

Almog String Manipulation

Generated by Doxygen 1.9.1

1 File Index	1
1.1 File List	1
2 File Documentation	3
2.1 Almog_String_Manipulation.h File Reference	3
2.1.1 Detailed Description	6
2.1.2 Macro Definition Documentation	6
2.1.2.1 asm_dprintCHAR	6
2.1.2.2 asm_dprintDOUBLE	7
2.1.2.3 asm_dprintERROR	7
2.1.2.4 asm_dprintFLOAT	7
2.1.2.5 asm_dprintINT	8
2.1.2.6 asm_dprintSIZE_T	8
2.1.2.7 asm_dprintSTRING	8
2.1.2.8 ASM_MALLOC	9
2.1.2.9 asm_max	9
2.1.2.10 ASM_MAX_LEN	9
2.1.2.11 asm_min	10
2.1.3 Function Documentation	10
2.1.3.1 asm_check_char_belong_to_base()	10
2.1.3.2 asm_copy_array_by_indexes()	11
2.1.3.3 asm_get_char_value_in_base()	11
2.1.3.4 asm_get_line()	12
2.1.3.5 asm_get_next_token_from_str()	13
2.1.3.6 asm_get_token_and_cut()	14
2.1.3.7 asm_isalnum()	15
2.1.3.8 asm_isalpha()	15
2.1.3.9 asm_isbdigit()	16
2.1.3.10 asm_iscntrl()	16
2.1.3.11 asm_isdigit()	17
2.1.3.12 asm_isgraph()	17
2.1.3.13 asm_islower()	17
2.1.3.14 asm_isodigit()	18
2.1.3.15 asm_isprint()	18
2.1.3.16 asm_ispunct()	19
2.1.3.17 asm_isspace()	19
2.1.3.18 asm_isupper()	20
2.1.3.19 asm_isxdigit()	20
2.1.3.20 asm_isXdigit()	20
2.1.3.21 asm_length()	21
2.1.3.22 asm_memset()	22
2.1.3.23 asm_pad_left()	23

2.1.3.24 asm_print_many_times()	24
2.1.3.25 asm_remove_char_from_string()	24
2.1.3.26 asm_shift_left()	24
2.1.3.27 asm_str2double()	25
2.1.3.28 asm_str2float()	26
2.1.3.29 asm_str2int()	27
2.1.3.30 asm_str2size_t()	28
2.1.3.31 asm_str_in_str()	28
2.1.3.32 asm_str_is_whitespace()	29
2.1.3.33 asm_strdup()	29
2.1.3.34 asm_strip_whitespace()	30
2.1.3.35 asm_strncat()	30
2.1.3.36 asm_strncmp()	31
2.1.3.37 asm_strncpy()	32
2.1.3.38 asm_tolower()	33
2.1.3.39 asm_toupper()	33
2.1.3.40 asm_trim_left_whitespace()	33
2.2 Almog_String_Manipulation.h	34
2.3 temp.c File Reference	42
2.3.1 Macro Definition Documentation	42
2.3.1.1 ALMOG_STRING_MANIPULATION_IMPLEMENTATION	42
2.3.2 Function Documentation	43
2.3.2.1 main()	43
2.4 temp.c	43
2.5 tests.c File Reference	43
2.5.1 Macro Definition Documentation	45
2.5.1.1 ALMOG_STRING_MANIPULATION_IMPLEMENTATION	45
2.5.1.2 NO_ERRORS	45
2.5.1.3 TEST_CASE	45
2.5.1.4 TEST_EQ_INT	45
2.5.1.5 TEST_EQ_SIZE	46
2.5.1.6 TEST_EQ_STR	46
2.5.1.7 TEST_NE_STR	46
2.5.1.8 TEST_WARN	46
2.5.2 Function Documentation	46
2.5.2.1 fill_sentinel()	47
2.5.2.2 is_nul_terminated_within()	47
2.5.2.3 main()	47
2.5.2.4 rand_ascii_printable()	47
2.5.2.5 test_ascii_classification_exhaustive_ranges()	48
2.5.2.6 test_ascii_classification_full_scan_0_127()	48
2.5.2.7 test_base_digit_helpers()	48

2.5.2.8 test_case_conversion_roundtrip()	48
2.5.2.9 test_copy_array_by_indexes_behavior_and_bounds()	49
2.5.2.10 test_get_line_tmpfile()	49
2.5.2.11 test_get_line_too_long()	49
2.5.2.12 test_get_next_word_from_line_current_behavior()	49
2.5.2.13 test_get_word_and_cut_edges()	50
2.5.2.14 test_left_pad_edges_and_sentinel()	50
2.5.2.15 test_left_shift_edges()	50
2.5.2.16 test_length_matches_strlen_small()	50
2.5.2.17 test_memset_basic_and_edges()	51
2.5.2.18 test_remove_char_form_string_edges()	51
2.5.2.19 test_str2double_exponent_basic()	51
2.5.2.20 test_str2double_exponent_edge_cases()	51
2.5.2.21 test_str2double_exponent_signed_mantissa()	52
2.5.2.22 test_str2float_double()	52
2.5.2.23 test_str2float_double_exponent_different_bases()	52
2.5.2.24 test_str2float_double_exponent_large_values()	52
2.5.2.25 test_str2float_double_exponent_whitespace()	53
2.5.2.26 test_str2float_exponent_basic()	53
2.5.2.27 test_str2float_exponent_edge_cases()	53
2.5.2.28 test_str2float_exponent_signed_mantissa()	53
2.5.2.29 test_str2float_exponent_with_trailing()	54
2.5.2.30 test_str2int()	54
2.5.2.31 test_str2size_t()	54
2.5.2.32 test_str_in_str_overlap_and_edges()	54
2.5.2.33 test_str_is_whitespace_edges()	55
2.5.2.34 test_strip_whitespace_properties()	55
2.5.2.35 test_strncat_current_behavior_and_sentinel()	55
2.5.2.36 test_strncmp_boolean_edges()	55
2.5.2.37 xorshift32()	56
2.5.3 Variable Documentation	56
2.5.3.1 g_tests_failed	56
2.5.3.2 g_tests_run	56
2.5.3.3 g_tests_warned	56
2.5.3.4 rng_state	57
2.6 tests.c	57

Chapter 1

File Index

1.1 File List

Here is a list of all files with brief descriptions:

Almog_String_Manipulation.h	
Lightweight string and line manipulation helpers	3
temp.c	42
tests.c	43

Chapter 2

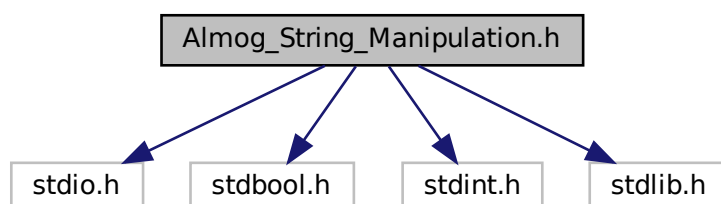
File Documentation

2.1 Almog_String_Manipulation.h File Reference

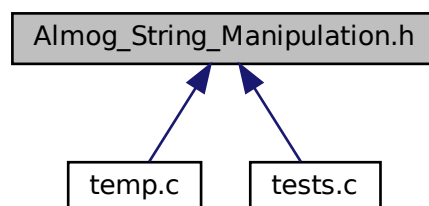
Lightweight string and line manipulation helpers.

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

Include dependency graph for Almog_String_Manipulation.h:



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



Macros

- `#define ASM_MALLOC` `malloc`
- `#define ASM_MAX_LEN` `(int)1e3`
Maximum number of characters processed in some string operations.
- `#define asm_dprintSTRING(expr)` `printf(#expr " = %s\n", expr)`
Debug-print a C string expression as "expr = value\n".
- `#define asm_dprintCHAR(expr)` `printf(#expr " = %c\n", expr)`
Debug-print a character expression as "expr = c\n".
- `#define asm_dprintINT(expr)` `printf(#expr " = %d\n", expr)`
Debug-print an integer expression as "expr = n\n".
- `#define asm_dprintFLOAT(expr)` `printf(#expr " = %g\n", expr)`
Debug-print a float expression as "expr = n\n".
- `#define asm_dprintDOUBLE(expr)` `printf(#expr " = %g\n", expr)`
Debug-print a double expression as "expr = n\n".
- `#define asm_dprintSIZE_T(expr)` `printf(#expr " = %zu\n", expr)`
Debug-print a size_t expression as "expr = n\n".
- `#define asm_dprintERROR(fmt, ...)`
- `#define asm_min(a, b) ((a) < (b) ? (a) : (b))`
Return the smaller of two values (macro).
- `#define asm_max(a, b) ((a) > (b) ? (a) : (b))`
Return the larger of two values (macro).

