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

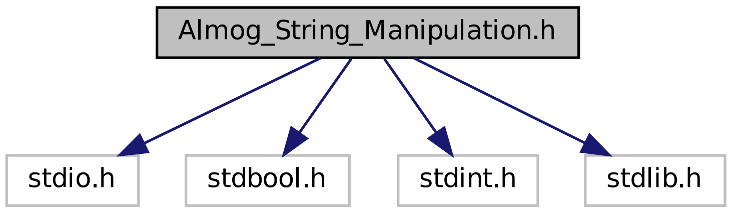
File Documentation

2.1 Almog_String_Manipulation.h File Reference

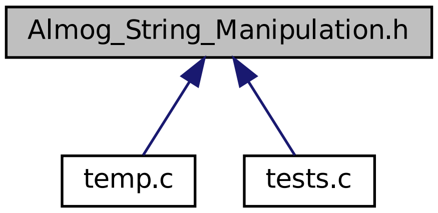
Lightweight string and line manipulation helpers.

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

Include dependency graph for Almog_String_Manipulation.h:



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



Macros

- #define ASM_MALLOC malloc
- #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_dprintERROR(fmt, ...)
- #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)

  Extract the next token into `dst` and remove the corresponding prefix from `src`.
- bool asm_isalnum (char c)

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

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

  Test for a binary digit (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_isodigit (const char c)

  Test for an octal digit (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).
- 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_pad_left (char ∗const s, const size_t padding, const char pad)

  Left-pad a string in-place.
- 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_from_string (char ∗const s, const size_t index)

  Remove a single character from a string by index.
- void asm_shift_left (char ∗const s, const size_t shift)

  Shift a string left in-place by `shift` characters.
- 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 with exponent support.
- float asm_str2float (const char ∗const s, const char ∗∗const end, const size_t base)

  Convert a string to float in the given base with exponent support.
- 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.
- char ∗ asm_strdup (const char ∗const s, size_t length)

  Allocate and copy up to `length` characters from `s`.
- int asm_strncat (char ∗const s1, const char ∗const s2, const size_t N)

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

  Compare up to N characters for equality (boolean result).
- int asm_strncpy (char ∗const s1, const char ∗const s2, const size_t N)

  Copy up to `N` characters from `s2` into `s1` (non-standard).
- 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.
- void asm_trim_left_whitespace (char ∗const s)

  Remove leading ASCII whitespace from a string 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 (does not skip 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 88 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 115 of file Almog_String_Manipulation.h.

2.1.2.3 asm_dprintERROR

```
#define asm_dprintERROR(
                fmt,
                ... )
```

Value:

```
    fprintf(stderr, "\n%s:%d:\n[Error] in function '%s':\n        " \
    fmt "\n\n", __FILE__, __LINE__, __func__, __VA_ARGS__)
```

Definition at line 126 of file Almog_String_Manipulation.h.

2.1.2.4 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 106 of file Almog_String_Manipulation.h.

2.1.2.5 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 97 of file Almog_String_Manipulation.h.

2.1.2.6 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 124 of file Almog_String_Manipulation.h.

2.1.2.7 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 79 of file Almog_String_Manipulation.h.

2.1.2.8 ASM_MALLOC

```
#define ASM_MALLOC malloc
```

Definition at line 50 of file Almog_String_Manipulation.h.

2.1.2.9 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 may be evaluated more than once. Do not pass expressions with side effects (e.g., ++i, function calls with state).

Definition at line 154 of file Almog_String_Manipulation.h.

2.1.2.10 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 ASM_MAX_LEN characters without encountering '
' or EOF, it prints an error to stderr and returns -1. In that error case, the buffer is truncated and null-terminated by overwriting the last stored character (so the resulting string length is ASM_MAX_LEN - 1).

Definition at line 69 of file Almog_String_Manipulation.h.

2.1.2.11 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 may be evaluated more than once. Do not pass expressions with side effects (e.g., ++i, function calls with state).

Definition at line 141 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 212 of file Almog_String_Manipulation.h.

References asm_dprintERROR, and 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 247 of file Almog_String_Manipulation.h.

Referenced by 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

Returns -1 if `c` is not valid for `base`.

Definition at line 269 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 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 reaches 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 truncated and null-terminated by overwriting the last stored character.

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

Definition at line 301 of file Almog_String_Manipulation.h.

References asm_dprintERROR, and 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 348 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 )
```

Extract the next token into `dst` and remove the corresponding prefix from `src`.

Calls asm_get_next_token_from_str(`dst`, `src`, `delimiter`) to extract a token from the beginning of `src` into `dst`. Then modifies `src` in-place by left-shifting it.

If `leave_delimiter` is true, `src` is left-shifted by the value returned from asm_get_next_token_from_str() (i.e., the delimiter—if present—remains as the first character in the updated `src`).

If `leave_delimiter` is false, `src` is left-shifted by that return value plus one (intended to also remove the delimiter).

Parameters

dst	Destination buffer for the extracted token (must be large enough for the token plus the null terminator).
src	Source buffer, modified in-place by this function.
delimiter	Delimiter character used to stop token extraction.
leave_delimiter	If true, do not remove the delimiter from `src`; if false, remove one additional character after the token.

Returns

1 if asm_get_next_token_from_str() returned a non-zero value, otherwise 0.

Note

This function always calls asm_shift_left() even when the returned value from asm_get_next_token_from_str() is 0. In particular, when `leave_delimiter` is false and the returned value is 0, `src` will be left-shifted by 1.

Definition at line 391 of file Almog_String_Manipulation.h.

References asm_get_next_token_from_str(), and asm_shift_left().

Referenced by 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 412 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 423 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_isbdigit()

