The Text Library

0.2.1

Generated by Doxygen 1.5.8

Mon Jan 24 01:12:43 2011

Contents

Chapter 1

C Basic Library: Text Library

Version:

0.2.1

Author:

Jun Woong (woong.jun at gmail.com)

Date:

last modified on 2011-01-24

1.1 Introduction

This document specifies the Text Library which belongs to the C Basic Library. The basic structure is from David Hanson's book, "C Interfaces and Implementations." I modified the original implementation to add missing but useful functions, to make it conform to the C standard and to enhance its readibility; for example a prefix is used more strictly in order to avoid the user namespace pollution.

Since the book explains its design and implementation in a very comprehensive way, not to mention the copyright issues, it is nothing but waste to repeat it here, so I finish this document by giving introduction to the library; how to use the facilities is deeply explained in files that define them.

The Text Library reserves identifiers starting with `text_` and `TEXT_`, and imports the Assertion Library (which requires the Exception Handling Library) and the Memory Management Library.

1.2 How to Use The Library

The Text Library is intended to aid string manipulation in C. In C, even a simple form of string handling like obtaining a sub-string requires lengthy code performing memory

allocation and deallocation. This is mainly because strings in C end with a null character, which interferes the storage for a single string from being shared for representing its sub-strings. The Text Library provides an alternative representation for strings, that is composed of a sequence of characters (not necessarily terminated by a null) and its length in byte. This representation helps many string operations to be efficient. In addition to it, the storage necessary for the strings is almost completely controled by the library; every allocation done by the library is remembered internally, and a user has, even if not complete, control over it.

For example, consider two typical cases to handle strings: obtaining a sub-string and appending a string to another string.

```
char *t;
t = malloc(strlen(s+n) + 1);
if (!t)
    ...
strcpy(t, s+n);
...
free(t);
```

This code shows a typical way in C to get a sub-string from a string `s` and saves it to `t`. Since the string length is often not predictable, it is essential for a product-level program to dynamically allocate storage for the sub-string, and it is obliged not to forget to release it. Using the Text Library, this construct changes to:

```
text_t ts, tt;
ts = text_put(s);
tt = text_sub(ts, m, 0);
```

where text_put() converts a C string `s` to a `text_t` string `ts`, and text_sub() gets a sub-string from it. text_put() and text_sub() allocate any necessary storage and a user does not have to take care of it.

```
char *s1, *s2, *t;
t = malloc(strlen(s1)+strlen(s2) + 1);
if (!t)
    ...
strcpy(t, s1);
strcat(t, s2);
...
free(t);
```

Similarly, this code appends a string `s2` to another string `s1`, and saves the result to `t`. Not to mention that the code repeats unnecessary scanning of strings, managing storages allocated for strings is quite burdensome. Compare this code to a version using the Text Library:

```
text_t ts1, ts2, tt;
ts1 = text_put(s1);
ts2 = text_put(s2);
tt = text_cat(ts1, ts2);
```

All things it has to do is to convert the C strings to their `text_t` string and to apply the string concatenating operation to them.

As you can see in the examples above, there is an extra expense to convert between C strings and `text_t` strings, but the merit that `text_t` strings bring is quite significant and that expense should not be that big if unnecessary conversions are eliminated by a good program design.

In general, referring to a character in a string is achieved by calculating an index of the character in the array for the string. In this library, a new and more convenient scheme to refer to certain positions in a string is introduced; see text_pos() for details. Just to mention one advantage the new scheme has, in order to refer to the end of a string, there is no need to call a string-length function or to inspect the `len` member of a `text_t` object; passing 0 to a position parameter of a library function is enough.

Managing the storage for `text_t` strings (called "the text space") is similar to record the state of the text space and to restore it to one of its previous states. Whenever the library allocates storage for a string, it acts as if it changes the state of the text space. A user code records the state when it wants and can deallocate any storage allocated after that record by restoring the text space to the remembered state; you might notice that the text space behaves like a stack containing the allocated chunks. text_save() and text_restore() explain more details.

1.2.1 Some Caveats

A null character that terminates a C string is not special in handling a `text_t` string. This means that a `text_t` string can have embedded null characters in it and all functions except for one converting to a C string treat a null indifferently from a normal character. Note that a `text_t` string does not need to end with a null character.

On the contrary, nothing prevents a `text_t` string from ending with a null character; to be precise, the string contains (rather than ends with) the null character as its part. It is sometimes useful to have a `text_t` string contain a null character, especially when converting it to a C string occurs very frequently; note that, however, placing a null character in a `text_t` string prohibits other strings from sharing the storage with it, which is to give up the major advantage the Text Library offers.

Functions in this library always generate a new string for the result. Comparing strcat() to text_cat() shows what this means:

```
strcat(s1, s2);
```

Assuming that the area `s1` points to is big enough to contain the result, strcat() modifies the string `s1` by appending `s2` to it. An equivalent `text_t` version is as follows:

```
t = text_cat(s1, s2);
```

where `t` is the resulting string and, `s1` and `s2` are unchanged. This difference, even if looks very small, often leads an unintended bug like writing this kind of code:

```
text_cat(s1, s2);
```

and expecting `s1` to point to the resulting string. The same caution goes for text_dup() and text_reverse(), too.

1.3 Boilerplate Code

A typical use of the Text Library starts with recording the state of the text space for managing storage:

```
text_save_t *chckpt;

chckpt = text_save();
```

Since the state of the text space is kept before any other Text Library functions are invoked, restoring the state to what is kept in `chckpt` effectively releases all storages the Text Library allocates. If you don't mind the memory leakage problem, you may ignore about saving and restoring the text space state.

Then, the program can generate a `text_t` string from a C string (text_box(), text_put() and text_gen()), convert a `text_string` back to a C string (text_get()), apply various string operations (text_sub(), text_cat(), text_dup() and text_reverse()) including mapping a string to another string (text_map()), compare two strings (text_cmp()), locate a character in a string (text_chr(), text_rchr(), text_upto(), text_rupto(), text_any(), text_many() and text_rmany()), and locate a string in another string (text_find(), text_rfind(), text_match() and text_rmatch()). To aid an access to the internal of strings, text_pos() and TEXT_ACCESS() are provided.

Finishing jobs using `text_t` strings, the following code that corresponds to the above call to text_save() restores the state of the text space:

```
text_restore(&chckpt);
```

As explained in text_save(), there is no requirement that text_save() and its corresponding text_restore() be called only once; see text_save() and text_restore() for details.