Functions

- `bool asm_check_char_belong_to_base` `(const char c, const size_t base)`
Check if a character is a valid digit in a given base.
- `void asm_copy_array_by_indexes` `(char *const target, const int start, const int end, const char *const src)`
Copy a substring from `src` into `target` by indices and null-terminate.
- `int asm_get_char_value_in_base` `(const char c, const size_t base)`
Convert a digit character to its numeric value in base-N.
- `int asm_get_line` `(FILE *fp, char *const dst)`
Read a single line from a stream into a buffer.
- `int asm_get_next_token_from_str` `(char *const dst, const char *const src, const char delimiter)`
Copy characters from the start of a string into a token buffer.
- `int asm_get_token_and_cut` `(char *const dst, char *src, const char delimiter, const bool leave_delimiter)`
Extract the next token into `dst` and remove the corresponding prefix from `src`.
- `bool asm_isalnum` `(char c)`
Test for an alphanumeric character (ASCII).
- `bool asm_isalpha` `(char c)`
Test for an alphabetic character (ASCII).
- `bool asm_isbigit` `(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_isdigit](#) (const char c)
- Test for an octal digit (ASCII).*

 - bool [asm_isprint](#) (char c)
- Test for any printable character including space (ASCII).*

 - bool [asm_isspace](#) (char c)
- Test for a punctuation character (ASCII).*

 - bool [asm_isspace](#) (char c)
- Test for a whitespace character (ASCII).*

 - bool [asm_isupper](#) (char c)
- Test for an uppercase letter (ASCII).*

 - bool [asm_isxdigit](#) (char c)
- Test for a hexadecimal digit (lowercase or decimal).*

 - bool [asm_isXdigit](#) (char c)
- Test for a hexadecimal digit (uppercase or decimal).*

 - size_t [asm_length](#) (const char *const str)
- Compute the length of a null-terminated C string.*

 - void * [asm_memset](#) (void *const des, const unsigned char value, const size_t n)
- Set a block of memory to a repeated byte value.*

 - void [asm_pad_left](#) (char *const s, const size_t padding, const char pad)
- Left-pad a string in-place.*

 - void [asm_print_many_times](#) (const char *const str, const size_t n)
- Print a string n times, then print a newline.*

 - void [asm_remove_char_from_string](#) (char *const s, const size_t index)
- Remove a single character from a string by index.*

 - void [asm_shift_left](#) (char *const s, const size_t shift)
- Shift a string left in-place by shift characters.*

 - int [asm_str_in_str](#) (const char *const src, const char *const word_to_search)
- Count occurrences of a substring within a string.*

 - double [asm_str2double](#) (const char *const s, const char **const end, const size_t base)
- Convert a string to double in the given base with exponent support.*

 - float [asm_str2float](#) (const char *const s, const char **const end, const size_t base)
- Convert a string to float in the given base with exponent support.*

 - int [asm_str2int](#) (const char *const s, const char **const end, const size_t base)
- Convert a string to int in the given base.*

 - size_t [asm_str2size_t](#) (const char *const s, const char **const end, const size_t base)
- Convert a string to size_t in the given base.*

 - void [asm_strip_whitespace](#) (char *const s)
- Remove all ASCII whitespace characters from a string in-place.*

 - bool [asm_str_is_whitespace](#) (const char *const s)
- Check whether a string contains only ASCII whitespace characters.*

 - char * [asm_strdup](#) (const char *const s, size_t length)
- Allocate and copy up to length characters from s.*

 - int [asm_strncat](#) (char *const s1, const char *const s2, const size_t N)
- Append up to N characters from s2 to the end of s1.*

 - int [asm_strncmp](#) (const char *s1, const char *s2, const size_t N)
- Compare up to N characters for equality (boolean result).*

 - int [asm_strncpy](#) (char *const s1, const char *const s2, const size_t N)
- Copy up to N characters from s2 into s1 (non-standard).*

 - void [asm_tolower](#) (char *const s)
- Convert all ASCII letters in a string to lowercase in-place.*

- void `asm_toupper` (char *const s)
Convert all ASCII letters in a string to uppercase in-place.
- void `asm_trim_left_whitespace` (char *const s)
Remove leading ASCII whitespace from a string in-place.

2.1.1 Detailed Description

Lightweight string and line manipulation helpers.

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

- Reading a single line from a FILE stream
- Measuring string length
- Extracting the next token from a string using a delimiter (does not skip whitespace)
- Cutting the extracted token from the source buffer (optionally also removing the delimiter)
- Copying a substring by indices
- Counting occurrences of a substring
- A boolean-style strcmp (returns 1 on equality, 0 otherwise)
- ASCII-only character classification helpers (isalnum, isalpha, ...)
- ASCII case conversion (toupper / tolower)
- In-place whitespace stripping and left padding
- Base-N string-to-number conversion for int, size_t, float, and double

Usage

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

Notes and limitations

- All destination buffers must be large enough; functions do not grow or allocate buffers.
- `asm_get_line` 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

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

Definition at line 90 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

<i>expr</i>	An expression that yields a double. The expression is evaluated exactly once.
-------------	---

Definition at line 117 of file [Almog_String_Manipulation.h](#).

2.1.2.3 asm_dprintERROR

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

Value:

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

Definition at line 128 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

<i>expr</i>	An expression that yields a float. The expression is evaluated exactly once.
-------------	--

Definition at line 108 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

<i>expr</i>	An expression that yields an int. The expression is evaluated exactly once.
-------------	---

Definition at line 99 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

<i>expr</i>	An expression that yields a <code>size_t</code> . The expression is evaluated exactly once.
-------------	---

Definition at line 126 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

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

Definition at line 81 of file [Almog_String_Manipulation.h](#).

2.1.2.8 ASM_MALLOC

```
#define ASM_MALLOC malloc
```

Definition at line 50 of file [Almog_String_Manipulation.h](#).

2.1.2.9 asm_max

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

Return the larger of two values (macro).

Parameters

<i>a</i>	First value.
<i>b</i>	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 156 of file [Almog_String_Manipulation.h](#).

2.1.2.10 ASM_MAX_LEN

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

Maximum number of characters processed in some string operations.

This constant 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 71 of file [Almog_String_Manipulation.h](#).

2.1.2.11 `asm_min`

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

Return the smaller of two values (macro).

Parameters

<i>a</i>	First value.
<i>b</i>	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 143 of file [Almog_String_Manipulation.h](#).

2.1.3 Function Documentation

2.1.3.1 `asm_check_char_belong_to_base()`

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

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

Parameters

<i>c</i>	Character to test (e.g., '0'–'9', 'a'–'z', 'A'–'Z').
<i>base</i>	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 214 of file [Almog_String_Manipulation.h](#).

References [asm_dprintERROR](#), and [asm_isdigit](#)().

Referenced by [asm_get_char_value_in_base\(\)](#), [asm_str2double\(\)](#), [asm_str2float\(\)](#), [asm_str2int\(\)](#), [asm_str2size_t\(\)](#), and [test_base_digit_helpers](#)().

2.1.3.2 asm_copy_array_by_indexes()

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

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

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

Parameters

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

Warning

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

Definition at line 249 of file [Almog_String_Manipulation.h](#).

Referenced by [test_copy_array_by_indexes_behavior_and_bounds](#)().

2.1.3.3 asm_get_char_value_in_base()

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

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

Parameters

<i>c</i>	Digit character ('0'–'9', 'a'–'z', 'A'–'Z').
<i>base</i>	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 271 of file [Almog_String_Manipulation.h](#).

References [asm_check_char_belong_to_base\(\)](#), [asm_isdigit\(\)](#), and [asm_isupper\(\)](#).

Referenced by [asm_str2double\(\)](#), [asm_str2float\(\)](#), [asm_str2int\(\)](#), [asm_str2size_t\(\)](#), and [test_base_digit_helpers\(\)](#).

2.1.3.4 asm_get_line()

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

Read a single line from a stream into a buffer.

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

Parameters

<i>fp</i>	Input stream (must be non-NULL).
<i>dst</i>	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 307 of file [Almog_String_Manipulation.h](#).

References [asm_dprintERROR](#), and [ASM_MAX_LEN](#).

Referenced by [test_get_line_tmpfile\(\)](#), and [test_get_line_too_long\(\)](#).

2.1.3.5 asm_get_next_token_from_str()

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

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

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

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

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

Parameters

<i>dst</i>	Destination buffer for the extracted token. Must be large enough to hold the token plus the null terminator.
<i>src</i>	Source C string to parse (not modified by this function).
<i>delimiter</i>	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 354 of file [Almog_String_Manipulation.h](#).

Referenced by [asm_get_token_and_cut\(\)](#), and [test_get_next_word_from_line_current_behavior\(\)](#).

