index in `src` of the delimiter or '\0' that stopped the copy.

Note

This function does not skip leading whitespace and does not treat newline ('
') specially; newlines are copied like any other character.

If `src` starts with `delimiter` or '\0', an empty token is produced (`dst` becomes ""), and 0 is returned.

Definition at line 331 of file Almog_String_Manipulation.h.

Referenced by asm_get_token_and_cut(), and test_get_next_word_from_line_current_behavior().

2.1.3.6 asm_get_token_and_cut()

```
int asm_get_token_and_cut (
             char *const dst,
             char * src,
             const char delimiter,
             const bool leave_delimiter )
```

Get the next word and cut the source string at that point.

Extracts the next word from `src` (per asm_get_next_word_from_line semantics) into `dst`. On success, `src` is modified in-place to remove the consumed prefix.

If `leave_delimiter` is true, the new `src` begins at the delimiter character. If false, the delimiter is skipped and the new `src` begins right after it.

Example (leave_delimiter == true):
```
char src[] = "abc,def";
char word[4];
asm_get_word_and_cut(word, src, ',', true);
// word == "abc"
// src  == ",def"
```

Parameters

dst	Destination buffer for the extracted word (large enough for the token and terminating null).
src	Source buffer. Modified in-place if a word is found.
delimiter	Delimiter character to stop at.
leave_delimiter	If true, the delimiter remains at the start of the updated `src`; if false, it is removed as well.

Returns

1 if a non-empty token was extracted into `dst`, 0 otherwise.

Note

Even when this function returns 0, it may still modify `src` if:

- leading whitespace was consumed, and/or
- `leave_delimiter` is false and the delimiter was the first non-whitespace character.

Definition at line 377 of file Almog_String_Manipulation.h.

References asm_get_next_token_from_str(), and asm_left_shift().

Referenced by main(), and test_get_word_and_cut_edges().

2.1.3.7 asm_isalnum()

```
bool asm_isalnum (
            char c )
```

Test for an alphanumeric character (ASCII).

Parameters

c	Character to test.

Returns

true if `c` is '0'–'9', 'A'–'Z', or 'a'–'z'; false otherwise.

Definition at line 395 of file Almog_String_Manipulation.h.

References asm_isalpha(), and asm_isdigit().

Referenced by test_ascii_classification_exhaustive_ranges(), and test_ascii_classification_full_scan_0_127().

2.1.3.8 asm_isalpha()

```
bool asm_isalpha (
            char c )
```

Test for an alphabetic character (ASCII).

Parameters

c	Character to test.

Returns

true if `c` is 'A'–'Z' or 'a'–'z'; false otherwise.

Definition at line 406 of file Almog_String_Manipulation.h.

References asm_islower(), and asm_isupper().

Referenced by asm_isalnum(), test_ascii_classification_exhaustive_ranges(), and test_ascii_classification_full_scan_0_127().

2.1.3.9 asm_iscntrl()

```
bool asm_iscntrl (
            char c )
```

Test for a control character (ASCII).

Parameters

c	Character to test.

Returns

true if `c` is in the range [0, 31] or 127; false otherwise.

Definition at line 417 of file Almog_String_Manipulation.h.

Referenced by test_ascii_classification_exhaustive_ranges().

2.1.3.10 asm_isdigit()

```
bool asm_isdigit (
            char c )
```

Test for a decimal digit (ASCII).

Parameters

c	Character to test.

Returns

true if `c` is '0'–'9'; false otherwise.

Definition at line 432 of file Almog_String_Manipulation.h.

Referenced by asm_check_char_belong_to_base(), asm_get_char_value_in_base(), asm_isalnum(), asm_isxdigit(), asm_isXdigit(), test_ascii_classification_exhaustive_ranges(), and test_ascii_classification_full_scan_0_127().

2.1.3.11 asm_isgraph()

```
bool asm_isgraph (
            char c )
```

Test for any printable character except space (ASCII).

Parameters

c	Character to test.

Returns

true if `c` is in the range [33, 126]; false otherwise.

Definition at line 447 of file Almog_String_Manipulation.h.

Referenced by asm_isprint(), test_ascii_classification_exhaustive_ranges(), and test_ascii_classification_full_scan_0_127().

2.1.3.12 asm_islower()

```
bool asm_islower (
            char c )
```

Test for a lowercase letter (ASCII).

Parameters

c	Character to test.

Returns

true if `c` is 'a'–'z'; false otherwise.

Definition at line 462 of file Almog_String_Manipulation.h.

Referenced by asm_isalpha(), asm_toupper(), test_ascii_classification_exhaustive_ranges(), and test_ascii_classification_full_scan_0

2.1.3.13 asm_isprint()

```
bool asm_isprint (
            char c )
```

Test for any printable character including space (ASCII).

Parameters

c	Character to test.

Returns

true if c is space (' ') or asm_isgraph(c) is true; false otherwise.

Definition at line 478 of file Almog_String_Manipulation.h.

References asm_isgraph().

Referenced by test_ascii_classification_exhaustive_ranges(), and test_ascii_classification_full_scan_0_127().

2.1.3.14 asm_ispunct()

```
bool asm_ispunct (
            char c )
```

Test for a punctuation character (ASCII).

Parameters

c	Character to test.

Returns

true if c is a printable, non-alphanumeric, non-space character; false otherwise.

Definition at line 490 of file Almog_String_Manipulation.h.

Referenced by test_ascii_classification_exhaustive_ranges().

2.1.3.15 asm_isspace()

```
bool asm_isspace (
            char c )
```

Test for a whitespace character (ASCII).

Parameters

c	Character to test.

Returns

true if `c` is one of ' ', '
', '\t', '\v', '\f', or '\r'; false otherwise.

Definition at line 506 of file Almog_String_Manipulation.h.

Referenced by asm_str2double(), asm_str2float(), asm_str2int(), asm_str2size_t(), asm_str_is_whitespace(), asm_strip_whitespace(), test_ascii_classification_exhaustive_ranges(), and test_strip_whitespace_properties().

2.1.3.16 asm_isupper()

```
bool asm_isupper (
            char c )
```

Test for an uppercase letter (ASCII).

Parameters

c	Character to test.

Returns

true if `c` is 'A'–'Z'; false otherwise.

Definition at line 522 of file Almog_String_Manipulation.h.

Referenced by asm_get_char_value_in_base(), asm_isalpha(), asm_tolower(), test_ascii_classification_exhaustive_ranges(), and test_ascii_classification_full_scan_0_127().

2.1.3.17 asm_isxdigit()

```
bool asm_isxdigit (
            char c )
```

Test for a hexadecimal digit (lowercase or decimal).

Parameters

c	Character to test.

Returns

true if `c` is '0'–'9' or 'a'–'f'; false otherwise.

Definition at line 537 of file Almog_String_Manipulation.h.

References asm_isdigit().

Referenced by test_ascii_classification_exhaustive_ranges().

2.1.3.18 asm_isXdigit()

```
bool asm_isXdigit (
             char c )
```

Test for a hexadecimal digit (uppercase or decimal).

Parameters

c	Character to test.

Returns

true if `c` is '0'–'9' or 'A'–'F'; false otherwise.

Definition at line 552 of file Almog_String_Manipulation.h.

References asm_isdigit().

Referenced by test_ascii_classification_exhaustive_ranges().

2.1.3.19 asm_left_pad()

```
void asm_left_pad (
             char *const s,
             const size_t padding,
             const char pad )
```

Left-pad a string in-place.

Shifts the contents of `s` to the right by `padding` positions and fills the vacated leading positions with `pad`.

Parameters

s	String to pad. Modified in-place.
padding	Number of leading spaces to insert.
pad	The padding character to insert.

Warning

The buffer backing `s` must have enough capacity for the original string length plus `padding` and the terminating null byte. No bounds checking is performed.

Definition at line 575 of file Almog_String_Manipulation.h.

References asm_length().

Referenced by main(), and test_left_pad_edges_and_sentinel().

2.1.3.20 asm_left_shift()

```
void asm_left_shift (
             char *const s,
             const size_t shift )
```

Shift a string left in-place by `shift` characters.

Removes the first `shift` characters from `s` by moving the remaining characters to the front. The resulting string is always null-terminated.

Parameters

s	String to modify in-place. Must be null-terminated.
shift	Number of characters to remove from the front.

Note

If `shift` is 0, `s` is unchanged.

If `shift` is greater than or equal to the string length, `s` becomes the empty string.

Definition at line 599 of file Almog_String_Manipulation.h.

References asm_length().

Referenced by asm_get_token_and_cut(), and test_left_shift_edges().

2.1.3.21 asm_length()

```
size_t asm_length (
             const char *const str )
```

Compute the length of a null-terminated C string.

Parameters

str	Null-terminated string (must be non-NULL).

Returns

The number of characters before the terminating null byte.

Note

If more than ASM_MAX_LEN characters are scanned without encountering a null terminator, an error is printed to stderr and **SIZE_MAX** is returned.

Definition at line 626 of file Almog_String_Manipulation.h.

References ASM_MAX_LEN.

Referenced by asm_left_pad(), asm_left_shift(), asm_remove_char_form_string(), asm_str_in_str(), asm_str_is_whitespace(), asm_strip_whitespace(), asm_strncat(), asm_tolower(), asm_toupper(), main(), and test_length_matches_strlen_small().

2.1.3.22 asm_memset()

```
void * asm_memset (
            void *const des,
            const unsigned char value,
            const size_t n )
```

Set a block of memory to a repeated byte value.

Writes `value` into each of the first `n` bytes of the memory region pointed to by `des`. This function mirrors the behavior of the standard C memset(), but implements it using a simple byte-wise loop.

Parameters

des	Destination memory block to modify. Must point to a valid buffer of at least `n` bytes.
value	Unsigned byte value to store repeatedly.
n	Number of bytes to set.

Returns

The original pointer `des`.

Note

This implementation performs no optimizations (such as word-sized writes); the memory block is filled one byte at a time.

Behavior is undefined if `des` overlaps with invalid or non-writable memory.