```
bool asm_isbdigit (
             const char c )
```

Test for a binary digit (ASCII).

Parameters

c	Character to test.

Returns

true if `c` is '0' or '1'; false otherwise.

Definition at line 434 of file Almog_String_Manipulation.h.

2.1.3.10 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 449 of file Almog_String_Manipulation.h.

Referenced by test_ascii_classification_exhaustive_ranges().

2.1.3.11 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 464 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().

2.1.3.12 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 479 of file Almog_String_Manipulation.h.

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

2.1.3.13 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 494 of file Almog_String_Manipulation.h.

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

2.1.3.14 asm_isodigit()

```
bool asm_isodigit (
             const char c )
```

Test for an octal digit (ASCII).

Parameters

c	Character to test.

Returns

true if `c` is '0'–'7'; false otherwise.

Definition at line 509 of file Almog_String_Manipulation.h.

2.1.3.15 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 525 of file Almog_String_Manipulation.h.

References asm_isgraph().

Referenced by test_ascii_classification_exhaustive_ranges(), and test_ascii_classification_full_scan_0_127().

2.1.3.16 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 537 of file Almog_String_Manipulation.h.

Referenced by test_ascii_classification_exhaustive_ranges().

2.1.3.17 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 553 of file Almog_String_Manipulation.h.

Referenced by asm_str2double(), asm_str2float(), asm_str2int(), asm_str2size_t(), asm_str_is_whitespace(), asm_strip_whitespace(), asm_trim_left_whitespace(), test_ascii_classification_exhaustive_ranges(), and test_strip_whitespace_prope

2.1.3.18 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 569 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().

2.1.3.19 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 584 of file Almog_String_Manipulation.h.

References asm_isdigit().

Referenced by test_ascii_classification_exhaustive_ranges().

2.1.3.20 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 599 of file Almog_String_Manipulation.h.

References asm_isdigit().

Referenced by test_ascii_classification_exhaustive_ranges().

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 618 of file Almog_String_Manipulation.h.

References asm_dprintERROR, and ASM_MAX_LEN.

Referenced by asm_pad_left(), asm_remove_char_from_string(), asm_shift_left(), asm_str_in_str(), asm_str_is_whitespace(), asm_strip_whitespace(), asm_strncat(), asm_tolower(), asm_toupper(), asm_trim_left_whitespace(), 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 653 of file Almog_String_Manipulation.h.

Referenced by test_memset_basic_and_edges().

2.1.3.23 asm_pad_left()

```
void asm_pad_left (
            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 676 of file Almog_String_Manipulation.h.

References asm_length().

Referenced by test_left_pad_edges_and_sentinel().

2.1.3.24 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 693 of file Almog_String_Manipulation.h.

2.1.3.25 asm_remove_char_from_string()

```
void asm_remove_char_from_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 713 of file Almog_String_Manipulation.h.

References asm_dprintERROR, and asm_length().

Referenced by asm_strip_whitespace(), and test_remove_char_form_string_edges().

2.1.3.26 asm_shift_left()

```
void asm_shift_left (
            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 742 of file Almog_String_Manipulation.h.

References asm_length().

Referenced by asm_get_token_and_cut(), asm_trim_left_whitespace(), and test_left_shift_edges().

2.1.3.27 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 with exponent support.

Parses an optional sign, then a sequence of base-N digits, optionally a fractional part separated by a '.' character, and optionally an exponent part indicated by 'e' or 'E' followed by an optional sign and decimal 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 double value. Returns 0.0 on invalid base.

Note

Only digits '0'–'9', 'a'–'z', and 'A'–'Z' are recognized as base-N digits for the mantissa (the part before the exponent).

The exponent is always parsed in base 10 and represents the power of the specified base. For example, "1.5e2" in base 10 means 1.5 ∗ 10^2 = 150, while "A.8e2" in base 16 means 10.5 ∗ 16^2 = 2688.

The exponent can be positive or negative (e.g., "1e-3" = 0.001).

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

Examples:

```
asm_str2double("1.5e2", NULL, 10)     // Returns 150.0
asm_str2double("-3.14e-1", NULL, 10) // Returns -0.314
asm_str2double("FF.0e1", NULL, 16)   // Returns 4080.0 (255 × 16^1)
```

Definition at line 816 of file Almog_String_Manipulation.h.

References asm_check_char_belong_to_base(), asm_dprintERROR, asm_get_char_value_in_base(), asm_isspace(), and asm_str2int().

Referenced by main(), test_str2double_exponent_basic(), test_str2double_exponent_edge_cases(), test_str2double_exponent_signe
test_str2float_double(), test_str2float_double_exponent_different_bases(), test_str2float_double_exponent_large_values(), and test_str2float_double_exponent_whitespace().

2.1.3.28 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 with exponent support.

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 for the mantissa (the part before the exponent).