2.1.3.6 asm_get_token_and_cut()

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

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

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

- If `leave_delimiter` is true:
 - `src` is 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

<i>dst</i>	Destination buffer for the extracted token (must be large enough for the token plus the null terminator).
<i>src</i>	Source buffer, modified in-place by this function.
<i>delimiter</i>	Delimiter character used to stop token extraction.
<i>leave_delimiter</i>	If true, do not remove the delimiter from <code>src</code> ; 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 400 of file [Almog_String_Manipulation.h](#).

References [asm_get_next_token_from_str\(\)](#), and [asm_shift_left\(\)](#).

Referenced by [test_get_word_and_cut_edges\(\)](#).

2.1.3.7 asm_isalnum()

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

Test for an alphanumeric character (ASCII).

Parameters

<i>c</i>	Character to test.
----------	--------------------

Returns

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

Definition at line 421 of file [Almog_String_Manipulation.h](#).

References [asm_isalpha\(\)](#), and [asm_isdigit\(\)](#).

Referenced by [test_ascii_classification_exhaustive_ranges\(\)](#), and [test_ascii_classification_full_scan_0_127\(\)](#).

2.1.3.8 asm_isalpha()

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

Test for an alphabetic character (ASCII).

Parameters

<i>c</i>	Character to test.
----------	--------------------

Returns

true if *c* is 'A'-'Z' or 'a'-'z'; false otherwise.

Definition at line 432 of file [Almog_String_Manipulation.h](#).

References [asm_islower\(\)](#), and [asm_isupper\(\)](#).

Referenced by [asm_isalnum\(\)](#), [test_ascii_classification_exhaustive_ranges\(\)](#), and [test_ascii_classification_full_scan_0_127\(\)](#).

2.1.3.9 `asm_isbdigit()`

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

Test for a binary digit (ASCII).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

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

Definition at line 443 of file [Almog_String_Manipulation.h](#).

2.1.3.10 `asm_iscntrl()`

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

Test for a control character (ASCII).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

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

Definition at line 458 of file [Almog_String_Manipulation.h](#).

Referenced by [test_ascii_classification_exhaustive_ranges\(\)](#).

2.1.3.11 asm_isdigit()

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

Test for a decimal digit (ASCII).

Parameters

<i>c</i>	Character to test.
----------	--------------------

Returns

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

Definition at line 473 of file [Almog_String_Manipulation.h](#).

Referenced by [asm_check_char_belong_to_base\(\)](#), [asm_get_char_value_in_base\(\)](#), [asm_isalnum\(\)](#), [asm_isxdigit\(\)](#), [asm_isXdigit\(\)](#), [test_ascii_classification_exhaustive_ranges\(\)](#), and [test_ascii_classification_full_scan_0_127\(\)](#).

2.1.3.12 asm_isgraph()

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

Test for any printable character except space (ASCII).

Parameters

<i>c</i>	Character to test.
----------	--------------------

Returns

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

Definition at line 488 of file [Almog_String_Manipulation.h](#).

Referenced by [asm_isprint\(\)](#), [test_ascii_classification_exhaustive_ranges\(\)](#), and [test_ascii_classification_full_scan_0_127\(\)](#).

2.1.3.13 asm_islower()

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

Test for a lowercase letter (ASCII).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

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

Definition at line 503 of file [Almog_String_Manipulation.h](#).

Referenced by [asm_isalpha\(\)](#), [asm_toupper\(\)](#), [test_ascii_classification_exhaustive_ranges\(\)](#), and [test_ascii_classification_full_scan_C](#).

2.1.3.14 asm_isodigit()

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

Test for an octal digit (ASCII).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

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

Definition at line 518 of file [Almog_String_Manipulation.h](#).

2.1.3.15 asm_isprint()

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

Test for any printable character including space (ASCII).

Parameters

<code>c</code>	Character to test.
----------------	--------------------

Returns

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

Definition at line 534 of file [Almog_String_Manipulation.h](#).

References [asm_isgraph\(\)](#).

Referenced by [test_ascii_classification_exhaustive_ranges\(\)](#), and [test_ascii_classification_full_scan_0_127\(\)](#).

2.1.3.16 asm_ispunct()

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

Test for a punctuation character (ASCII).

Parameters

<i>c</i>	Character to test.
----------	--------------------

Returns

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

Definition at line 546 of file [Almog_String_Manipulation.h](#).

Referenced by [test_ascii_classification_exhaustive_ranges\(\)](#).

2.1.3.17 asm_isspace()

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

Test for a whitespace character (ASCII).

Parameters

<i>c</i>	Character to test.
----------	--------------------

Returns

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

Definition at line 562 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_prop](#).

2.1.3.18 asm_isupper()

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

Test for an uppercase letter (ASCII).

Parameters

<i>c</i>	Character to test.
----------	--------------------

Returns

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

Definition at line 578 of file [Almog_String_Manipulation.h](#).

Referenced by [asm_get_char_value_in_base\(\)](#), [asm_isalpha\(\)](#), [asm_tolower\(\)](#), [test_ascii_classification_exhaustive_ranges\(\)](#), and [test_ascii_classification_full_scan_0_127\(\)](#).

2.1.3.19 asm_isxdigit()

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

Test for a hexadecimal digit (lowercase or decimal).

Parameters

<i>c</i>	Character to test.
----------	--------------------

Returns

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

Definition at line 593 of file [Almog_String_Manipulation.h](#).

References [asm_isdigit\(\)](#).

Referenced by [test_ascii_classification_exhaustive_ranges\(\)](#).

2.1.3.20 asm_isXdigit()

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

Test for a hexadecimal digit (uppercase or decimal).

Parameters

<i>c</i>	Character to test.
----------	--------------------

Returns

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

Definition at line 608 of file [Almog_String_Manipulation.h](#).

References [asm_isdigit\(\)](#).

Referenced by [test_ascii_classification_exhaustive_ranges\(\)](#).

2.1.3.21 asm_length()

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

Compute the length of a null-terminated C string.

Parameters

<i>str</i>	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 627 of file [Almog_String_Manipulation.h](#).

References [asm_dprintERROR](#), and [ASM_MAX_LEN](#).

Referenced by [asm_pad_left\(\)](#), [asm_remove_char_from_string\(\)](#), [asm_shift_left\(\)](#), [asm_str_in_str\(\)](#), [asm_str_is_whitespace\(\)](#), [asm_strip_whitespace\(\)](#), [asm_strncat\(\)](#), [asm_tolower\(\)](#), [asm_toupper\(\)](#), [asm_trim_left_whitespace\(\)](#), and [test_length_matches_strlen_small\(\)](#).

2.1.3.22 `asm_memset()`

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

Set a block of memory to a repeated byte value.

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

Parameters

<i>des</i>	Destination memory block to modify. Must point to a valid buffer of at least <i>n</i> bytes.
<i>value</i>	Unsigned byte value to store repeatedly.
<i>n</i>	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 662 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

<i>s</i>	String to pad. Modified in-place.
<i>padding</i>	Number of leading spaces to insert.
<i>pad</i>	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 685 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

<i>str</i>	String to print (as-is with <code>printf("%s", ...)</code>).
<i>n</i>	Number of times to print <i>str</i> .

Definition at line 702 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

<i>s</i>	String to modify in-place. Must be null-terminated.
<i>index</i>	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 722 of file [Almog_String_Manipulation.h](#).

References [asm_dprintERROR](#), and [asm_length\(\)](#).

Referenced by [asm_strip_whitespace\(\)](#), and [test_remove_char_from_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

<i>s</i>	String to modify in-place. Must be null-terminated.
<i>shift</i>	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 751 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

<i>s</i>	String to convert. Leading ASCII whitespace is skipped.
<i>end</i>	If non-NULL, *end is set to point to the first character not used in the conversion.
<i>base</i>	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 828 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

<i>s</i>	String to convert. Leading ASCII whitespace is skipped.
<i>end</i>	If non-NULL, *end is set to point to the first character not used in the conversion.
<i>base</i>	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 918 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_large\(\)](#), [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

<i>s</i>	String to convert. Leading ASCII whitespace is skipped.
<i>end</i>	If non-NULL, *end is set to point to the first character not used in the conversion.
<i>base</i>	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, and 0 is returned. For *end*:

- if a negative sign is encountered, this implementation leaves *end pointing to the original *s* (before whitespace skip);
- if the base is invalid, this implementation sets *end to the first non-whitespace character (i.e., *s* after whitespace skip).

Definition at line 996 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

<i>s</i>	String to convert. Leading ASCII whitespace is skipped.
<i>end</i>	If non-NULL, *end is set to point to the first character not used in the conversion.
<i>base</i>	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 1041 of file [Almog_String_Manipulation.h](#).

References [asm_check_char_belong_to_base\(\)](#), [asm_dprintERROR](#), [asm_get_char_value_in_base\(\)](#), and [asm_isspace\(\)](#).

Referenced by [test_str2size_t\(\)](#).

2.1.3.31 `asm_str_in_str()`

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

Count occurrences of a substring within a string.

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

Parameters

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

Returns

The number of (possibly overlapping) occurrences found.