Definition at line 661 of file Almog_String_Manipulation.h.

Referenced by test_memset_basic_and_edges().

2.1.3.23 asm_print_many_times()

```
void asm_print_many_times (
             const char *const str,
             const size_t n )
```

Print a string `n` times, then print a newline.

Parameters

str	String to print (as-is with printf("%s", ...)).
n	Number of times to print `str`.

Definition at line 676 of file Almog_String_Manipulation.h.

2.1.3.24 asm_remove_char_form_string()

```
void asm_remove_char_form_string (
             char *const s,
             const size_t index )
```

Remove a single character from a string by index.

Deletes the character at position `index` from `s` by shifting subsequent characters one position to the left.

Parameters

s	String to modify in-place. Must be null-terminated.
index	Zero-based index of the character to remove.

Note

If `index` is out of range, an error is printed to stderr and the string is left unchanged.

Definition at line 696 of file Almog_String_Manipulation.h.

References asm_length().

Referenced by asm_strip_whitespace(), and test_remove_char_form_string_edges().

2.1.3.25 asm_str2double()

```
double asm_str2double (
             const char *const s,
             const char **const end,
             const size_t base )
```

Convert a string to double in the given base.

Parses an optional sign, then a sequence of base-N digits, and optionally a fractional part separated by a '.' character.

Parameters

s	String to convert. Leading ASCII whitespace is skipped.
end	If non-NULL, ∗end is set to point to the first character not used in the conversion.
base	Numeric base in the range [2, 36].

Returns

The converted double value. Returns 0.0 on invalid base.

Note

Only digits '0'–'9', 'a'–'z', and 'A'–'Z' are recognized as base-N digits. No exponent notation (e.g., 'e' or 'p') is supported.

On invalid base, an error is printed to stderr, ∗end (if non-NULL) is set to `s`, and 0.0 is returned.

Definition at line 756 of file Almog_String_Manipulation.h.

References asm_check_char_belong_to_base(), asm_get_char_value_in_base(), and asm_isspace().

Referenced by test_str2float_double().

2.1.3.26 asm_str2float()

```
float asm_str2float (
             const char *const s,
             const char **const end,
             const size_t base )
```

Convert a string to float in the given base.

Identical to asm_str2double semantically, but returns a float and uses float arithmetic for the fractional part.

Parameters

s	String to convert. Leading ASCII whitespace is skipped.
end	If non-NULL, ∗end is set to point to the first character not used in the conversion.
base	Numeric base in the range [2, 36].

Returns

The converted float value. Returns 0.0f on invalid base.

Note

Only digits '0'–'9', 'a'–'z', and 'A'–'Z' are recognized as base-N digits. No exponent notation (e.g., 'e' or 'p') is supported.

On invalid base, an error is printed to stderr, ∗end (if non-NULL) is set to `s`, and 0.0f is returned.

Definition at line 818 of file Almog_String_Manipulation.h.

References asm_check_char_belong_to_base(), asm_get_char_value_in_base(), and asm_isspace().

Referenced by test_str2float_double().

2.1.3.27 asm_str2int()

```
int asm_str2int (
            const char *const s,
            const char **const end,
            const size_t base )
```

Convert a string to int in the given base.

Parses an optional sign and then a sequence of base-N digits.

Parameters

s	String to convert. Leading ASCII whitespace is skipped.
end	If non-NULL, ∗end is set to point to the first character not used in the conversion.
base	Numeric base in the range [2, 36].

Returns

The converted int value. Returns 0 on invalid base.

Note

Only digits '0'–'9', 'a'–'z', and 'A'–'Z' are recognized as base-N digits.

On invalid base, an error is printed to stderr, ∗end (if non-NULL) is set to `s`, and 0 is returned.

Definition at line 877 of file Almog_String_Manipulation.h.

References asm_check_char_belong_to_base(), asm_get_char_value_in_base(), and asm_isspace().

Referenced by test_str2int().

2.1.3.28 asm_str2size_t()

```
size_t asm_str2size_t (
             const char *const s,
             const char **const end,
             const size_t base )
```

Convert a string to size_t in the given base.

Parses an optional leading '+' sign, then a sequence of base-N digits. Negative numbers are rejected.

Parameters

s	String to convert. Leading ASCII whitespace is skipped.
end	If non-NULL, ∗end is set to point to the first character not used in the conversion.
base	Numeric base in the range [2, 36].

Returns

The converted size_t value. Returns 0 on invalid base or if a negative sign is encountered.

Note

On invalid base or a negative sign, an error is printed to stderr, ∗end (if non-NULL) is set to `s`, and 0 is returned.

Definition at line 922 of file Almog_String_Manipulation.h.

References asm_check_char_belong_to_base(), asm_get_char_value_in_base(), and asm_isspace().

Referenced by test_str2size_t().

2.1.3.29 asm_str_in_str()

```
int asm_str_in_str (
             const char *const src,
             const char *const word_to_search )
```

Count occurrences of a substring within a string.

Counts how many times `word_to_search` appears in `src`. Occurrences may overlap.

Parameters

src	The string to search in (must be null-terminated).
word_to_search	The substring to find (must be null-terminated and non-empty).

Returns

The number of (possibly overlapping) occurrences found.

Note

If `word_to_search` is the empty string, the behavior is not well-defined and should be avoided.

Definition at line 726 of file Almog_String_Manipulation.h.

References asm_length(), and asm_strncmp().

Referenced by test_str_in_str_overlap_and_edges().

2.1.3.30 asm_str_is_whitespace()

```
bool asm_str_is_whitespace (
             const char *const s )
```

Check whether a string contains only ASCII whitespace characters.

Parameters

s	Null-terminated string to test.

Returns

true if every character in `s` satisfies asm_isspace(), or if `s` is the empty string; false otherwise.

Definition at line 990 of file Almog_String_Manipulation.h.

References asm_isspace(), and asm_length().

Referenced by test_str_is_whitespace_edges().

2.1.3.31 asm_strip_whitespace()

```
void asm_strip_whitespace (
             char *const s )
```

Remove all ASCII whitespace characters from a string in-place.

Scans `s` and deletes all characters for which asm_isspace() is true, compacting the string and preserving the original order of non-whitespace characters.

Parameters

s	String to modify in-place. Must be null-terminated.

Definition at line 969 of file Almog_String_Manipulation.h.

References asm_isspace(), asm_length(), and asm_remove_char_form_string().

Referenced by test_strip_whitespace_properties().

2.1.3.32 asm_strncat()

```
int asm_strncat (
            char *const s1,
            const char *const s2,
            const int N )
```

Append up to `N` characters from `s2` to the end of `s1`.

Appends characters from `s2` to the end of `s1` until either:

- `N` characters were appended, or
- a '\0' is encountered in `s2`.

After appending, this implementation writes a terminating '\0' to `s1`.

Parameters

s1	Destination string buffer (must be null-terminated).
s2	Source string buffer (must be null-terminated).
N	Maximum number of characters to append. If N == 0, the limit defaults to ASM_MAX_LEN.

Returns

The number of characters appended to `s1`.

Warning

This function uses ASM_MAX_LEN as an upper bound for the resulting length (excluding the terminating '\0'). The caller must ensure `s1` has capacity of at least ASM_MAX_LEN bytes.

Definition at line 1021 of file Almog_String_Manipulation.h.

References asm_length(), and ASM_MAX_LEN.

Referenced by test_strncat_current_behavior_and_sentinel().

2.1.3.33 asm_strncmp()

```
int asm_strncmp (
            const char * s1,
            const char * s2,
            const int N )
```

Compare up to N characters for equality (boolean result).

Returns 1 if the first N characters of s1 and s2 are all equal; otherwise returns 0. Unlike the standard C strncmp, which returns 0 on equality and a non-zero value on inequality/order, this function returns a boolean-like result (1 == equal, 0 == different).

Parameters

s1	First string (may be shorter than N).
s2	Second string (may be shorter than N).
N	Number of characters to compare.

Returns

1 if equal for the first N characters, 0 otherwise.

Note

If either string ends before N characters and the other does not, the strings are considered different.

Definition at line 1061 of file Almog_String_Manipulation.h.

Referenced by asm_str_in_str(), and test_strncmp_boolean_edges().

2.1.3.34 asm_tolower()

```
void asm_tolower (
            char *const s )
```

Convert all ASCII letters in a string to lowercase in-place.

Parameters

s	String to modify in-place. Must be null-terminated.

Definition at line 1081 of file Almog_String_Manipulation.h.

References asm_isupper(), and asm_length().

Referenced by test_case_conversion_roundtrip().

2.1.3.35 asm_toupper()

```
void asm_toupper (
              char *const s )
```

Convert all ASCII letters in a string to uppercase in-place.

Parameters

s	String to modify in-place. Must be null-terminated.

Definition at line 1096 of file Almog_String_Manipulation.h.

References asm_islower(), and asm_length().

Referenced by test_case_conversion_roundtrip().

