NAME

cmark - CommonMark parsing, manipulating, and rendering

DESCRIPTION

Simple Interface

char * **cmark_markdown_to_html**(*const char *text*, *size_t len*, *int options*)

Convert *text* (assumed to be a UTF-8 encoded string with length *len*) from CommonMark Markdown to HTML, returning a null-terminated, UTF-8-encoded string. It is the caller's responsibility to free the returned buffer.

Node Structure

```
typedef enum {
  /* Error status */
  CMARK_NODE_NONE,

  /* Block */
  CMARK_NODE_DOCUMENT,
  CMARK_NODE_BLOCK_QUOTE,
  CMARK_NODE_LIST,
  CMARK_NODE_ITEM,
  CMARK_NODE_CODE_BLOCK,
  CMARK_NODE_HTML_BLOCK,
  CMARK_NODE_CUSTOM_BLOCK,
  CMARK_NODE_PARAGRAPH,
  CMARK_NODE_HEADING,
  CMARK_NODE_THEMATIC_BREAK,

  CMARK_NODE_FIRST_BLOCK = CMARK_NODE_DOCUMENT,
  CMARK_NODE_LAST_BLOCK = CMARK_NODE_THEMATIC_BREAK,

  /* Inline */
  CMARK_NODE_TEXT,
  CMARK_NODE_SOFTBREAK,
  CMARK_NODE_LINEBREAK,
  CMARK_NODE_CODE,
  CMARK_NODE_HTML_INLINE,
  CMARK_NODE_CUSTOM_INLINE,
  CMARK_NODE_EMPH,
  CMARK_NODE_STRONG,
  CMARK_NODE_LINK,
  CMARK_NODE_IMAGE,

  CMARK_NODE_FIRST_INLINE = CMARK_NODE_TEXT,
  CMARK_NODE_LAST_INLINE = CMARK_NODE_IMAGE
} cmark_node_type;
```

```
typedef enum {
  CMARK_NO_LIST,
  CMARK_BULLET_LIST,
  CMARK_ORDERED_LIST
} cmark_list_type;
```

```
typedef enum {
  CMARK_NO_DELIM,
  CMARK_PERIOD_DELIM,
  CMARK_PAREN_DELIM
} cmark_delim_type;
```

Custom memory allocator support

```
typedef struct cmark_mem {
  void *(*calloc)(size_t, size_t);
  void *(*realloc)(void *, size_t);
  void (*free)(void *);
} cmark_mem;
```

Defines the memory allocation functions to be used by CMark when parsing and allocating a document tree

cmark_mem * **cmark_get_default_mem_allocator**()

Returns a pointer to the default memory allocator.

Creating and Destroying Nodes

cmark_node * **cmark_node_new**(*cmark_node_type type*)

Creates a new node of type *type*. Note that the node may have other required properties, which it is the caller's responsibility to assign.

cmark_node * **cmark_node_new_with_mem**(*cmark_node_type type*, *cmark_mem *mem*)

Same as cmark_node_new, but explicitly listing the memory allocator used to allocate the node. Note: be sure to use the same allocator for every node in a tree, or bad things can happen.

void **cmark_node_free**(*cmark_node *node*)

Frees the memory allocated for a node and any children.

Tree Traversal

cmark_node * **cmark_node_next**(*cmark_node *node*)

Returns the next node in the sequence after *node*, or NULL if there is none.

cmark_node * **cmark_node_previous**(*cmark_node *node*)

Returns the previous node in the sequence after *node*, or NULL if there is none.

cmark_node * **cmark_node_parent**(*cmark_node *node*)

Returns the parent of *node*, or NULL if there is none.

cmark_node * **cmark_node_first_child**(*cmark_node *node*)

Returns the first child of *node*, or NULL if *node* has no children.

cmark_node * **cmark_node_last_child**(*cmark_node *node*)

Returns the last child of *node*, or NULL if *node* has no children.