The exponent is always parsed in base 10 and represents the power of the specified base. For example, "1.5e2" in base 10 means 1.5 ∗ 10^2 = 150, while "A.8e2" in base 16 means 10.5 ∗ 16^2 = 2688.

The exponent can be positive or negative (e.g., "1e-3" = 0.001).

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

Examples:

```
asm_str2float("1.5e2", NULL, 10)    // Returns 150.0f
asm_str2float("-3.14e-1", NULL, 10) // Returns -0.314f
asm_str2float("FF.0e1", NULL, 16)   // Returns 4080.0f (255 × 16^1)
```

Definition at line 903 of file Almog_String_Manipulation.h.

References asm_check_char_belong_to_base(), asm_dprintERROR, asm_get_char_value_in_base(), asm_isspace(), and asm_str2int().

Referenced by main(), test_str2float_double(), test_str2float_double_exponent_different_bases(), test_str2float_double_exponent_lar
test_str2float_double_exponent_whitespace(), test_str2float_exponent_basic(), test_str2float_exponent_edge_cases(), test_str2float_exponent_signed_mantissa(), and test_str2float_exponent_with_trailing().

2.1.3.29 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 977 of file Almog_String_Manipulation.h.

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

Referenced by asm_str2double(), asm_str2float(), and test_str2int().

2.1.3.30 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 1022 of file Almog_String_Manipulation.h.

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

Referenced by test_str2size_t().

2.1.3.31 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 773 of file Almog_String_Manipulation.h.

References asm_length(), and asm_strncmp().

Referenced by test_str_in_str_overlap_and_edges().

2.1.3.32 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 1090 of file Almog_String_Manipulation.h.

References asm_isspace(), and asm_length().

Referenced by test_str_is_whitespace_edges().

2.1.3.33 asm_strdup()

```
char * asm_strdup (
            const char *const s,
            size_t length )
```

Allocate and copy up to `length` characters from `s`.

Allocates a new buffer of size (length + 1) bytes using ASM_MALLOC, copies up to `length` characters from `s`, and always null-terminates the result.

Parameters

s	Source string (must be null-terminated).
length	Maximum number of characters to copy (excluding '\0').

Returns

Newly allocated string, or NULL if allocation fails.

Note

This is not the same as POSIX strdup(): it does not compute length by itself and may intentionally truncate.

Definition at line 1115 of file Almog_String_Manipulation.h.

References ASM_MALLOC, and asm_strncpy().

2.1.3.34 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 1069 of file Almog_String_Manipulation.h.

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

Referenced by test_strip_whitespace_properties().

2.1.3.35 asm_strncat()

```
int asm_strncat (
            char *const s1,
            const char *const s2,
            const size_t 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 1143 of file Almog_String_Manipulation.h.

References asm_dprintERROR, asm_length(), and ASM_MAX_LEN.

Referenced by test_strncat_current_behavior_and_sentinel().

2.1.3.36 asm_strncmp()

```
int asm_strncmp (
            const char * s1,
            const char * s2,
            const size_t 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 1185 of file Almog_String_Manipulation.h.

References ASM_MAX_LEN.

Referenced by asm_str_in_str(), and test_strncmp_boolean_edges().

2.1.3.37 asm_strncpy()

```
int asm_strncpy (
            char *const s1,
            const char *const s2,
            const size_t N )
```

Copy up to `N` characters from `s2` into `s1` (non-standard).

Copies N characters from `s2` into `s1` and then writes a terminating '\0'.

Parameters

s1	Destination string buffer (must be null-terminated).
s2	Source string buffer (must be null-terminated).
N	Maximum number of characters to copy from `s2`.

Returns

The number of characters copied (i.e., (n)).

Definition at line 1213 of file Almog_String_Manipulation.h.

Referenced by asm_strdup().

2.1.3.38 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 1231 of file Almog_String_Manipulation.h.

References asm_isupper(), and asm_length().

Referenced by test_case_conversion_roundtrip().

2.1.3.39 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 1246 of file Almog_String_Manipulation.h.

References asm_islower(), and asm_length().

Referenced by test_case_conversion_roundtrip().

2.1.3.40 asm_trim_left_whitespace()

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

Remove leading ASCII whitespace from a string in-place.

Finds the first character in `s` for which asm_isspace() is false and left-shifts the string so that character becomes the first character.

Parameters

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

Definition at line 1264 of file Almog_String_Manipulation.h.