1.4 Future Directions

1.4.1 Replacing Stack-based Storage Management

The stack-like storage management by text_save() and text_restore() needs to be replaced so that other libraries are free to use the Text Library. With the current approach, invoking a clean-up function of a library that calls text_restore() for the library's texts can also destroy the storage for the program's texts. Since this effectively discourages libraries not to use the Text Library, it would be better to hire a lifetime-based approach like that used in the Arena Library.

1.4.2 Minor Changes

If the stack-like strategy for managing the storage is not replaced as described above, detecting some erroneous sequences of text_save() and text_restore() would be useful. For more information, see the example given in text_save().

1.5 Contact Me

Visit http://project.woong.org to get the lastest version of this library. Only a small portion of my homepage (http://www.woong.org) is maintained in English, thus one who is not good at Korean would have difficulty when navigating most of other pages served in Korean. If you think the information you are looking for is on pages written in Korean you cannot read, do not hesitate to send me an email asking for help.

Any comments about the library are welcomed. If you have a proposal or question on the library just email me, and then I will reply as soon as possible.

1.6 Copyright

CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Chapter 2

Todo List

Global text_save Some improvements are possible and planned:

- text_save() and text_restore() can be improved to detect an erroneous call shown in the above example;
- the stack-like storage management by text_save() and text_restore() unnecessarily keeps the Text Library from being used in ohter libraries. For example, text_restore() invoked by a clean-up function of a library can destroy the storage for texts that are still in use by a program. The approach used by the Arena Library would be more appropriate.

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Data Structure Documentation

5.1 text_t Struct Reference

implements a text.

`#include <text.h>`

Data Fields

- int len
- const char * str

5.1.1 Detailed Description

implements a text.

struct `text_t` implements a text that is alternative representation of character strings. The C representation of strings using the terminating null character has some drawbacks:

- it has to scan all characters in a string to determine its length, and
- it too often has to copy a string even when not necessary. For example, suppose concatenating a string to another string and those strings are not immutable. To determine the byte into which the former string is to be put, one has to compute the length of the later string, which requires to scan all characters in it. Besides, one has to somehow allocate storage to contain the result because there is no way for two different strings to share the same sequence of characters in the storage. Jobs like scanning a string and allocation of new storage can be avoided by getting rid of the terminating null character and keeping the length of a string in a separate place.

Because the null character does not play a role of terminating a string anymore, there is no need to treat it specially. Therefore, a text represented by `text_t` is allowed to

contain the null character anywhere in it, and other `text_t` functions never treat it in a special way. Nevertheless, be warned that, if a text which has the null character embedded in it is converted to a C string, the resulting string might not work as expected because of the embedded null character.

The Text Library is not intended to completely replace the C representation of strings. To perform string operations other than those provided by the Text Library, a user has to convert a text back to a C string and then apply ordinary string functions to the result. Such a conversion between those two representations is the cost for the benefit the Text Library confers. To minimize the cost, some basic text operations like comparison and mapping are also supported.

`text_t` intentionally reveals its interals, so that a user can readily get the length of a text and can access to the text as necessary. Modifying a text, however, makes a program behave in an unpredictable way, which is the reason the `str` member is const-qualified.

Most functions in this library take and return a `text_t` value, not a pointer to it. This design approach simplifies an implementation since they never need to allocate a descriptor for a text. The size of `text_t` being not so big, passing its value would cause no big penalty on performance.

5.1.2 Field Documentation

5.1.2.1 int text_t::len

length of string

5.1.2.2 const char* text_t::str

string (possibly not having null character

The documentation for this struct was generated from the following file:

- text.h

Chapter 6

File Documentation

6.1 text.c File Reference

Source for Text Library (CBL).

```
#include <stddef.h>
#include <string.h>
#include <limits.h>
#include "cbl/assert.h"
#include "cbl/memory.h"
#include "text.h"
```

Include dependency graph for text.c:

Data Structures

- struct **text_save_t**
- struct **chunk**

Defines

- #define **IDX**(i, len) (((i) <= 0)? (i) + (len): (i) - 1)
- #define **ISATEND**(s, n) ((s).str+(s).len == current → avail && (n) <= current → limit-current → avail)
- #define **EQUAL**(s, i, t) (memcmp(&(s).str[i], (t).str, (t).len) == 0)

- #define **SWAP**(i, j)

Functions

- int() text_pos (text_t s, int i)
  normalizes a text position.
- text_t() text_box (const char *str, int len)
  boxes a null-terminated string to construct a text.
- text_t() text_sub (text_t s, int i, int j)
  constructs a sub-text of a text.
- text_t() text_put (const char *str)
  constructs a text from a null-terminated string.
- text_t() text_gen (const char str[ ], int size)
  constructs a text from an array of characters.
- char *() text_get (char *str, int size, text_t s)
  converts a text to a C string.
- text_t() text_dup (text_t s, int n)
  constructs a text by duplicating another text.
- text_t() text_cat (text_t s1, text_t s2)
  constructs a text by concatenating two texts.
- text_t() text_reverse (text_t s)
  constructs a text by reversing a text.
- text_t() text_map (text_t s, const text_t *from, const text_t *to)
  constructs a text by converting a text based on a specified mapping.
- int() text_cmp (text_t s1, text_t s2)
  compares two texts.
- text_save_t *() text_save (void)
  saves the current top of the text space.
- void() text_restore (text_save_t **save)
  restores a saved state of the text space.
- int() text_chr (text_t s, int i, int j, int c)
  finds the first occurrence of a character in a text.

- int() text_rchr (text_t s, int i, int j, int c)

  finds the last occurrence of a character in a text.

- int() text_upto (text_t s, int i, int j, text_t set)

  finds the first occurrence of any character from a set in a text.

- int() text_rupto (text_t s, int i, int j, text_t set)

  finds the last occurrence of any character from a set in a text.

- int() text_find (text_t s, int i, int j, text_t str)

  finds the first occurrence of a text in a text.

- int() text_rfind (text_t s, int i, int j, text_t str)

  finds the last occurrence of a text in a text.

- int() text_any (text_t s, int i, text_t set)

  checks if a character of a specified position matches any character from a set.

- int() text_many (text_t s, int i, int j, text_t set)

  finds the end of a span consisted of characters from a set.

- int() text_rmany (text_t s, int i, int j, text_t set)

  finds the start of a span consisted of characters from a set.

- int() text_match (text_t s, int i, int j, text_t str)

  checks if a text starts with another text.

- int() text_rmatch (text_t s, int i, int j, text_t str)

  checks if a text ends with another text.