Iterator

An iterator will walk through a tree of nodes, starting from a root node, returning one node at a time, together with information about whether the node is being entered or exited. The iterator will first descend to a child node, if there is one. When there is no child, the iterator will go to the next sibling. When there is no next sibling, the iterator will return to the parent (but with a *cmark_event_type* of CMARK_EVENT_EXIT*). The iterator will return CMARK_EVENT_DONE* when it reaches the root node again. One natural application is an HTML renderer, where an ENTER *event outputs an open tag and an EXIT* event outputs a close tag. An iterator might also be used to transform an AST in some systematic way, for example, turning all level-3 headings into regular paragraphs.

```
void
usage_example(cmark_node *root) {
    cmark_event_type ev_type;
    cmark_iter *iter = cmark_iter_new(root);

    while ((ev_type = cmark_iter_next(iter)) != CMARK_EVENT_DONE) {
      cmark_node *cur = cmark_iter_get_node(iter);
      // Do something with ‘cur‘ and ‘ev_type‘
    }

    cmark_iter_free(iter);
  }
```

Iterators will never return EXIT events for leaf nodes, which are nodes of type:

- CMARK_NODE_HTML_BLOCK
- CMARK_NODE_THEMATIC_BREAK
- CMARK_NODE_CODE_BLOCK
- CMARK_NODE_TEXT
- CMARK_NODE_SOFTBREAK
- CMARK_NODE_LINEBREAK
- CMARK_NODE_CODE
- CMARK_NODE_HTML_INLINE

Nodes must only be modified after an EXIT event, or an ENTER event for leaf nodes.

```
typedef enum {
  CMARK_EVENT_NONE,
  CMARK_EVENT_DONE,
  CMARK_EVENT_ENTER,
  CMARK_EVENT_EXIT
} cmark_event_type;
```

cmark_iter * **cmark_iter_new**(*cmark_node *root*)

Creates a new iterator starting at *root*. The current node and event type are undefined until *cmark_iter_next* is called for the first time. The memory allocated for the iterator should be released using *cmark_iter_free* when it is no longer needed.

void **cmark_iter_free**(*cmark_iter *iter*)

Frees the memory allocated for an iterator.

cmark_event_type **cmark_iter_next**(*cmark_iter *iter*)

Advances to the next node and returns the event type (CMARK_EVENT_ENTER, CMARK_EVENT_EXIT or CMARK_EVENT_DONE).

cmark_node * **cmark_iter_get_node**(*cmark_iter *iter*)

Returns the current node.

cmark_event_type **cmark_iter_get_event_type**(*cmark_iter *iter*)

Returns the current event type.

cmark_node * **cmark_iter_get_root**(*cmark_iter *iter*)

Returns the root node.

void **cmark_iter_reset**(*cmark_iter *iter*, *cmark_node *current*, *cmark_event_type event_type*)

Resets the iterator so that the current node is *current* and the event type is *event_type*. The new current node must be a descendant of the root node or the root node itself.

Accessors

void * **cmark_node_get_user_data**(*cmark_node *node*)

Returns the user data of *node*.

int **cmark_node_set_user_data**(*cmark_node *node*, *void *user_data*)

Sets arbitrary user data for *node*. Returns 1 on success, 0 on failure.

cmark_node_type **cmark_node_get_type**(*cmark_node *node*)

Returns the type of *node*, or CMARK_NODE_NONE *on error.*

const char * **cmark_node_get_type_string**(*cmark_node *node*)

Like *cmark_node_get_type*, but returns a string representation of the type, or "<unknown>".

const char * **cmark_node_get_literal**(*cmark_node *node*)

Returns the string contents of *node*, or an empty string if none is set. Returns NULL if called on a node that does not have string content.

int **cmark_node_set_literal**(*cmark_node *node*, *const char *content*)

Sets the string contents of *node*. Returns 1 on success, 0 on failure.

int **cmark_node_get_heading_level**(*cmark_node *node*)

Returns the heading level of *node,* or 0 if *node* is not a heading.

int **cmark_node_set_heading_level**(*cmark_node *node*, *int level*)

Sets the heading level of *node*, returning 1 on success and 0 on error.

cmark_list_type **cmark_node_get_list_type**(*cmark_node *node*)

Returns the list type of *node*, or CMARK_NO_LIST *if node is not a list.*

int **cmark_node_set_list_type**(*cmark_node *node*, *cmark_list_type type*)

Sets the list type of *node,* returning 1 on success and 0 on error.

cmark_delim_type **cmark_node_get_list_delim**(*cmark_node *node*)

Returns the list delimiter type of *node*, or CMARK_NO_DELIM *if node is not a list.*

int **cmark_node_set_list_delim**(*cmark_node *node*, *cmark_delim_type delim*)

Sets the list delimiter type of *node*, returning 1 on success and 0 on error.

int **cmark_node_get_list_start**(*cmark_node *node*)

Returns starting number of *node*, if it is an ordered list, otherwise 0.

int **cmark_node_set_list_start**(*cmark_node *node*, *int start*)

Sets starting number of *node*, if it is an ordered list. Returns 1 on success, 0 on failure.

int **cmark_node_get_list_tight**(*cmark_node *node*)

Returns 1 if *node* is a tight list, 0 otherwise.

int **cmark_node_set_list_tight**(*cmark_node *node*, *int tight*)

Sets the "tightness" of a list. Returns 1 on success, 0 on failure.