2.2 Almog_String_Manipulation.h

```

#ifndef ALMOG_STRING_MANIPULATION_H_
#define ALMOG_STRING_MANIPULATION_H_

#include <stdio.h>
#include <stdbool.h>

#ifndef ASM_MAX_LEN
#define ASM_MAX_LEN (int)1e3
#endif

#define asm_dprintSTRING(expr) printf(#expr " = %s\n", expr)

#define asm_dprintCHAR(expr) printf(#expr " = %c\n", expr)

#define asm_dprintINT(expr) printf(#expr " = %d\n", expr)

#define asm_dprintFLOAT(expr) printf(#expr " = %#g\n", expr)

#define asm_dprintDOUBLE(expr) printf(#expr " = %#g\n", expr)

#define asm_dprintSIZE_T(expr) printf(#expr " = %zu\n", expr)

#define asm_min(a, b) ((a) < (b) ? (a) : (b))

#define asm_max(a, b) ((a) > (b) ? (a) : (b))

bool    asm_check_char_belong_to_base(const char c, const size_t base);
void    asm_copy_array_by_indexes(char * const target, const int start, const int end, const char *
  const src);
int     asm_get_char_value_in_base(const char c, const size_t base);
int     asm_get_line(FILE *fp, char * const dst);
int     asm_get_next_token_from_str(char * const dst, const char * const src, const char delimiter);
int     asm_get_token_and_cut(char * const dst, char *src, const char delimiter, const bool
  leave_delimiter);
bool    asm_isalnum(const char c);
bool    asm_isalpha(const char c);
bool    asm_iscntrl(const char c);
bool    asm_isdigit(const char c);
bool    asm_isgraph(const char c);
bool    asm_islower(const char c);
bool    asm_isprint(const char c);
bool    asm_ispunct(const char c);
bool    asm_isspace(const char c);
bool    asm_isupper(const char c);
bool    asm_isxdigit(const char c);
bool    asm_isXdigit(const char c);
void    asm_left_pad(char * const s, const size_t padding, const char pad);
void    asm_left_shift(char * const s, const size_t shift);
size_t  asm_length(const char * const str);
void *  asm_memset(void * const des, const unsigned char value, const size_t n);
void    asm_print_many_times(const char * const str, const size_t n);
void    asm_remove_char_form_string(char * const s, const size_t index);
int     asm_str_in_str(const char * const src, const char * const word_to_search);
```

```
double  asm_str2double(const char * const s, const char ** const end, const size_t base);
float   asm_str2float(const char * const s, const char ** const end, const size_t base);
int     asm_str2int(const char * const s, const char ** const end, const size_t base);
size_t  asm_str2size_t(const char * const s, const char ** const end, const size_t base);
void    asm_strip_whitespace(char * const s);
bool    asm_str_is_whitespace(const char * const s);
int     asm_strncat(char * const s1, const char * const s2, const int N);
int     asm_strncmp(const char * const s1, const char * const s2, const int N);
void    asm_tolower(char * const s);
void    asm_toupper(char * const s);

#endif /*ALMOG_STRING_MANIPULATION_H_*/

#ifdef ALMOG_STRING_MANIPULATION_IMPLEMENTATION
#undef ALMOG_STRING_MANIPULATION_IMPLEMENTATION

bool asm_check_char_belong_to_base(const char c, const size_t base)
{
    if (base > 36 || base < 2) {
        #ifndef NO_ERRORS
        fprintf(stderr, "%s:%d:\n%s:\n[Error] Supported bases are [2...36]. Inputted: %zu\n\n", __FILE__, __LINE__, __func__, base);
        #endif
        return false;
    }
    if (base <= 10) {
        return c >= '0' && c <= '9'+(char)base-10;
    }
    if (base > 10) {
        return asm_isdigit(c) || (c >= 'A' && c <= ('A'+(char)base-11)) || (c >= 'a' && c <= ('a'+(char)base-11));
    }

    return false;
}

void asm_copy_array_by_indexes(char * const target, const int start, const int end, const char * const src)
{
    if (start > end) return;
    int j = 0;
    for (int i = start; i <= end; i++) {
        target[j] = src[i];
        j++;
    }
    if (target[j-1] != '\0') {
        target[j] = '\0';
    }
}

int asm_get_char_value_in_base(const char c, const size_t base)
{
    if (!asm_check_char_belong_to_base(c, base)) return -1;
    if (asm_isdigit(c)) {
        return c - '0';
    } else if (asm_isupper(c)) {
        return c - 'A' + 10;
    } else {
        return c - 'a' + 10;
    }
}

int asm_get_line(FILE *fp, char * const dst)
{
    int i = 0;
    int c;
    while ((c = fgetc(fp)) != '\n' && c != EOF) {
        dst[i++] = c;
        if (i >= ASM_MAX_LEN) {
            #ifndef NO_ERRORS
            fprintf(stderr, "%s:%d:\nIn function '%s':\n[Error] index exceeds ASM_MAX_LEN. Line in file is too long.\n\n", __FILE__, __LINE__, __func__);
            #endif
            return -1;
        }
    }
    dst[i] = '\0';
    if (c == EOF && i == 0) {
        return -1;
    }
    return i;
}

int asm_get_next_token_from_str(char * const dst, const char * const src, const char delimiter)
{
    int i = 0, j = 0;
    char c;
    while ((c = src[i]) != delimiter && c != '\0') {
        dst[j++] = src[i++];
    }

    dst[j] = '\0';

    return j;
}

int asm_get_token_and_cut(char * const dst, char *src, const char delimiter, const bool
 leave_delimiter)
{
    int new_src_start_index = asm_get_next_token_from_str(dst, src, delimiter);

    if (leave_delimiter) {
        asm_left_shift(src, new_src_start_index);
    } else {
        asm_left_shift(src, new_src_start_index + 1);
    }
    return new_src_start_index ? 1 : 0;
}

bool asm_isalnum(char c)
{
    return asm_isalpha(c) || asm_isdigit(c);
}

bool asm_isalpha(char c)
{
    return asm_isupper(c) || asm_islower(c);
}

bool asm_iscntrl(char c)
{
    if ((c >= 0 && c <= 31) || c == 127) {
        return true;
    } else {
        return false;
    }
}

bool asm_isdigit(char c)
{
    if (c >= '0' && c <= '9') {
        return true;
    } else {
        return false;
    }
}

bool asm_isgraph(char c)
{
    if (c >= 33 && c <= 126) {
        return true;
    } else {
        return false;
    }
}

bool asm_islower(char c)
{
    if (c >= 'a' && c <= 'z') {
        return true;
    } else {
        return false;
    }
}

bool asm_isprint(char c)
{
    return asm_isgraph(c) || c == ' ';
}

bool asm_ispunct(char c)
{
    if ((c >= 33 && c <= 47) || (c >= 58 && c <= 64) || (c >= 91 && c <= 96) || (c >= 123 && c <=
 126)) {
        return true;
    } else {
        return false;
    }
}

bool asm_isspace(char c)
{
    if (c == ' ' || c == '\n' || c == '\t' ||
        c == '\v'|| c == '\f' || c == '\r') {
```

```
        return true;
    } else {
        return false;
    }
}

bool asm_isupper(char c)
{
    if (c >= 'A' && c <= 'Z') {
        return true;
    } else {
        return false;
    }
}

bool asm_isxdigit(char c)
{
    if ((c >= 'a' && c <= 'f') || asm_isdigit(c)) {
        return true;
    } else {
        return false;
    }
}

bool asm_isXdigit(char c)
{
    if ((c >= 'A' && c <= 'F') || asm_isdigit(c)) {
        return true;
    } else {
        return false;
    }
}

void asm_left_pad(char * const s, const size_t padding, const char pad)
{
    int len = (int)asm_length(s);
    for (int i = len; i >= 0; i--) {
        s[i+(int)padding] = s[i];
    }
    for (int i = 0; i < (int)padding; i++) {
        s[i] = pad;
    }
}

void asm_left_shift(char * const s, const size_t shift)
{
    size_t len = asm_length(s);

    if (shift == 0) return;
    if (len <= shift) {
        s[0] = '\0';
        return;
    }

    size_t i;
    for (i = shift; i < len; i++) {
        s[i-shift] = s[i];
    }
    s[i-shift] = '\0';
}

size_t asm_length(const char * const str)
{
    char c;
    size_t i = 0;

    while ((c = str[i++]) != '\0') {
        if (i > ASM_MAX_LEN) {
            #ifndef NO_ERRORS
            fprintf(stderr, "%s:%d:\n%s:\n[Error] index exceeds ASM_MAX_LEN_LINE. Probably no NULL termination.\n\n", __FILE__, __LINE__, __func__);
            #endif
            return __SIZE_MAX__;
        }
    }
    return --i;
}

void * asm_memset(void * const des, const unsigned char value, const size_t n)
{
    unsigned char *ptr = (unsigned char *)des;
    for (size_t i = n; i-- > 0;) {
        *ptr++ = value;
    }
    return des;
}

