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

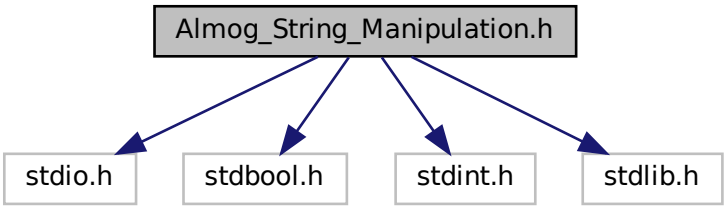
File Documentation

2.1 Almog_String_Manipulation.h File Reference

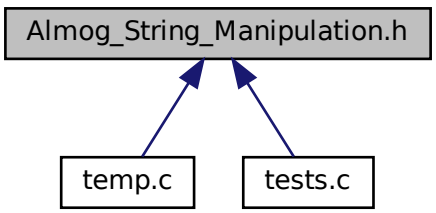
Lightweight string and line manipulation helpers.

```
#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_FREE free
- #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, ...)

  Print a formatted error message to stderr with file/line/function context.
- #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).
- #define asm_length(str) __asm_length(str, __FILE__, __LINE__, __func__)

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, char *file_name, int line_num, char *function_name)

  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 size_t src_len, const char *const word_to_search, const char **first_occurrence)

  Count occurrences of a substring within a string.
- int asm_str_in_str_case_insensitive (const char *const src, const size_t src_len, const char *const word_↩ to_search, const char **first_occurrence)

  Count occurrences of a substring within a string, case-insensitively (ASCII-only; even indices only).
- 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)
- 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_strncmp_case_insensitive (const char *const s1, const char *const s2, const size_t N)

  Compare up to N characters for equality, ASCII case-insensitively.
- 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, const size_t len)

  Convert ASCII uppercase letters to lowercase in-place, up to a limit or until a sentinel character is encountered.
- void asm_toupper (char *const s, const size_t len)

  Convert ASCII lowercase letters to uppercase in-place, up to a limit or until a sentinel character is encountered.
- 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 from the source buffer (optionally also removing the delimiter)
- Copying a substring by indices
- Counting occurrences of a substring (see asm_str_in_str() notes for the exact scan pattern)
- A boolean-style strncmp (returns 1 on equality, 0 otherwise)
- Case-insensitive boolean-style strncmp (ASCII-only; see asm_strncmp_case_insensitive() notes/constraints)
- 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 stores at most ASM_MAX_LEN - 1 characters (plus '\0'). Lines longer than that cause an early return with an error message. asm_length uses ASM_MAX_LEN as a sanity limit when scanning for '\0'.
- 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 98 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 125 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__)
```

Print a formatted error message to stderr with file/line/function context.

Parameters

fmt	printf-style format string.
...	printf-style arguments for `fmt`.

Note

This macro requires at least one variadic argument in addition to `fmt` (because it unconditionally uses **VA_ARGS**).

Definition at line 147 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 116 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 107 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 134 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 89 of file Almog_String_Manipulation.h.

2.1.2.8 ASM_FREE

```
#define ASM_FREE free
```

Definition at line 58 of file Almog_String_Manipulation.h.

2.1.2.9 asm_length

```
#define asm_length(
            str ) __asm_length(str, __FILE__, __LINE__, __func__)
```

Definition at line 197 of file Almog_String_Manipulation.h.

2.1.2.10 ASM_MALLOC

```
#define ASM_MALLOC malloc
```

Definition at line 53 of file Almog_String_Manipulation.h.

2.1.2.11 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 175 of file Almog_String_Manipulation.h.

2.1.2.12 ASM_MAX_LEN

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

Maximum number of characters processed in some string operations.

This constant is used as a fixed, caller-provided buffer size / sanity limit:

- asm_get_line() writes at most ASM_MAX_LEN - 1 characters to the destination buffer and always reserves 1 byte for the terminating '\0'.
- asm_length() uses ASM_MAX_LEN as a safety bound while searching for '\0' (it returns SIZE_MAX if no terminator is found within that bound).

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 79 of file Almog_String_Manipulation.h.

2.1.2.13 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 162 of file Almog_String_Manipulation.h.

2.1.3 Function Documentation

2.1.3.1 __asm_length()

```
size_t __asm_length (
            const char *const str,
            char * file_name,
            int line_num,
            char * function_name )
```

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 654 of file Almog_String_Manipulation.h.

References asm_dprintERROR, and ASM_MAX_LEN.

2.1.3.2 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 236 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().

2.1.3.3 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.

Note

If `start > end`, this function returns immediately and leaves `target` unchanged (it does not write a terminator in that case).

If the copied range includes a '\0' from `src`, `target` will also contain that '\0' at the corresponding position and no extra '\0' is appended.

Definition at line 276 of file Almog_String_Manipulation.h.

Referenced by test_copy_array_by_indexes_behavior_and_bounds().

2.1.3.4 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 298 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().

2.1.3.5 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.

On the "line too long" error path, this function returns immediately after truncating `dst` and does not consume the rest of the current line from `fp`. A subsequent call will continue reading from the same (still-unfinished) line.

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

Definition at line 334 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().

2.1.3.6 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 381 of file Almog_String_Manipulation.h.

Referenced by asm_get_token_and_cut(), and test_get_next_word_from_line_current_behavior().

2.1.3.7 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 shifted left by the token length.
  - If a delimiter was present, it becomes the first character of the updated `src`.
- If `leave_delimiter` is false:
  - If a delimiter is present immediately after the token, `src` is shifted left by (token length + 1), removing exactly one delimiter.
  - If no delimiter is present (the token reaches '\0'), `src` is set to the empty string.

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 a non-empty token was extracted (token length != 0), otherwise 0. (Note: an empty token may still cause `src` to change, e.g., when `src` begins with the delimiter and `leave_delimiter` is false, the delimiter is removed but 0 is returned.)

Note

This function does not skip whitespace. Any leading whitespace is part of the extracted token (until the delimiter or '\0').

Definition at line 427 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().

2.1.3.8 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 448 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().

2.1.3.9 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 459 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().

2.1.3.10 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 470 of file Almog_String_Manipulation.h.

2.1.3.11 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 485 of file Almog_String_Manipulation.h.

Referenced by test_ascii_classification_exhaustive_ranges().

2.1.3.12 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 500 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().

2.1.3.13 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 515 of file Almog_String_Manipulation.h.

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

2.1.3.14 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 530 of file Almog_String_Manipulation.h.

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

2.1.3.15 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 545 of file Almog_String_Manipulation.h.

2.1.3.16 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 561 of file Almog_String_Manipulation.h.

References asm_isgraph().

Referenced by test_ascii_classification_exhaustive_ranges(), and test_ascii_classification_full_scan_0_127().

2.1.3.17 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 573 of file Almog_String_Manipulation.h.

Referenced by test_ascii_classification_exhaustive_ranges().

2.1.3.18 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 589 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

2.1.3.19 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 605 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().

2.1.3.20 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 620 of file Almog_String_Manipulation.h.

References asm_isdigit().

Referenced by test_ascii_classification_exhaustive_ranges().

2.1.3.21 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 635 of file Almog_String_Manipulation.h.

References asm_isdigit().

Referenced by test_ascii_classification_exhaustive_ranges().

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 689 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 pad characters 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 712 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 729 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 749 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 778 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 is parsed via asm_str2int(), which skips leading ASCII whitespace. As a result, strings like "1e 2" may be accepted (expo=2) even though standard C conversions typically stop at the 'e'.

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 934 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 is parsed via asm_str2int(), which skips leading ASCII whitespace. As a result, strings like "1e 2" may be accepted (expo=2) even though standard C conversions typically stop at the 'e'.

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 1024 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 1098 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 1143 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 size_t src_len,
            const char *const word_to_search,
            const char ** first_occurrence )
```

Count occurrences of a substring within a string.

Scans `src` from left to right and tests for a match of `word_to_search` starting at each index (i = 0, 1, 2, \dots) until either:

- a terminating '\0' is encountered in `src`, or
- (i) reaches the configured cap.

On each index (i), this function checks equality using asm_strncmp(src + i, word_to_search, asm_length(word_to_search)). Matches may overlap.

Parameters

src	String to search in (must be null-terminated).
src_len	If non-zero, limits the scan to start indices (i <) `src_len` (the scan also stops earlier at '\0'). If zero, ASM_MAX_LEN is used as the cap.
word_to_search	Substring to find (must be null-terminated).
first_occurrence	Output parameter. On the first match found, this function stores a pointer to the matched position within `src` into `*first_occurrence`.

Returns

The number of matches found. Matches may overlap.

Note

If no matches are found, `*first_occurrence` is left unchanged. If you need a deterministic "not found" value, initialize it to NULL before calling.

Definition at line 822 of file Almog_String_Manipulation.h.

References asm_length, ASM_MAX_LEN, and asm_strncmp().

Referenced by test_str_in_str_overlap_and_edges().

2.1.3.32 asm_str_in_str_case_insensitive()

```
int asm_str_in_str_case_insensitive (
            const char *const src,
            const size_t src_len,
            const char *const word_to_search,
            const char ** first_occurrence )
```

Count occurrences of a substring within a string, case-insensitively (ASCII-only; even indices only).

Scans `src` from left to right and tests for a match of `word_to_search` starting at each index (i = 0, 1, 2, \dots) until either:

- a terminating '\0' is encountered in `src`, or
- (i) reaches the configured cap.

Matching is performed by calling: asm_strncmp_case_insensitive(src + i, word_to_search, asm_length(word_to_search)).

Parameters

src	String to search in (must be null-terminated).
src_len	If non-zero, limits the scan to start indices (i <) `src_len` (the scan also stops earlier at '\0'). If zero, ASM_MAX_LEN is used as the cap.
word_to_search	Substring to find (must be null-terminated).
first_occurrence	Output parameter. On the first match found, this function stores a pointer to the matched
Generated by Doxygen	position within `src` into `*first_occurrence`.

Returns

The number of matches found. Matches may overlap.

Warning

Due to the current implementation of asm_strncmp_case_insensitive() (it lowercases exactly N bytes in temporary buffers), this function can invoke undefined behavior when a tested position `src` + i is shorter than N bytes (i.e., too close to the terminating '\0'). To keep behavior defined, the caller should bound the scan so that every tested start index has at least N characters available, e.g.: let needle_len = asm_length(word_to_search) and hay_len = asm_length(src); then use src_len <= (hay_len >= needle_len ? hay_len - needle_len + 1 : 0).

Note

If no matches are found, `*first_occurrence` is left unchanged. If you need a deterministic "not found" value, initialize it to NULL before calling.

ASCII-only: locale-specific case mappings are not supported.

Definition at line 880 of file Almog_String_Manipulation.h.

References asm_length, ASM_MAX_LEN, and asm_strncmp_case_insensitive().

2.1.3.33 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 1211 of file Almog_String_Manipulation.h.

References asm_isspace(), and asm_length.

Referenced by test_str_is_whitespace_edges().

2.1.3.34 asm_strdup()

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

```
@brief Allocate and copy up to @p length characters from @p s.