Note

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

Definition at line 782 of file [Almog_String_Manipulation.h](#).

References [asm_length\(\)](#), and [asm_strncmp\(\)](#).

Referenced by [test_str_in_str_overlap_and_edges\(\)](#).

2.1.3.32 asm_str_is_whitespace()

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

Check whether a string contains only ASCII whitespace characters.

Parameters

<code>s</code>	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 1109 of file [Almog_String_Manipulation.h](#).

References [asm_isspace\(\)](#), and [asm_length\(\)](#).

Referenced by [test_str_is_whitespace_edges\(\)](#).

2.1.3.33 asm_strdup()

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

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

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

Parameters

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

Returns

Newly allocated string.

Note

This is not the same as POSIX `strdup()`: it does not compute length by itself and may intentionally truncate. Allocation failure is not handled: if `ASM_MALLOC` returns `NULL`, this implementation will pass `NULL` to [asm_strncpy\(\)](#), resulting in undefined behavior.

Definition at line 1137 of file [Almog_String_Manipulation.h](#).

References [ASM_MALLOC](#), and [asm_strncpy\(\)](#).

2.1.3.34 asm_strip_whitespace()

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

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

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

Parameters

<i>s</i>	String to modify in-place. Must be null-terminated.
----------	---

Definition at line 1088 of file [Almog_String_Manipulation.h](#).

References [asm_isspace\(\)](#), [asm_length\(\)](#), and [asm_remove_char_from_string\(\)](#).

Referenced by [test_strip_whitespace_properties\(\)](#).

2.1.3.35 asm_strncat()

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

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

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

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

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

Parameters

<i>s1</i>	Destination string buffer (must be null-terminated).
<i>s2</i>	Source string buffer (must be null-terminated).
<i>N</i>	Maximum number of characters to append. If <i>N</i> == 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 1166 of file [Almog_String_Manipulation.h](#).

References [asm_dprintERROR](#), [asm_length\(\)](#), and [ASM_MAX_LEN](#).

Referenced by [test_strncat_current_behavior_and_sentinel\(\)](#).

2.1.3.36 asm_strncmp()

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

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

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

Parameters

<i>s1</i>	First string (may be shorter than N).
<i>s2</i>	Second string (may be shorter than N).
<i>N</i>	Number of characters to compare. If <i>N</i> == 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 1211 of file [Almog_String_Manipulation.h](#).

References [ASM_MAX_LEN](#).

Referenced by [asm_str_in_str\(\)](#), and [test_strncmp_boolean_edges\(\)](#).

2.1.3.37 asm_strncpy()

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

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

Copies 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 1244 of file [Almog_String_Manipulation.h](#).

Referenced by [asm_strdup\(\)](#).

2.1.3.38 asm_tolower()

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

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

Parameters

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

Definition at line 1262 of file [Almog_String_Manipulation.h](#).

References [asm_isupper\(\)](#), and [asm_length\(\)](#).

Referenced by [test_case_conversion_roundtrip\(\)](#).

2.1.3.39 asm_toupper()

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

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

Parameters

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

Definition at line 1277 of file [Almog_String_Manipulation.h](#).

References [asm_islower\(\)](#), and [asm_length\(\)](#).

Referenced by [test_case_conversion_roundtrip\(\)](#).

2.1.3.40 asm_trim_left_whitespace()

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

Remove leading ASCII whitespace from a string in-place.

Finds the first character in *s* for which [asm_isspace\(\)](#) is false and left-shifts the string so that character becomes the first character.

Parameters

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

Definition at line 1295 of file [Almog_String_Manipulation.h](#).

References [asm_isspace\(\)](#), [asm_length\(\)](#), and [asm_shift_left\(\)](#).

2.2 Almog_String_Manipulation.h

```

00001
00041 #ifndef ALMOG_STRING_MANIPULATION_H_
00042 #define ALMOG_STRING_MANIPULATION_H_
00043
00044 #include <stdio.h>
00045 #include <stdbool.h>
00046 #include <stdint.h>
00047
00048 #ifndef ASM_MALLOC
00049 #include <stdlib.h>
00050 #define ASM_MALLOC malloc
00051 #endif
00052
00070 #ifndef ASM_MAX_LEN
00071 #define ASM_MAX_LEN (int)1e3
00072 #endif
00073
00081 #define asm_dprintSTRING(expr) printf(#expr " = %s\n", expr)
00082
00090 #define asm_dprintCHAR(expr) printf(#expr " = %c\n", expr)
00091
00099 #define asm_dprintINT(expr) printf(#expr " = %d\n", expr)
00100
00108 #define asm_dprintFLOAT(expr) printf(#expr " = %#g\n", expr)
00109
00117 #define asm_dprintDOUBLE(expr) printf(#expr " = %#g\n", expr)
00118
00126 #define asm_dprintSIZE_T(expr) printf(#expr " = %zu\n", expr)
00127
00128 #define asm_dprintERROR(fmt, ...) \
00129     fprintf(stderr, "\n%s:%d:\n[Error] in function '%s':\n"      " \
00130     fmt "\n\n", __FILE__, __LINE__, __func__, __VA_ARGS__)
00131
00143 #define asm_min(a, b) ((a) < (b) ? (a) : (b))
00144
00156 #define asm_max(a, b) ((a) > (b) ? (a) : (b))
00157
00158 bool    asm_check_char_belong_to_base(const char c, const size_t base);
00159 void    asm_copy_array_by_indexes(char * const target, const int start, const int end, const char *
const src);
00160 int     asm_get_char_value_in_base(const char c, const size_t base);
00161 int     asm_get_line(FILE *fp, char * const dst);
00162 int     asm_get_next_token_from_str(char * const dst, const char * const src, const char delimiter);
00163 int     asm_get_token_and_cut(char * const dst, char *src, const char delimiter, const bool
leave_delimiter);
00164 bool    asm_isalnum(const char c);
00165 bool    asm_isalpha(const char c);
00166 bool    asm_isbdigit(const char c);
00167 bool    asm_iscntrl(const char c);
00168 bool    asm_isdigit(const char c);
00169 bool    asm_isgraph(const char c);
00170 bool    asm_islower(const char c);
00171 bool    asm_isodigit(const char c);
00172 bool    asm_isprint(const char c);
00173 bool    asm_isspace(const char c);
00174 bool    asm_isspace(const char c);
00175 bool    asm_isupper(const char c);
00176 bool    asm_isxdigit(const char c);
00177 bool    asm_isXdigit(const char c);
00178 size_t  asm_length(const char * const str);
00179 void *  asm_memset(void * const des, const unsigned char value, const size_t n);
00180 void    asm_pad_left(char * const s, const size_t padding, const char pad);
00181 void    asm_print_many_times(const char * const str, const size_t n);
00182 void    asm_remove_char_from_string(char * const s, const size_t index);
00183 void    asm_shift_left(char * const s, const size_t shift);
00184 int     asm_str_in_str(const char * const src, const char * const word_to_search);
00185 double  asm_str2double(const char * const s, const char ** const end, const size_t base);
00186 float   asm_str2float(const char * const s, const char ** const end, const size_t base);
00187 int     asm_str2int(const char * const s, const char ** const end, const size_t base);
00188 size_t  asm_str2size_t(const char * const s, const char ** const end, const size_t base);
00189 void    asm_strip_whitespace(char * const s);
00190 bool    asm_str_is_whitespace(const char * const s);
00191 char *  asm_strdup(const char * const s, size_t length);
00192 int     asm_strncat(char * const s1, const char * const s2, const size_t N);
00193 int     asm_strncmp(const char * const s1, const char * const s2, const size_t N);
00194 int     asm_strncpy(char * const s1, const char * const s2, const size_t N);