```

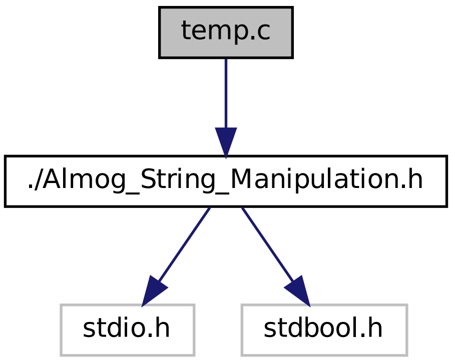
```c
void asm_print_many_times(const char * const str, const size_t n)
{
    for (size_t i = 0; i < n; i++) {
        printf("%s", str);
    }
    printf("\n");
}

void asm_remove_char_form_string(char * const s, const size_t index)
{
    size_t len = asm_length(s);
    if (len == 0) return;
    if (index >= len) {
        #ifndef NO_ERRORS
        fprintf(stderr, "%s:%d:\n%s:\n[Error] index exceeds array length.\n\n", __FILE__, __LINE__,
 __func__);
        #endif
        return;
    }

    for (size_t i = index; i < len; i++) {
        s[i] = s[i+1];
    }
}

int asm_str_in_str(const char * const src, const char * const word_to_search)
{
    int i = 0, num_of_accur = 0;
    while (src[i] != '\0') {
        if (asm_strncmp(src+i, word_to_search, asm_length(word_to_search))) {
            num_of_accur++;
        }
        i++;
    }
    return num_of_accur;
}

double asm_str2double(const char * const s, const char ** const end, const size_t base)
{
    if (base < 2 || base > 36) {
        #ifndef NO_ERRORS
        fprintf(stderr, "%s:%d:\n%s:\n[Error] Supported bases are [2...36]. Input: %zu\n\n", __FILE__,
 __LINE__, __func__, base);
        #endif
        if (end) *end = s;
        return 0.0f;
    }

    int num_of_whitespace = 0;
    while (asm_isspace(s[num_of_whitespace])) {
        num_of_whitespace++;
    }

    int i = 0;
    if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
        i++;
    }
    int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;

    size_t left = 0;
    for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
        left = base * left + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
    }
    if (s[i+num_of_whitespace] != '.') {
        if (end) *end = s + i + num_of_whitespace;
        return (left * sign);
    }

    i++; /* skip the point */

    double right = 0;
    size_t divider = base;
    for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
        right = right + asm_get_char_value_in_base(s[i+num_of_whitespace], base) / (double)divider;
        divider *= base;
    }

    if (end) *end = s + i + num_of_whitespace;

    return sign * (left + right);
}

float asm_str2float(const char * const s, const char ** const end, const size_t base)
{
    if (base < 2 || base > 36) {
        #ifndef NO_ERRORS
        fprintf(stderr, "%s:%d:\n%s:\n[Error] Supported bases are [2...36]. Input: %zu\n\n", __FILE__,
__LINE__, __func__, base);
        #endif
        if (end) *end = s;
        return 0.0f;
    }
    int num_of_whitespace = 0;
    while (asm_isspace(s[num_of_whitespace])) {
        num_of_whitespace++;
    }

    int i = 0;
    if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
        i++;
    }
    int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;

    int left = 0;
    for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
        left = base * left + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
    }
    if (s[i+num_of_whitespace] != '.') {
        if (end) *end = s + i + num_of_whitespace;
        return left * sign;
    }

    i++; /* skip the point */

    float right = 0;
    size_t divider = base;
    for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
        right = right + asm_get_char_value_in_base(s[i+num_of_whitespace], base) / (float)divider;
        divider *= base;
    }

    if (end) *end = s + i + num_of_whitespace;

    return sign * (left + right);
}

int asm_str2int(const char * const s, const char ** const end, const size_t base)
{
    if (base < 2 || base > 36) {
        #ifndef NO_ERRORS
        fprintf(stderr, "%s:%d:\n%s:\n[Error] Supported bases are [2...36]. Input: %zu\n\n", __FILE__,
__LINE__, __func__, base);
        #endif
        if (end) *end = s;
        return 0;
    }
    int num_of_whitespace = 0;
    while (asm_isspace(s[num_of_whitespace])) {
        num_of_whitespace++;
    }

    int n = 0, i = 0;
    if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
        i++;
    }
    int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;

    for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
        n = base * n + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
    }

    if (end) *end = s + i+num_of_whitespace;

    return n * sign;
}

size_t asm_str2size_t(const char * const s, const char ** const end, const size_t base)
{
    if (end) *end = s;

    int num_of_whitespace = 0;
    while (asm_isspace(s[num_of_whitespace])) {
        num_of_whitespace++;
    }

    if (s[0+num_of_whitespace] == '-') {
        #ifndef NO_ERRORS
        fprintf(stderr, "%s:%d:\n%s:\n[Error] Unable to convert a negative number to size_t.\n\n",
__FILE__, __LINE__, __func__);
        #endif
        return 0;
    }

    if (base < 2 || base > 36) {
        #ifndef NO_ERRORS
        fprintf(stderr, "%s:%d:\n%s:\n[Error] Supported bases are [2...36]. Input: %zu\n\n", __FILE__, __LINE__, __func__, base);
        #endif
        if (end) *end = s+num_of_whitespace;
        return 0;
    }

    size_t n = 0, i = 0;
    if (s[0+num_of_whitespace] == '+') {
        i++;
    }

    for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
        n = base * n + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
    }

    if (end) *end = s + i+num_of_whitespace;

    return n;
}

void asm_strip_whitespace(char * const s)
{
    size_t len = asm_length(s);
    size_t i;
    for (i = 0; i < len; i++) {
        if (asm_isspace(s[i])) {
            asm_remove_char_form_string(s, i);
            len--;
            i--;
        }
    }
    s[i] = '\0';
}

bool asm_str_is_whitespace(const char * const s)
{
    for (size_t i = 0; i < asm_length(s); i++) {
        if (!asm_isspace(s[i])) {
            return false;
        }
    }

    return true;
}

int asm_strncat(char * const s1, const char * const s2, const int N)
{
    size_t len_s1 = asm_length(s1);

    int limit = N;
    if (limit == 0) {
        limit = ASM_MAX_LEN;
    }

    int i = 0;
    while (i < limit && s2[i] != '\0') {
        if (len_s1 + (size_t)i >= ASM_MAX_LEN) {
            #ifdef NO_ERRORS
            fprintf(stderr, "%s:%d:\n%s:\n[Error] s2 or the first N=%d digit of s2 does not fit into s1.\n\n", __FILE__, __LINE__, __func__, N);
            #endif
            return i;
        }

        s1[len_s1+i] = s2[i];
        i++;
    }
    return i;
}

int asm_strncmp(const char *s1, const char *s2, const int N)
{
    int i = 0;
    while (i < N) {
        if (s1[i] == '\0' && s2[i] == '\0') {
            break;
        }
        if (s1[i] != s2[i] || (s1[i] == '\0') || (s2[i] == '\0')) {
            return 0;
        }
        i++;
    }
    return 1;
}

```

```
void asm_tolower(char * const s)
{
    size_t len = asm_length(s);
    for (size_t i = 0; i < len; i++) {
        if (asm_isupper(s[i])) {
            s[i] += 'a' - 'A';
        }
    }
}

void asm_toupper(char * const s)
{
    size_t len = asm_length(s);
    for (size_t i = 0; i < len; i++) {
        if (asm_islower(s[i])) {
            s[i] += 'A' - 'a';
        }
    }
}

#ifdef NO_ERRORS
#undef NO_ERRORS
#endif

#endif /*ALMOG_STRING_MANIPULATION_IMPLEMENTATION*/

```

2.3 temp.c File Reference

`#include "./Almog_String_Manipulation.h"`

Include dependency graph for temp.c:



Macros

- #define ALMOG_STRING_MANIPULATION_IMPLEMENTATION

Functions

- int main (void)

2.3.1 Macro Definition Documentation

2.3.1.1 ALMOG_STRING_MANIPULATION_IMPLEMENTATION

```
#define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
```

Definition at line 1 of file temp.c.

2.3.2 Function Documentation

2.3.2.1 main()

```
int main (
            void  )
```

Definition at line 4 of file temp.c.

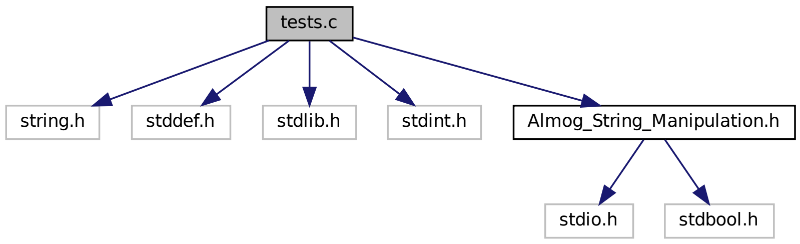
References asm_copy_array_by_indexes(), asm_dprintSTRING, asm_get_token_and_cut(), asm_left_pad(), and asm_length().

2.4 temp.c

```
#define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
#include "./Almog_String_Manipulation.h"

int main(void)
{
    char str[] = "1012";
    char str1[5] = {'2'};

    asm_copy_array_by_indexes(str1, 0, asm_length(str), str);

    asm_dprintSTRING(str);
    asm_dprintSTRING(str1);

    asm_get_token_and_cut(str1, str, '0', false);

    asm_dprintSTRING(str);
    asm_dprintSTRING(str1);

    asm_left_pad(str1, 2, '*');
    asm_dprintSTRING(str1);


    return 0;
}
```

2.5 tests.c File Reference

```
#include <string.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdint.h>
#include "Almog_String_Manipulation.h"
```
Include dependency graph for tests.c:



Macros

- #define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
- #define NO_ERRORS
- #define TEST_CASE(expr)
- #define TEST_WARN(expr, msg)
- #define TEST_EQ_INT(a, b) TEST_CASE((a) == (b))
- #define TEST_EQ_SIZE(a, b) TEST_CASE((a) == (b))
- #define TEST_EQ_STR(a, b) TEST_CASE(strcmp((a), (b)) == 0)
- #define TEST_NE_STR(a, b) TEST_CASE(strcmp((a), (b)) != 0)

Functions

- static void fill_sentinel (unsigned char ∗buf, size_t n, unsigned char v)
- static bool is_nul_terminated_within (const char ∗s, size_t cap)
- static uint32_t xorshift32 (void)
- static char rand_ascii_printable (void)
- static void test_ascii_classification_exhaustive_ranges (void)
- static void test_ascii_classification_full_scan_0_127 (void)
- static void test_case_conversion_roundtrip (void)
- static void test_length_matches_strlen_small (void)
- static void test_memset_basic_and_edges (void)
- static void test_copy_array_by_indexes_behavior_and_bounds (void)
- static void test_left_shift_edges (void)
- static void test_left_pad_edges_and_sentinel (void)
- static void test_remove_char_form_string_edges (void)
- static void test_strip_whitespace_properties (void)
- static void test_str_is_whitespace_edges (void)
- static void test_strncmp_boolean_edges (void)
- static void test_str_in_str_overlap_and_edges (void)

- static void test_base_digit_helpers (void)
- static void test_str2int (void)
- static void test_str2size_t (void)
- static void test_str2float_double (void)
- static void test_get_next_word_from_line_current_behavior (void)
- static void test_get_word_and_cut_edges (void)
- static void test_get_line_tmpfile (void)
- static void test_get_line_too_long (void)
- static void test_strncat_current_behavior_and_sentinel (void)
- int main (void)

Variables

- static int g_tests_run = 0
- static int g_tests_failed = 0
- static int g_tests_warned = 0
- static uint32_t rng_state = 0xC0FFEE01u

2.5.1 Macro Definition Documentation

2.5.1.1 ALMOG_STRING_MANIPULATION_IMPLEMENTATION

```
#define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
```

Definition at line 9 of file tests.c.

2.5.1.2 NO_ERRORS

```
#define NO_ERRORS
```

Definition at line 10 of file tests.c.