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

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

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

- *

  Note

  Allocation failure is not handled: if ASM_MALLOC returns NULL, this

- * implementation will pass NULL to asm_strncpy(), resulting in undefined
- * behavior.
- *

  Note

  The returned pointer must be released with the matching deallocator

- * for ASM_MALLOC.

Definition at line 1241 of file Almog_String_Manipulation.h.

References ASM_MALLOC, and asm_strncpy().

Referenced by asm_strncmp_case_insensitive().

2.1.3.35 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 1190 of file Almog_String_Manipulation.h.

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

Referenced by test_strip_whitespace_properties().

2.1.3.36 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 buffer size and enforces a maximum resulting string length of ASM_MAX_LEN - 1 (excluding the terminating '\0'). The caller must ensure `s1` has capacity of at least ASM_MAX_LEN bytes.

Definition at line 1270 of file Almog_String_Manipulation.h.

References asm_dprintERROR, asm_length, and ASM_MAX_LEN.

Referenced by test_strncat_current_behavior_and_sentinel().

2.1.3.37 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. If N == 0, this implementation compares up to ASM_MAX_LEN characters.

Returns

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

Note

If both strings terminate ('\0') at the same position before `N`, they are considered equal.

If either string ends before the other (within `N`), the strings are considered different.

Definition at line 1315 of file Almog_String_Manipulation.h.

References ASM_MAX_LEN.

Referenced by asm_str_in_str(), and test_strncmp_boolean_edges().

2.1.3.38 asm_strncmp_case_insensitive()

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

Compare up to N characters for equality, ASCII case-insensitively.

Returns 1 if the first `N` characters of `s1` and `s2` are equal when compared case-insensitively using ASCII rules; otherwise returns 0.

Internally, this implementation duplicates both strings (up to `N` bytes), lowercases exactly `N` bytes in each duplicate, and then compares.

Parameters

s1	First string (must be null-terminated).
s2	Second string (must be null-terminated).
N	Number of characters to compare.

Returns

1 if equal (case-insensitive) for the first `N` characters, otherwise 0.

Warning

For defined behavior, N should not exceed the length (in bytes) of either input string (i.e., avoid N > asm_length(s1) or N > asm_length(s2)). Otherwise, the internal lowercasing step may read and modify uninitialized bytes in the temporary buffers.

Note

ASCII-only: locale-specific case mappings are not supported.

Definition at line 1354 of file Almog_String_Manipulation.h.

References ASM_FREE, ASM_MAX_LEN, asm_strdup(), and asm_tolower().

Referenced by asm_str_in_str_case_insensitive().

2.1.3.39 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 characters from `s2` into `s1` until either:

- `N` characters were copied, or
- a '\0' is encountered in `s2`, and then writes a terminating '\0' to `s1`.

This differs from the standard strncpy(): it does not pad with additional '\0' bytes up to `N`.

Parameters

s1	Destination string buffer (need not be null-terminated on entry).
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 1401 of file Almog_String_Manipulation.h.

References ASM_MAX_LEN.

Referenced by asm_strdup().

2.1.3.40 asm_tolower()

```
void asm_tolower (
            char *const s,
            const size_t len )
```

Convert ASCII uppercase letters to lowercase in-place, up to a limit or until a sentinel character is encountered.

Iterates over `s` and converts each ASCII uppercase letter ('A'–'Z') to its lowercase form. The loop stops when either:

- `len` bytes have been processed, or
- the character '\0' is encountered in `s`.

Parameters

s	Buffer to modify in-place.
len	Maximum number of bytes to examine/modify.

Note

ASCII-only; not locale aware.

Definition at line 1428 of file Almog_String_Manipulation.h.

References asm_isupper().

Referenced by asm_strncmp_case_insensitive(), and test_case_conversion_roundtrip().

2.1.3.41 asm_toupper()

```
void asm_toupper (
            char *const s,
            const size_t len )
```

Convert ASCII lowercase letters to uppercase in-place, up to a limit or until a sentinel character is encountered.

Iterates over `s` and converts each ASCII lowercase letter ('a'–'z') to its uppercase form. The loop stops when either:

- `len` bytes have been processed, or
- the character '\0' is encountered in `s`.

Parameters

s	Buffer to modify in-place.
len	Maximum number of bytes to examine/modify.

Note

ASCII-only; not locale aware.

Definition at line 1451 of file Almog_String_Manipulation.h.

References asm_islower().

Referenced by test_case_conversion_roundtrip().

2.1.3.42 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 1468 of file Almog_String_Manipulation.h.