```

```

00195 void    asm_tolower(char * const s);
00196 void    asm_toupper(char * const s);
00197 void    asm_trim_left_whitespace(char *s);
00198
00199 #endif /*ALMOG_STRING_MANIPULATION_H_*/
00200
00201 #ifdef ALMOG_STRING_MANIPULATION_IMPLEMENTATION
00202 #undef ALMOG_STRING_MANIPULATION_IMPLEMENTATION
00203
00214 bool asm_check_char_belong_to_base(const char c, const size_t base)
00215 {
00216     if (base > 36 || base < 2) {
00217         #ifndef ASM_NO_ERRORS
00218             asm_dprintERROR("Supported bases are [2...36]. Inputted: %zu", base);
00219         #endif
00220         return false;
00221     }
00222     if (base <= 10) {
00223         return c >= '0' && c <= '9'+(char)base-10;
00224     }
00225     if (base > 10) {
00226         return asm_isdigit(c) || (c >= 'A' && c <= ('A'+(char)base-11)) || (c >= 'a' && c <=
('a'+(char)base-11));
00227     }
00228
00229     return false;
00230 }
00231
00249 void asm_copy_array_by_indexes(char * const target, const int start, const int end, const char * const
src)
00250 {
00251     if (start > end) return;
00252     int j = 0;
00253     for (int i = start; i <= end; i++) {
00254         target[j] = src[i];
00255         j++;
00256     }
00257     if (target[j-1] != '\0') {
00258         target[j] = '\0';
00259     }
00260 }
00261
00271 int asm_get_char_value_in_base(const char c, const size_t base)
00272 {
00273     if (!asm_check_char_belong_to_base(c, base)) return -1;
00274     if (asm_isdigit(c)) {
00275         return c - '0';
00276     } else if (asm_isupper(c)) {
00277         return c - 'A' + 10;
00278     } else {
00279         return c - 'a' + 10;
00280     }
00281 }
00282
00307 int asm_get_line(FILE *fp, char * const dst)
00308 {
00309     int i = 0;
00310     int c;
00311     while ((c = fgetc(fp)) != '\n' && c != EOF) {
00312         dst[i++] = c;
00313         if (i >= ASM_MAX_LEN) {
00314             #ifndef ASM_NO_ERRORS
00315                 asm_dprintERROR("%s", "index exceeds ASM_MAX_LEN. Line in file is too long.");
00316             #endif
00317             dst[i-1] = '\0';
00318             return -1;
00319         }
00320     }
00321     dst[i] = '\0';
00322     if (c == EOF && i == 0) {
00323         return -1;
00324     }
00325     return i;
00326 }
00327
00354 int asm_get_next_token_from_str(char * const dst, const char * const src, const char delimiter)
00355 {
00356     int i = 0, j = 0;
00357     char c;
00358     while ((c = src[i]) != delimiter && c != '\0') {
00359         dst[j++] = src[i++];
00360     }
00361
00362     dst[j] = '\0';
00363
00364     return j;
00365 }

```

```

00366
00400 int asm_get_token_and_cut(char * const dst, char *src, const char delimiter, const bool
    leave_delimiter)
00401 {
00402     int new_src_start_index = asm_get_next_token_from_str(dst, src, delimiter);
00403     bool delimiter_at_start = src[new_src_start_index] == delimiter;
00404
00405     if (leave_delimiter) {
00406         asm_shift_left(src, new_src_start_index);
00407     } else if (delimiter_at_start) {
00408         asm_shift_left(src, new_src_start_index + 1);
00409     } else {
00410         src[0] = '\0';
00411     }
00412     return new_src_start_index ? 1 : 0;
00413 }
00414
00421 bool asm_isalnum(char c)
00422 {
00423     return asm_isalpha(c) || asm_isdigit(c);
00424 }
00425
00432 bool asm_isalpha(char c)
00433 {
00434     return asm_isupper(c) || asm_islower(c);
00435 }
00436
00443 bool asm_isbdigit(const char c)
00444 {
00445     if (c == '0' || c == '1') {
00446         return true;
00447     } else {
00448         return false;
00449     }
00450 }
00451
00458 bool asm_iscntrl(char c)
00459 {
00460     if ((c >= 0 && c <= 31) || c == 127) {
00461         return true;
00462     } else {
00463         return false;
00464     }
00465 }
00466
00473 bool asm_isdigit(char c)
00474 {
00475     if (c >= '0' && c <= '9') {
00476         return true;
00477     } else {
00478         return false;
00479     }
00480 }
00481
00488 bool asm_isgraph(char c)
00489 {
00490     if (c >= 33 && c <= 126) {
00491         return true;
00492     } else {
00493         return false;
00494     }
00495 }
00496
00503 bool asm_islower(char c)
00504 {
00505     if (c >= 'a' && c <= 'z') {
00506         return true;
00507     } else {
00508         return false;
00509     }
00510 }
00511
00518 bool asm_isodigit(const char c)
00519 {
00520     if ((c >= '0' && c <= '7')) {
00521         return true;
00522     } else {
00523         return false;
00524     }
00525 }
00526
00534 bool asm_isprint(char c)
00535 {
00536     return asm_isgraph(c) || c == ' ';
00537 }
00538
00546 bool asm_ispunct(char c)

```

```

00547 {
00548     if ((c >= 33 && c <= 47) || (c >= 58 && c <= 64) || (c >= 91 && c <= 96) || (c >= 123 && c <=
126)) {
00549         return true;
00550     } else {
00551         return false;
00552     }
00553 }
00554
00562 bool asm_isspace(char c)
00563 {
00564     if (c == ' ' || c == '\n' || c == '\t' ||
00565         c == '\v' || c == '\f' || c == '\r') {
00566         return true;
00567     } else {
00568         return false;
00569     }
00570 }
00571
00578 bool asm_isupper(char c)
00579 {
00580     if (c >= 'A' && c <= 'Z') {
00581         return true;
00582     } else {
00583         return false;
00584     }
00585 }
00586
00593 bool asm_isxdigit(char c)
00594 {
00595     if ((c >= 'a' && c <= 'f') || asm_isdigit(c)) {
00596         return true;
00597     } else {
00598         return false;
00599     }
00600 }
00601
00608 bool asm_isXdigit(char c)
00609 {
00610     if ((c >= 'A' && c <= 'F') || asm_isdigit(c)) {
00611         return true;
00612     } else {
00613         return false;
00614     }
00615 }
00616
00627 size_t asm_length(const char * const str)
00628 {
00629     char c;
00630     size_t i = 0;
00631
00632     while ((c = str[i++]) != '\0') {
00633         if (i > ASM_MAX_LEN) {
00634             #ifndef ASM_NO_ERRORS
00635                 asm_dprintfERROR("%s", "index exceeds ASM_MAX_LEN. Probably no NULL termination.");
00636             #endif
00637             return SIZE_MAX;
00638         }
00639     }
00640     return --i;
00641 }
00642
00662 void * asm_memset(void * const des, const unsigned char value, const size_t n)
00663 {
00664     unsigned char *ptr = (unsigned char *)des;
00665     for (size_t i = n; i-- > 0;) {
00666         *ptr++ = value;
00667     }
00668     return des;
00669 }
00670
00685 void asm_pad_left(char * const s, const size_t padding, const char pad)
00686 {
00687     int len = (int)asm_length(s);
00688     for (int i = len; i >= 0; i--) {
00689         s[i+(int)padding] = s[i];
00690     }
00691     for (int i = 0; i < (int)padding; i++) {
00692         s[i] = pad;
00693     }
00694 }
00695
00702 void asm_print_many_times(const char * const str, const size_t n)
00703 {
00704     for (size_t i = 0; i < n; i++) {
00705         printf("%s", str);
00706     }

```

```

00707     printf("\n");
00708 }
00709
00722 void asm_remove_char_from_string(char * const s, const size_t index)
00723 {
00724     size_t len = asm_length(s);
00725     if (len == 0) return;
00726     if (index >= len) {
00727         #ifndef ASM_NO_ERRORS
00728             asm_dprintERROR("%s", "index exceeds array length.");
00729         #endif
00730         return;
00731     }
00732
00733     for (size_t i = index; i < len; i++) {
00734         s[i] = s[i+1];
00735     }
00736 }
00737
00751 void asm_shift_left(char * const s, const size_t shift)
00752 {
00753     size_t len = asm_length(s);
00754
00755     if (shift == 0) return;
00756     if (len <= shift) {
00757         s[0] = '\0';
00758         return;
00759     }
00760
00761     size_t i;
00762     for (i = shift; i < len; i++) {
00763         s[i-shift] = s[i];
00764     }
00765     s[i-shift] = '\0';
00766 }
00767
00782 int asm_str_in_str(const char * const src, const char * const word_to_search)
00783 {
00784     int i = 0, num_of_accur = 0;
00785     while (src[i] != '\0') {
00786         if (asm_strncmp(src+i, word_to_search, asm_length(word_to_search))) {
00787             num_of_accur++;
00788         }
00789         i++;
00790     }
00791     return num_of_accur;
00792 }
00793
00828 double asm_str2double(const char * const s, const char ** const end, const size_t base)
00829 {
00830     if (base < 2 || base > 36) {
00831         #ifndef ASM_NO_ERRORS
00832             asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
00833         #endif
00834         if (end) *end = s;
00835         return 0.0;
00836     }
00837     int num_of_whitespace = 0;
00838     while (asm_isspace(s[num_of_whitespace])) {
00839         num_of_whitespace++;
00840     }
00841
00842     int i = 0;
00843     if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
00844         i++;
00845     }
00846     int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;
00847
00848     size_t left = 0;
00849     double right = 0.0;
00850     int expo = 0;
00851     for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
00852         left = base * left + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
00853     }
00854
00855     if (s[i+num_of_whitespace] == '.') {
00856         i++; /* skip the point */
00857
00858         size_t divider = base;
00859         for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
00860             right = right + asm_get_char_value_in_base(s[i+num_of_whitespace], base) /
(double)divider;
00861             divider *= base;
00862         }
00863     }
00864
00865     if ((s[i+num_of_whitespace] == 'e') || (s[i+num_of_whitespace] == 'E')) {