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

2.2 Almog_String_Manipulation.h

```
00001
00041 #ifndef ALMOG_STRING_MANIPULATION_H_
00042 #define ALMOG_STRING_MANIPULATION_H_
00043
00044 #include <stdio.h>
00045 #include <stdbool.h>
00046 #include <stdint.h>
00047
00048 #ifndef ASM_MALLOC
00049 #include <stdlib.h>
00050 #define ASM_MALLOC malloc
00051 #endif
00052
00068 #ifndef ASM_MAX_LEN
00069 #define ASM_MAX_LEN (int)1e3
00070 #endif
00071
00079 #define asm_dprintSTRING(expr) printf(#expr " = %s\n", expr)
00080
00088 #define asm_dprintCHAR(expr) printf(#expr " = %c\n", expr)
00089
00097 #define asm_dprintINT(expr) printf(#expr " = %d\n", expr)
00098
00106 #define asm_dprintFLOAT(expr) printf(#expr " = %#g\n", expr)
00107
00115 #define asm_dprintDOUBLE(expr) printf(#expr " = %#g\n", expr)
00116
00124 #define asm_dprintSIZE_T(expr) printf(#expr " = %zu\n", expr)
00125
00126 #define asm_dprintERROR(fmt, ...) \
00127     fprintf(stderr, "\n%s:%d:\n[Error] in function '%s':\n        " \
00128     fmt "\n\n", __FILE__, __LINE__, __func__, __VA_ARGS__)
00129
00141 #define asm_min(a, b) ((a) < (b) ? (a) : (b))
00142
00154 #define asm_max(a, b) ((a) > (b) ? (a) : (b))
00155
00156 bool    asm_check_char_belong_to_base(const char c, const size_t base);
00157 void    asm_copy_array_by_indexes(char * const target, const int start, const int end, const char *
         const src);
00158 int     asm_get_char_value_in_base(const char c, const size_t base);
00159 int     asm_get_line(FILE *fp, char * const dst);
00160 int     asm_get_next_token_from_str(char * const dst, const char * const src, const char delimiter);
00161 int     asm_get_token_and_cut(char * const dst, char *src, const char delimiter, const bool
         leave_delimiter);
00162 bool    asm_isalnum(const char c);
00163 bool    asm_isalpha(const char c);
00164 bool    asm_isbdigit(const char c);
00165 bool    asm_iscntrl(const char c);
00166 bool    asm_isdigit(const char c);
00167 bool    asm_isgraph(const char c);
00168 bool    asm_islower(const char c);
```

```
bool    asm_isodigit(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);
size_t  asm_length(const char * const str);
void *  asm_memset(void * const des, const unsigned char value, const size_t n);
void    asm_pad_left(char * const s, const size_t padding, const char pad);
void    asm_print_many_times(const char * const str, const size_t n);
void    asm_remove_char_from_string(char * const s, const size_t index);
void    asm_shift_left(char * const s, const size_t shift);
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);
char *  asm_strdup(const char * const s, size_t length);
int     asm_strncat(char * const s1, const char * const s2, const size_t N);
int     asm_strncmp(const char * const s1, const char * const s2, const size_t N);
int     asm_strncpy(char * const s1, const char * const s2, const size_t N);
void    asm_tolower(char * const s);
void    asm_toupper(char * const s);
void    asm_trim_left_whitespace(char *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 ASM_NO_ERRORS
        asm_dprintERROR("Supported bases are [2...36]. Inputted: %zu", 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 ASM_NO_ERRORS
            asm_dprintERROR("%s", "index exceeds ASM_MAX_LEN. Line in file is too long.");
```

```c
            #endif
            dst[i-1] = '\0';
            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);
    bool delimiter_at_start = src[new_src_start_index] == delimiter;

    if (leave_delimiter) {
        asm_shift_left(src, new_src_start_index);
    } else if (delimiter_at_start) {
        asm_shift_left(src, new_src_start_index + 1);
    } else {
        src[0] = '\0';
    }
    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_isbdigit(const char c)
{
    if (c == '0' || c == '1') {
        return true;
    } else {
        return false;
    }
}

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;
    }
}