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

2.2 Almog_String_Manipulation.h

```
00001
00044 #ifndef ALMOG_STRING_MANIPULATION_H_
00045 #define ALMOG_STRING_MANIPULATION_H_
00046
00047 #include <stdio.h>
00048 #include <stdbool.h>
00049 #include <stdint.h>
00050
00051 #ifndef ASM_MALLOC
00052 #include <stdlib.h>
00053 #define ASM_MALLOC malloc
00054 #endif
00055
00056 #ifndef ASM_FREE
00057 #include <stdlib.h>
00058 #define ASM_FREE free
00059 #endif
00060
00078 #ifndef ASM_MAX_LEN
00079 #define ASM_MAX_LEN (int)1e3
00080 #endif
```

```
00081
00089 #define asm_dprintSTRING(expr) printf(#expr " = %s\n", expr)
00090
00098 #define asm_dprintCHAR(expr) printf(#expr " = %c\n", expr)
00099
00107 #define asm_dprintINT(expr) printf(#expr " = %d\n", expr)
00108
00116 #define asm_dprintFLOAT(expr) printf(#expr " = %#g\n", expr)
00117
00125 #define asm_dprintDOUBLE(expr) printf(#expr " = %#g\n", expr)
00126
00134 #define asm_dprintSIZE_T(expr) printf(#expr " = %zu\n", expr)
00135
00147 #define asm_dprintERROR(fmt, ...) \
00148     fprintf(stderr, "\n%s:%d:\n[Error] in function '%s':\n        " \
00149     fmt "\n\n", __FILE__, __LINE__, __func__, __VA_ARGS__)
00150
00162 #define asm_min(a, b) ((a) < (b) ? (a) : (b))
00163
00175 #define asm_max(a, b) ((a) > (b) ? (a) : (b))
00176
00177 bool    asm_check_char_belong_to_base(const char c, const size_t base);
00178 void    asm_copy_array_by_indexes(char * const target, const int start, const int end, const char *
       const src);
00179 int     asm_get_char_value_in_base(const char c, const size_t base);
00180 int     asm_get_line(FILE *fp, char * const dst);
00181 int     asm_get_next_token_from_str(char * const dst, const char * const src, const char delimiter);
00182 int     asm_get_token_and_cut(char * const dst, char *src, const char delimiter, const bool
       leave_delimiter);
00183 bool    asm_isalnum(const char c);
00184 bool    asm_isalpha(const char c);
00185 bool    asm_isbdigit(const char c);
00186 bool    asm_iscntrl(const char c);
00187 bool    asm_isdigit(const char c);
00188 bool    asm_isgraph(const char c);
00189 bool    asm_islower(const char c);
00190 bool    asm_isodigit(const char c);
00191 bool    asm_isprint(const char c);
00192 bool    asm_ispunct(const char c);
00193 bool    asm_isspace(const char c);
00194 bool    asm_isupper(const char c);
00195 bool    asm_isxdigit(const char c);
00196 bool    asm_isXdigit(const char c);
00197 #define asm_length(str) __asm_length(str, __FILE__, __LINE__, __func__)
00198 size_t  __asm_length(const char * const str, char *file_name, int line_num, char *function_name);
00199 void *  asm_memset(void * const des, const unsigned char value, const size_t n);
00200 void    asm_pad_left(char * const s, const size_t padding, const char pad);
00201 void    asm_print_many_times(const char * const str, const size_t n);
00202 void    asm_remove_char_from_string(char * const s, const size_t index);
00203 void    asm_shift_left(char * const s, const size_t shift);
00204 int     asm_str_in_str(const char * const src, const size_t src_len, const char * const
       word_to_search, const char **first_occurrence);
00205 int     asm_str_in_str_case_insensitive(const char * const src, const size_t src_len, const char *
       const word_to_search, const char **first_occurrence);
00206 double  asm_str2double(const char * const s, const char ** const end, const size_t base);
00207 float   asm_str2float(const char * const s, const char ** const end, const size_t base);
00208 int     asm_str2int(const char * const s, const char ** const end, const size_t base);
00209 size_t  asm_str2size_t(const char * const s, const char ** const end, const size_t base);
00210 void    asm_strip_whitespace(char * const s);
00211 bool    asm_str_is_whitespace(const char * const s);
00212 char *  asm_strdup(const char * const s, size_t length);
00213 int     asm_strncat(char * const s1, const char * const s2, const size_t N);
00214 int     asm_strncmp(const char * const s1, const char * const s2, const size_t N);
00215 int     asm_strncmp_case_insensitive(const char * const s1, const char * const s2, const size_t N);
00216 int     asm_strncpy(char * const s1, const char * const s2, const size_t N);
00217 void    asm_tolower(char * const s, const size_t len);
00218 void    asm_toupper(char * const s, const size_t len);
00219 void    asm_trim_left_whitespace(char *s);
00220
00221 #endif /*ALMOG_STRING_MANIPULATION_H_*/
00222
00223 #ifdef ALMOG_STRING_MANIPULATION_IMPLEMENTATION
00224 #undef ALMOG_STRING_MANIPULATION_IMPLEMENTATION
00225
00236 bool asm_check_char_belong_to_base(const char c, const size_t base)
00237 {
00238     if (base > 36 || base < 2) {
00239         #ifndef ASM_NO_ERRORS
00240         asm_dprintERROR("Supported bases are [2...36]. Inputted: %zu", base);
00241         #endif
00242         return false;
00243     }
00244     if (base <= 10) {
00245         return c >= '0' && c <= '9'+(char)base-10;
00246     }
00247     if (base > 10) {
00248         return asm_isdigit(c) || (c >= 'A' && c <= ('A'+(char)base-11)) || (c >= 'a' && c <=
```

```
      ('a'+(char)base-11));
00249     }
00250
00251     return false;
00252 }
00253
00276 void asm_copy_array_by_indexes(char * const target, const int start, const int end, const char * const
      src)
00277 {
00278     if (start > end) return;
00279     int j = 0;
00280     for (int i = start; i <= end; i++) {
00281         target[j] = src[i];
00282         j++;
00283     }
00284     if (target[j-1] != '\0') {
00285         target[j] = '\0';
00286     }
00287 }
00288
00298 int asm_get_char_value_in_base(const char c, const size_t base)
00299 {
00300     if (!asm_check_char_belong_to_base(c, base)) return -1;
00301     if (asm_isdigit(c)) {
00302         return c - '0';
00303     } else if (asm_isupper(c)) {
00304         return c - 'A' + 10;
00305     } else {
00306         return c - 'a' + 10;
00307     }
00308 }
00309
00334 int asm_get_line(FILE *fp, char * const dst)
00335 {
00336     int i = 0;
00337     int c;
00338     while ((c = fgetc(fp)) != '\n' && c != EOF) {
00339         dst[i++] = (char)c;
00340         if (i >= ASM_MAX_LEN) {
00341             #ifndef ASM_NO_ERRORS
00342             asm_dprintERROR("%s", "index exceeds ASM_MAX_LEN. Line in file is too long.");
00343             #endif
00344             dst[i-1] = '\0';
00345             return -1;
00346         }
00347     }
00348     dst[i] = '\0';
00349     if (c == EOF && i == 0) {
00350         return -1;
00351     }
00352     return i;
00353 }
00354
00381 int asm_get_next_token_from_str(char * const dst, const char * const src, const char delimiter)
00382 {
00383     int i = 0, j = 0;
00384     char c;
00385     while ((c = src[i]) != delimiter && c != '\0') {
00386         dst[j++] = src[i++];
00387     }
00388
00389     dst[j] = '\0';
00390
00391     return j;
00392 }
00393
00427 int asm_get_token_and_cut(char * const dst, char *src, const char delimiter, const bool
      leave_delimiter)
00428 {
00429     int new_src_start_index = asm_get_next_token_from_str(dst, src, delimiter);
00430     bool delimiter_at_start = src[new_src_start_index] == delimiter;
00431
00432     if (leave_delimiter) {
00433         asm_shift_left(src, new_src_start_index);
00434     } else if (delimiter_at_start) {
00435         asm_shift_left(src, new_src_start_index + 1);
00436     } else {
00437         src[0] = '\0';
00438     }
00439     return new_src_start_index ? 1 : 0;
00440 }
00441
00448 bool asm_isalnum(char c)
00449 {
00450     return asm_isalpha(c) || asm_isdigit(c);
00451 }
00452
```

```
00459 bool asm_isalpha(char c)
00460 {
00461     return asm_isupper(c) || asm_islower(c);
00462 }
00463
00470 bool asm_isbdigit(const char c)
00471 {
00472     if (c == '0' || c == '1') {
00473         return true;
00474     } else {
00475         return false;
00476     }
00477 }
00478
00485 bool asm_iscntrl(char c)
00486 {
00487     if ((c >= 0 && c <= 31) || c == 127) {
00488         return true;
00489     } else {
00490         return false;
00491     }
00492 }
00493
00500 bool asm_isdigit(char c)
00501 {
00502     if (c >= '0' && c <= '9') {
00503         return true;
00504     } else {
00505         return false;
00506     }
00507 }
00508
00515 bool asm_isgraph(char c)
00516 {
00517     if (c >= 33 && c <= 126) {
00518         return true;
00519     } else {
00520         return false;
00521     }
00522 }
00523
00530 bool asm_islower(char c)
00531 {
00532     if (c >= 'a' && c <= 'z') {
00533         return true;
00534     } else {
00535         return false;
00536     }
00537 }
00538
00545 bool asm_isodigit(const char c)
00546 {
00547     if ((c >= '0' && c <= '7')) {
00548         return true;
00549     } else {
00550         return false;
00551     }
00552 }
00553
00561 bool asm_isprint(char c)
00562 {
00563     return asm_isgraph(c) || c == ' ';
00564 }
00565
00573 bool asm_ispunct(char c)
00574 {
00575     if ((c >= 33 && c <= 47) || (c >= 58 && c <= 64) || (c >= 91 && c <= 96) || (c >= 123 && c <=
       126)) {
00576         return true;
00577     } else {
00578         return false;
00579     }
00580 }
00581
00589 bool asm_isspace(char c)
00590 {
00591     if (c == ' ' || c == '\n' || c == '\t' ||
00592         c == '\v'|| c == '\f' || c == '\r') {
00593         return true;
00594     } else {
00595         return false;
00596     }
00597 }
00598
00605 bool asm_isupper(char c)
00606 {
00607     if (c >= 'A' && c <= 'Z') {
```

```
00608         return true;
00609     } else {
00610         return false;
00611     }
00612 }
00613
00620 bool asm_isxdigit(char c)
00621 {
00622     if ((c >= 'a' && c <= 'f') || asm_isdigit(c)) {
00623         return true;
00624     } else {
00625         return false;
00626     }
00627 }
00628
00635 bool asm_isXdigit(char c)
00636 {
00637     if ((c >= 'A' && c <= 'F') || asm_isdigit(c)) {
00638         return true;
00639     } else {
00640         return false;
00641     }
00642 }
00643
00654 size_t  __asm_length(const char * const str, char *file_name, int line_num, char *function_name)
00655 {
00656     char c;
00657     size_t i = 0;
00658
00659     while ((c = str[i++]) != '\0') {
00660         if (i > ASM_MAX_LEN) {
00661             #ifndef ASM_NO_ERRORS
00662             asm_dprintERROR("index exceeds ASM_MAX_LEN. Probably no NULL termination.\nCalled in
      function: '%s' in: %s:%d", function_name, file_name, line_num);
00663             #endif
00664             return SIZE_MAX;
00665         }
00666     }
00667     return --i;
00668 }
00669
00689 void * asm_memset(void * const des, const unsigned char value, const size_t n)
00690 {
00691     unsigned char *ptr = (unsigned char *)des;
00692     for (size_t i = n; i-- > 0;) {
00693         *ptr++ = value;
00694     }
00695     return des;
00696 }
00697
00712 void asm_pad_left(char * const s, const size_t padding, const char pad)
00713 {
00714     int len = (int)asm_length(s);
00715     for (int i = len; i >= 0; i--) {
00716         s[i+(int)padding] = s[i];
00717     }
00718     for (int i = 0; i < (int)padding; i++) {
00719         s[i] = pad;
00720     }
00721 }
00722
00729 void asm_print_many_times(const char * const str, const size_t n)
00730 {
00731     for (size_t i = 0; i < n; i++) {
00732         printf("%s", str);
00733     }
00734     printf("\n");
00735 }
00736
00749 void asm_remove_char_from_string(char * const s, const size_t index)
00750 {
00751     size_t len = asm_length(s);
00752     if (len == 0) return;
00753     if (index >= len) {
00754         #ifndef ASM_NO_ERRORS
00755         asm_dprintERROR("%s", "index exceeds array length.");
00756         #endif
00757         return;
00758     }
00759
00760     for (size_t i = index; i < len; i++) {
00761         s[i] = s[i+1];
00762     }
00763 }
00764
00778 void asm_shift_left(char * const s, const size_t shift)
00779 {
```

```
00780     size_t len = asm_length(s);
00781
00782     if (shift == 0) return;
00783     if (len <= shift) {
00784         s[0] = '\0';
00785         return;
00786     }
00787
00788     size_t i;
00789     for (i = shift; i < len; i++) {
00790         s[i-shift] = s[i];
00791     }
00792     s[i-shift] = '\0';
00793 }
00794
00822 int asm_str_in_str(const char * const src, const size_t src_len, const char * const word_to_search,
       const char **first_occurrence)
00823 {
00824     size_t word_to_search_len = asm_length(word_to_search);
00825     if (word_to_search_len == 0) {
00826         if (first_occurrence) *first_occurrence = (const char *)src;
00827         return 0;
00828     }
00829     size_t num_of_accur = 0;
00830     size_t n = src_len == 0 ? ASM_MAX_LEN : src_len;
00831     for (size_t i = 0; src[i] != '\0' && i < n - word_to_search_len; i++) {
00832         if (asm_strncmp(src+i, word_to_search, asm_length(word_to_search))) {
00833             num_of_accur++;
00834             if (num_of_accur == 1) {
00835                 if (first_occurrence) *first_occurrence = &(src[i]);
00836             }
00837         }
00838     }
00839     return (int)num_of_accur;
00840 }
00841
00880 int asm_str_in_str_case_insensitive(const char * const src, const size_t src_len, const char * const
       word_to_search, const char **first_occurrence)
00881 {
00882     size_t word_to_search_len = asm_length(word_to_search);
00883     if (word_to_search_len == 0) {
00884         if (first_occurrence) *first_occurrence = (const char *)src;
00885         return 0;
00886     }
00887     size_t num_of_accur = 0;
00888     size_t n = src_len == 0 ? ASM_MAX_LEN : src_len;
00889     for (size_t i = 0; src[i] != '\0' && i < n - word_to_search_len; i++) {
00890         if (asm_strncmp_case_insensitive(src+i, word_to_search, asm_length(word_to_search))) {
00891             num_of_accur++;
00892             if (num_of_accur == 1) {
00893                 if (first_occurrence) *first_occurrence = &(src[i]);
00894             }
00895         }
00896     }
00897     return (int)num_of_accur;
00898 }
00899
00934 double asm_str2double(const char * const s, const char ** const end, const size_t base)
00935 {
00936     if (base < 2 || base > 36) {
00937         #ifndef ASM_NO_ERRORS
00938         asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
00939         #endif
00940         if (end) *end = s;
00941         return 0.0;
00942     }
00943     int num_of_whitespace = 0;
00944     while (asm_isspace(s[num_of_whitespace])) {
00945         num_of_whitespace++;
00946     }
00947
00948     int i = 0;
00949     if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
00950         i++;
00951     }
00952     int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;
00953
00954     size_t left = 0;
00955     double right = 0.0;
00956     int expo = 0;
00957     for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
00958         left = base * left + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
00959     }
00960
00961     if (s[i+num_of_whitespace] == '.') {
00962         i++; /* skip the point */
00963
```

```
00964         size_t divider = base;
00965         for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
00966             right = right + asm_get_char_value_in_base(s[i+num_of_whitespace], base) /
      (double)divider;
00967             divider *= base;
00968         }
00969     }
00970
00971     if ((s[i+num_of_whitespace] == 'e') || (s[i+num_of_whitespace] == 'E')) {
00972         expo = asm_str2int(&(s[i+num_of_whitespace+1]), end, 10);
00973     } else {
00974         if (end) *end = s + i + num_of_whitespace;
00975     }
00976
00977     double res = sign * (left + right);
00978
00979     if (expo > 0) {
00980         for (int index = 0; index < expo; index++) {
00981             res *= (double)base;
00982         }
00983     } else {
00984         for (int index = 0; index > expo; index--) {
00985             res /= (double)base;
00986         }
00987     }
00988
00989     return res;
00990 }
00991
01024 float asm_str2float(const char * const s, const char ** const end, const size_t base)
01025 {
01026     if (base < 2 || base > 36) {
01027         #ifndef ASM_NO_ERRORS
01028         asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
01029         #endif
01030         if (end) *end = s;
01031         return 0.0f;
01032     }
01033     int num_of_whitespace = 0;
01034     while (asm_isspace(s[num_of_whitespace])) {
01035         num_of_whitespace++;
01036     }
01037
01038     int i = 0;
01039     if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
01040         i++;
01041     }
01042     int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;
01043
01044     int left = 0;
01045     float right = 0.0f;
01046     int expo = 0;
01047     for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
01048         left = (int)base * left + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
01049     }
01050
01051     if (s[i+num_of_whitespace] == '.') {
01052         i++; /* skip the point */
01053
01054         size_t divider = base;
01055         for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
01056             right = right + asm_get_char_value_in_base(s[i+num_of_whitespace], base) / (float)divider;
01057             divider *= base;
01058         }
01059     }
01060
01061     if ((s[i+num_of_whitespace] == 'e') || (s[i+num_of_whitespace] == 'E')) {
01062         expo = asm_str2int(&(s[i+num_of_whitespace+1]), end, 10);
01063     } else {
01064         if (end) *end = s + i + num_of_whitespace;
01065     }
01066
01067     float res = sign * (left + right);
01068
01069     if (expo > 0) {
01070         for (int index = 0; index < expo; index++) {
01071             res *= (float)base;
01072         }
01073     } else {
01074         for (int index = 0; index > expo; index--) {
01075             res /= (float)base;
01076         }
01077     }
01078
01079     return res;
01080 }
01081
```

```
01098 int asm_str2int(const char * const s, const char ** const end, const size_t base)
01099 {
01100     if (base < 2 || base > 36) {
01101         #ifndef ASM_NO_ERRORS
01102         asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
01103         #endif
01104         if (end) *end = s;
01105         return 0;
01106     }
01107     int num_of_whitespace = 0;
01108     while (asm_isspace(s[num_of_whitespace])) {
01109         num_of_whitespace++;
01110     }
01111
01112     int n = 0, i = 0;
01113     if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
01114         i++;
01115     }
01116     int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;
01117
01118     for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
01119         n = (int)base * n + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
01120     }
01121
01122     if (end) *end = s + i+num_of_whitespace;
01123
01124     return n * sign;
01125 }
01126
01143 size_t asm_str2size_t(const char * const s, const char ** const end, const size_t base)
01144 {
01145     if (end) *end = s;
01146
01147     int num_of_whitespace = 0;
01148     while (asm_isspace(s[num_of_whitespace])) {
01149         num_of_whitespace++;
01150     }
01151
01152     if (s[0+num_of_whitespace] == '-') {
01153         #ifndef ASM_NO_ERRORS
01154         asm_dprintERROR("%s", "Unable to convert a negative number to size_t.");
01155         #endif
01156         return 0;
01157     }
01158
01159     if (base < 2 || base > 36) {
01160         #ifndef ASM_NO_ERRORS
01161         asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
01162         #endif
01163         if (end) *end = s+num_of_whitespace;
01164         return 0;
01165     }
01166
01167     size_t n = 0, i = 0;
01168     if (s[0+num_of_whitespace] == '+') {
01169         i++;
01170     }
01171
01172     for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
01173         n = base * n + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
01174     }
01175
01176     if (end) *end = s + i+num_of_whitespace;
01177
01178     return n;
01179 }
01180
01190 void asm_strip_whitespace(char * const s)
01191 {
01192     size_t len = asm_length(s);
01193     size_t i;
01194     for (i = 0; i < len; i++) {
01195         if (asm_isspace(s[i])) {
01196             asm_remove_char_from_string(s, i);
01197             len--;
01198             i--;
01199         }
01200     }
01201     s[i] = '\0';
01202 }
01203
01211 bool asm_str_is_whitespace(const char * const s)
01212 {
01213     size_t len = asm_length(s);
01214     for (size_t i = 0; i < len; i++) {
01215         if (!asm_isspace(s[i])) {
01216             return false;
```