```

```

00866         expo = asm_str2int(&(s[i+num_of_whitespace+1]), end, 10);
00867     } else {
00868         if (end) *end = s + i + num_of_whitespace;
00869     }
00870
00871     double res = sign * (left + right);
00872
00873     if (expo > 0) {
00874         for (int index = 0; index < expo; index++) {
00875             res *= (double)base;
00876         }
00877     } else {
00878         for (int index = 0; index > expo; index--) {
00879             res /= (double)base;
00880         }
00881     }
00882
00883     return res;
00884 }
00885
00918 float asm_str2float(const char * const s, const char ** const end, const size_t base)
00919 {
00920     if (base < 2 || base > 36) {
00921         #ifndef ASM_NO_ERRORS
00922             asm_dprintfERROR("Supported bases are [2...36]. Input: %zu", base);
00923         #endif
00924         if (end) *end = s;
00925         return 0.0f;
00926     }
00927     int num_of_whitespace = 0;
00928     while (asm_isspace(s[num_of_whitespace])) {
00929         num_of_whitespace++;
00930     }
00931
00932     int i = 0;
00933     if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
00934         i++;
00935     }
00936     int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;
00937
00938     int left = 0;
00939     float right = 0.0f;
00940     int expo = 0;
00941     for (; asm_check_char_belongs_to_base(s[i+num_of_whitespace], base); i++) {
00942         left = base * left + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
00943     }
00944
00945     if (s[i+num_of_whitespace] == '.') {
00946         i++; /* skip the point */
00947
00948         size_t divider = base;
00949         for (; asm_check_char_belongs_to_base(s[i+num_of_whitespace], base); i++) {
00950             right = right + asm_get_char_value_in_base(s[i+num_of_whitespace], base) / (float)divider;
00951             divider *= base;
00952         }
00953     }
00954
00955     if ((s[i+num_of_whitespace] == 'e') || (s[i+num_of_whitespace] == 'E')) {
00956         expo = asm_str2int(&(s[i+num_of_whitespace+1]), end, 10);
00957     } else {
00958         if (end) *end = s + i + num_of_whitespace;
00959     }
00960
00961     float res = sign * (left + right);
00962
00963     if (expo > 0) {
00964         for (int index = 0; index < expo; index++) {
00965             res *= (float)base;
00966         }
00967     } else {
00968         for (int index = 0; index > expo; index--) {
00969             res /= (float)base;
00970         }
00971     }
00972
00973     return res;
00974 }
00975
00996 int asm_str2int(const char * const s, const char ** const end, const size_t base)
00997 {
00998     if (base < 2 || base > 36) {
00999         #ifndef ASM_NO_ERRORS
01000             asm_dprintfERROR("Supported bases are [2...36]. Input: %zu", base);
01001         #endif
01002         if (end) *end = s;
01003         return 0;
01004     }

```

```

01005     int num_of_whitespace = 0;
01006     while (asm_isspace(s[num_of_whitespace])) {
01007         num_of_whitespace++;
01008     }
01009
01010     int n = 0, i = 0;
01011     if (s[0+num_of_whitespace] == '-' || s[0+num_of_whitespace] == '+') {
01012         i++;
01013     }
01014     int sign = s[0+num_of_whitespace] == '-' ? -1 : 1;
01015
01016     for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
01017         n = base * n + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
01018     }
01019
01020     if (end) *end = s + i+num_of_whitespace;
01021
01022     return n * sign;
01023 }
01024
01041 size_t asm_str2size_t(const char * const s, const char ** const end, const size_t base)
01042 {
01043     if (end) *end = s;
01044
01045     int num_of_whitespace = 0;
01046     while (asm_isspace(s[num_of_whitespace])) {
01047         num_of_whitespace++;
01048     }
01049
01050     if (s[0+num_of_whitespace] == '-') {
01051         #ifndef ASM_NO_ERRORS
01052         asm_dprintERROR("%s", "Unable to convert a negative number to size_t.");
01053         #endif
01054         return 0;
01055     }
01056
01057     if (base < 2 || base > 36) {
01058         #ifndef ASM_NO_ERRORS
01059         asm_dprintERROR("Supported bases are [2...36]. Input: %zu", base);
01060         #endif
01061         if (end) *end = s+num_of_whitespace;
01062         return 0;
01063     }
01064
01065     size_t n = 0, i = 0;
01066     if (s[0+num_of_whitespace] == '+') {
01067         i++;
01068     }
01069
01070     for (; asm_check_char_belong_to_base(s[i+num_of_whitespace], base); i++) {
01071         n = base * n + asm_get_char_value_in_base(s[i+num_of_whitespace], base);
01072     }
01073
01074     if (end) *end = s + i+num_of_whitespace;
01075
01076     return n;
01077 }
01078
01088 void asm_strip_whitespace(char * const s)
01089 {
01090     size_t len = asm_length(s);
01091     size_t i;
01092     for (i = 0; i < len; i++) {
01093         if (asm_isspace(s[i])) {
01094             asm_remove_char_from_string(s, i);
01095             len--;
01096             i--;
01097         }
01098     }
01099     s[i] = '\0';
01100 }
01101
01109 bool asm_str_is_whitespace(const char * const s)
01110 {
01111     size_t len = asm_length(s);
01112     for (size_t i = 0; i < len; i++) {
01113         if (!asm_isspace(s[i])) {
01114             return false;
01115         }
01116     }
01117
01118     return true;
01119 }
01120
01137 char * asm_strdup(const char * const s, size_t length)
01138 {
01139     char * res = (char *)ASM_MALLOC(sizeof(char) * length+1);

```

```

01140     asm_strncpy((char * const)res, s, length);
01141
01142     return res;
01143 }
01144
01166 int asm_strncat(char * const s1, const char * const s2, const size_t N)
01167 {
01168     size_t len_s1 = asm_length(s1);
01169
01170     int limit = N;
01171     if (limit == 0) {
01172         limit = ASM_MAX_LEN;
01173     }
01174
01175     int i = 0;
01176     while (i < limit && s2[i] != '\0') {
01177         if (len_s1 + (size_t)i >= ASM_MAX_LEN-1) {
01178             #ifndef ASM_NO_ERRORS
01179                 asm_dprintERROR("s2 or the first N=%zu digit of s2 does not fit into s1.", N);
01180             #endif
01181             return i;
01182         }
01183
01184         s1[len_s1+(size_t)i] = s2[i];
01185         i++;
01186     }
01187     s1[len_s1+(size_t)i] = '\0';
01188
01189     return i;
01190 }
01191
01211 int asm_strncmp(const char *s1, const char *s2, const size_t N)
01212 {
01213     size_t n = N == 0 ? ASM_MAX_LEN : N;
01214     size_t i = 0;
01215     while (i < n) {
01216         if (s1[i] == '\0' && s2[i] == '\0') {
01217             break;
01218         }
01219         if (s1[i] != s2[i] || (s1[i] == '\0') || (s2[i] == '\0')) {
01220             return 0;
01221         }
01222         i++;
01223     }
01224     return 1;
01225 }
01226
01244 int asm_strncpy(char * const s1, const char * const s2, const size_t N)
01245 {
01246     size_t n = N;
01247
01248     size_t i;
01249     for (i = 0; i < n && s2[i] != '\0'; i++) {
01250         s1[i] = s2[i];
01251     }
01252     s1[i] = '\0';
01253
01254     return i;
01255 }
01256
01262 void asm_tolower(char * const s)
01263 {
01264     size_t len = asm_length(s);
01265     for (size_t i = 0; i < len; i++) {
01266         if (asm_isupper(s[i])) {
01267             s[i] += 'a' - 'A';
01268         }
01269     }
01270 }
01271
01277 void asm_toupper(char * const s)
01278 {
01279     size_t len = asm_length(s);
01280     for (size_t i = 0; i < len; i++) {
01281         if (asm_islower(s[i])) {
01282             s[i] += 'A' - 'a';
01283         }
01284     }
01285 }
01286
01295 void asm_trim_left_whitespace(char * const s)
01296 {
01297     size_t len = asm_length(s);
01298
01299     if (len == 0) return;
01300     size_t i;
01301     for (i = 0; i < len; i++) {

```

```

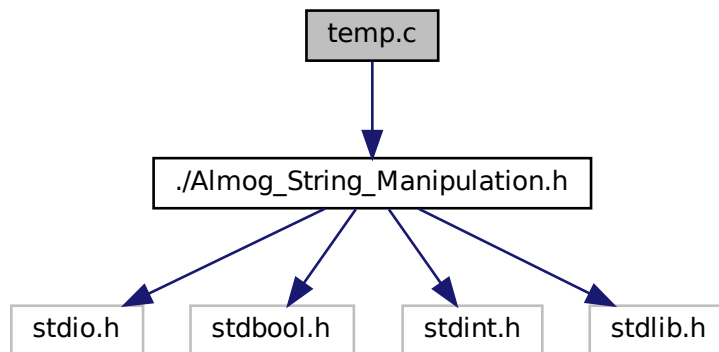
01302         if (!asm_isspace(s[i])) {
01303             break;
01304         }
01305     }
01306     asm_shift_left(s, i);
01307 }
01308
01309 #ifdef ASM_NO_ERRORS
01310 #undef ASM_NO_ERRORS
01311 #endif
01312
01313 #endif /*ALMOG_STRING_MANIPULATION_IMPLEMENTATION*/
01314

```

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\(\)](#).