*const char ** **cmark_node_get_fence_info**(*cmark_node *node*)

Returns the info string from a fenced code block.

int **cmark_node_set_fence_info**(*cmark_node *node*, *const char *info*)

Sets the info string in a fenced code block, returning 1 on success and 0 on failure.

*const char ** **cmark_node_get_url**(*cmark_node *node*)

Returns the URL of a link or image *node*, or an empty string if no URL is set. Returns NULL if called on a node that is not a link or image.

int **cmark_node_set_url**(*cmark_node *node, const char *url*)

Sets the URL of a link or image *node*. Returns 1 on success, 0 on failure.

*const char ** **cmark_node_get_title**(*cmark_node *node*)

Returns the title of a link or image *node*, or an empty string if no title is set. Returns NULL if called on a node that is not a link or image.

int **cmark_node_set_title**(*cmark_node *node, const char *title*)

Sets the title of a link or image *node*. Returns 1 on success, 0 on failure.

*const char ** **cmark_node_get_on_enter**(*cmark_node *node*)

Returns the literal "on enter" text for a custom *node*, or an empty string if no on_enter is set. Returns NULL if called on a non-custom node.

int **cmark_node_set_on_enter**(*cmark_node *node, const char *on_enter*)

Sets the literal text to render "on enter" for a custom *node*. Any children of the node will be rendered after this text. Returns 1 on success 0 on failure.

*const char ** **cmark_node_get_on_exit**(*cmark_node *node*)

Returns the literal "on exit" text for a custom *node*, or an empty string if no on_exit is set. Returns NULL if called on a non-custom node.

int **cmark_node_set_on_exit**(*cmark_node *node, const char *on_exit*)

Sets the literal text to render "on exit" for a custom *node*. Any children of the node will be rendered before this text. Returns 1 on success 0 on failure.

int **cmark_node_get_start_line**(*cmark_node *node*)

Returns the line on which *node* begins.

int **cmark_node_get_start_column**(*cmark_node *node*)

Returns the column at which *node* begins.

int **cmark_node_get_end_line**(*cmark_node *node*)

Returns the line on which *node* ends.

int **cmark_node_get_end_column**(*cmark_node *node*)

Returns the column at which *node* ends.

Tree Manipulation

void **cmark_node_unlink**(*cmark_node *node*)

Unlinks a *node*, removing it from the tree, but not freeing its memory. (Use *cmark_node_free* for that.)

int **cmark_node_insert_before**(*cmark_node *node*, *cmark_node *sibling*)

Inserts *sibling* before *node*. Returns 1 on success, 0 on failure.

int **cmark_node_insert_after**(*cmark_node *node*, *cmark_node *sibling*)

Inserts *sibling* after *node*. Returns 1 on success, 0 on failure.

int **cmark_node_replace**(*cmark_node *oldnode*, *cmark_node *newnode*)

Replaces *oldnode* with *newnode* and unlinks *oldnode* (but does not free its memory). Returns 1 on success, 0 on failure.

int **cmark_node_prepend_child**(*cmark_node *node*, *cmark_node *child*)

Adds *child* to the beginning of the children of *node*. Returns 1 on success, 0 on failure.

int **cmark_node_append_child**(*cmark_node *node*, *cmark_node *child*)

Adds *child* to the end of the children of *node*. Returns 1 on success, 0 on failure.

void **cmark_consolidate_text_nodes**(*cmark_node *root*)

Consolidates adjacent text nodes.