```

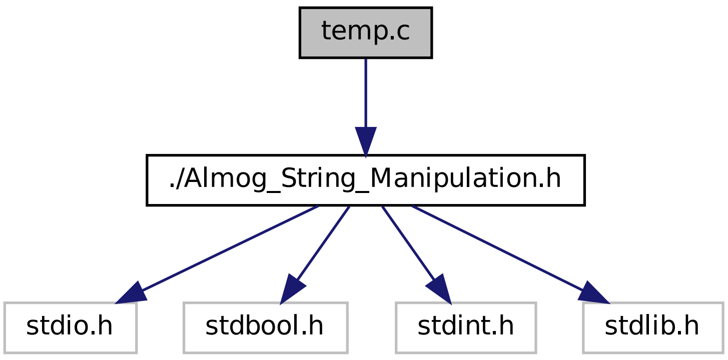
```
bool asm_islower(char c)
{
    if (c >= 'a' && c <= 'z') {
        return true;
    } else {
        return false;
    }
}

bool asm_isodigit(const char c)
{
    if ((c >= '0' && c <= '7')) {
        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;
    }
}

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 ASM_NO_ERRORS
            asm_dprintERROR("%s", "index exceeds ASM_MAX_LEN. Probably no NULL termination.");
            #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;
}

void asm_pad_left(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_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_from_string(char * const s, const size_t index)
{
    size_t len = asm_length(s);
    if (len == 0) return;
    if (index >= len) {
        #ifndef ASM_NO_ERRORS
        asm_dprintERROR("%s", "index exceeds array length.");
        #endif
        return;
    }

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

void asm_shift_left(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';
}

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 ASM_NO_ERRORS
        asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
        #endif
        if (end) *end = s;
        return 0.0;
    }
    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;
    double right = 0.0;
    int expo = 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] == '.') {
        i++; /* skip the point */

        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 ((s[i+num_of_whitespace] == 'e') || (s[i+num_of_whitespace] == 'E')) {
        expo = asm_str2int(&(s[i+num_of_whitespace+1]), end, 10);
    } else {
        if (end) *end = s + i + num_of_whitespace;
    }

    double res = sign * (left + right);

    if (expo > 0) {
        for (int index = 0; index < expo; index++) {
            res *= (double)base;
        }
    } else {
        for (int index = 0; index > expo; index--) {
            res /= (double)base;
        }
    }

    return res;
}

float asm_str2float(const char * const s, const char ** const end, const size_t base)
{
    if (base < 2 || base > 36) {
        #ifndef ASM_NO_ERRORS
        asm_dprintERROR("Supported bases are [2...36]. Input: %zu", 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;
    float right = 0.0f;
    int expo = 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] == '.') {
        i++; /* skip the point */

        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 ((s[i+num_of_whitespace] == 'e') || (s[i+num_of_whitespace] == 'E')) {
        expo = asm_str2int(&(s[i+num_of_whitespace+1]), end, 10);
    } else {
        if (end) *end = s + i + num_of_whitespace;
    }

    float res = sign * (left + right);

    if (expo > 0) {
        for (int index = 0; index < expo; index++) {
            res *= (float)base;
        }
    } else {
        for (int index = 0; index > expo; index--) {
            res /= (float)base;
        }
    }

    return res;
}

int asm_str2int(const char * const s, const char ** const end, const size_t base)
{
    if (base < 2 || base > 36) {
        #ifndef ASM_NO_ERRORS
        asm_dprintERROR("Supported bases are [2...36]. Input: %zu", 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 ASM_NO_ERRORS
        asm_dprintERROR("%s", "Unable to convert a negative number to size_t.");
        #endif
        return 0;
    }

    if (base < 2 || base > 36) {
        #ifndef ASM_NO_ERRORS
        asm_dprintERROR("Supported bases are [2...36]. Input: %zu", 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_from_string(s, i);
            len--;
            i--;
        }
    }
    s[i] = '\0';
}

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

    return true;
}

char * asm_strdup(const char * const s, size_t length)
{
    char * res = (char *)ASM_MALLOC(sizeof(char) * length+1);
    asm_strncpy((char * const)res, s, length);

    return res;
}

int asm_strncat(char * const s1, const char * const s2, const size_t 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-1) {
            #ifndef ASM_NO_ERRORS
            asm_dprintERROR("s2 or the first N=%zu digit of s2 does not fit into s1.", N);
            #endif
            return i;
        }

        s1[len_s1+(size_t)i] = s2[i];
        i++;
    }
    s1[len_s1+(size_t)i] = '\0';

    return i;
}

int asm_strncmp(const char *s1, const char *s2, const size_t N)
{
    size_t n = N == 0 ? ASM_MAX_LEN : N;
    size_t 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;
}

int asm_strncpy(char * const s1, const char * const s2, const size_t N)
{
    size_t n = N;

    size_t i;
    for (i = 0; i < n && s2[i] != '\0'; i++) {
        s1[i] = s2[i];
    }
    s1[i] = '\0';

    return i;
}

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';
        }
    }
}

void asm_trim_left_whitespace(char * const s)
{
    size_t len = asm_length(s);

    if (len == 0) return;
    size_t i;
    for (i = 0; i < len; i++) {
        if (!asm_isspace(s[i])) {
            break;
        }
    }
    asm_shift_left(s, i);
}

#ifdef ASM_NO_ERRORS
#undef ASM_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_dprintDOUBLE, asm_dprintFLOAT, asm_str2double(), and asm_str2float().

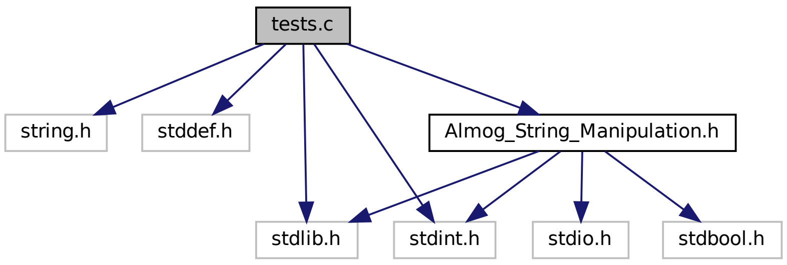
2.4 temp.c

```
00001 #define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
00002 #include "./Almog_String_Manipulation.h"
00003
00004 int main(void)
00005 {
00006     char str[] = "-1.1e-1";
00007
00008     asm_dprintFLOAT(asm_str2float(str, NULL, 10));
00009     asm_dprintDOUBLE(asm_str2double(str, NULL, 10));
00010
00011
00012
00013     return 0;
00014 }
```

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)
- static void test_str2float_exponent_basic (void)
- static void test_str2float_exponent_signed_mantissa (void)
- static void test_str2float_exponent_edge_cases (void)
- static void test_str2float_exponent_with_trailing (void)
- static void test_str2double_exponent_basic (void)
- static void test_str2double_exponent_signed_mantissa (void)
- static void test_str2double_exponent_edge_cases (void)
- static void test_str2float_double_exponent_different_bases (void)
- static void test_str2float_double_exponent_whitespace (void)
- static void test_str2float_double_exponent_large_values (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(), and test_get_line_tmpfile().

2.5.2.3 main()