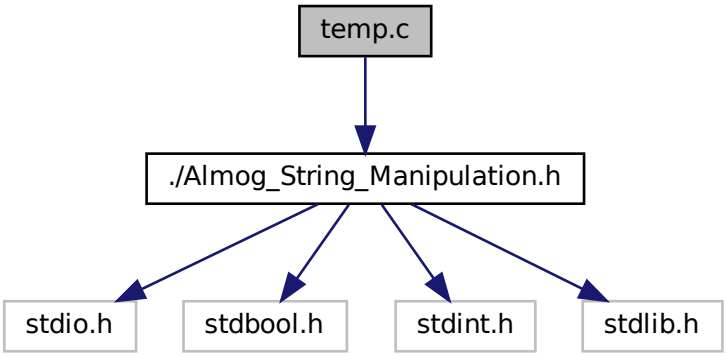
```
01217         }
01218     }
01219
01220     return true;
01221 }
01222
01241 char * asm_strdup(const char * const s, size_t length)
01242 {
01243     char * res = (char *)ASM_MALLOC(sizeof(char) * length+1);
01244     asm_strncpy((char * const)res, s, length);
01245
01246     return res;
01247 }
01248
01270 int asm_strncat(char * const s1, const char * const s2, const size_t N)
01271 {
01272     size_t len_s1 = asm_length(s1);
01273
01274     size_t limit = N;
01275     if (limit == 0) {
01276         limit = ASM_MAX_LEN;
01277     }
01278
01279     size_t i = 0;
01280     while (i < limit && s2[i] != '\0') {
01281         if (len_s1 + (size_t)i >= ASM_MAX_LEN-1) {
01282             #ifndef ASM_NO_ERRORS
01283             asm_dprintERROR("s2 or the first N=%zu digit of s2 does not fit into s1.", N);
01284             #endif
01285             return (int)i;
01286         }
01287
01288         s1[len_s1+i] = s2[i];
01289         i++;
01290     }
01291     s1[len_s1+i] = '\0';
01292
01293     return (int)i;
01294 }
01295
01315 int asm_strncmp(const char *s1, const char *s2, const size_t N)
01316 {
01317     size_t n = N == 0 ? ASM_MAX_LEN : N;
01318     size_t i = 0;
01319     while (i < n) {
01320         if (s1[i] == '\0' && s2[i] == '\0') {
01321             break;
01322         }
01323         if (s1[i] != s2[i] || (s1[i] == '\0') || (s2[i] == '\0')) {
01324             return 0;
01325         }
01326         i++;
01327     }
01328     return 1;
01329 }
01330
01354 int asm_strncmp_case_insensitive(const char * const s1, const char * const s2, const size_t N)
01355 {
01356     size_t n = N == 0 ? ASM_MAX_LEN : N;
01357     size_t i = 0;
01358
01359     char *s1dup = asm_strdup(s1, n);
01360     char *s2dup = asm_strdup(s2, n);
01361
01362     asm_tolower(s1dup, N);
01363     asm_tolower(s2dup, N);
01364
01365     while (i < n) {
01366         if (s1dup[i] == '\0' && s2dup[i] == '\0') {
01367             break;
01368         }
01369         if (s1dup[i] != s2dup[i] || (s1dup[i] == '\0') || (s2dup[i] == '\0')) {
01370             free(s1dup);
01371             free(s2dup);
01372             return 0;
01373         }
01374         i++;
01375     }
01376
01377
01378     ASM_FREE(s1dup);
01379     ASM_FREE(s2dup);
01380
01381     return 1;
01382 }
01383
01401 int asm_strncpy(char * const s1, const char * const s2, const size_t N)
```

```
01402 {
01403     size_t n = N == 0 ? ASM_MAX_LEN : N;
01404 
01405     size_t i;
01406     for (i = 0; i < n && s2[i] != '\0'; i++) {
01407         s1[i] = s2[i];
01408     }
01409     s1[i] = '\0';
01410 
01411     return (int)i;
01412 }
01413 
01428 void asm_tolower(char * const s, const size_t len)
01429 {
01430     for (size_t i = 0; i < len && s[i] != '\0'; i++) {
01431         if (asm_isupper(s[i])) {
01432             s[i] += 'a' - 'A';
01433         }
01434     }
01435 }
01436 
01451 void asm_toupper(char * const s, const size_t len)
01452 {
01453     for (size_t i = 0; i < len && s[i] != '\0'; i++) {
01454         if (asm_islower(s[i])) {
01455             s[i] += 'A' - 'a';
01456         }
01457     }
01458 }
01459 
01468 void asm_trim_left_whitespace(char * const s)
01469 {
01470     size_t len = asm_length(s);
01471 
01472     if (len == 0) return;
01473     size_t i;
01474     for (i = 0; i < len; i++) {
01475         if (!asm_isspace(s[i])) {
01476             break;
01477         }
01478     }
01479     asm_shift_left(s, i);
01480 }
01481 
01482 #ifdef ASM_NO_ERRORS
01483 #undef ASM_NO_ERRORS
01484 #endif
01485 
01486 #endif /*ALMOG_STRING_MANIPULATION_IMPLEMENTATION*/
01487 
```

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().