2.5.1.3 TEST_CASE

```
#define TEST_CASE(
              expr )
```

Value:

```
do {                                                                        \
    g_tests_run++;                                                          \
    if (!(expr)) {                                                          \
        g_tests_failed++;                                                   \
        fprintf(stderr, "[FAIL] %s:%d: %s\n", __FILE__, __LINE__, #expr);  \
    }                                                                       \
} while (0)
```

Definition at line 19 of file tests.c.

2.5.1.4 TEST_EQ_INT

```
#define TEST_EQ_INT(
             a,
             b ) TEST_CASE((a) == (b))
```

Definition at line 38 of file tests.c.

2.5.1.5 TEST_EQ_SIZE

```
#define TEST_EQ_SIZE(
             a,
             b ) TEST_CASE((a) == (b))
```

Definition at line 39 of file tests.c.

2.5.1.6 TEST_EQ_STR

```
#define TEST_EQ_STR(
             a,
             b ) TEST_CASE(strcmp((a), (b)) == 0)
```

Definition at line 40 of file tests.c.

2.5.1.7 TEST_NE_STR

```
#define TEST_NE_STR(
             a,
             b ) TEST_CASE(strcmp((a), (b)) != 0)
```

Definition at line 41 of file tests.c.

2.5.1.8 TEST_WARN

```
#define TEST_WARN(
             expr,
             msg )
```

Value:

```
    do {                                                                    \
        g_tests_run++;                                                      \
        if (!(expr)) {                                                      \
            g_tests_warned++;                                               \
            fprintf(stderr, "[WARN] %s:%d: %s | %s\n", __FILE__, __LINE__,   \
                    #expr, msg);                                            \
        }                                                                   \
    } while (0)
```

Definition at line 28 of file tests.c.

2.5.2 Function Documentation

2.5.2.1 fill_sentinel()

```
static void fill_sentinel (
            unsigned char * buf,
            size_t n,
            unsigned char v )  [static]
```

Definition at line 43 of file tests.c.

Referenced by test_copy_array_by_indexes_behavior_and_bounds(), test_get_line_too_long(), test_left_pad_edges_and_sentinel(), test_memset_basic_and_edges(), and test_strncat_current_behavior_and_sentinel().

2.5.2.2 is_nul_terminated_within()

```
static bool is_nul_terminated_within (
            const char * s,
            size_t cap )  [static]
```

Definition at line 48 of file tests.c.

Referenced by test_case_conversion_roundtrip(), test_get_line_tmpfile(), and test_strncat_current_behavior_and_sentinel().

2.5.2.3 main()

```
int main (
            void  )
```

Definition at line 797 of file tests.c.

References g_tests_failed, g_tests_run, g_tests_warned, test_ascii_classification_exhaustive_ranges(), test_ascii_classification_full_
test_base_digit_helpers(), test_case_conversion_roundtrip(), test_copy_array_by_indexes_behavior_and_bounds(),
test_get_line_tmpfile(), test_get_line_too_long(), test_get_next_word_from_line_current_behavior(), test_get_word_and_cut_edges()
test_left_pad_edges_and_sentinel(), test_left_shift_edges(), test_length_matches_strlen_small(), test_memset_basic_and_edges(),
test_remove_char_form_string_edges(), test_str2float_double(), test_str2int(), test_str2size_t(), test_str_in_str_overlap_and_edges(),
test_str_is_whitespace_edges(), test_strip_whitespace_properties(), test_strncat_current_behavior_and_sentinel(),
and test_strncmp_boolean_edges().

2.5.2.4 rand_ascii_printable()

```
static char rand_ascii_printable (
            void ) [static]
```

Definition at line 68 of file tests.c.

References xorshift32().

Referenced by test_case_conversion_roundtrip(), test_length_matches_strlen_small(), and test_strip_whitespace_properties().

2.5.2.5 test_ascii_classification_exhaustive_ranges()

```
static void test_ascii_classification_exhaustive_ranges (
            void ) [static]
```

Definition at line 82 of file tests.c.

References asm_isalnum(), asm_isalpha(), asm_iscntrl(), asm_isdigit(), asm_isgraph(), asm_islower(), asm_isprint(), asm_ispunct(), asm_isspace(), asm_isupper(), asm_isxdigit(), asm_isXdigit(), and TEST_CASE.

Referenced by main().

2.5.2.6 test_ascii_classification_full_scan_0_127()

```
static void test_ascii_classification_full_scan_0_127 (
            void ) [static]
```

Definition at line 153 of file tests.c.

References asm_isalnum(), asm_isalpha(), asm_isdigit(), asm_isgraph(), asm_islower(), asm_isprint(), asm_isupper(), and TEST_CASE.

Referenced by main().

2.5.2.7 test_base_digit_helpers()

```
static void test_base_digit_helpers (
            void ) [static]
```

Definition at line 460 of file tests.c.

References asm_check_char_belong_to_base(), asm_get_char_value_in_base(), TEST_CASE, and TEST_EQ_INT.

Referenced by main().

2.5.2.8 test_case_conversion_roundtrip()

```
static void test_case_conversion_roundtrip (
             void )  [static]
```

Definition at line 181 of file tests.c.

References asm_tolower(), asm_toupper(), is_nul_terminated_within(), rand_ascii_printable(), TEST_CASE, TEST_EQ_STR, and xorshift32().

Referenced by main().

2.5.2.9 test_copy_array_by_indexes_behavior_and_bounds()

```
static void test_copy_array_by_indexes_behavior_and_bounds (
             void )  [static]
```

Definition at line 257 of file tests.c.

References asm_copy_array_by_indexes(), fill_sentinel(), TEST_CASE, and TEST_EQ_STR.

Referenced by main().

2.5.2.10 test_get_line_tmpfile()

```
static void test_get_line_tmpfile (
             void )  [static]
```

Definition at line 689 of file tests.c.

References asm_get_line(), ASM_MAX_LEN, g_tests_warned, is_nul_terminated_within(), TEST_CASE, TEST_EQ_INT, and TEST_EQ_STR.

Referenced by main().

2.5.2.11 test_get_line_too_long()

```
static void test_get_line_too_long (
             void )  [static]
```

Definition at line 733 of file tests.c.

References asm_get_line(), ASM_MAX_LEN, fill_sentinel(), g_tests_warned, and TEST_EQ_INT.

Referenced by main().

2.5.2.12 test_get_next_word_from_line_current_behavior()

```
static void test_get_next_word_from_line_current_behavior (
            void ) [static]
```

Definition at line 606 of file tests.c.

References asm_get_next_token_from_str(), TEST_CASE, TEST_EQ_INT, and TEST_EQ_STR.

Referenced by main().

2.5.2.13 test_get_word_and_cut_edges()

```
static void test_get_word_and_cut_edges (
            void ) [static]
```

Definition at line 651 of file tests.c.

References asm_get_token_and_cut(), TEST_CASE, and TEST_EQ_STR.

Referenced by main().

2.5.2.14 test_left_pad_edges_and_sentinel()

```
static void test_left_pad_edges_and_sentinel (
            void ) [static]
```

Definition at line 319 of file tests.c.

References asm_left_pad(), fill_sentinel(), TEST_CASE, and TEST_EQ_STR.

Referenced by main().

2.5.2.15 test_left_shift_edges()

```
static void test_left_shift_edges (
            void ) [static]
```

Definition at line 294 of file tests.c.

References asm_left_shift(), and TEST_EQ_STR.

Referenced by main().

2.5.2.16 test_length_matches_strlen_small()

```
static void test_length_matches_strlen_small (
            void ) [static]
```

Definition at line 227 of file tests.c.

References asm_length(), rand_ascii_printable(), TEST_EQ_SIZE, and xorshift32().

Referenced by main().

2.5.2.17 test_memset_basic_and_edges()

```
static void test_memset_basic_and_edges (
            void ) [static]
```

Definition at line 241 of file tests.c.

References asm_memset(), fill_sentinel(), and TEST_CASE.

Referenced by main().

2.5.2.18 test_remove_char_form_string_edges()

```
static void test_remove_char_form_string_edges (
            void ) [static]
```

Definition at line 358 of file tests.c.

References asm_remove_char_form_string(), and TEST_EQ_STR.

Referenced by main().

2.5.2.19 test_str2float_double()

```
static void test_str2float_double (
            void ) [static]
```

Definition at line 562 of file tests.c.

References asm_str2double(), asm_str2float(), and TEST_CASE.

Referenced by main().

2.5.2.20 test_str2int()

```
static void test_str2int (
            void ) [static]
```

Definition at line 495 of file tests.c.

References asm_str2int(), and TEST_CASE.

Referenced by main().

2.5.2.21 test_str2size_t()

```
static void test_str2size_t (
            void ) [static]
```

Definition at line 531 of file tests.c.

References asm_str2size_t(), and TEST_CASE.

Referenced by main().

2.5.2.22 test_str_in_str_overlap_and_edges()

```
static void test_str_in_str_overlap_and_edges (
            void ) [static]
```

Definition at line 448 of file tests.c.

References asm_str_in_str(), and TEST_EQ_INT.

Referenced by main().

2.5.2.23 test_str_is_whitespace_edges()

```
static void test_str_is_whitespace_edges (
            void ) [static]
```

Definition at line 423 of file tests.c.

References asm_str_is_whitespace(), and TEST_CASE.

Referenced by main().

2.5.2.24 test_strip_whitespace_properties()

```
static void test_strip_whitespace_properties (
             void  )  [static]
```

Definition at line 387 of file tests.c.

References asm_isspace(), asm_strip_whitespace(), rand_ascii_printable(), TEST_CASE, TEST_EQ_STR, and xorshift32().

Referenced by main().

2.5.2.25 test_strncat_current_behavior_and_sentinel()

```
static void test_strncat_current_behavior_and_sentinel (
             void  )  [static]
```

Definition at line 761 of file tests.c.

References asm_strncat(), fill_sentinel(), is_nul_terminated_within(), TEST_CASE, TEST_EQ_INT, and TEST_WARN.

Referenced by main().