Variables

- const text_t **text_ucase** = { 26, "ABCDEFGHIJKLMNOPQRSTUVWXYZ" }
- const text_t **text_lcase** = { 26, "abcdefghijklmnopqrstuvwxyz" }
- const text_t **text_digits** = { 10, "0123456789" }
- const text_t **text_null** = { 0, "" }

6.1.1 Detailed Description

Source for Text Library (CBL).

6.1.2 Define Documentation

6.1.2.1 #define SWAP(i, j)

Value:

```
do {                \
                    \
        int t = i;      \
        i = j;          \
        j = t;          \
    } while(0)
```

6.1.3 Function Documentation

6.1.3.1 int() text_any (text_t *s*, int *i*, text_t *set*)

checks if a character of a specified position matches any character from a set.

text_any() checks if a character of a specified position by `i` in a text `s` matches any character from a set `set`. `i` specifies the left position of a character. If it matches, text_any() returns the right positive position of the character or 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
   c  a  c  a  o  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_any(t, 2, text_box("ca", 2)) gives 3 because `a` matches. If the set containing characters to find is empty, text_any() always fails and returns 0.

Note that giving to `i` the last position (7 or 0 in the example text) makes text_any() fail and return 0; that does not cause the assertion to fail since it is a valid position.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `set`

Parameters:

← ***s*** text in which character is to be found

← ***i*** left position of character to match

← ***set*** set text containing characters to find

Returns:

right positive position of matched character or 0

6.1.3.2 text_t() text_box (const char * *str*, int *len*)

boxes a null-terminated string to construct a text.

text_box() "boxes" a constant string or a string whose storage is already allocated properly by a user. Unlike text_put(), text_box() does not copy a given string and the length of a text is granted by a user. text_box() is useful especially when constructing a text representation for a string literal:

```
text_t t = text_box("sample", 6);
```

Note, in the above example, that the terminating null character is excluded by the length given to text_box(). If a user gives 7 for the length, the resulting text includes a null character, which constructs a different text from what the above call makes.

An empty text whose length is 0 is allowed. It can be constructed simply as in the following example:

```
text_t empty = text_box("", 0);
```

and a predefined empty text, `text_null` is also provided for convenience.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid string or length given for `str` or `len`

Parameters:

← ***str*** string to box for text representation

← ***len*** length of string to box

Returns:

text containing given string

6.1.3.3 text_t() text_cat (text_t *s1*, text_t *s2*)

constructs a text by concatenating two texts.

text_cat() constructs a new text by concatenating `s2` to `s1`.

Possible exceptions: assert_exceptfail, memory_exceptfail

Unchecked errors: invalid text given for `s1` or `s2`

Warning:

Unlike strcat() in the standard library, text_cat() does not change a given text by concatenation, but creates a new text by concatenating `s2` to `s1`, which means only the returned text has the concatenated result.

Parameters:

← ***s1*** text to which another text is to be concatenated

← ***s2*** text to concatenate

Returns:

concatenated text

6.1.3.4 int() text_chr (text_t *s*, int *i*, int *j*, int *c*)

finds the first occurrence of a character in a text.

text_chr() finds the first occurrence of a character `c` in the specified range of a text `s`. The range is specified by `i` and `j`. If found, text_chr() returns the left position of the found character. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
  e  v  e  n  t  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_chr(t, -6, 5, 'e') gives 1 while text_chr(t, -6, 5, 's') does 0.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s`

Parameters:

← ***s*** text in which character is to be found
← ***i*** range specified
← ***j*** range specified
← ***c*** character to find

Returns:

left positive position of found character or 0

6.1.3.5 int() text_cmp (text_t *s1*, text_t *s2*)

compares two texts.

text_cmp() compares two texts as strcmp() does strings except that a null character is not treated specially by the former.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s1` or `s2`

Parameters:

← ***s1*** text to compare
← ***s2*** text to compare

Returns:

comparison result

Return values:

negative `s1` compares less than `s2`
0 `s1` compares equal to `s2`
positive `s1` compares larger than `s2`

6.1.3.6 text_t() text_dup (text_t *s*, int *n*)

constructs a text by duplicating another text.

text_dup() takes a text and constructs a text that duplicates the original text `n` times. For example, the following call

```
text_dup(text_box("sample", 6), 3);
```

constructs as the result a text: `samplesamplesample`

Possible exceptions: assert_exceptfail, memory_exceptfail

Unchecked errors: invalid text given for `s`

Warning:

Note that text_dup() does not change a given text, but creates a new text that duplicates a given text. Do not forget, in this library, a text is immutable.

Parameters:

← ***s*** text to duplicate

← ***n*** number of duplication

Returns:

text duplicated

6.1.3.7 int() text_find (text_t *s*, int *i*, int *j*, text_t *str*)

finds the first occurrence of a text in a text.

text_find() finds the first occurrence of a text `str` in the specified range of a text `s`. The range is specified by `i` and `j`. If found, text_find() returns the left position of the character starting the found text. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
   c  a  c  a  o  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_find(t, 6, -6, text_box("ca", 2)) gives 1. If `str` is empty, text_find() always succeeds and returns the left positive position of the specified range.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `str`

Parameters:

← ***s*** text in which another text is to be found

← ***i*** range specified

← ***j*** range specified

← ***str*** text to find

Returns:

left positive position of found text or 0

6.1.3.8 text_t() text_gen (const char *str*[ ], int *size*)

constructs a text from an array of characters.

text_gen() copies `size` characters from `str` to the text space and returns a text representing the copied characters. The terminating null character is considered an ordinary character if any. Because it always copies given characters, the storage for the original array can be safely released after a text for it has been generated.

text_gen() is useful when a caller wants to construct a text that embodies the terminating null character with allocating storage for it. text_put() allocates storage but always precludes the null character, and text_box() can make the resulting text embody the null character but allocates no storage. text_gen() is added to fill the gap.

Possible exceptions: assert_exceptfail, memory_exceptfail

Unchecked errors: invalid string given for `str`, invalid size given for `size`

Parameters:

← ***str*** null terminated string to copy for text representation

← ***size*** length of string

Returns:

text containing given string

6.1.3.9 char*() text_get (char * *str*, int *size*, text_t *s*)

converts a text to a C string.

text_get() is used when converting a text to a C string that is null-terminated. There are two ways to provide a buffer into which the resulting C string is to be written. If `str` is not a null pointer, text_get() assumes that a user provides the buffer whose size is `size`, and tries to write the conversion result to it. If its specified size is not enough to contain the result, it raises an exception due to assertion failure. If `str` is a null pointer, `size` is ignored and text_get() allocates a proper buffer to contain the reulsting string. The Text Library never deallocates the buffer allocated by text_ger(), thus a user has to set it free when it is no longer necessary.