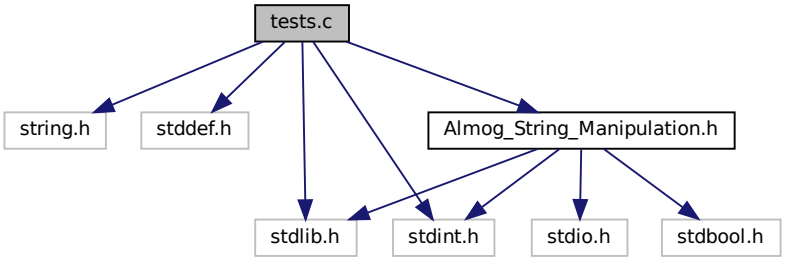
2.4 temp.c

```
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         }                                                               \
```

```
00036     } while (0)
00037
00038 #define TEST_EQ_INT(a, b) TEST_CASE((a) == (b))
00039 #define TEST_EQ_SIZE(a, b) TEST_CASE((a) == (b))
00040 #define TEST_EQ_STR(a, b) TEST_CASE(strcmp((a), (b)) == 0)
00041 #define TEST_NE_STR(a, b) TEST_CASE(strcmp((a), (b)) != 0)
00042
00043 static void fill_sentinel(unsigned char *buf, size_t n, unsigned char v)
00044 {
00045     for (size_t i = 0; i < n; i++) buf[i] = v;
00046 }
00047
00048 static bool is_nul_terminated_within(const char *s, size_t cap)
00049 {
00050     for (size_t i = 0; i < cap; i++) {
00051         if (s[i] == '\0') return true;
00052     }
00053     return false;
00054 }
00055
00056 /* Simple deterministic RNG for fuzz-ish tests */
00057 static uint32_t rng_state = 0xC0FFEE01u;
00058 static uint32_t xorshift32(void)
00059 {
00060     uint32_t x = rng_state;
00061     x ^= x << 13;
00062     x ^= x >> 17;
00063     x ^= x << 5;
00064     rng_state = x;
00065     return x;
00066 }
00067
00068 static char rand_ascii_printable(void)
00069 {
00070     /* printable ASCII range 32..126 */
00071     return (char)(32 + (xorshift32() % 95));
00072 }
00073
00074 /* ---------------- Coverage checks ----------------
00075  * We can't reliably "assert all symbols exist" at runtime, but we can at least
00076  * ensure we have tests for every IMPLEMENTED function by calling it at least
00077  * once in this file.
00078  */
00079
00080 /* ---------------- Tests: ASCII classification ---------------- */
00081
00082 static void test_ascii_classification_exhaustive_ranges(void)
00083 {
00084     /* Check key boundaries and a few midpoints for each function. */
00085     TEST_CASE(asm_isdigit('0'));
00086     TEST_CASE(asm_isdigit('9'));
00087     TEST_CASE(!asm_isdigit('/'));
00088     TEST_CASE(!asm_isdigit(':'));
00089
00090     TEST_CASE(asm_isupper('A'));
00091     TEST_CASE(asm_isupper('Z'));
00092     TEST_CASE(!asm_isupper('@'));
00093     TEST_CASE(!asm_isupper('['));
00094
00095     TEST_CASE(asm_islower('a'));
00096     TEST_CASE(asm_islower('z'));
00097     TEST_CASE(!asm_islower('`'));
00098     TEST_CASE(!asm_islower('{'));
00099
00100     TEST_CASE(asm_isalpha('A'));
00101     TEST_CASE(asm_isalpha('z'));
00102     TEST_CASE(!asm_isalpha('0'));
00103
00104     TEST_CASE(asm_isalnum('A'));
00105     TEST_CASE(asm_isalnum('9'));
00106     TEST_CASE(!asm_isalnum('_'));
00107     TEST_CASE(!asm_isalnum(' '));
00108
00109     TEST_CASE(asm_isspace(' '));
00110     TEST_CASE(asm_isspace('\n'));
00111     TEST_CASE(asm_isspace('\t'));
00112     TEST_CASE(asm_isspace('\r'));
00113     TEST_CASE(asm_isspace('\v'));
00114     TEST_CASE(asm_isspace('\f'));
00115     TEST_CASE(!asm_isspace('X'));
00116
00117     TEST_CASE(asm_isgraph('!'));
00118     TEST_CASE(asm_isgraph('~'));
00119     TEST_CASE(!asm_isgraph(' '));
00120
00121     TEST_CASE(asm_isprint(' '));
00122     TEST_CASE(asm_isprint('!'));
```

```
00123     TEST_CASE(!asm_isprint('\n'));
00124
00125     TEST_CASE(asm_ispunct('!'));
00126     TEST_CASE(asm_ispunct('/'));
00127     TEST_CASE(asm_ispunct(':'));
00128     TEST_CASE(!asm_ispunct('A'));
00129     TEST_CASE(!asm_ispunct('0'));
00130     TEST_CASE(!asm_ispunct(' '));
00131
00132     TEST_CASE(asm_iscntrl('\0'));
00133     TEST_CASE(asm_iscntrl('\n'));
00134     TEST_CASE(asm_iscntrl(127));
00135     TEST_CASE(!asm_iscntrl('A'));
00136
00137     /* Hex digit helpers (your impl splits by case) */
00138     TEST_CASE(asm_isxdigit('0'));
00139     TEST_CASE(asm_isxdigit('9'));
00140     TEST_CASE(asm_isxdigit('a'));
00141     TEST_CASE(asm_isxdigit('f'));
00142     TEST_CASE(!asm_isxdigit('g'));
00143     TEST_CASE(!asm_isxdigit('A'));
00144
00145     TEST_CASE(asm_isXdigit('0'));
00146     TEST_CASE(asm_isXdigit('9'));
00147     TEST_CASE(asm_isXdigit('A'));
00148     TEST_CASE(asm_isXdigit('F'));
00149     TEST_CASE(!asm_isXdigit('G'));
00150     TEST_CASE(!asm_isXdigit('a'));
00151 }
00152
00153 static void test_ascii_classification_full_scan_0_127(void)
00154 {
00155     /* Property checks over ASCII 0..127. */
00156     for (int c = 0; c <= 127; c++) {
00157         char ch = (char)c;
00158
00159         /* isalnum == isalpha || isdigit */
00160         TEST_CASE(asm_isalnum(ch) == (asm_isalpha(ch) || asm_isdigit(ch)));
00161
00162         /* isprint == isgraph || ' ' */
00163         TEST_CASE(asm_isprint(ch) == (asm_isgraph(ch) || ch == ' '));
00164
00165         /* isalpha implies not digit */
00166         if (asm_isalpha(ch)) {
00167             TEST_CASE(!asm_isdigit(ch));
00168         }
00169
00170         /* upper and lower are disjoint */
00171         if (asm_isupper(ch)) TEST_CASE(!asm_islower(ch));
00172         if (asm_islower(ch)) TEST_CASE(!asm_isupper(ch));
00173
00174         /* graph implies print */
00175         if (asm_isgraph(ch)) TEST_CASE(asm_isprint(ch));
00176     }
00177 }
00178
00179 /* ---------------- Tests: case conversion ---------------- */
00180
00181 static void test_case_conversion_roundtrip(void)
00182 {
00183     for (int i = 0; i < 200; i++) {
00184         char s[128];
00185         char a[128];
00186         char b[128];
00187
00188         /* random printable string length 0..40 */
00189         size_t n = (size_t)(xorshift32() % 41);
00190         for (size_t j = 0; j < n; j++) s[j] = rand_ascii_printable();
00191         s[n] = '\0';
00192
00193         strcpy(a, s);
00194         strcpy(b, s);
00195
00196         asm_tolower(a);
00197         asm_toupper(a);
00198         asm_toupper(b);
00199         asm_tolower(b);
00200
00201         /* Not equal generally, but must still be valid strings and stable */
00202         TEST_CASE(is_nul_terminated_within(a, sizeof(a)));
00203         TEST_CASE(is_nul_terminated_within(b, sizeof(b)));
00204
00205         /* toupper(toupper(x)) == toupper(x) */
00206         char u1[128], u2[128];
00207         strcpy(u1, s);
00208         strcpy(u2, s);
00209         asm_toupper(u1);
```

```
00210         asm_toupper(u2);
00211         asm_toupper(u2);
00212         TEST_EQ_STR(u1, u2);
00213
00214         /* tolower(tolower(x)) == tolower(x) */
00215         char l1[128], l2[128];
00216         strcpy(l1, s);
00217         strcpy(l2, s);
00218         asm_tolower(l1);
00219         asm_tolower(l2);
00220         asm_tolower(l2);
00221         TEST_EQ_STR(l1, l2);
00222     }
00223 }
00224
00225 /* ---------------- Tests: asm_length ---------------- */
00226
00227 static void test_length_matches_strlen_small(void)
00228 {
00229     for (int i = 0; i < 200; i++) {
00230         char s[256];
00231         size_t n = (size_t)(xorshift32() % 200);
00232         for (size_t j = 0; j < n; j++) s[j] = rand_ascii_printable();
00233         s[n] = '\0';
00234
00235         TEST_EQ_SIZE(asm_length(s), strlen(s));
00236     }
00237 }
00238
00239 /* ---------------- Tests: asm_memset ---------------- */
00240
00241 static void test_memset_basic_and_edges(void)
00242 {
00243     unsigned char buf[32];
00244     fill_sentinel(buf, sizeof(buf), 0xCC);
00245
00246     void *ret = asm_memset(buf, 0xAB, sizeof(buf));
00247     TEST_CASE(ret == buf);
00248     for (size_t i = 0; i < sizeof(buf); i++) TEST_CASE(buf[i] == 0xAB);
00249
00250     fill_sentinel(buf, sizeof(buf), 0xCC);
00251     asm_memset(buf, 0xAB, 0);
00252     for (size_t i = 0; i < sizeof(buf); i++) TEST_CASE(buf[i] == 0xCC);
00253 }
00254
00255 /* ---------------- Tests: asm_copy_array_by_indexes ---------------- */
00256
00257 static void test_copy_array_by_indexes_behavior_and_bounds(void)
00258 {
00259     const char *src = "abcdef";
00260     char out[16];
00261
00262     asm_copy_array_by_indexes(out, 1, 3, src); /* inclusive end in impl */
00263     TEST_EQ_STR(out, "bcd");
00264
00265     asm_copy_array_by_indexes(out, 0, 0, src);
00266     TEST_EQ_STR(out, "a");
00267
00268     asm_copy_array_by_indexes(out, 5, 5, src);
00269     TEST_EQ_STR(out, "f");
00270
00271     asm_copy_array_by_indexes(out, 0, 6, src); /* copies '\0' too */
00272     TEST_EQ_STR(out, "abcdef");
00273
00274     /* Sentinel around output buffer to detect overwrite beyond out[16] */
00275     struct {
00276         unsigned char pre[8];
00277         char out2[8];
00278         unsigned char post[8];
00279     } box;
00280
00281     fill_sentinel(box.pre, sizeof(box.pre), 0xA5);
00282     fill_sentinel((unsigned char *)box.out2, sizeof(box.out2), 0xCC);
00283     fill_sentinel(box.post, sizeof(box.post), 0x5A);
00284
00285     /* copy "ab" plus '\0' => should fit exactly */
00286     asm_copy_array_by_indexes(box.out2, 0, 1, "ab");
00287     TEST_EQ_STR(box.out2, "ab");
00288     for (size_t i = 0; i < sizeof(box.pre); i++) TEST_CASE(box.pre[i] == 0xA5);
00289     for (size_t i = 0; i < sizeof(box.post); i++) TEST_CASE(box.post[i] == 0x5A);
00290 }
00291
00292 /* ---------------- Tests: shifting/padding ---------------- */
00293
00294 static void test_left_shift_edges(void)
00295 {
00296     char s[64];
```

```
00297
00298     strcpy(s, "abcdef");