```
int main (
            void )
```

Definition at line 1076 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_str2double_exponent_basic(), test_str2double_exponent_edge_cases(),
test_str2double_exponent_signed_mantissa(), test_str2float_double(), test_str2float_double_exponent_different_bases(),
test_str2float_double_exponent_large_values(), test_str2float_double_exponent_whitespace(), test_str2float_exponent_basic(),
test_str2float_exponent_edge_cases(), test_str2float_exponent_signed_mantissa(), test_str2float_exponent_with_trailing(),
test_str2int(), test_str2size_t(), test_str_in_str_overlap_and_edges(), test_str_is_whitespace_edges(), test_strip_whitespace_properti
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_pad_left(), 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_shift_left(), 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_from_string(), and TEST_EQ_STR.

Referenced by main().

2.5.2.19 test_str2double_exponent_basic()

```
static void test_str2double_exponent_basic (
             void  )  [static]
```

Definition at line 931 of file tests.c.

References asm_str2double(), and TEST_CASE.

Referenced by main().

2.5.2.20 test_str2double_exponent_edge_cases()

```
static void test_str2double_exponent_edge_cases (
             void  )  [static]
```

Definition at line 980 of file tests.c.

References asm_str2double(), and TEST_CASE.

Referenced by main().

2.5.2.21 test_str2double_exponent_signed_mantissa()

```
static void test_str2double_exponent_signed_mantissa (
             void  )  [static]
```

Definition at line 960 of file tests.c.

References asm_str2double(), and TEST_CASE.

Referenced by main().

2.5.2.22 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().

2.5.2.23 test_str2float_double_exponent_different_bases()

```
static void test_str2float_double_exponent_different_bases (
             void  )  [static]
```

Definition at line 1006 of file tests.c.

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

Referenced by main().

2.5.2.24 test_str2float_double_exponent_large_values()

```
static void test_str2float_double_exponent_large_values (
             void  )  [static]
```

Definition at line 1049 of file tests.c.

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

Referenced by main().

2.5.2.25 test_str2float_double_exponent_whitespace()

```
static void test_str2float_double_exponent_whitespace (
             void  )  [static]
```

Definition at line 1033 of file tests.c.

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

Referenced by main().

2.5.2.26 test_str2float_exponent_basic()

```
static void test_str2float_exponent_basic (
             void  )  [static]
```

Definition at line 791 of file tests.c.

References asm_str2float(), and TEST_CASE.

Referenced by main().

2.5.2.27 test_str2float_exponent_edge_cases()

```
static void test_str2float_exponent_edge_cases (
             void  )  [static]
```

Definition at line 865 of file tests.c.

References asm_str2float(), and TEST_CASE.

Referenced by main().

2.5.2.28 test_str2float_exponent_signed_mantissa()

```
static void test_str2float_exponent_signed_mantissa (
             void  )  [static]
```

Definition at line 832 of file tests.c.

References asm_str2float(), and TEST_CASE.

Referenced by main().

2.5.2.29 test_str2float_exponent_with_trailing()

```
static void test_str2float_exponent_with_trailing (
             void  )  [static]
```

Definition at line 912 of file tests.c.

References asm_str2float(), and TEST_CASE.

Referenced by main().

2.5.2.30 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().

2.5.2.31 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().

2.5.2.32 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().

2.5.2.33 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().

2.5.2.34 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().

2.5.2.35 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(), TEST_CASE, TEST_EQ_INT, and TEST_EQ_STR.

Referenced by main().

2.5.2.36 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().

2.5.2.37 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

```
00001 /* written by AI */
00002 /* test_almog_string_manipulation.c */
00003
00004 #include <string.h>
00005 #include <stddef.h>
00006 #include <stdlib.h>
00007 #include <stdint.h>
00008
00009 #define ALMOG_STRING_MANIPULATION_IMPLEMENTATION
00010 #define NO_ERRORS
00011 #include "Almog_String_Manipulation.h"
00012
00013 /* ---------------- Test harness ---------------- */
00014
00015 static int g_tests_run = 0;
00016 static int g_tests_failed = 0;
00017 static int g_tests_warned = 0;
00018
00019 #define TEST_CASE(expr)                                                       \
00020     do {                                                                      \
00021         g_tests_run++;                                                        \
00022         if (!(expr)) {                                                        \
00023             g_tests_failed++;                                                 \
00024             fprintf(stderr, "[FAIL] %s:%d: %s\n", __FILE__, __LINE__, #expr); \
00025         }                                                                     \
00026     } while (0)
00027
00028 #define TEST_WARN(expr, msg)                                                  \
00029     do {                                                                      \
00030         g_tests_run++;                                                        \
00031         if (!(expr)) {                                                        \
00032             g_tests_warned++;                                                 \
00033             fprintf(stderr, "[WARN] %s:%d: %s | %s\n", __FILE__, __LINE__,    \
00034                     #expr, msg);                                              \
00035         }                                                                     \
    } 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));

        /* 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];
```

```c

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

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

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

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

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

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

    strcpy(box.s, "x");
    asm_pad_left(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_from_string(s, 1);
    TEST_EQ_STR(s, "acd");

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

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

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

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

    strcpy(s, "abcd");
    asm_remove_char_from_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";
```

```c
        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);
}

```

```c
/* 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_EQ_STR(box.s1, "abcDEF");

    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);
}

/* ---------------- Tests: str2float/double with exponent notation ---------------- */

static void test_str2float_exponent_basic(void)
{
    const char *end = NULL;
    float v;