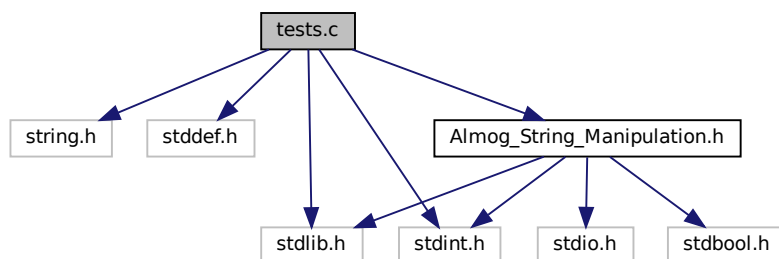
2.4 temp.c

```
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_properties\(\)](#), [test_strncat_current_behavior_and_sentinel\(\)](#), and [test_strncmp_boolean_edges\(\)](#).

2.5.2.4 rand_ascii_printable()

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

Definition at line 68 of file [tests.c](#).

References [xorshift32\(\)](#).

Referenced by [test_case_conversion_roundtrip\(\)](#), [test_length_matches_strlen_small\(\)](#), and [test_strip_whitespace_properties\(\)](#).

2.5.2.5 test_ascii_classification_exhaustive_ranges()

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

Definition at line 82 of file [tests.c](#).

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

Referenced by [main\(\)](#).

2.5.2.6 test_ascii_classification_full_scan_0_127()

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

Definition at line 153 of file [tests.c](#).

References [asm_isalnum\(\)](#), [asm_isalpha\(\)](#), [asm_isdigit\(\)](#), [asm_isgraph\(\)](#), [asm_islower\(\)](#), [asm_isprint\(\)](#), [asm_isupper\(\)](#), and [TEST_CASE](#).

Referenced by [main\(\)](#).

2.5.2.7 test_base_digit_helpers()

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

Definition at line 460 of file [tests.c](#).

References [asm_check_char_belong_to_base\(\)](#), [asm_get_char_value_in_base\(\)](#), [TEST_CASE](#), and [TEST_EQ_INT](#).

Referenced by [main\(\)](#).

2.5.2.8 test_case_conversion_roundtrip()

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

Definition at line 181 of file [tests.c](#).

References [asm_tolower\(\)](#), [asm_toupper\(\)](#), [is_nul_terminated_within\(\)](#), [rand_ascii_printable\(\)](#), [TEST_CASE](#), [TEST_EQ_STR](#), and [xorshift32\(\)](#).

Referenced by [main\(\)](#).

2.5.2.9 test_copy_array_by_indexes_behavior_and_bounds()

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

Definition at line 257 of file [tests.c](#).

References [asm_copy_array_by_indexes\(\)](#), [fill_sentinel\(\)](#), [TEST_CASE](#), and [TEST_EQ_STR](#).

Referenced by [main\(\)](#).

2.5.2.10 test_get_line_tmpfile()

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

Definition at line 689 of file [tests.c](#).

References [asm_get_line\(\)](#), [ASM_MAX_LEN](#), [g_tests_warned](#), [is_nul_terminated_within\(\)](#), [TEST_CASE](#), [TEST_EQ_INT](#), and [TEST_EQ_STR](#).

Referenced by [main\(\)](#).

2.5.2.11 test_get_line_too_long()

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

Definition at line 733 of file [tests.c](#).

References [asm_get_line\(\)](#), [ASM_MAX_LEN](#), [fill_sentinel\(\)](#), [g_tests_warned](#), and [TEST_EQ_INT](#).

Referenced by [main\(\)](#).

2.5.2.12 test_get_next_word_from_line_current_behavior()

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

Definition at line 606 of file [tests.c](#).

References [asm_get_next_token_from_str\(\)](#), [TEST_CASE](#), [TEST_EQ_INT](#), and [TEST_EQ_STR](#).

Referenced by [main\(\)](#).

2.5.2.13 test_get_word_and_cut_edges()

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

Definition at line 651 of file [tests.c](#).

References [asm_get_token_and_cut\(\)](#), [TEST_CASE](#), and [TEST_EQ_STR](#).

Referenced by [main\(\)](#).

2.5.2.14 test_left_pad_edges_and_sentinel()

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

Definition at line 319 of file [tests.c](#).

References [asm_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_isctrl('\0'));
00133     TEST_CASE(asm_isctrl('\n'));
00134     TEST_CASE(asm_isctrl(127));
00135     TEST_CASE(!asm_isctrl('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 }