00299     asm_shift_left(s, 0);
00300     TEST_EQ_STR(s, "abcdef");
00301
00302     strcpy(s, "abcdef");
00303     asm_shift_left(s, 1);
00304     TEST_EQ_STR(s, "bcdef");
00305
00306     strcpy(s, "abcdef");
00307     asm_shift_left(s, 5);
00308     TEST_EQ_STR(s, "f");
00309
00310     strcpy(s, "abcdef");
00311     asm_shift_left(s, 6);
00312     TEST_EQ_STR(s, "");
00313
00314     strcpy(s, "abcdef");
00315     asm_shift_left(s, 1000);
00316     TEST_EQ_STR(s, "");
00317 }
00318
00319 static void test_left_pad_edges_and_sentinel(void)
00320 {
00321     {
00322         char s[64] = "abc";
00323         asm_pad_left(s, 0, ' ');
00324         TEST_EQ_STR(s, "abc");
00325     }
00326     {
00327         char s[64] = "abc";
00328         asm_pad_left(s, 4, ' ');
00329         TEST_EQ_STR(s, "    abc");
00330     }
00331     {
00332         char s[64] = "";
00333         asm_pad_left(s, 3, '_');
00334         TEST_EQ_STR(s, "___");
00335     }
00336
00337     /* Sentinel structure: ensure we don't write before start */
00338     struct {
00339         unsigned char pre[8];
00340         char s[32];
00341         unsigned char post[8];
00342     } box;
00343
00344     fill_sentinel(box.pre, sizeof(box.pre), 0x11);
00345     fill_sentinel((unsigned char *)box.s, sizeof(box.s), 0xCC);
00346     fill_sentinel(box.post, sizeof(box.post), 0x22);
00347
00348     strcpy(box.s, "x");
00349     asm_pad_left(box.s, 5, '0');
00350     TEST_EQ_STR(box.s, "00000x");
00351
00352     for (size_t i = 0; i < sizeof(box.pre); i++) TEST_CASE(box.pre[i] == 0x11);
00353     for (size_t i = 0; i < sizeof(box.post); i++) TEST_CASE(box.post[i] == 0x22);
00354 }
00355
00356 /* ---------------- Tests: remove/strip/whitespace ---------------- */
00357
00358 static void test_remove_char_form_string_edges(void)
00359 {
00360     char s[64];
00361
00362     strcpy(s, "abcd");
00363     asm_remove_char_from_string(s, 1);
00364     TEST_EQ_STR(s, "acd");
00365
00366     strcpy(s, "abcd");
00367     asm_remove_char_from_string(s, 0);
00368     TEST_EQ_STR(s, "bcd");
00369
00370     strcpy(s, "abcd");
00371     asm_remove_char_from_string(s, 3);
00372     TEST_EQ_STR(s, "abc");
00373
00374     strcpy(s, "a");
00375     asm_remove_char_from_string(s, 0);
00376     TEST_EQ_STR(s, "");
00377
00378     strcpy(s, "");
00379     asm_remove_char_from_string(s, 0);
00380     TEST_EQ_STR(s, "");
00381
00382     strcpy(s, "abcd");
00383     asm_remove_char_from_string(s, 999);
```

```
00384     TEST_EQ_STR(s, "abcd");
00385 }
00386
00387 static void test_strip_whitespace_properties(void)
00388 {
00389     char s[128];
00390
00391     strcpy(s, "  a \t b\nc  ");
00392     asm_strip_whitespace(s);
00393     TEST_EQ_STR(s, "abc");
00394
00395     strcpy(s, "no_spaces");
00396     asm_strip_whitespace(s);
00397     TEST_EQ_STR(s, "no_spaces");
00398
00399     strcpy(s, " \t\r\n");
00400     asm_strip_whitespace(s);
00401     TEST_EQ_STR(s, "");
00402
00403     /* Property: result has no whitespace chars */
00404     for (int i = 0; i < 100; i++) {
00405         size_t n = (size_t)(xorshift32() % 60);
00406         for (size_t j = 0; j < n; j++) {
00407             /* mix whitespace and printable */
00408             uint32_t r = xorshift32() % 10;
00409             if (r == 0) s[j] = ' ';
00410             else if (r == 1) s[j] = '\n';
00411             else if (r == 2) s[j] = '\t';
00412             else s[j] = rand_ascii_printable();
00413         }
00414         s[n] = '\0';
00415
00416         asm_strip_whitespace(s);
00417         for (size_t k = 0; s[k] != '\0'; k++) {
00418             TEST_CASE(!asm_isspace(s[k]));
00419         }
00420     }
00421 }
00422
00423 static void test_str_is_whitespace_edges(void)
00424 {
00425     TEST_CASE(asm_str_is_whitespace(" \t\r\n") == true);
00426     TEST_CASE(asm_str_is_whitespace("") == true); /* current behavior */
00427     TEST_CASE(asm_str_is_whitespace(" x ") == false);
00428 }
00429
00430 /* ---------------- Tests: asm_strncmp (boolean) ---------------- */
00431
00432 static void test_strncmp_boolean_edges(void)
00433 {
00434     TEST_CASE(asm_strncmp("abc", "abc", 3) == 1);
00435     TEST_CASE(asm_strncmp("abc", "abd", 3) == 0);
00436     TEST_CASE(asm_strncmp("ab", "abc", 3) == 0);
00437     TEST_CASE(asm_strncmp("abc", "ab", 3) == 0);
00438
00439     TEST_CASE(asm_strncmp("abc", "XYZ", 0) == 1);
00440
00441     TEST_CASE(asm_strncmp("", "", 5) == 1);
00442     TEST_CASE(asm_strncmp("", "a", 1) == 0);
00443     TEST_CASE(asm_strncmp("a", "", 1) == 0);
00444 }
00445
00446 /* ---------------- Tests: asm_str_in_str ---------------- */
00447
00448 static void test_str_in_str_overlap_and_edges(void)
00449 {
00450     TEST_EQ_INT(asm_str_in_str("aaaa", "aa"), 3);
00451     TEST_EQ_INT(asm_str_in_str("hello world", "lo"), 1);
00452     TEST_EQ_INT(asm_str_in_str("abc", "abcd"), 0);
00453     TEST_EQ_INT(asm_str_in_str("abababa", "aba"), 3);
00454
00455     /* Do not pass empty needle: undefined-ish for your implementation. */
00456 }
00457
00458 /* ---------------- Tests: base digit helpers ---------------- */
00459
00460 static void test_base_digit_helpers(void)
00461 {
00462     TEST_CASE(asm_check_char_belong_to_base('0', 2) == true);
00463     TEST_CASE(asm_check_char_belong_to_base('1', 2) == true);
00464     TEST_CASE(asm_check_char_belong_to_base('2', 2) == false);
00465
00466     TEST_CASE(asm_check_char_belong_to_base('9', 10) == true);
00467     TEST_CASE(asm_check_char_belong_to_base('a', 10) == false);
00468
00469     TEST_CASE(asm_check_char_belong_to_base('a', 16) == true);
00470     TEST_CASE(asm_check_char_belong_to_base('f', 16) == true);
```

```
00471     TEST_CASE(asm_check_char_belong_to_base('g', 16) == false);
00472     TEST_CASE(asm_check_char_belong_to_base('A', 16) == true);
00473     TEST_CASE(asm_check_char_belong_to_base('F', 16) == true);
00474     TEST_CASE(asm_check_char_belong_to_base('G', 16) == false);
00475
00476     TEST_CASE(asm_check_char_belong_to_base('z', 36) == true);
00477     TEST_CASE(asm_check_char_belong_to_base('Z', 36) == true);
00478
00479     TEST_EQ_INT(asm_get_char_value_in_base('0', 10), 0);
00480     TEST_EQ_INT(asm_get_char_value_in_base('9', 10), 9);
00481     TEST_EQ_INT(asm_get_char_value_in_base('A', 16), 10);
00482     TEST_EQ_INT(asm_get_char_value_in_base('f', 16), 15);
00483     TEST_EQ_INT(asm_get_char_value_in_base('Z', 36), 35);
00484
00485     TEST_EQ_INT(asm_get_char_value_in_base('g', 16), -1);
00486
00487     /* base validity errors should return false / -1 */
00488     TEST_CASE(asm_check_char_belong_to_base('0', 1) == false);
00489     TEST_CASE(asm_check_char_belong_to_base('0', 37) == false);
00490     TEST_EQ_INT(asm_get_char_value_in_base('0', 1), -1);
00491 }
00492
00493 /* ---------------- Tests: str2int/size_t/float/double ---------------- */
00494
00495 static void test_str2int(void)
00496 {
00497     const char *end = NULL;
00498
00499     {
00500         char s[] = "  -1011zzz";
00501         int v = asm_str2int(s, &end, 2);
00502         TEST_CASE(v == -11);
00503         TEST_CASE(*end == 'z');
00504     }
00505     {
00506         char s[] = "+7fff!";
00507         int v = asm_str2int(s, &end, 16);
00508         TEST_CASE(v == 0x7fff);
00509         TEST_CASE(*end == '!');
00510     }
00511     {
00512         char s[] = "   +0";
00513         int v = asm_str2int(s, &end, 10);
00514         TEST_CASE(v == 0);
00515         TEST_CASE(*end == '\0');
00516     }
00517     {
00518         char s[] = "xyz";
00519         int v = asm_str2int(s, &end, 10);
00520         TEST_CASE(v == 0);
00521         TEST_CASE(*end == 'x');
00522     }
00523     {
00524         char s[] = "123";
00525         int v = asm_str2int(s, &end, 1);
00526         TEST_CASE(v == 0);
00527         TEST_CASE(end == s);
00528     }
00529 }
00530
00531 static void test_str2size_t(void)
00532 {
00533     const char *end = NULL;
00534
00535     {
00536         char s[] = " +1f!";
00537         size_t v = asm_str2size_t(s, &end, 16);
00538         TEST_CASE(v == 31u);
00539         TEST_CASE(*end == '!');
00540     }
00541     {
00542         char s[] = "  -1";
00543         size_t v = asm_str2size_t(s, &end, 10);
00544         TEST_CASE(v == 0);
00545         TEST_CASE(end == s);
00546     }
00547     {
00548         char s[] = "  +0009x";
00549         size_t v = asm_str2size_t(s, &end, 10);
00550         TEST_CASE(v == 9u);
00551         TEST_CASE(*end == 'x');
00552     }
00553     {
00554         char s[] = "  123";
00555         size_t v = asm_str2size_t(s, &end, 37);
00556         TEST_CASE(v == 0);
00557         /* current implementation sets *end = s+num_of_whitespace on invalid base */
```

```
00558         TEST_CASE(end == s + 2);
00559     }
00560 }
00561
00562 static void test_str2float_double(void)
00563 {
00564     const char *end = NULL;
00565
00566     {
00567         char s[] = " 10.5x";
00568         float v = asm_str2float(s, &end, 10);
00569         TEST_CASE(v > 10.49f && v < 10.51f);
00570         TEST_CASE(*end == 'x');
00571     }
00572     {
00573         char s[] = "-a.bQ";
00574         double v = asm_str2double(s, &end, 16);