    /* Basic positive exponents */
    v = asm_str2float("1e2", &end, 10);
    TEST_CASE(v > 99.9f && v < 100.1f);
    TEST_CASE(*end == '\0');

    v = asm_str2float("1.5e3", &end, 10);
    TEST_CASE(v > 1499.9f && v < 1500.1f);
    TEST_CASE(*end == '\0');

    v = asm_str2float("5e2", &end, 10);
    TEST_CASE(v > 499.9f && v < 500.1f);
    TEST_CASE(*end == '\0');

    /* Basic negative exponents */
    v = asm_str2float("1e-2", &end, 10);
    TEST_CASE(v > 0.0099f && v < 0.0101f);
    TEST_CASE(*end == '\0');

    v = asm_str2float("5e-1", &end, 10);
    TEST_CASE(v > 0.49f && v < 0.51f);
    TEST_CASE(*end == '\0');

    v = asm_str2float("2.5e-3", &end, 10);
    TEST_CASE(v > 0.00249f && v < 0.00251f);
    TEST_CASE(*end == '\0');

    /* Exponent with explicit positive sign */
    v = asm_str2float("1e+2", &end, 10);
    TEST_CASE(v > 99.9f && v < 100.1f);
    TEST_CASE(*end == '\0');

    v = asm_str2float("3.5e+1", &end, 10);
    TEST_CASE(v > 34.9f && v < 35.1f);
    TEST_CASE(*end == '\0');
}

static void test_str2float_exponent_signed_mantissa(void)
{
    const char *end = NULL;
    float v;

    /* Negative mantissa with positive exponent */
    v = asm_str2float("-1e2", &end, 10);
    TEST_CASE(v > -100.1f && v < -99.9f);
    TEST_CASE(*end == '\0');

    v = asm_str2float("-2.5e3", &end, 10);
    TEST_CASE(v > -2500.1f && v < -2499.9f);
    TEST_CASE(*end == '\0');

    /* Negative mantissa with negative exponent */
    v = asm_str2float("-1.0e-2", &end, 10);
    TEST_CASE(v > -0.0101f && v < -0.0099f);
    TEST_CASE(*end == '\0');

    v = asm_str2float("-5e-1", &end, 10);
    TEST_CASE(v > -0.51f && v < -0.49f);
    TEST_CASE(*end == '\0');

    /* Positive sign with exponent */
    v = asm_str2float("+1.5e2", &end, 10);
    TEST_CASE(v > 149.9f && v < 150.1f);
    TEST_CASE(*end == '\0');

    v = asm_str2float("+3e-2", &end, 10);
    TEST_CASE(v > 0.0299f && v < 0.0301f);
    TEST_CASE(*end == '\0');
}

static void test_str2float_exponent_edge_cases(void)
{
    const char *end = NULL;
    float v;

    /* Zero exponent */
    v = asm_str2float("5e0", &end, 10);
    TEST_CASE(v > 4.99f && v < 5.01f);
    TEST_CASE(*end == '\0');

    v = asm_str2float("3.14e0", &end, 10);
    TEST_CASE(v > 3.13f && v < 3.15f);
    TEST_CASE(*end == '\0');

    /* Zero mantissa */
    v = asm_str2float("0e5", &end, 10);
    TEST_CASE(v > -0.01f && v < 0.01f);
    TEST_CASE(*end == '\0');

    v = asm_str2float("0.0e-3", &end, 10);
    TEST_CASE(v > -0.01f && v < 0.01f);
    TEST_CASE(*end == '\0');

    /* No integer part */
    v = asm_str2float(".5e2", &end, 10);
    TEST_CASE(v > 49.9f && v < 50.1f);
    TEST_CASE(*end == '\0');

    v = asm_str2float(".25e-1", &end, 10);
    TEST_CASE(v > 0.0249f && v < 0.0251f);
    TEST_CASE(*end == '\0');

    /* No fractional part */
    v = asm_str2float("10.e2", &end, 10);
    TEST_CASE(v > 999.9f && v < 1000.1f);
    TEST_CASE(*end == '\0');

    /* Uppercase E */
    v = asm_str2float("1E2", &end, 10);
    TEST_CASE(v > 99.9f && v < 100.1f);
    TEST_CASE(*end == '\0');
    v = asm_str2float("5E-3", &end, 10);
    TEST_CASE(v > 0.00499f && v < 0.00501f);
    TEST_CASE(*end == '\0');
}

static void test_str2float_exponent_with_trailing(void)
{
    const char *end = NULL;
    float v;

    /* Exponent with trailing characters */
    v = asm_str2float("1.5e2xyz", &end, 10);
    TEST_CASE(v > 149.9f && v < 150.1f);
    TEST_CASE(*end == 'x');

    v = asm_str2float("3e-1!", &end, 10);
    TEST_CASE(v > 0.29f && v < 0.31f);
    TEST_CASE(*end == '!');

    v = asm_str2float("  -2.5e3  ", &end, 10);
    TEST_CASE(v > -2500.1f && v < -2499.9f);
    TEST_CASE(*end == ' ');
}

static void test_str2double_exponent_basic(void)
{
    const char *end = NULL;
    double v;