2.5.2.26 test_strncmp_boolean_edges()

```
static void test_strncmp_boolean_edges (
             void  )  [static]
```

Definition at line 432 of file tests.c.

References asm_strncmp(), and TEST_CASE.

Referenced by main().

2.5.2.27 xorshift32()

```
static uint32_t xorshift32 (
             void  )  [static]
```

Definition at line 58 of file tests.c.

References rng_state.

Referenced by rand_ascii_printable(), test_case_conversion_roundtrip(), test_length_matches_strlen_small(), and test_strip_whitespace_properties().

2.5.3 Variable Documentation

2.5.3.1 g_tests_failed

```
int g_tests_failed = 0  [static]
```

Definition at line 16 of file tests.c.

Referenced by main().

2.5.3.2 g_tests_run

```
int g_tests_run = 0  [static]
```

Definition at line 15 of file tests.c.

Referenced by main().

2.5.3.3 g_tests_warned

```
int g_tests_warned = 0  [static]
```

Definition at line 17 of file tests.c.

Referenced by main(), test_get_line_tmpfile(), and test_get_line_too_long().

2.5.3.4 rng_state

```
uint32_t rng_state = 0xC0FFEE01u  [static]
```

Definition at line 57 of file tests.c.

Referenced by xorshift32().

2.6 tests.c

```c
/* written by AI */
/* test_almog_string_manipulation.c */

#include <string.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdint.h>

#define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
#define NO_ERRORS
#include "Almog_String_Manipulation.h"

/* ---------------- Test harness ---------------- */

static int g_tests_run = 0;
static int g_tests_failed = 0;
static int g_tests_warned = 0;

#define TEST_CASE(expr)                                                       \
    do {                                                                      \
        g_tests_run++;                                                        \
        if (!(expr)) {                                                        \
            g_tests_failed++;                                                 \
            fprintf(stderr, "[FAIL] %s:%d: %s\n", __FILE__, __LINE__, #expr); \
        }                                                                     \
    } while (0)

#define TEST_WARN(expr, msg)                                                  \
    do {                                                                      \
        g_tests_run++;                                                        \
        if (!(expr)) {                                                        \
            g_tests_warned++;                                                 \
            fprintf(stderr, "[WARN] %s:%d: %s | %s\n", __FILE__, __LINE__,    \
                    #expr, msg);                                              \
        }                                                                     \
    } while (0)

#define TEST_EQ_INT(a, b) TEST_CASE((a) == (b))
#define TEST_EQ_SIZE(a, b) TEST_CASE((a) == (b))
#define TEST_EQ_STR(a, b) TEST_CASE(strcmp((a), (b)) == 0)
#define TEST_NE_STR(a, b) TEST_CASE(strcmp((a), (b)) != 0)

static void fill_sentinel(unsigned char *buf, size_t n, unsigned char v)
{
    for (size_t i = 0; i < n; i++) buf[i] = v;
}

static bool is_nul_terminated_within(const char *s, size_t cap)
{
    for (size_t i = 0; i < cap; i++) {
        if (s[i] == '\0') return true;
    }
    return false;
}

/* Simple deterministic RNG for fuzz-ish tests */
static uint32_t rng_state = 0xC0FFEE01u;
static uint32_t xorshift32(void)
{
    uint32_t x = rng_state;
    x ^= x « 13;
    x ^= x » 17;
    x ^= x « 5;
    rng_state = x;
    return x;
}

static char rand_ascii_printable(void)
{
    /* printable ASCII range 32..126 */
    return (char)(32 + (xorshift32() % 95));
}

/* ---------------- Coverage checks ----------------
 * We can't reliably "assert all symbols exist" at runtime, but we can at least
 * ensure we have tests for every IMPLEMENTED function by calling it at least
 * once in this file.
 */

/* ---------------- Tests: ASCII classification ---------------- */

static void test_ascii_classification_exhaustive_ranges(void)
{
    /* Check key boundaries and a few midpoints for each function. */
    TEST_CASE(asm_isdigit('0'));
    TEST_CASE(asm_isdigit('9'));
    TEST_CASE(!asm_isdigit('/'));
    TEST_CASE(!asm_isdigit(':'));

    TEST_CASE(asm_isupper('A'));
    TEST_CASE(asm_isupper('Z'));
    TEST_CASE(!asm_isupper('@'));
    TEST_CASE(!asm_isupper('['));

    TEST_CASE(asm_islower('a'));
    TEST_CASE(asm_islower('z'));
    TEST_CASE(!asm_islower('`'));
    TEST_CASE(!asm_islower('{'));

    TEST_CASE(asm_isalpha('A'));
    TEST_CASE(asm_isalpha('z'));
    TEST_CASE(!asm_isalpha('0'));

    TEST_CASE(asm_isalnum('A'));
    TEST_CASE(asm_isalnum('9'));
    TEST_CASE(!asm_isalnum('_'));
    TEST_CASE(!asm_isalnum(' '));

    TEST_CASE(asm_isspace(' '));
    TEST_CASE(asm_isspace('\n'));
    TEST_CASE(asm_isspace('\t'));
    TEST_CASE(asm_isspace('\r'));
    TEST_CASE(asm_isspace('\v'));
    TEST_CASE(asm_isspace('\f'));
    TEST_CASE(!asm_isspace('X'));

    TEST_CASE(asm_isgraph('!'));
    TEST_CASE(asm_isgraph('~'));
    TEST_CASE(!asm_isgraph(' '));

    TEST_CASE(asm_isprint(' '));
    TEST_CASE(asm_isprint('!'));
    TEST_CASE(!asm_isprint('\n'));

    TEST_CASE(asm_ispunct('!'));
    TEST_CASE(asm_ispunct('/'));
    TEST_CASE(asm_ispunct(':'));
    TEST_CASE(!asm_ispunct('A'));
    TEST_CASE(!asm_ispunct('0'));
    TEST_CASE(!asm_ispunct(' '));

    TEST_CASE(asm_iscntrl('\0'));
    TEST_CASE(asm_iscntrl('\n'));
    TEST_CASE(asm_iscntrl(127));
    TEST_CASE(!asm_iscntrl('A'));

    /* Hex digit helpers (your impl splits by case) */
    TEST_CASE(asm_isxdigit('0'));
    TEST_CASE(asm_isxdigit('9'));
    TEST_CASE(asm_isxdigit('a'));
    TEST_CASE(asm_isxdigit('f'));
    TEST_CASE(!asm_isxdigit('g'));
    TEST_CASE(!asm_isxdigit('A'));

    TEST_CASE(asm_isXdigit('0'));
    TEST_CASE(asm_isXdigit('9'));
    TEST_CASE(asm_isXdigit('A'));
    TEST_CASE(asm_isXdigit('F'));
    TEST_CASE(!asm_isXdigit('G'));
    TEST_CASE(!asm_isXdigit('a'));
}

static void test_ascii_classification_full_scan_0_127(void)
{
    /* Property checks over ASCII 0..127. */
    for (int c = 0; c <= 127; c++) {
        char ch = (char)c;

        /* isalnum == isalpha || isdigit */
        TEST_CASE(asm_isalnum(ch) == (asm_isalpha(ch) || asm_isdigit(ch)));

        /* isprint == isgraph || ' ' */
        TEST_CASE(asm_isprint(ch) == (asm_isgraph(ch) || ch == ' '));

        /* isalpha implies not digit */
        if (asm_isalpha(ch)) {
            TEST_CASE(!asm_isdigit(ch));
        }

        /* upper and lower are disjoint */
        if (asm_isupper(ch)) TEST_CASE(!asm_islower(ch));
        if (asm_islower(ch)) TEST_CASE(!asm_isupper(ch));
```

```c

        /* graph implies print */
        if (asm_isgraph(ch)) TEST_CASE(asm_isprint(ch));
    }
}

/* ---------------- Tests: case conversion ---------------- */

static void test_case_conversion_roundtrip(void)
{
    for (int i = 0; i < 200; i++) {
        char s[128];
        char a[128];
        char b[128];

        /* random printable string length 0..40 */
        size_t n = (size_t)(xorshift32() % 41);
        for (size_t j = 0; j < n; j++) s[j] = rand_ascii_printable();
        s[n] = '\0';

        strcpy(a, s);
        strcpy(b, s);

        asm_tolower(a);
        asm_toupper(a);
        asm_toupper(b);
        asm_tolower(b);

        /* Not equal generally, but must still be valid strings and stable */
        TEST_CASE(is_nul_terminated_within(a, sizeof(a)));
        TEST_CASE(is_nul_terminated_within(b, sizeof(b)));

        /* toupper(toupper(x)) == toupper(x) */
        char u1[128], u2[128];
        strcpy(u1, s);
        strcpy(u2, s);
        asm_toupper(u1);
        asm_toupper(u2);
        asm_toupper(u2);
        TEST_EQ_STR(u1, u2);

        /* tolower(tolower(x)) == tolower(x) */
        char l1[128], l2[128];
        strcpy(l1, s);
        strcpy(l2, s);
        asm_tolower(l1);
        asm_tolower(l2);
        asm_tolower(l2);
        TEST_EQ_STR(l1, l2);
    }
}

/* ---------------- Tests: asm_length ---------------- */

static void test_length_matches_strlen_small(void)
{
    for (int i = 0; i < 200; i++) {
        char s[256];
        size_t n = (size_t)(xorshift32() % 200);
        for (size_t j = 0; j < n; j++) s[j] = rand_ascii_printable();
        s[n] = '\0';

        TEST_EQ_SIZE(asm_length(s), strlen(s));