Parsing

Simple interface:

```
cmark_node *document = cmark_parse_document("Hello *world*", 13,
                                            CMARK_OPT_DEFAULT);
```

Streaming interface:

```
cmark_parser *parser = cmark_parser_new(CMARK_OPT_DEFAULT);
FILE *fp = fopen("myfile.md", "rb");
while ((bytes = fread(buffer, 1, sizeof(buffer), fp)) > 0) {
    cmark_parser_feed(parser, buffer, bytes);
    if (bytes < sizeof(buffer)) {
        break;
    }
}
document = cmark_parser_finish(parser);
cmark_parser_free(parser);
```

cmark_parser * **cmark_parser_new**(*int options*)

Creates a new parser object.

cmark_parser * **cmark_parser_new_with_mem**(*int options*, *cmark_mem *mem*)

Creates a new parser object with the given memory allocator

void **cmark_parser_free**(*cmark_parser *parser*)

Frees memory allocated for a parser object.

void **cmark_parser_feed**(*cmark_parser *parser*, *const char *buffer*, *size_t len*)

Feeds a string of length *len* to *parser*.

cmark_node * **cmark_parser_finish**(*cmark_parser *parser*)

Finish parsing and return a pointer to a tree of nodes.

cmark_node * **cmark_parse_document**(*const char *buffer*, *size_t len*, *int options*)

Parse a CommonMark document in *buffer* of length *len*. Returns a pointer to a tree of nodes. The memory allocated for the node tree should be released using *cmark_node_free* when it is no longer needed.

cmark_node * **cmark_parse_file**(*FILE *f*, *int options*)

Parse a CommonMark document in file *f*, returning a pointer to a tree of nodes. The memory allocated for the node tree should be released using *cmark_node_free* when it is no longer needed.

Rendering

char * **cmark_render_xml**(*cmark_node *root*, *int options*)

Render a *node* tree as XML. It is the caller's responsibility to free the returned buffer.

char * **cmark_render_html**(*cmark_node *root*, *int options*)

Render a *node* tree as an HTML fragment. It is up to the user to add an appropriate header and footer. It is the caller's responsibility to free the returned buffer.

char * **cmark_render_man**(*cmark_node *root*, *int options*, *int width*)

Render a *node* tree as a groff man page, without the header. It is the caller's responsibility to free the returned buffer.

char * **cmark_render_commonmark**(*cmark_node *root*, *int options*, *int width*)

Render a *node* tree as a commonmark document. It is the caller’s responsibility to free the returned buffer.

char * **cmark_render_latex**(*cmark_node *root*, *int options*, *int width*)

Render a *node* tree as a LaTeX document. It is the caller’s responsibility to free the returned buffer.

Options

#define CMARK_OPT_DEFAULT 0

Default options.

Options affecting rendering

#define CMARK_OPT_SOURCEPOS (1 << 1)

Include a data-sourcepos attribute on all block elements.

#define CMARK_OPT_HARDBREAKS (1 << 2)

Render softbreak elements as hard line breaks.

#define CMARK_OPT_SAFE (1 << 3)

CMARK_OPT_SAFE is defined here for API compatibility, but it no longer has any effect. "Safe" mode is now the default: set CMARK_OPT_UNSAFE to disable it.

#define CMARK_OPT_UNSAFE (1 << 17)

Render raw HTML and unsafe links (javascript:, vbscript:, file:, and data:, except for image/png, image/gif, image/jpeg, or image/webp mime types). By default, raw HTML is replaced by a placeholder HTML comment. Unsafe links are replaced by empty strings.

#define CMARK_OPT_NOBREAKS (1 << 4)

Render softbreak elements as spaces.

Options affecting parsing

#define CMARK_OPT_NORMALIZE (1 << 8)

Legacy option (no effect).

#define CMARK_OPT_VALIDATE_UTF8 (1 << 9)

Validate UTF-8 in the input before parsing, replacing illegal sequences with the replacement character U+FFFD.

#define CMARK_OPT_SMART (1 << 10)

Convert straight quotes to curly, --- to em dashes, -- to en dashes.

Version information

int **cmark_version**(*void*)

The library version as integer for runtime checks. Also available as macro CMARK_VERSION for compile time checks.

- Bits 16-23 contain the major version.
- Bits 8-15 contain the minor version.
- Bits 0-7 contain the patchlevel.

In hexadecimal format, the number 0x010203 represents version 1.2.3.

*const char ** **cmark_version_string**(*void*)

The library version string for runtime checks. Also available as macro CMARK_VERSION_STRING for compile time checks.

AUTHORS

John MacFarlane, Vicent Marti, Karlis Gaņgis, Nick Wellnhofer.