Possible exceptions: assert_exceptfail, memory_exceptfail

Unchecked errors: invalid text given for `s`, invalid buffer or size given for `str` or `size`

Parameters:

→ ***str*** buffer into which converted string to be written

← ***size*** size of given buffer

← ***s*** text to convert to C string

Returns:

pointer to buffer containing C string

6.1.3.10 int() text_many (text_t *s*, int *i*, int *j*, text_t *set*)

finds the end of a span consisted of characters from a set.

If the specified range of a text `s` starts with a character from a set `set`, text_many() returns the right positive position ending a span consisted of characters from the set. The range is specified by `i` and `j`. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
  c  a  c  a  o  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_many(t, 2, 6, text_box("ca", 2)) gives 5. If the set containing characters to find is empty, text_many() always fails and returns 0.

Since text_many() checks the range starts with a character from a given set, text_many() is often called after text_upto().

The original code in the book is modified to form a more compact form.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `set`

Parameters:

← ***s*** text in which character to be found

← ***i*** range specified

← ***j*** range specified

← ***set*** set text containing characters to find

Returns:

right positive position of span or 0

6.1.3.11 text_t() text_map (text_t *s*, const text_t * *from*, const text_t * *to*)

constructs a text by converting a text based on a specified mapping.

text_map() converts a text based on a mapping that is described by two pointers to texts. Both pointers to describe a mapping should be a null pointers or non-null pointers; it is not allowed for only one of them to be a null pointer.

When they are non-null, they should point to texts whose lengths equal. text_map() takes a text and copies it converting any occurrence of characters in a text pointed by `from` to corresponding characters in a text pointed by `to`, where the corresponding characters are determined by their positions in a text. Ohter characters are copied unchagned.

Once a mapping is set by calling text_map() with non-null text pointers, text_map() can be called with a null pointers for `from` and `to`, in which case the latest mapping is used for conversion. Calling with a null pointers is highly recommended whenever possible, since constructing a mapping table from two texts costs time.

For example, after the following call:

```
result = text_map(t, &text_upper, &text_lower);
```

`result` is a text copied from `t` converting any uppercase letters in it to corresponding lowercase letters.

Possible exceptions: assert_exceptfail, memory_exceptfail

Unchecked errors: invalid text given for `s`, `from` or `to`

Parameters:

← ***s*** text to convert

← ***from*** pointer to text describing mapping

← ***to*** pointer to text describing mapping

Returns:

converted text

6.1.3.12 int() text_match (text_t *s*, int *i*, int *j*, text_t *str*)

checks if a text starts with another text.

If the specified range of a text `s` starts with a text `str`, text_match() returns the right positive position ending the matched text. The range is specified by `i` and `j`. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
   c  a  c  a  o  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_match(t, 3, 7, text_box("ca", 2)) gives 5. If `str` is empty, text_match() always succeeds and returns the left positive position of the specified range.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `str`

Parameters:

← ***s*** text in which another text to be found

← ***i*** range specified

← ***j*** range specified

← ***str*** text to find

Returns:

right positive position ending matched text or 0

6.1.3.13 int() text_pos (text_t *s*, int *i*)

normalizes a text position.

A text position may be negative and it is often necessary to normalize it into the positive range. text_pos() takes a text position and adjusts it to the positive range. For example, given a text:

```
1  2  3  4  5  (positive positions)
  t  e  s  t
-4 -3 -2 -1 0  (non-positive positions)
  0  1  2  3   (array indices)
```

both text_pos(t, 2) and text_pos(t, -3) give 2.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s`

Parameters:

← ***s*** string for which position is to be normalized

← ***i*** position to normalize

Returns:

nomalized positive position

6.1.3.14 text_t() text_put (const char * *str*)

constructs a text from a null-terminated string.

text_put() copies a null-terminated string to the text space and returns a text representing the copied string. The resulting text does not contain the terminating null character. Because it always copies a given string, the storage for the original string can be safely released after a text for it has been generated.

Possible exceptions: assert_exceptfail, memory_exceptfail

Unchecked errors: invalid string given for `str`

Parameters:

← ***str*** null terminated string to copy for text representation

Returns:

text containing given string

6.1.3.15 int() text_rchr (text_t *s*, int *i*, int *j*, int *c*)

finds the last occurrence of a character in a text.

text_rchr() finds the last occurrence of a character `c` in the specified range of a text `s`. The range is specified by `i` and `j`. If found, text_rchr() returns the left position of the found character. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
   e  v  e  n  t  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_rchr(t, -6, 5, 'e') gives 3 while text_rchr(t, -6, 5, 's') does 0. The "r" in its name stands for "right" since what it does can be seen as scanning a given text from the right end.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s`

Parameters:

← ***s*** text in which character is to be found

← ***i*** range specified

← ***j*** range specified

← ***c*** character to find

Returns:

left positive position of found character or 0

6.1.3.16 void() text_restore (text_save_t ** *save*)

restores a saved state of the text space.

text_restore() gets the text space to a state returned by text_save(). As explained in text_save(), any text and state generated after saving the state to be reverted are invalidated, thus they should not be used. See text_save() for more details.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid saved state given for `save`

Parameters:

← ***save*** pointer to saved state of text space

Returns:

nothing

6.1.3.17 text_t() text_reverse (text_t *s*)

constructs a text by reversing a text.

text_reverse() constructs a text by reversing a given text.

Possible exceptions: assert_exceptfail, memory_exceptfail

Unchecked errors: invalid text given for `s`

Warning:

text_reverse() does not change a given text, but creates a new text by reversing a given text.

Parameters:

← ***s*** text to reverse

Returns:

reversed text

6.1.3.18 int() text_rfind (text_t *s*, int *i*, int *j*, text_t *str*)

finds the last occurrence of a text in a text.

text_rfind() finds the last occurrence of a text `str` in the specified range of a text `s`. The range is specified by `i` and `j`. If found, text_rfind() returns the left position of the character starting the found text. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
   c  a  c  a  o  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_rfind(t, -6, 6, text_box("ca", 2)) gives 3. If `str` is empty, text_rfind() always succeeds and returns the right positive position of the specified range.

The "r" in its name stands for "right" since what it does can be seen as scanning a given text from the right end.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `str`

Parameters:

← ***s*** text in which another text is to be found

← ***i*** range specified

← ***j*** range specified

← ***str*** text to find

Returns:

left positive position of found text or 0

6.1.3.19 int() text_rmany (text_t *s*, int *i*, int *j*, text_t *set*)

finds the start of a span consisted of characters from a set.