    }
}

/* ---------------- Tests: asm_memset ---------------- */

static void test_memset_basic_and_edges(void)
{
    unsigned char buf[32];
    fill_sentinel(buf, sizeof(buf), 0xCC);

    void *ret = asm_memset(buf, 0xAB, sizeof(buf));
    TEST_CASE(ret == buf);
    for (size_t i = 0; i < sizeof(buf); i++) TEST_CASE(buf[i] == 0xAB);

    fill_sentinel(buf, sizeof(buf), 0xCC);
    asm_memset(buf, 0xAB, 0);
    for (size_t i = 0; i < sizeof(buf); i++) TEST_CASE(buf[i] == 0xCC);
}

/* ---------------- Tests: asm_copy_array_by_indexes ---------------- */

static void test_copy_array_by_indexes_behavior_and_bounds(void)
{
    const char *src = "abcdef";
    char out[16];

    asm_copy_array_by_indexes(out, 1, 3, src); /* inclusive end in impl */
    TEST_EQ_STR(out, "bcd");

    asm_copy_array_by_indexes(out, 0, 0, src);
    TEST_EQ_STR(out, "a");

    asm_copy_array_by_indexes(out, 5, 5, src);
    TEST_EQ_STR(out, "f");

    asm_copy_array_by_indexes(out, 0, 6, src); /* copies '\0' too */
    TEST_EQ_STR(out, "abcdef");

    /* Sentinel around output buffer to detect overwrite beyond out[16] */
    struct {
        unsigned char pre[8];
        char out2[8];
        unsigned char post[8];
    } box;

    fill_sentinel(box.pre, sizeof(box.pre), 0xA5);
    fill_sentinel((unsigned char *)box.out2, sizeof(box.out2), 0xCC);
    fill_sentinel(box.post, sizeof(box.post), 0x5A);

    /* copy "ab" plus '\0' => should fit exactly */
    asm_copy_array_by_indexes(box.out2, 0, 1, "ab");
    TEST_EQ_STR(box.out2, "ab");
    for (size_t i = 0; i < sizeof(box.pre); i++) TEST_CASE(box.pre[i] == 0xA5);
    for (size_t i = 0; i < sizeof(box.post); i++) TEST_CASE(box.post[i] == 0x5A);
}

/* ---------------- Tests: shifting/padding ---------------- */

static void test_left_shift_edges(void)
{
    char s[64];

    strcpy(s, "abcdef");
    asm_left_shift(s, 0);
    TEST_EQ_STR(s, "abcdef");

    strcpy(s, "abcdef");
    asm_left_shift(s, 1);
    TEST_EQ_STR(s, "bcdef");

    strcpy(s, "abcdef");
    asm_left_shift(s, 5);
    TEST_EQ_STR(s, "f");

    strcpy(s, "abcdef");
    asm_left_shift(s, 6);
    TEST_EQ_STR(s, "");

    strcpy(s, "abcdef");
    asm_left_shift(s, 1000);
    TEST_EQ_STR(s, "");
}

static void test_left_pad_edges_and_sentinel(void)
{
    {
        char s[64] = "abc";
        asm_left_pad(s, 0, ' ');
        TEST_EQ_STR(s, "abc");
    }
    {
        char s[64] = "abc";
        asm_left_pad(s, 4, ' ');
        TEST_EQ_STR(s, "    abc");
    }
    {
        char s[64] = "";
        asm_left_pad(s, 3, '_');
        TEST_EQ_STR(s, "___");
    }

    /* Sentinel structure: ensure we don't write before start */
    struct {
        unsigned char pre[8];
        char s[32];
        unsigned char post[8];
    } box;

    fill_sentinel(box.pre, sizeof(box.pre), 0x11);
    fill_sentinel((unsigned char *)box.s, sizeof(box.s), 0xCC);
    fill_sentinel(box.post, sizeof(box.post), 0x22);
```

```c

    strcpy(box.s, "x");
    asm_left_pad(box.s, 5, '0');
    TEST_EQ_STR(box.s, "00000x");

    for (size_t i = 0; i < sizeof(box.pre); i++) TEST_CASE(box.pre[i] == 0x11);
    for (size_t i = 0; i < sizeof(box.post); i++) TEST_CASE(box.post[i] == 0x22);
}

/* ---------------- Tests: remove/strip/whitespace ---------------- */

static void test_remove_char_form_string_edges(void)
{
    char s[64];

    strcpy(s, "abcd");
    asm_remove_char_form_string(s, 1);
    TEST_EQ_STR(s, "acd");

    strcpy(s, "abcd");
    asm_remove_char_form_string(s, 0);
    TEST_EQ_STR(s, "bcd");

    strcpy(s, "abcd");
    asm_remove_char_form_string(s, 3);
    TEST_EQ_STR(s, "abc");

    strcpy(s, "a");
    asm_remove_char_form_string(s, 0);
    TEST_EQ_STR(s, "");

    strcpy(s, "");
    asm_remove_char_form_string(s, 0);
    TEST_EQ_STR(s, "");

    strcpy(s, "abcd");
    asm_remove_char_form_string(s, 999);
    TEST_EQ_STR(s, "abcd");
}

static void test_strip_whitespace_properties(void)
{
    char s[128];

    strcpy(s, "  a \t b\nc  ");
    asm_strip_whitespace(s);
    TEST_EQ_STR(s, "abc");

    strcpy(s, "no_spaces");
    asm_strip_whitespace(s);
    TEST_EQ_STR(s, "no_spaces");

    strcpy(s, " \t\r\n");
    asm_strip_whitespace(s);
    TEST_EQ_STR(s, "");

    /* Property: result has no whitespace chars */
    for (int i = 0; i < 100; i++) {
        size_t n = (size_t)(xorshift32() % 60);
        for (size_t j = 0; j < n; j++) {
            /* mix whitespace and printable */
            uint32_t r = xorshift32() % 10;
            if (r == 0) s[j] = ' ';
            else if (r == 1) s[j] = '\n';
            else if (r == 2) s[j] = '\t';
            else s[j] = rand_ascii_printable();
        }
        s[n] = '\0';

        asm_strip_whitespace(s);
        for (size_t k = 0; s[k] != '\0'; k++) {
            TEST_CASE(!asm_isspace(s[k]));
        }
    }
}

static void test_str_is_whitespace_edges(void)
{
    TEST_CASE(asm_str_is_whitespace(" \t\r\n") == true);
    TEST_CASE(asm_str_is_whitespace("") == true); /* current behavior */
    TEST_CASE(asm_str_is_whitespace(" x ") == false);
}

/* ---------------- Tests: asm_strncmp (boolean) ---------------- */

static void test_strncmp_boolean_edges(void)
{
    TEST_CASE(asm_strncmp("abc", "abc", 3) == 1);
    TEST_CASE(asm_strncmp("abc", "abd", 3) == 0);
    TEST_CASE(asm_strncmp("ab", "abc", 3) == 0);
    TEST_CASE(asm_strncmp("abc", "ab", 3) == 0);

    TEST_CASE(asm_strncmp("abc", "XYZ", 0) == 1);

    TEST_CASE(asm_strncmp("", "", 5) == 1);
    TEST_CASE(asm_strncmp("", "a", 1) == 0);
    TEST_CASE(asm_strncmp("a", "", 1) == 0);
}

/* ---------------- Tests: asm_str_in_str ---------------- */

static void test_str_in_str_overlap_and_edges(void)
{
    TEST_EQ_INT(asm_str_in_str("aaaa", "aa"), 3);
    TEST_EQ_INT(asm_str_in_str("hello world", "lo"), 1);
    TEST_EQ_INT(asm_str_in_str("abc", "abcd"), 0);
    TEST_EQ_INT(asm_str_in_str("abababa", "aba"), 3);

    /* Do not pass empty needle: undefined-ish for your implementation. */
}

/* ---------------- Tests: base digit helpers ---------------- */

static void test_base_digit_helpers(void)
{
    TEST_CASE(asm_check_char_belong_to_base('0', 2) == true);
    TEST_CASE(asm_check_char_belong_to_base('1', 2) == true);
    TEST_CASE(asm_check_char_belong_to_base('2', 2) == false);

    TEST_CASE(asm_check_char_belong_to_base('9', 10) == true);
    TEST_CASE(asm_check_char_belong_to_base('a', 10) == false);

    TEST_CASE(asm_check_char_belong_to_base('a', 16) == true);
    TEST_CASE(asm_check_char_belong_to_base('f', 16) == true);
    TEST_CASE(asm_check_char_belong_to_base('g', 16) == false);
    TEST_CASE(asm_check_char_belong_to_base('A', 16) == true);
    TEST_CASE(asm_check_char_belong_to_base('F', 16) == true);
    TEST_CASE(asm_check_char_belong_to_base('G', 16) == false);

    TEST_CASE(asm_check_char_belong_to_base('z', 36) == true);
    TEST_CASE(asm_check_char_belong_to_base('Z', 36) == true);

    TEST_EQ_INT(asm_get_char_value_in_base('0', 10), 0);
    TEST_EQ_INT(asm_get_char_value_in_base('9', 10), 9);
    TEST_EQ_INT(asm_get_char_value_in_base('A', 16), 10);
    TEST_EQ_INT(asm_get_char_value_in_base('f', 16), 15);
    TEST_EQ_INT(asm_get_char_value_in_base('Z', 36), 35);

    TEST_EQ_INT(asm_get_char_value_in_base('g', 16), -1);

    /* base validity errors should return false / -1 */
    TEST_CASE(asm_check_char_belong_to_base('0', 1) == false);
    TEST_CASE(asm_check_char_belong_to_base('0', 37) == false);
    TEST_EQ_INT(asm_get_char_value_in_base('0', 1), -1);
}

/* ---------------- Tests: str2int/size_t/float/double ---------------- */

static void test_str2int(void)
{
    const char *end = NULL;

    {
        char s[] = "  -1011zzz";
        int v = asm_str2int(s, &end, 2);
        TEST_CASE(v == -11);
        TEST_CASE(*end == 'z');
    }
    {
        char s[] = "+7fff!";
        int v = asm_str2int(s, &end, 16);
        TEST_CASE(v == 0x7fff);
        TEST_CASE(*end == '!');
    }
    {
        char s[] = "   +0";
        int v = asm_str2int(s, &end, 10);
        TEST_CASE(v == 0);
        TEST_CASE(*end == '\0');
    }
    {
        char s[] = "xyz";
        int v = asm_str2int(s, &end, 10);
        TEST_CASE(v == 0);
        TEST_CASE(*end == 'x');
    }
    {
        char s[] = "123";
        int v = asm_str2int(s, &end, 1);
        TEST_CASE(v == 0);
        TEST_CASE(end == s);
    }