```

```

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_from_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 sl[16];
00771         unsigned char post[8];
00772     } box;
00773
00774     fill_sentinel(box.pre, sizeof(box.pre), 0xAA);
00775     fill_sentinel((unsigned char *)box.sl, sizeof(box.sl), 0xCC);
00776     fill_sentinel(box.post, sizeof(box.post), 0xBB);
00777
00778     strcpy(box.sl, "abc");
00779
00780     int n = asm_strncat(box.sl, "DEF", 3);
00781     TEST_EQ_INT(n, 3);
00782
00783     TEST_EQ_STR(box.sl, "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 > 0.0000000009 && vd < 0.0000000011);
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_from_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

Almog_String_Manipulation.h, [3](#)

asm_check_char_belong_to_base, [10](#)

asm_copy_array_by_indexes, [11](#)

asm_dprintCHAR, [6](#)

asm_dprintDOUBLE, [7](#)

asm_dprintERROR, [7](#)

asm_dprintFLOAT, [7](#)

asm_dprintINT, [8](#)

asm_dprintSIZE_T, [8](#)

asm_dprintSTRING, [8](#)

asm_get_char_value_in_base, [11](#)

asm_get_line, [12](#)

asm_get_next_token_from_str, [13](#)

asm_get_token_and_cut, [14](#)

asm_isalnum, [15](#)

asm_isalpha, [15](#)

asm_isbdigit, [16](#)

asm_iscntrl, [16](#)

asm_isdigit, [16](#)

asm_isgraph, [17](#)

asm_islower, [17](#)

asm_isodigit, [18](#)

asm_isprint, [18](#)

asm_ispunct, [19](#)

asm_isspace, [19](#)

asm_isupper, [19](#)

asm_isXdigit, [20](#)

asm_isxdigit, [20](#)

asm_length, [21](#)

ASM_MALLOC, [8](#)

asm_max, [9](#)

ASM_MAX_LEN, [9](#)

asm_memset, [21](#)

asm_min, [9](#)

asm_pad_left, [23](#)

asm_print_many_times, [23](#)

asm_remove_char_from_string, [24](#)

asm_shift_left, [24](#)

asm_str2double, [25](#)

asm_str2float, [26](#)

asm_str2int, [27](#)

asm_str2size_t, [27](#)

asm_str_in_str, [28](#)

asm_str_is_whitespace, [29](#)

asm_strdup, [29](#)

asm_strip_whitespace, [30](#)

asm_strncat, [30](#)

asm_strncmp, [31](#)

asm_strncpy, [32](#)

asm_tolower, [32](#)

asm_toupper, [33](#)

asm_trim_left_whitespace, [33](#)

ALMOG_STRING_MANIPULATION_IMPLEMENTATION

temp.c, [42](#)

tests.c, [45](#)

asm_check_char_belong_to_base

Almog_String_Manipulation.h, [10](#)

asm_copy_array_by_indexes

Almog_String_Manipulation.h, [11](#)

asm_dprintCHAR

Almog_String_Manipulation.h, [6](#)

asm_dprintDOUBLE

Almog_String_Manipulation.h, [7](#)

asm_dprintERROR

Almog_String_Manipulation.h, [7](#)

asm_dprintFLOAT

Almog_String_Manipulation.h, [7](#)

asm_dprintINT

Almog_String_Manipulation.h, [8](#)

asm_dprintSIZE_T

Almog_String_Manipulation.h, [8](#)

asm_dprintSTRING

Almog_String_Manipulation.h, [8](#)

asm_get_char_value_in_base

Almog_String_Manipulation.h, [11](#)

asm_get_line

Almog_String_Manipulation.h, [12](#)

asm_get_next_token_from_str

Almog_String_Manipulation.h, [13](#)

asm_get_token_and_cut

Almog_String_Manipulation.h, [14](#)

asm_isalnum

Almog_String_Manipulation.h, [15](#)

asm_isalpha

Almog_String_Manipulation.h, [15](#)

asm_isbdigit

Almog_String_Manipulation.h, [16](#)

asm_iscntrl

Almog_String_Manipulation.h, [16](#)

asm_isdigit

Almog_String_Manipulation.h, [16](#)

asm_isgraph

Almog_String_Manipulation.h, [17](#)

asm_islower

Almog_String_Manipulation.h, [17](#)

asm_isodigit

Almog_String_Manipulation.h, [18](#)

asm_isprint

- Almog_String_Manipulation.h, [18](#)
- asm_ispunct
 - Almog_String_Manipulation.h, [19](#)
- asm_isspace
 - Almog_String_Manipulation.h, [19](#)
- asm_isupper
 - Almog_String_Manipulation.h, [19](#)
- asm_isXdigit
 - Almog_String_Manipulation.h, [20](#)
- asm_isxdigit
 - Almog_String_Manipulation.h, [20](#)
- asm_length
 - Almog_String_Manipulation.h, [21](#)
- ASM_MALLOC
 - Almog_String_Manipulation.h, [8](#)
- asm_max
 - Almog_String_Manipulation.h, [9](#)
- ASM_MAX_LEN
 - Almog_String_Manipulation.h, [9](#)
- asm_memset
 - Almog_String_Manipulation.h, [21](#)
- asm_min
 - Almog_String_Manipulation.h, [9](#)
- asm_pad_left
 - Almog_String_Manipulation.h, [23](#)
- asm_print_many_times
 - Almog_String_Manipulation.h, [23](#)
- asm_remove_char_from_string
 - Almog_String_Manipulation.h, [24](#)
- asm_shift_left
 - Almog_String_Manipulation.h, [24](#)
- asm_str2double
 - Almog_String_Manipulation.h, [25](#)
- asm_str2float
 - Almog_String_Manipulation.h, [26](#)
- asm_str2int
 - Almog_String_Manipulation.h, [27](#)
- asm_str2size_t
 - Almog_String_Manipulation.h, [27](#)
- asm_str_in_str
 - Almog_String_Manipulation.h, [28](#)
- asm_str_is_whitespace
 - Almog_String_Manipulation.h, [29](#)
- asm_strdup
 - Almog_String_Manipulation.h, [29](#)
- asm_strip_whitespace
 - Almog_String_Manipulation.h, [30](#)
- asm_strncat
 - Almog_String_Manipulation.h, [30](#)
- asm_strncmp
 - Almog_String_Manipulation.h, [31](#)
- asm_strncpy
 - Almog_String_Manipulation.h, [32](#)
- asm_tolower
 - Almog_String_Manipulation.h, [32](#)
- asm_toupper
 - Almog_String_Manipulation.h, [33](#)
- asm_trim_left_whitespace
- Almog_String_Manipulation.h, [33](#)
- fill_sentinel
 - tests.c, [46](#)
- g_tests_failed
 - tests.c, [56](#)
- g_tests_run
 - tests.c, [56](#)
- g_tests_warned
 - tests.c, [56](#)
- is_nul_terminated_within
 - tests.c, [47](#)
- main
 - temp.c, [43](#)
 - tests.c, [47](#)
- NO_ERRORS
 - tests.c, [45](#)
- rand_ascii_printable
 - tests.c, [47](#)
- rng_state
 - tests.c, [56](#)
- temp.c, [42](#)
- ALMOG_STRING_MANIPULATION_IMPLEMENTATION, [42](#)
- main, [43](#)
- test_ascii_classification_exhaustive_ranges
 - tests.c, [47](#)
- test_ascii_classification_full_scan_0_127
 - tests.c, [48](#)
- test_base_digit_helpers
 - tests.c, [48](#)
- TEST_CASE
 - tests.c, [45](#)
- test_case_conversion_roundtrip
 - tests.c, [48](#)
- test_copy_array_by_indexes_behavior_and_bounds
 - tests.c, [48](#)
- TEST_EQ_INT
 - tests.c, [45](#)
- TEST_EQ_SIZE
 - tests.c, [45](#)
- TEST_EQ_STR
 - tests.c, [46](#)
- test_get_line_tmpfile
 - tests.c, [49](#)
- test_get_line_too_long
 - tests.c, [49](#)
- test_get_next_word_from_line_current_behavior
 - tests.c, [49](#)
- test_get_word_and_cut_edges
 - tests.c, [49](#)
- test_left_pad_edges_and_sentinel
 - tests.c, [50](#)
- test_left_shift_edges

- tests.c, [50](#)
- test_length_matches_strlen_small
 - tests.c, [50](#)
- test_memset_basic_and_edges
 - tests.c, [50](#)
- TEST_NE_STR
 - tests.c, [46](#)
- test_remove_char_form_string_edges
 - tests.c, [51](#)
- test_str2double_exponent_basic
 - tests.c, [51](#)
- test_str2double_exponent_edge_cases
 - tests.c, [51](#)
- test_str2double_exponent_signed_mantissa
 - tests.c, [51](#)
- test_str2float_double
 - tests.c, [52](#)
- test_str2float_double_exponent_different_bases
 - tests.c, [52](#)
- test_str2float_double_exponent_large_values
 - tests.c, [52](#)
- test_str2float_double_exponent_whitespace
 - tests.c, [52](#)
- test_str2float_exponent_basic
 - tests.c, [53](#)
- test_str2float_exponent_edge_cases
 - tests.c, [53](#)
- test_str2float_exponent_signed_mantissa
 - tests.c, [53](#)
- test_str2float_exponent_with_trailing
 - tests.c, [53](#)
- test_str2int
 - tests.c, [54](#)
- test_str2size_t
 - tests.c, [54](#)
- test_str_in_str_overlap_and_edges
 - tests.c, [54](#)
- test_str_is_whitespace_edges
 - tests.c, [54](#)
- test_strip_whitespace_properties
 - tests.c, [55](#)
- test_strncat_current_behavior_and_sentinel
 - tests.c, [55](#)
- test_strncmp_boolean_edges
 - tests.c, [55](#)
- TEST_WARN
 - tests.c, [46](#)
- tests.c, [43](#)
 - ALMOG_STRING_MANIPULATION_IMPLEMENTATION,
 - [45](#)
 - fill_sentinel, [46](#)
 - g_tests_failed, [56](#)
 - g_tests_run, [56](#)
 - g_tests_warned, [56](#)
 - is_nul_terminated_within, [47](#)
 - main, [47](#)
 - NO_ERRORS, [45](#)
 - rand_ascii_printable, [47](#)
 - rng_state, [56](#)
 - test_ascii_classification_exhaustive_ranges, [47](#)
 - test_ascii_classification_full_scan_0_127, [48](#)
 - test_base_digit_helpers, [48](#)
 - TEST_CASE, [45](#)
 - test_case_conversion_roundtrip, [48](#)
 - test_copy_array_by_indexes_behavior_and_bounds,
 - [48](#)
 - TEST_EQ_INT, [45](#)
 - TEST_EQ_SIZE, [45](#)
 - TEST_EQ_STR, [46](#)
 - test_get_line_tmpfile, [49](#)
 - test_get_line_too_long, [49](#)
 - test_get_next_word_from_line_current_behavior,
 - [49](#)
 - test_get_word_and_cut_edges, [49](#)
 - test_left_pad_edges_and_sentinel, [50](#)
 - test_left_shift_edges, [50](#)
 - test_length_matches_strlen_small, [50](#)
 - test_memset_basic_and_edges, [50](#)
 - TEST_NE_STR, [46](#)
 - test_remove_char_form_string_edges, [51](#)
 - test_str2double_exponent_basic, [51](#)
 - test_str2double_exponent_edge_cases, [51](#)
 - test_str2double_exponent_signed_mantissa, [51](#)
 - test_str2float_double, [52](#)
 - test_str2float_double_exponent_different_bases,
 - [52](#)
 - test_str2float_double_exponent_large_values, [52](#)
 - test_str2float_double_exponent_whitespace, [52](#)
 - test_str2float_exponent_basic, [53](#)
 - test_str2float_exponent_edge_cases, [53](#)
 - test_str2float_exponent_signed_mantissa, [53](#)
 - test_str2float_exponent_with_trailing, [53](#)
 - test_str2int, [54](#)
 - test_str2size_t, [54](#)
 - test_str_in_str_overlap_and_edges, [54](#)
 - test_str_is_whitespace_edges, [54](#)
 - test_strip_whitespace_properties, [55](#)
 - test_strncat_current_behavior_and_sentinel, [55](#)
 - test_strncmp_boolean_edges, [55](#)
 - TEST_WARN, [46](#)
 - xorshift32, [55](#)
- xorshift32
 - tests.c, [55](#)