If the specified range of a text `s` ends with a character from a set `set`, text_rmany() returns the left positive position starting a span consisted of characters from the set. The range is specified by `i` and `j`. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
   c  a  c  a  o  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_rmany(t, 3, 7, text_box("aos", 3)) gives 4. The "r" in its name stands for "right" since what it does can be seen as scanning a given text from the right end. If the set containing characters to find is empty, text_rmany() always fails and returns 0.

Since text_rmany() checks the range ends with a character from a given set, text_rmany() is often called after text_rupto().

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `set`

Parameters:

← ***s*** text in which character to be found

← ***i*** range specified

← ***j*** range specified

← ***set*** set text containing characters to find

Returns:

right positive position of span or 0

6.1.3.20 int() text_rmatch (text_t *s*, int *i*, int *j*, text_t *str*)

checks if a text ends with another text.

If the specified range of a text `s` ends with a text `str`, text_rmatch() returns the left positive position starting the matched text. The range is specified by `i` and `j`. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
   c  a  c  a  o  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_rmatch(t, 3, 7, text_box("os", 2)) gives 5. If `str` is empty, text_rmatch() always succeeds and returns the right positive position of the specified range.

The "r" in its name stands for "right" since what it does can be seen as scanning a given text from the right end.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `str`

Parameters:

← ***s*** text in which another text to be found

← ***i*** range specified

← ***j*** range specified

← ***str*** text to find

Returns:

left positive position starting matched text or 0

6.1.3.21 int() text_rupto (text_t *s*, int *i*, int *j*, text_t *set*)

finds the last occurrence of any character from a set in a text.

text_rupto() finds the last occurrence of any character from a set `set` in the specified range of a text `s`. The range is specified by `i` and `j`. If found, text_rupto() returns the left position of the found character. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
   e  v  e  n  t  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_rupto(t, -6, 5, text_box("escape", 6)) gives 3. If the set containing characters to find is empty, text_rupto() always fails and returns 0.

The "r" in its name stands for "right" since what it does can be seen as scanning a given text from the right end.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `set`

Parameters:

← ***s*** text in which character is to be found

← ***i*** range specified

← ***j*** range specified

← ***set*** set text containing characters to find

Returns:

left positive position of found character or 0

6.1.3.22 text_save_t*() text_save (void)

saves the current top of the text space.

text_save() saves the current state of the text space and returns it. The text space to provide storages for texts can be seen as a stack and storages allocated by text_*() (except that allocated by text_get()) can be seen as piled up in the stack, thus any storage being used by the Text Library after a call to text_save() can be set free by calling text_restore() with the saved state. After text_restore(), any text constructed after the text_save() call is invalidated and should not be used. In addition, other saved states, if any, get also invalidated if the text space gets back to a previous state by a state saved before they are generated. For example, after the following code:

```
h = text_save();
...
g = text_save();
...
text_restore(h);
```

calling text_restore() with `g` makes the program behave in an unpredicatble way since the last call to text_restore() with `h` invalidates `g`.

Possible exceptions: memory_exceptfail

Unchecked errors: none

Returns:

saved state of text space

Todo

Some improvements are possible and planned:

- text_save() and text_restore() can be improved to detect an erroneous call shown in the above example;
- the stack-like storage management by text_save() and text_restore() unnecessarily keeps the Text Library from being used in ohter libraries. For example, text_restore() invoked by a clean-up function of a library can destroy the storage for texts that are still in use by a program. The approach used by the Arena Library would be more appropriate.

6.1.3.23 text_t() text_sub (text_t *s*, int *i*, int *j*)

constructs a sub-text of a text.

text_sub() constructs a sub-text from characters between two specified positions in a text. Positions in a text are specified as in the Doubly-Linked List Library:

```
1  2  3  4  5  6  7    (positive positions)
  s  a  m  p  l  e
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

Given the above text, a sub-string `amp` can be specified as [2:5], [2:-2], [-5:5] or [-5:-2]. Furthermore, the order in which the positions are given does not matter, which means [5:2] indicates the same sequence of characters as [2:5]. In conclusion, the following calls to text_sub() gives the same sub-text.