00575         TEST_CASE(v < -10.68 && v > -10.70);
00576         TEST_CASE(*end == 'Q');
00577     }
00578     {
00579         char s[] = "  123.";
00580         double v = asm_str2double(s, &end, 10);
00581         TEST_CASE(v > 122.99 && v < 123.01);
00582         TEST_CASE(*end == '\0');
00583     }
00584     {
00585         char s[] = "  .5";
00586         double v = asm_str2double(s, &end, 10);
00587         TEST_CASE(v > 0.49 && v < 0.51);
00588         TEST_CASE(*end == '\0');
00589     }
00590     {
00591         char s[] = "  -.";
00592         double v = asm_str2double(s, &end, 10);
00593         TEST_CASE(v == 0.0);
00594         TEST_CASE(*end == '\0');
00595     }
00596     {
00597         char s[] = "12.3";
00598         double v = asm_str2double(s, &end, 37);
00599         TEST_CASE(v == 0.0);
00600         TEST_CASE(end == s);
00601     }
00602 }
00603
00604 /* ---------------- Tests: tokenization helpers ---------------- */
00605
00606 static void test_get_next_word_from_line_current_behavior(void)
00607 {
00608     /* Your implementation:
00609      * - does NOT skip whitespace
00610      * - stops only on delimiter or '\0'
00611      * - returns length (j), not consumed index
00612      */
00613     {
00614         char src[] = "abc,def";
00615         char w[64] = {0};
00616         int r = asm_get_next_token_from_str(w, src, ',');
00617         TEST_EQ_INT(r, 3);
00618         TEST_EQ_STR(w, "abc");
00619     }
00620     {
00621         char src[] = ",def";
00622         char w[64] = {0};
00623         int r = asm_get_next_token_from_str(w, src, ',');
00624         TEST_EQ_INT(r, 0);
00625         TEST_EQ_STR(w, "");
00626     }
00627     {
00628         char src[] = "  abc,def";
00629         char w[64] = {0};
00630         int r = asm_get_next_token_from_str(w, src, ',');
00631         TEST_EQ_INT(r, 5);
00632         TEST_EQ_STR(w, "  abc");
00633     }
00634     {
00635         char src[] = "abc\ndef";
00636         char w[64] = {0};
00637         int r = asm_get_next_token_from_str(w, src, ',');
00638         TEST_EQ_INT(r, (int)strlen(src));
00639         TEST_EQ_STR(w, "abc\ndef");
00640     }
00641
00642     /* Doc mismatch detection (warn, not fail) */
00643     {
00644         char src[] = "  abc,def";
```

```
00645         char w[64] = {0};
00646         asm_get_next_token_from_str(w, src, ',');
00647         TEST_CASE(strcmp(w, "  abc") == 0);
00648     }
00649 }
00650
00651 static void test_get_word_and_cut_edges(void)
00652 {
00653     {
00654         char src[64] = "abc,def";
00655         char w[64] = {0};
00656         int ok = asm_get_token_and_cut(w, src, ',', true);
00657         TEST_CASE(ok == 1);
00658         TEST_EQ_STR(w, "abc");
00659         TEST_EQ_STR(src, ",def");
00660     }
00661     {
00662         char src[64] = "abc,def";
00663         char w[64] = {0};
00664         int ok = asm_get_token_and_cut(w, src, ',', false);
00665         TEST_CASE(ok == 1);
00666         TEST_EQ_STR(w, "abc");
00667         TEST_EQ_STR(src, "def");
00668     }
00669     {
00670         char src[64] = ",def";
00671         char w[64] = {0};
00672         int ok = asm_get_token_and_cut(w, src, ',', true);
00673         TEST_CASE(ok == 0);
00674         TEST_EQ_STR(w, "");
00675         TEST_EQ_STR(src, ",def");
00676     }
00677     {
00678         char src[64] = "nodelem";
00679         char w[64] = {0};
00680         int ok = asm_get_token_and_cut(w, src, ',', false);
00681         TEST_CASE(ok == 1);
00682         TEST_EQ_STR(w, "nodelem");
00683         TEST_EQ_STR(src, "");
00684     }
00685 }
00686
00687 /* ---------------- Tests: asm_get_line ---------------- */
00688
00689 static void test_get_line_tmpfile(void)
00690 {
00691     FILE *fp = tmpfile();
00692     if (!fp) {
00693         fprintf(stderr,
00694                 "[WARN] tmpfile() unavailable; skipping asm_get_line tests\n");
00695         g_tests_warned++;
00696         return;
00697     }
00698
00699     fputs("hello\n", fp);
00700     fputs("\n", fp);
00701     fputs("world", fp);
00702     rewind(fp);
00703
00704     {
00705         char line[ASM_MAX_LEN + 1];
00706         int n = asm_get_line(fp, line);
00707         TEST_EQ_INT(n, 5);
00708         TEST_EQ_STR(line, "hello");
00709         TEST_CASE(is_nul_terminated_within(line, sizeof(line)));
00710     }
00711     {
00712         char line[ASM_MAX_LEN + 1];
00713         int n = asm_get_line(fp, line);
00714         TEST_EQ_INT(n, 0);
00715         TEST_EQ_STR(line, "");
00716     }
00717     {
00718         char line[ASM_MAX_LEN + 1];
00719         int n = asm_get_line(fp, line);
00720         TEST_EQ_INT(n, 5);
00721         TEST_EQ_STR(line, "world");
00722     }
00723     {
00724         char line[ASM_MAX_LEN + 1];
00725         int n = asm_get_line(fp, line);
00726         TEST_EQ_INT(n, -1);
00727     }
00728
00729     fclose(fp);
00730 }
00731
```

```
00732 /* Optional: test overflow condition using ASM_MAX_LEN+1 chars before '\n' */
00733 static void test_get_line_too_long(void)
00734 {
00735     FILE *fp = tmpfile();
00736     if (!fp) {
00737         fprintf(stderr,
00738                 "[WARN] tmpfile() unavailable; skipping long-line test\n");
00739         g_tests_warned++;
00740         return;
00741     }
00742
00743     for (int i = 0; i < ASM_MAX_LEN + 5; i++) fputc('a', fp);
00744     fputc('\n', fp);
00745     rewind(fp);
00746
00747     char line[ASM_MAX_LEN + 1];
00748     fill_sentinel((unsigned char *)line, sizeof(line), 0xCC);
00749
00750     int n = asm_get_line(fp, line);
00751     TEST_EQ_INT(n, -1);
00752
00753     /* On error, your docs say not guaranteed NUL terminated. We only ensure
00754        we didn't write past buffer size (can't directly prove; but at least
00755        array exists). */
00756     fclose(fp);
00757 }
00758
00759 /* ---------------- Tests: asm_strncat ---------------- */
00760
00761 static void test_strncat_current_behavior_and_sentinel(void)
00762 {
00763     /* Current impl does NOT append '\0' (bug-like).
00764        We test both:
00765        - it copies correct bytes
00766        - it does not clobber past allowed region
00767      */
00768     struct {
00769         unsigned char pre[8];
00770         char s1[16];
00771         unsigned char post[8];
00772     } box;
00773
00774     fill_sentinel(box.pre, sizeof(box.pre), 0xAA);
00775     fill_sentinel((unsigned char *)box.s1, sizeof(box.s1), 0xCC);
00776     fill_sentinel(box.post, sizeof(box.post), 0xBB);
00777
00778     strcpy(box.s1, "abc");
00779
00780     int n = asm_strncat(box.s1, "DEF", 3);
00781     TEST_EQ_INT(n, 3);
00782
00783     TEST_EQ_STR(box.s1, "abcDEF");
00784
00785     for (size_t i = 0; i < sizeof(box.pre); i++) TEST_CASE(box.pre[i] == 0xAA);
00786     for (size_t i = 0; i < sizeof(box.post); i++) TEST_CASE(box.post[i] == 0xBB);
00787 }
00788
00789 /* ---------------- Tests: str2float/double with exponent notation ---------------- */
00790
00791 static void test_str2float_exponent_basic(void)
00792 {
00793     const char *end = NULL;
00794     float v;
00795
00796     /* Basic positive exponents */
00797     v = asm_str2float("1e2", &end, 10);
00798     TEST_CASE(v > 99.9f && v < 100.1f);
00799     TEST_CASE(*end == '\0');
00800
00801     v = asm_str2float("1.5e3", &end, 10);
00802     TEST_CASE(v > 1499.9f && v < 1500.1f);
00803     TEST_CASE(*end == '\0');
00804
00805     v = asm_str2float("5e2", &end, 10);
00806     TEST_CASE(v > 499.9f && v < 500.1f);
00807     TEST_CASE(*end == '\0');
00808
00809     /* Basic negative exponents */
00810     v = asm_str2float("1e-2", &end, 10);
00811     TEST_CASE(v > 0.0099f && v < 0.0101f);
00812     TEST_CASE(*end == '\0');
00813
00814     v = asm_str2float("5e-1", &end, 10);
00815     TEST_CASE(v > 0.49f && v < 0.51f);
00816     TEST_CASE(*end == '\0');
00817
00818     v = asm_str2float("2.5e-3", &end, 10);
```

```
00819     TEST_CASE(v > 0.00249f && v < 0.00251f);
00820     TEST_CASE(*end == '\0');
00821
00822     /* Exponent with explicit positive sign */
00823     v = asm_str2float("1e+2", &end, 10);
00824     TEST_CASE(v > 99.9f && v < 100.1f);
00825     TEST_CASE(*end == '\0');
00826
00827     v = asm_str2float("3.5e+1", &end, 10);
00828     TEST_CASE(v > 34.9f && v < 35.1f);
00829     TEST_CASE(*end == '\0');
00830 }
00831
00832 static void test_str2float_exponent_signed_mantissa(void)
00833 {
00834     const char *end = NULL;
00835     float v;
00836
00837     /* Negative mantissa with positive exponent */
00838     v = asm_str2float("-1e2", &end, 10);
00839     TEST_CASE(v > -100.1f && v < -99.9f);
00840     TEST_CASE(*end == '\0');
00841
00842     v = asm_str2float("-2.5e3", &end, 10);
00843     TEST_CASE(v > -2500.1f && v < -2499.9f);
00844     TEST_CASE(*end == '\0');
00845
00846     /* Negative mantissa with negative exponent */
00847     v = asm_str2float("-1.0e-2", &end, 10);
00848     TEST_CASE(v > -0.0101f && v < -0.0099f);
00849     TEST_CASE(*end == '\0');
00850
00851     v = asm_str2float("-5e-1", &end, 10);
00852     TEST_CASE(v > -0.51f && v < -0.49f);
00853     TEST_CASE(*end == '\0');
00854
00855     /* Positive sign with exponent */
00856     v = asm_str2float("+1.5e2", &end, 10);