    /* Basic positive exponents */
    v = asm_str2double("1e2", &end, 10);
    TEST_CASE(v > 99.99 && v < 100.01);
    TEST_CASE(*end == '\0');

    v = asm_str2double("1.5e3", &end, 10);
    TEST_CASE(v > 1499.99 && v < 1500.01);
    TEST_CASE(*end == '\0');

    /* Basic negative exponents */
    v = asm_str2double("1e-2", &end, 10);
    TEST_CASE(v > 0.0099 && v < 0.0101);
    TEST_CASE(*end == '\0');

    v = asm_str2double("-1.0e-2", &end, 10);
    TEST_CASE(v > -0.0101 && v < -0.0099);
    TEST_CASE(*end == '\0');

    /* Higher precision than float */
    v = asm_str2double("3.141592653589793e0", &end, 10);
    TEST_CASE(v > 3.141592653 && v < 3.141592654);
    TEST_CASE(*end == '\0');
}

static void test_str2double_exponent_signed_mantissa(void)
{
    const char *end = NULL;
    double v;

    /* Negative mantissa with exponents */
    v = asm_str2double("-2.5e3", &end, 10);
    TEST_CASE(v > -2500.01 && v < -2499.99);
    TEST_CASE(*end == '\0');

    v = asm_str2double("-5e-1", &end, 10);
    TEST_CASE(v > -0.51 && v < -0.49);
    TEST_CASE(*end == '\0');

    /* Positive sign */
    v = asm_str2double("+1.5e2", &end, 10);
    TEST_CASE(v > 149.99 && v < 150.01);
    TEST_CASE(*end == '\0');
}

static void test_str2double_exponent_edge_cases(void)
{
    const char *end = NULL;
    double v;

    /* Zero exponent */
    v = asm_str2double("5e0", &end, 10);
    TEST_CASE(v > 4.99 && v < 5.01);
    TEST_CASE(*end == '\0');

    /* Zero mantissa */
    v = asm_str2double("0e5", &end, 10);
    TEST_CASE(v > -0.01 && v < 0.01);
    TEST_CASE(*end == '\0');

    /* No integer part */
    v = asm_str2double(".5e2", &end, 10);
    TEST_CASE(v > 49.99 && v < 50.01);
    TEST_CASE(*end == '\0');

    /* Uppercase E */
    v = asm_str2double("1E2", &end, 10);
    TEST_CASE(v > 99.99 && v < 100.01);
    TEST_CASE(*end == '\0');
}

static void test_str2float_double_exponent_different_bases(void)
{
    const char *end = NULL;
    float vf;
    double vd;

    /* Binary with exponent (base 2)
     * 1.1e3 in base 2 = 1.5 * 2^3 = 1.5 * 8 = 12 */
    vf = asm_str2float("1.1e3", &end, 2);
    TEST_CASE(vf > 11.9f && vf < 12.1f);
    TEST_CASE(*end == '\0');

    vd = asm_str2double("1.1e3", &end, 2);
    TEST_CASE(vd > 11.99 && vd < 12.01);
    TEST_CASE(*end == '\0');

    /* Octal with exponent (base 8)
     * 7.4e2 in base 8 = (7 + 4/8) * 8^2 = 7.5 * 64 = 480 */
    vf = asm_str2float("7.4e2", &end, 8);
    TEST_CASE(vf > 479.9f && vf < 480.1f);
    TEST_CASE(*end == '\0');

    vd = asm_str2double("7.4e2", &end, 8);
    TEST_CASE(vd > 479.99 && vd < 480.01);
    TEST_CASE(*end == '\0');
}

static void test_str2float_double_exponent_whitespace(void)
{
    const char *end = NULL;
    float vf;
    double vd;

    /* Leading whitespace */
    vf = asm_str2float("  \t\n1.5e2", &end, 10);
    TEST_CASE(vf > 149.9f && vf < 150.1f);
    TEST_CASE(*end == '\0');

    vd = asm_str2double("  \t\n-2.5e-3", &end, 10);
    TEST_CASE(vd > -0.00251 && vd < -0.00249);
    TEST_CASE(*end == '\0');
}

static void test_str2float_double_exponent_large_values(void)
{
    const char *end = NULL;
    float vf;
    double vd;

    /* Larger exponents */
    vf = asm_str2float("1e5", &end, 10);
    TEST_CASE(vf > 99999.0f && vf < 100001.0f);
    TEST_CASE(*end == '\0');

    vd = asm_str2double("1e10", &end, 10);
    TEST_CASE(vd > 9999999999.0 && vd < 10000000001.0);
    TEST_CASE(*end == '\0');

    /* Very small exponents */
    vf = asm_str2float("1e-5", &end, 10);
    TEST_CASE(vf > 0.000009f && vf < 0.000011f);
    TEST_CASE(*end == '\0');

    vd = asm_str2double("1e-10", &end, 10);
    TEST_CASE(vd > 0.00000000009 && vd < 0.00000000011);
    TEST_CASE(*end == '\0');
}

/* ---------------- 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_str2float_exponent_basic();
    test_str2float_exponent_signed_mantissa();
    test_str2float_exponent_edge_cases();
    test_str2float_exponent_with_trailing();
    test_str2double_exponent_basic();
    test_str2double_exponent_signed_mantissa();
    test_str2double_exponent_edge_cases();
    test_str2float_double_exponent_different_bases();
    test_str2float_double_exponent_whitespace();
    test_str2float_double_exponent_large_values();

    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