```
text_sub(t, 2, 5);
text_sub(t, -5: 5);
text_sub(t, -2: -5);
text_sub(t, 2: -2);
```

Since a user is not allowed to modify the resulting text and it need not end with a null character, text_sub() does not have to allocate storage for the result.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s`

Warning:

> Do not assume that the resulting text always share the same storage as the original text. An implementation might change not to guarantee it, and there is already an exception to that assumption - when text_sub() returns an empty text.

Parameters:

- ← ***s*** text from which sub-text to be constructed
- ← ***i*** position for sub-text
- ← ***j*** position for sub-text

Returns:

sub-text constructed

6.1.3.24 int() text_upto (text_t *s*, int *i*, int *j*, text_t *set*)

finds the first occurrence of any character from a set in a text.

text_upto() finds the first occurrence of any character from a set `set` in the specified range of a text `s`. The range is specified by `i` and `j`. If found, text_upto() returns the left position of the found character. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
  e  v  e  n  t  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_upto(t, -6, 5, text_box("vwxyz", 5)) gives 2. If the set containing characters to find is empty, text_upto() always fails and returns 0.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `set`

Parameters:

← ***s*** text in which character is to be found

← ***i*** range specified

← ***j*** range specified

← ***set*** set text containing characters to find

Returns:

left positive position of found character or 0

6.2 text.h File Reference

Documentation for Text Library (CBL).

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

Data Structures

- struct text_t

  implements a text.

Defines

- #define TEXT_ACCESS(t, i) ((t).str[((i) <= 0)? (i)+(t).len: (i)-1])

  accesses with a position a character in a text.

Typedefs

- typedef struct text_save_t text_save_t

  represents information on the top of the stack-like text space.

Functions

text creating functions:

- text_t text_put (const char *)

  constructs a text from a null-terminated string.

- text_t **text_gen** (const char *, int)
- text_t text_box (const char *, int)

  boxes a null-terminated string to construct a text.

- char * text_get (char *, int, text_t)

  converts a text to a C string.

text positioning functions:

- int text_pos (text_t, int)
  normalizes a text position.

text handling functions:

- text_t text_sub (text_t, int, int)
  constructs a sub-text of a text.
- text_t text_cat (text_t, text_t)
  constructs a text by concatenating two texts.
- text_t text_dup (text_t, int)
  constructs a text by duplicating another text.
- text_t text_reverse (text_t)
  constructs a text by reversing a text.
- text_t text_map (text_t, const text_t *, const text_t *)
  constructs a text by converting a text based on a specified mapping.

text comparing functions:

- int text_cmp (text_t, text_t)
  compares two texts.

text analyzing functions (character):

- int text_chr (text_t, int, int, int)
  finds the first occurrence of a character in a text.
- int text_rchr (text_t, int, int, int)
  finds the last occurrence of a character in a text.
- int text_upto (text_t, int, int, text_t)
  finds the first occurrence of any character from a set in a text.
- int text_rupto (text_t, int, int, text_t)
  finds the last occurrence of any character from a set in a text.
- int text_any (text_t, int, text_t)
  checks if a character of a specified position matches any character from a set.
- int text_many (text_t, int, int, text_t)
  finds the end of a span consisted of characters from a set.
- int text_rmany (text_t, int, int, text_t)

finds the start of a span consisted of characters from a set.

text analyzing functions (string):

- int text_find (text_t, int, int, text_t)
  finds the first occurrence of a text in a text.
- int text_rfind (text_t, int, int, text_t)
  finds the last occurrence of a text in a text.
- int text_match (text_t, int, int, text_t)
  checks if a text starts with another text.
- int text_rmatch (text_t, int, int, text_t)
  checks if a text ends with another text.

text space managing functions:

- text_save_t * text_save (void)
  saves the current top of the text space.
- void text_restore (text_save_t **save)
  restores a saved state of the text space.

Variables

- const text_t **text_ucase**
- const text_t **text_lcase**
- const text_t **text_digits**
- const text_t **text_null**

6.2.1 Detailed Description

Documentation for Text Library (CBL).

Header for Text Library (CBL).

6.2.2 Define Documentation

6.2.2.1 #define TEXT_ACCESS(t, i) ((t).str[((i) <= 0)? (i)+(t).len: (i)-1])

accesses with a position a character in a text.

TEXT_ACCESS() is useful when accessing a character in a text when a position number is known. The position can be negative, but has to be within a valid range. For

example, given a text of 4 characters, a valid positive range for the position is from 1 to 4 and a valid negative range from -4 to -1; 0 is never allowed. No validity check for the range is performed.

Possible exceptions: none

Unchecked errors: invalid position given for `i`

6.2.3 Typedef Documentation

6.2.3.1 typedef struct text_save_t text_save_t

represents information on the top of the stack-like text space.

An object of the type `text_save_t` is used when remembering the current top of the stack-like text space and restoring the space to make it have as the current the top remembered in the `text_save_t` object. For more details, see struct `text_save_t`, struct `chunk`, text_save() and text_restore().

6.2.4 Function Documentation

6.2.4.1 int text_any (text_t *s*, int *i*, text_t *set*)

checks if a character of a specified position matches any character from a set.

text_any() checks if a character of a specified position by `i` in a text `s` matches any character from a set `set`. `i` specifies the left position of a character. If it matches, text_any() returns the right positive position of the character or 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
   c  a  c  a  o  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_any(t, 2, text_box("ca", 2)) gives 3 because `a` matches. If the set containing characters to find is empty, text_any() always fails and returns 0.

Note that giving to `i` the last position (7 or 0 in the example text) makes text_any() fail and return 0; that does not cause the assertion to fail since it is a valid position.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `set`

Parameters:

← ***s*** text in which character is to be found
← ***i*** left position of character to match
← ***set*** set text containing characters to find

Returns:

right positive position of matched character or 0

6.2.4.2 text_t text_box (const char * *str*, int *len*)

boxes a null-terminated string to construct a text.

text_box() "boxes" a constant string or a string whose storage is already allocated properly by a user. Unlike text_put(), text_box() does not copy a given string and the length of a text is granted by a user. text_box() is useful especially when constructing a text representation for a string literal:

```
text_t t = text_box("sample", 6);
```

Note, in the above example, that the terminating null character is excluded by the length given to text_box(). If a user gives 7 for the length, the resulting text includes a null character, which constructs a different text from what the above call makes.

An empty text whose length is 0 is allowed. It can be constructed simply as in the following example:

```
text_t empty = text_box("", 0);
```

and a predefined empty text, `text_null` is also provided for convenience.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid string or length given for `str` or `len`

Parameters:

← ***str*** string to box for text representation

← ***len*** length of string to box

Returns:

text containing given string

6.2.4.3 text_t text_cat (text_t *s1*, text_t *s2*)

constructs a text by concatenating two texts.

text_cat() constructs a new text by concatenating `s2` to `s1`.

Possible exceptions: assert_exceptfail, memory_exceptfail

Unchecked errors: invalid text given for `s1` or `s2`

Warning:

Unlike strcat() in the standard library, text_cat() does not change a given text by concatenation, but creates a new text by concatenating `s2` to `s1`, which means only the returned text has the concatenated result.

Parameters:

← ***s1*** text to which another text is to be concatenated

← ***s2*** text to concatenate

Returns:

concatenated text

6.2.4.4 int text_chr (text_t *s*, int *i*, int *j*, int *c*)

finds the first occurrence of a character in a text.

text_chr() finds the first occurrence of a character `c` in the specified range of a text `s`. The range is specified by `i` and `j`. If found, text_chr() returns the left position of the found character. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
  e  v  e  n  t  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_chr(t, -6, 5, 'e') gives 1 while text_chr(t, -6, 5, 's') does 0.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s`

Parameters:

← ***s*** text in which character is to be found

← ***i*** range specified

← ***j*** range specified

← ***c*** character to find

Returns:

left positive position of found character or 0

6.2.4.5 int text_cmp (text_t *s1*, text_t *s2*)

compares two texts.

text_cmp() compares two texts as strcmp() does strings except that a null character is not treated specially by the former.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s1` or `s2`

Parameters:

← ***s1*** text to compare

← ***s2*** text to compare

Returns:

comparison result

Return values:

negative `s1` compares less than `s2`

0 `s1` compares equal to `s2`

positive `s1` compares larger than `s2`

6.2.4.6 text_t text_dup (text_t *s*, int *n*)

constructs a text by duplicating another text.

text_dup() takes a text and constructs a text that duplicates the original text `n` times. For example, the following call