}

static void test_str2size_t(void)
{
    const char *end = NULL;

    {
        char s[] = " +1f!";
        size_t v = asm_str2size_t(s, &end, 16);
        TEST_CASE(v == 31u);
        TEST_CASE(*end == '!');
    }
    {
        char s[] = "  -1";
        size_t v = asm_str2size_t(s, &end, 10);
        TEST_CASE(v == 0);
        TEST_CASE(end == s);
    }
    {
        char s[] = "  +0009x";
        size_t v = asm_str2size_t(s, &end, 10);
        TEST_CASE(v == 9u);
        TEST_CASE(*end == 'x');
    }
    {
        char s[] = "  123";
        size_t v = asm_str2size_t(s, &end, 37);
        TEST_CASE(v == 0);
        /* current implementation sets *end = s+num_of_whitespace on invalid base */
        TEST_CASE(end == s + 2);
    }
}

static void test_str2float_double(void)
{
    const char *end = NULL;

    {
        char s[] = " 10.5x";
        float v = asm_str2float(s, &end, 10);
        TEST_CASE(v > 10.49f && v < 10.51f);
        TEST_CASE(*end == 'x');
    }
    {
        char s[] = "-a.bQ";
        double v = asm_str2double(s, &end, 16);
        TEST_CASE(v < -10.68 && v > -10.70);
        TEST_CASE(*end == 'Q');
    }
    {
        char s[] = "  123.";
        double v = asm_str2double(s, &end, 10);
        TEST_CASE(v > 122.99 && v < 123.01);
        TEST_CASE(*end == '\0');
    }
    {
        char s[] = "  .5";
        double v = asm_str2double(s, &end, 10);
        TEST_CASE(v > 0.49 && v < 0.51);
        TEST_CASE(*end == '\0');
    }
    {
        char s[] = "  -.";
        double v = asm_str2double(s, &end, 10);
        TEST_CASE(v == 0.0);
        TEST_CASE(*end == '\0');
    }
    {
        char s[] = "12.3";
        double v = asm_str2double(s, &end, 37);
        TEST_CASE(v == 0.0);
        TEST_CASE(end == s);
    }
}

/* ---------------- Tests: tokenization helpers ---------------- */

static void test_get_next_word_from_line_current_behavior(void)
{
    /* Your implementation:
     * - does NOT skip whitespace
     * - stops only on delimiter or '\0'
     * - returns length (j), not consumed index
     */
    {
        char src[] = "abc,def";
        char w[64] = {0};
        int r = asm_get_next_token_from_str(w, src, ',');
        TEST_EQ_INT(r, 3);
        TEST_EQ_STR(w, "abc");
    }
    {
        char src[] = ",def";
        char w[64] = {0};
        int r = asm_get_next_token_from_str(w, src, ',');
        TEST_EQ_INT(r, 0);
        TEST_EQ_STR(w, "");
    }
    {
        char src[] = "  abc,def";
        char w[64] = {0};
        int r = asm_get_next_token_from_str(w, src, ',');
        TEST_EQ_INT(r, 5);
        TEST_EQ_STR(w, "  abc");
    }
    {
        char src[] = "abc\ndef";
        char w[64] = {0};
        int r = asm_get_next_token_from_str(w, src, ',');
        TEST_EQ_INT(r, (int)strlen(src));
        TEST_EQ_STR(w, "abc\ndef");
    }

    /* Doc mismatch detection (warn, not fail) */
    {
        char src[] = "  abc,def";
        char w[64] = {0};
        asm_get_next_token_from_str(w, src, ',');
        TEST_CASE(strcmp(w, "  abc") == 0);
    }
}

static void test_get_word_and_cut_edges(void)
{
    {
        char src[64] = "abc,def";
        char w[64] = {0};
        int ok = asm_get_token_and_cut(w, src, ',', true);
        TEST_CASE(ok == 1);
        TEST_EQ_STR(w, "abc");
        TEST_EQ_STR(src, ",def");
    }
    {
        char src[64] = "abc,def";
        char w[64] = {0};
        int ok = asm_get_token_and_cut(w, src, ',', false);
        TEST_CASE(ok == 1);
        TEST_EQ_STR(w, "abc");
        TEST_EQ_STR(src, "def");
    }
    {
        char src[64] = ",def";
        char w[64] = {0};
        int ok = asm_get_token_and_cut(w, src, ',', true);
        TEST_CASE(ok == 0);
        TEST_EQ_STR(w, "");
        TEST_EQ_STR(src, ",def");
    }
    {
        char src[64] = "nodelem";
        char w[64] = {0};
        int ok = asm_get_token_and_cut(w, src, ',', false);
        TEST_CASE(ok == 1);
        TEST_EQ_STR(w, "nodelem");
        TEST_EQ_STR(src, "");
    }
}

/* ---------------- Tests: asm_get_line ---------------- */

static void test_get_line_tmpfile(void)
{
    FILE *fp = tmpfile();
    if (!fp) {
        fprintf(stderr,
                "[WARN] tmpfile() unavailable; skipping asm_get_line tests\n");
        g_tests_warned++;
        return;
    }

    fputs("hello\n", fp);
    fputs("\n", fp);
    fputs("world", fp);
    rewind(fp);

    {
        char line[ASM_MAX_LEN + 1];
        int n = asm_get_line(fp, line);
        TEST_EQ_INT(n, 5);
        TEST_EQ_STR(line, "hello");
        TEST_CASE(is_nul_terminated_within(line, sizeof(line)));
    }
    {
        char line[ASM_MAX_LEN + 1];
        int n = asm_get_line(fp, line);
        TEST_EQ_INT(n, 0);
        TEST_EQ_STR(line, "");
    }
    {
        char line[ASM_MAX_LEN + 1];
        int n = asm_get_line(fp, line);
        TEST_EQ_INT(n, 5);
        TEST_EQ_STR(line, "world");
    }
    {
        char line[ASM_MAX_LEN + 1];
        int n = asm_get_line(fp, line);
        TEST_EQ_INT(n, -1);
    }

    fclose(fp);
}

/* Optional: test overflow condition using ASM_MAX_LEN+1 chars before '\n' */
static void test_get_line_too_long(void)
{
    FILE *fp = tmpfile();
    if (!fp) {
        fprintf(stderr,
                "[WARN] tmpfile() unavailable; skipping long-line test\n");
        g_tests_warned++;
        return;
    }

    for (int i = 0; i < ASM_MAX_LEN + 5; i++) fputc('a', fp);
    fputc('\n', fp);
    rewind(fp);

    char line[ASM_MAX_LEN + 1];
    fill_sentinel((unsigned char *)line, sizeof(line), 0xCC);

    int n = asm_get_line(fp, line);
    TEST_EQ_INT(n, -1);

    /* On error, your docs say not guaranteed NUL terminated. We only ensure
       we didn't write past buffer size (can't directly prove; but at least
       array exists). */
    fclose(fp);
}

/* ---------------- Tests: asm_strncat ---------------- */

static void test_strncat_current_behavior_and_sentinel(void)
{
    /* Current impl does NOT append '\0' (bug-like).
       We test both:
       - it copies correct bytes
       - it does not clobber past allowed region
     */
    struct {
        unsigned char pre[8];
        char s1[16];
        unsigned char post[8];
    } box;

    fill_sentinel(box.pre, sizeof(box.pre), 0xAA);
    fill_sentinel((unsigned char *)box.s1, sizeof(box.s1), 0xCC);
    fill_sentinel(box.post, sizeof(box.post), 0xBB);

    strcpy(box.s1, "abc");

    int n = asm_strncat(box.s1, "DEF", 3);
    TEST_EQ_INT(n, 3);

    TEST_CASE(box.s1[0] == 'a' && box.s1[1] == 'b' && box.s1[2] == 'c');
    TEST_CASE(box.s1[3] == 'D' && box.s1[4] == 'E' && box.s1[5] == 'F');

    /* warn if it *is* NUL terminated (meaning you fixed it) */
    TEST_WARN(!is_nul_terminated_within(box.s1, 7),
              "asm_strncat appears to NUL-terminate now; update tests to "
              "expect \"abcDEF\" as a proper C-string.");

    for (size_t i = 0; i < sizeof(box.pre); i++) TEST_CASE(box.pre[i] == 0xAA);
    for (size_t i = 0; i < sizeof(box.post); i++) TEST_CASE(box.post[i] == 0xBB);
}

/* ---------------- Main ---------------- */

int main(void)
{
    test_ascii_classification_exhaustive_ranges();
    test_ascii_classification_full_scan_0_127();

    test_case_conversion_roundtrip();

    test_length_matches_strlen_small();

    test_memset_basic_and_edges();

    test_copy_array_by_indexes_behavior_and_bounds();

    test_left_shift_edges();
    test_left_pad_edges_and_sentinel();

    test_remove_char_form_string_edges();
    test_strip_whitespace_properties();
    test_str_is_whitespace_edges();

    test_strncmp_boolean_edges();
    test_str_in_str_overlap_and_edges();

    test_base_digit_helpers();
    test_str2int();
    test_str2size_t();
    test_str2float_double();

    test_get_next_word_from_line_current_behavior();
    test_get_word_and_cut_edges();

    test_get_line_tmpfile();
    test_get_line_too_long();

    test_strncat_current_behavior_and_sentinel();

    if (g_tests_failed == 0) {
        if (g_tests_warned == 0) {
            printf("[OK] %d tests passed\n", g_tests_run);
        } else {
            printf("[OK] %d tests passed, %d warnings\n", g_tests_run,
                   g_tests_warned);
        }
        return 0;
    }

    fprintf(stderr, "[FAIL] %d/%d tests failed (%d warnings)\n", g_tests_failed,
            g_tests_run, g_tests_warned);
    return 1;
}
```

Index