00857     TEST_CASE(v > 149.9f && v < 150.1f);
00858     TEST_CASE(*end == '\0');
00859
00860     v = asm_str2float("+3e-2", &end, 10);
00861     TEST_CASE(v > 0.0299f && v < 0.0301f);
00862     TEST_CASE(*end == '\0');
00863 }
00864
00865 static void test_str2float_exponent_edge_cases(void)
00866 {
00867     const char *end = NULL;
00868     float v;
00869
00870     /* Zero exponent */
00871     v = asm_str2float("5e0", &end, 10);
00872     TEST_CASE(v > 4.99f && v < 5.01f);
00873     TEST_CASE(*end == '\0');
00874
00875     v = asm_str2float("3.14e0", &end, 10);
00876     TEST_CASE(v > 3.13f && v < 3.15f);
00877     TEST_CASE(*end == '\0');
00878
00879     /* Zero mantissa */
00880     v = asm_str2float("0e5", &end, 10);
00881     TEST_CASE(v > -0.01f && v < 0.01f);
00882     TEST_CASE(*end == '\0');
00883
00884     v = asm_str2float("0.0e-3", &end, 10);
00885     TEST_CASE(v > -0.01f && v < 0.01f);
00886     TEST_CASE(*end == '\0');
00887
00888     /* No integer part */
00889     v = asm_str2float(".5e2", &end, 10);
00890     TEST_CASE(v > 49.9f && v < 50.1f);
00891     TEST_CASE(*end == '\0');
00892
00893     v = asm_str2float(".25e-1", &end, 10);
00894     TEST_CASE(v > 0.0249f && v < 0.0251f);
00895     TEST_CASE(*end == '\0');
00896
00897     /* No fractional part */
00898     v = asm_str2float("10.e2", &end, 10);
00899     TEST_CASE(v > 999.9f && v < 1000.1f);
00900     TEST_CASE(*end == '\0');
00901
00902     /* Uppercase E */
00903     v = asm_str2float("1E2", &end, 10);
00904     TEST_CASE(v > 99.9f && v < 100.1f);
00905     TEST_CASE(*end == '\0');
```

```
00906
00907     v = asm_str2float("5E-3", &end, 10);
00908     TEST_CASE(v > 0.00499f && v < 0.00501f);
00909     TEST_CASE(*end == '\0');
00910 }
00911
00912 static void test_str2float_exponent_with_trailing(void)
00913 {
00914     const char *end = NULL;
00915     float v;
00916
00917     /* Exponent with trailing characters */
00918     v = asm_str2float("1.5e2xyz", &end, 10);
00919     TEST_CASE(v > 149.9f && v < 150.1f);
00920     TEST_CASE(*end == 'x');
00921
00922     v = asm_str2float("3e-1!", &end, 10);
00923     TEST_CASE(v > 0.29f && v < 0.31f);
00924     TEST_CASE(*end == '!');
00925
00926     v = asm_str2float("  -2.5e3  ", &end, 10);
00927     TEST_CASE(v > -2500.1f && v < -2499.9f);
00928     TEST_CASE(*end == ' ');
00929 }
00930
00931 static void test_str2double_exponent_basic(void)
00932 {
00933     const char *end = NULL;
00934     double v;
00935
00936     /* Basic positive exponents */
00937     v = asm_str2double("1e2", &end, 10);
00938     TEST_CASE(v > 99.99 && v < 100.01);
00939     TEST_CASE(*end == '\0');
00940
00941     v = asm_str2double("1.5e3", &end, 10);
00942     TEST_CASE(v > 1499.99 && v < 1500.01);
00943     TEST_CASE(*end == '\0');
00944
00945     /* Basic negative exponents */
00946     v = asm_str2double("1e-2", &end, 10);
00947     TEST_CASE(v > 0.0099 && v < 0.0101);
00948     TEST_CASE(*end == '\0');
00949
00950     v = asm_str2double("-1.0e-2", &end, 10);
00951     TEST_CASE(v > -0.0101 && v < -0.0099);
00952     TEST_CASE(*end == '\0');
00953
00954     /* Higher precision than float */
00955     v = asm_str2double("3.141592653589793e0", &end, 10);
00956     TEST_CASE(v > 3.141592653 && v < 3.141592654);
00957     TEST_CASE(*end == '\0');
00958 }
00959
00960 static void test_str2double_exponent_signed_mantissa(void)
00961 {
00962     const char *end = NULL;
00963     double v;
00964
00965     /* Negative mantissa with exponents */
00966     v = asm_str2double("-2.5e3", &end, 10);
00967     TEST_CASE(v > -2500.01 && v < -2499.99);
00968     TEST_CASE(*end == '\0');
00969
00970     v = asm_str2double("-5e-1", &end, 10);
00971     TEST_CASE(v > -0.51 && v < -0.49);
00972     TEST_CASE(*end == '\0');
00973
00974     /* Positive sign */
00975     v = asm_str2double("+1.5e2", &end, 10);
00976     TEST_CASE(v > 149.99 && v < 150.01);
00977     TEST_CASE(*end == '\0');
00978 }
00979
00980 static void test_str2double_exponent_edge_cases(void)
00981 {
00982     const char *end = NULL;
00983     double v;
00984
00985     /* Zero exponent */
00986     v = asm_str2double("5e0", &end, 10);
00987     TEST_CASE(v > 4.99 && v < 5.01);
00988     TEST_CASE(*end == '\0');
00989
00990     /* Zero mantissa */
00991     v = asm_str2double("0e5", &end, 10);
00992     TEST_CASE(v > -0.01 && v < 0.01);
```

```
00993     TEST_CASE(*end == '\0');
00994
00995     /* No integer part */
00996     v = asm_str2double(".5e2", &end, 10);
00997     TEST_CASE(v > 49.99 && v < 50.01);
00998     TEST_CASE(*end == '\0');
00999
01000     /* Uppercase E */
01001     v = asm_str2double("1E2", &end, 10);
01002     TEST_CASE(v > 99.99 && v < 100.01);
01003     TEST_CASE(*end == '\0');
01004 }
01005
01006 static void test_str2float_double_exponent_different_bases(void)
01007 {
01008     const char *end = NULL;
01009     float vf;
01010     double vd;
01011
01012     /* Binary with exponent (base 2)
01013      * 1.1e3 in base 2 = 1.5 * 2^3 = 1.5 * 8 = 12 */
01014     vf = asm_str2float("1.1e3", &end, 2);
01015     TEST_CASE(vf > 11.9f && vf < 12.1f);
01016     TEST_CASE(*end == '\0');
01017
01018     vd = asm_str2double("1.1e3", &end, 2);
01019     TEST_CASE(vd > 11.99 && vd < 12.01);
01020     TEST_CASE(*end == '\0');
01021
01022     /* Octal with exponent (base 8)
01023      * 7.4e2 in base 8 = (7 + 4/8) * 8^2 = 7.5 * 64 = 480 */
01024     vf = asm_str2float("7.4e2", &end, 8);
01025     TEST_CASE(vf > 479.9f && vf < 480.1f);
01026     TEST_CASE(*end == '\0');
01027
01028     vd = asm_str2double("7.4e2", &end, 8);
01029     TEST_CASE(vd > 479.99 && vd < 480.01);
01030     TEST_CASE(*end == '\0');
01031 }
01032
01033 static void test_str2float_double_exponent_whitespace(void)
01034 {
01035     const char *end = NULL;
01036     float vf;
01037     double vd;
01038
01039     /* Leading whitespace */
01040     vf = asm_str2float("  \t\n1.5e2", &end, 10);
01041     TEST_CASE(vf > 149.9f && vf < 150.1f);
01042     TEST_CASE(*end == '\0');
01043
01044     vd = asm_str2double("  \t\n-2.5e-3", &end, 10);
01045     TEST_CASE(vd > -0.00251 && vd < -0.00249);
01046     TEST_CASE(*end == '\0');
01047 }
01048
01049 static void test_str2float_double_exponent_large_values(void)
01050 {
01051     const char *end = NULL;
01052     float vf;
01053     double vd;
01054
01055     /* Larger exponents */
01056     vf = asm_str2float("1e5", &end, 10);
01057     TEST_CASE(vf > 99999.0f && vf < 100001.0f);
01058     TEST_CASE(*end == '\0');
01059
01060     vd = asm_str2double("1e10", &end, 10);
01061     TEST_CASE(vd > 9999999999.0 && vd < 10000000001.0);
01062     TEST_CASE(*end == '\0');
01063
01064     /* Very small exponents */
01065     vf = asm_str2float("1e-5", &end, 10);
01066     TEST_CASE(vf > 0.000009f && vf < 0.000011f);
01067     TEST_CASE(*end == '\0');
01068
01069     vd = asm_str2double("1e-10", &end, 10);
01070     TEST_CASE(vd > 0.00000000009 && vd < 0.00000000011);
01071     TEST_CASE(*end == '\0');
01072 }
01073
01074 /* ---------------- Main ---------------- */
01075
01076 int main(void)
01077 {
01078     test_ascii_classification_exhaustive_ranges();
01079     test_ascii_classification_full_scan_0_127();
```

```
01080
01081     test_case_conversion_roundtrip();
01082
01083     test_length_matches_strlen_small();
01084
01085     test_memset_basic_and_edges();
01086
01087     test_copy_array_by_indexes_behavior_and_bounds();
01088
01089     test_left_shift_edges();
01090     test_left_pad_edges_and_sentinel();
01091
01092     test_remove_char_form_string_edges();
01093     test_strip_whitespace_properties();
01094     test_str_is_whitespace_edges();
01095
01096     test_strncmp_boolean_edges();
01097     test_str_in_str_overlap_and_edges();
01098
01099     test_base_digit_helpers();
01100     test_str2int();
01101     test_str2size_t();
01102     test_str2float_double();
01103
01104     test_str2float_exponent_basic();
01105     test_str2float_exponent_signed_mantissa();
01106     test_str2float_exponent_edge_cases();
01107     test_str2float_exponent_with_trailing();
01108     test_str2double_exponent_basic();
01109     test_str2double_exponent_signed_mantissa();
01110     test_str2double_exponent_edge_cases();
01111     test_str2float_double_exponent_different_bases();
01112     test_str2float_double_exponent_whitespace();
01113     test_str2float_double_exponent_large_values();
01114
01115     test_get_next_word_from_line_current_behavior();
01116     test_get_word_and_cut_edges();
01117
01118     test_get_line_tmpfile();
01119     test_get_line_too_long();
01120
01121     test_strncat_current_behavior_and_sentinel();
01122
01123     if (g_tests_failed == 0) {
01124         if (g_tests_warned == 0) {
01125             printf("[OK] %d tests passed\n", g_tests_run);
01126         } else {
01127             printf("[OK] %d tests passed, %d warnings\n", g_tests_run,
01128                    g_tests_warned);
01129         }
01130         return 0;
01131     }
01132
01133     fprintf(stderr, "[FAIL] %d/%d tests failed (%d warnings)\n", g_tests_failed,
01134             g_tests_run, g_tests_warned);
01135     return 1;
01136 }
```

Index