```
text_dup(text_box("sample", 6), 3);
```

constructs as the result a text: `samplesamplesample`

Possible exceptions: assert_exceptfail, memory_exceptfail

Unchecked errors: invalid text given for `s`

Warning:

Note that text_dup() does not change a given text, but creates a new text that duplicates a given text. Do not forget, in this library, a text is immutable.

Parameters:

← ***s*** text to duplicate

← ***n*** number of duplication

Returns:

text duplicated

6.2.4.7 int text_find (text_t *s*, int *i*, int *j*, text_t *str*)

finds the first occurrence of a text in a text.

text_find() finds the first occurrence of a text `str` in the specified range of a text `s`. The range is specified by `i` and `j`. If found, text_find() returns the left position of the character starting the found text. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
  c  a  c  a  o  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_find(t, 6, -6, text_box("ca", 2)) gives 1. If `str` is empty, text_find() always succeeds and returns the left positive position of the specified range.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `str`

Parameters:

← ***s*** text in which another text is to be found

← ***i*** range specified

← ***j*** range specified

← ***str*** text to find

Returns:

left positive position of found text or 0

6.2.4.8 char* text_get (char * *str*, int *size*, text_t *s*)

converts a text to a C string.

text_get() is used when converting a text to a C string that is null-terminated. There are two ways to provide a buffer into which the resulting C string is to be written. If `str` is not a null pointer, text_get() assumes that a user provides the buffer whose size is `size`, and tries to write the conversion result to it. If its specified size is not enough to contain the result, it raises an exception due to assertion failure. If `str` is a null pointer, `size` is ignored and text_get() allocates a proper buffer to contain the reulsting string. The Text Library never deallocates the buffer allocated by text_ger(), thus a user has to set it free when it is no longer necessary.

Possible exceptions: assert_exceptfail, memory_exceptfail

Unchecked errors: invalid text given for `s`, invalid buffer or size given for `str` or `size`

Parameters:

→ ***str*** buffer into which converted string to be written

← ***size*** size of given buffer

← ***s*** text to convert to C string

Returns:

pointer to buffer containing C string

6.2.4.9 int text_many (text_t *s*, int *i*, int *j*, text_t *set*)

finds the end of a span consisted of characters from a set.

If the specified range of a text `s` starts with a character from a set `set`, text_many() returns the right positive position ending a span consisted of characters from the set.

The range is specified by `i` and `j`. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
  c  a  c  a  o  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_many(t, 2, 6, text_box("ca", 2)) gives 5. If the set containing characters to find is empty, text_many() always fails and returns 0.

Since text_many() checks the range starts with a character from a given set, text_many() is often called after text_upto().

The original code in the book is modified to form a more compact form.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `set`

Parameters:

← ***s*** text in which character to be found

← ***i*** range specified

← ***j*** range specified

← ***set*** set text containing characters to find

Returns:

right positive position of span or 0

6.2.4.10 text_t text_map (text_t *s*, const text_t * *from*, const text_t * *to*)

constructs a text by converting a text based on a specified mapping.

text_map() converts a text based on a mapping that is described by two pointers to texts. Both pointers to describe a mapping should be a null pointers or non-null pointers; it is not allowed for only one of them to be a null pointer.

When they are non-null, they should point to texts whose lengths equal. text_map() takes a text and copies it converting any occurrence of characters in a text pointed by `from` to corresponding characters in a text pointed by `to`, where the corresponding characters are determined by their positions in a text. Ohter characters are copied unchagned.

Once a mapping is set by calling text_map() with non-null text pointers, text_map() can be called with a null pointers for `from` and `to`, in which case the latest mapping is used for conversion. Calling with a null pointers is highly recommended whenever possible, since constructing a mapping table from two texts costs time.

For example, after the following call:

```
result = text_map(t, &text_upper, &text_lower);
```

`result` is a text copied from `t` converting any uppercase letters in it to corresponding lowercase letters.

Possible exceptions: assert_exceptfail, memory_exceptfail

Unchecked errors: invalid text given for `s`, `from` or `to`

Parameters:

← ***s*** text to convert

← ***from*** pointer to text describing mapping

← ***to*** pointer to text describing mapping

Returns:

converted text

6.2.4.11 int text_match (text_t *s*, int *i*, int *j*, text_t *str*)

checks if a text starts with another text.

If the specified range of a text `s` starts with a text `str`, text_match() returns the right positive position ending the matched text. The range is specified by `i` and `j`. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
   c  a  c  a  o  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_match(t, 3, 7, text_box("ca", 2)) gives 5. If `str` is empty, text_match() always succeeds and returns the left positive position of the specified range.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `str`

Parameters:

← ***s*** text in which another text to be found

← ***i*** range specified

← ***j*** range specified

← ***str*** text to find

Returns:

right positive position ending matched text or 0

6.2.4.12 int text_pos (text_t *s*, int *i*)

normalizes a text position.

A text position may be negative and it is often necessary to normalize it into the positive range. text_pos() takes a text position and adjusts it to the positive range. For example, given a text:

```
1  2  3  4  5  (positive positions)
  t  e  s  t
-4 -3 -2 -1 0  (non-positive positions)
  0  1  2  3   (array indices)
```

both text_pos(t, 2) and text_pos(t, -3) give 2.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s`

Parameters:

← ***s*** string for which position is to be normalized

← ***i*** position to normalize

Returns:

nomalized positive position

6.2.4.13 text_t text_put (const char * *str*)

constructs a text from a null-terminated string.

text_put() copies a null-terminated string to the text space and returns a text representing the copied string. The resulting text does not contain the terminating null character. Because it always copies a given string, the storage for the original string can be safely released after a text for it has been generated.

Possible exceptions: assert_exceptfail, memory_exceptfail

Unchecked errors: invalid string given for `str`

Parameters:

← ***str*** null terminated string to copy for text representation

Returns:

text containing given string

6.2.4.14 int text_rchr (text_t *s*, int *i*, int *j*, int *c*)

finds the last occurrence of a character in a text.

text_rchr() finds the last occurrence of a character `c` in the specified range of a text `s`. The range is specified by `i` and `j`. If found, text_rchr() returns the left position of the found character. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
  e  v  e  n  t  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_rchr(t, -6, 5, 'e') gives 3 while text_rchr(t, -6, 5, 's') does 0. The "r" in its name stands for "right" since what it does can be seen as scanning a given text from the right end.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s`

Parameters:

← ***s*** text in which character is to be found

← ***i*** range specified

← ***j*** range specified

← ***c*** character to find

Returns:

left positive position of found character or 0

6.2.4.15 void text_restore (text_save_t ** *save*)

restores a saved state of the text space.

text_restore() gets the text space to a state returned by text_save(). As explained in text_save(), any text and state generated after saving the state to be reverted are invalidated, thus they should not be used. See text_save() for more details.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid saved state given for `save`

Parameters:

← ***save*** pointer to saved state of text space

Returns:

nothing

6.2.4.16 text_t text_reverse (text_t *s*)

constructs a text by reversing a text.

text_reverse() constructs a text by reversing a given text.

Possible exceptions: assert_exceptfail, memory_exceptfail

Unchecked errors: invalid text given for `s`

Warning:

text_reverse() does not change a given text, but creates a new text by reversing a given text.

Parameters:

← ***s*** text to reverse

Returns:

reversed text

6.2.4.17 int text_rfind (text_t *s*, int *i*, int *j*, text_t *str*)

finds the last occurrence of a text in a text.

text_rfind() finds the last occurrence of a text `str` in the specified range of a text `s`. The range is specified by `i` and `j`. If found, text_rfind() returns the left position of the character starting the found text. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
   c  a  c  a  o  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_rfind(t, -6, 6, text_box("ca", 2)) gives 3. If `str` is empty, text_rfind() always succeeds and returns the right positive position of the specified range.

The "r" in its name stands for "right" since what it does can be seen as scanning a given text from the right end.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `str`

Parameters:

← ***s*** text in which another text is to be found
← ***i*** range specified
← ***j*** range specified
← ***str*** text to find

Returns:

left positive position of found text or 0

6.2.4.18 int text_rmany (text_t *s*, int *i*, int *j*, text_t *set*)

finds the start of a span consisted of characters from a set.

If the specified range of a text `s` ends with a character from a set `set`, text_rmany() returns the left positive position starting a span consisted of characters from the set. The range is specified by `i` and `j`. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
   c  a  c  a  o  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_rmany(t, 3, 7, text_box("aos", 3)) gives 4. The "r" in its name stands for "right" since what it does can be seen as scanning a given text from the right end. If the set containing characters to find is empty, text_rmany() always fails and returns 0.

Since text_rmany() checks the range ends with a character from a given set, text_rmany() is often called after text_rupto().

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `set`

Parameters:

← ***s*** text in which character to be found

← ***i*** range specified

← ***j*** range specified

← ***set*** set text containing characters to find

Returns:

right positive position of span or 0

6.2.4.19 int text_rmatch (text_t *s*, int *i*, int *j*, text_t *str*)

checks if a text ends with another text.

If the specified range of a text `s` ends with a text `str`, text_rmatch() returns the left positive position starting the matched text. The range is specified by `i` and `j`. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
   c  a  c  a  o  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_rmatch(t, 3, 7, text_box("os", 2)) gives 5. If `str` is empty, text_rmatch() always succeeds and returns the right positive position of the specified range.

The "r" in its name stands for "right" since what it does can be seen as scanning a given text from the right end.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `str`

Parameters:

← ***s*** text in which another text to be found

← ***i*** range specified

← ***j*** range specified

← ***str*** text to find

Returns:

left positive position starting matched text or 0

6.2.4.20 int text_rupto (text_t *s*, int *i*, int *j*, text_t *set*)

finds the last occurrence of any character from a set in a text.

text_rupto() finds the last occurrence of any character from a set `set` in the specified range of a text `s`. The range is specified by `i` and `j`. If found, text_rupto() returns the left position of the found character. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
  e  v  e  n  t  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_rupto(t, -6, 5, text_box("escape", 6)) gives 3. If the set containing characters to find is empty, text_rupto() always fails and returns 0.

The "r" in its name stands for "right" since what it does can be seen as scanning a given text from the right end.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `set`

Parameters:

← ***s*** text in which character is to be found

← ***i*** range specified

← ***j*** range specified

← ***set*** set text containing characters to find

Returns:

left positive position of found character or 0

6.2.4.21 text_save_t* text_save (void)

saves the current top of the text space.

text_save() saves the current state of the text space and returns it. The text space to provide storages for texts can be seen as a stack and storages allocated by text_*() (except that allocated by text_get()) can be seen as piled up in the stack, thus any storage being used by the Text Library after a call to text_save() can be set free by calling text_restore() with the saved state. After text_restore(), any text constructed after the text_save() call is invalidated and should not be used. In addition, other saved states, if any, get also invalidated if the text space gets back to a previous state by a state saved before they are generated. For example, after the following code:

```
h = text_save();
...
g = text_save();
...
text_restore(h);
```

calling text_restore() with `g` makes the program behave in an unpredicatble way since the last call to text_restore() with `h` invalidates `g`.

Possible exceptions: memory_exceptfail

Unchecked errors: none

Returns:

saved state of text space

Todo

Some improvements are possible and planned:

- text_save() and text_restore() can be improved to detect an erroneous call shown in the above example;
- the stack-like storage management by text_save() and text_restore() unnecessarily keeps the Text Library from being used in ohter libraries. For example, text_restore() invoked by a clean-up function of a library can destroy the storage for texts that are still in use by a program. The approach used by the Arena Library would be more appropriate.

6.2.4.22 text_t text_sub (text_t *s*, int *i*, int *j*)

constructs a sub-text of a text.

text_sub() constructs a sub-text from characters between two specified positions in a text. Positions in a text are specified as in the Doubly-Linked List Library:

```
1  2  3  4  5  6  7    (positive positions)
  s  a  m  p  l  e
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

Given the above text, a sub-string `amp` can be specified as [2:5], [2:-2], [-5:5] or [-5:-2]. Furthermore, the order in which the positions are given does not matter, which means [5:2] indicates the same sequence of characters as [2:5]. In conclusion, the following calls to text_sub() gives the same sub-text.

```
text_sub(t, 2, 5);
text_sub(t, -5: 5);
text_sub(t, -2: -5);
text_sub(t, 2: -2);
```

Since a user is not allowed to modify the resulting text and it need not end with a null character, text_sub() does not have to allocate storage for the result.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s`

Warning:

Do not assume that the resulting text always share the same storage as the original text. An implementation might change not to guarantee it, and there is already an exception to that assumption - when text_sub() returns an empty text.

Parameters:

← ***s*** text from which sub-text to be constructed

← ***i*** position for sub-text

← ***j*** position for sub-text

Returns:

sub-text constructed

6.2.4.23 int text_upto (text_t *s*, int *i*, int *j*, text_t *set*)

finds the first occurrence of any character from a set in a text.

text_upto() finds the first occurrence of any character from a set `set` in the specified range of a text `s`. The range is specified by `i` and `j`. If found, text_upto() returns the left position of the found character. It returns 0 otherwise. For example, given the following text:

```
1  2  3  4  5  6  7    (positive positions)
  e  v  e  n  t  s
-6 -5 -4 -3 -2 -1 0    (non-positive positions)
```

text_upto(t, -6, 5, text_box("vwxyz", 5)) gives 2. If the set containing characters to find is empty, text_upto() always fails and returns 0.

Possible exceptions: assert_exceptfail

Unchecked errors: invalid text given for `s` or `set`

Parameters:

← ***s*** text in which character is to be found

← ***i*** range specified

← ***j*** range specified

← ***set*** set text containing characters to find

Returns:

left positive position of found character or 0

Index