# Contents

1 About The Extended C Library: libxc 1

2 File Index 3

2.1 File List 3

3 File Documentation 5

3.1 libxc-0.0.4/xcfg/xcfg.c File Reference 5

3.1.1 Detailed Description 6

3.1.2 Function Documentation 6

3.1.2.1 `xcfg_configure` 6

3.1.2.2 `xcfg_from_array` 7

3.1.2.3 `xcfg_from_env` 8

3.1.2.4 `xcfg_from_file` 9

3.1.2.5 `xcfg_get` 10

3.1.2.6 `xcfg_get_all` 10

3.1.2.7 `xcfg_get_f` 11

3.1.2.8 `xcfg_get_i` 12

3.1.2.9 `xcfg_get_u` 12

3.1.2.10 `xcfg_save_config` 13

3.1.2.11 `xcfg_set` 14

3.1.2.12 `xcfg_shutdown` 14

3.2 xcfg/xcfg.c File Reference 15

3.2.1 Detailed Description 15

3.2.2 Function Documentation 16

3.2.2.1 `xcfg_configure` 16

3.2.2.2 `xcfg_from_array` 16

3.2.2.3 `xcfg_from_env` 18

3.2.2.4 `xcfg_from_file` 18

3.2.2.5 `xcfg_get` 20

3.2.2.6 `xcfg_get_all` 20

3.2.2.7 `xcfg_get_f` 21

3.2.2.8 `xcfg_get_i` 21
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.2.9</td>
<td>xcfg_get_u</td>
</tr>
<tr>
<td>3.2.2.10</td>
<td>xcfg_save_config</td>
</tr>
<tr>
<td>3.2.2.11</td>
<td>xcfg_set</td>
</tr>
<tr>
<td>3.2.2.12</td>
<td>xcfg_shutdown</td>
</tr>
<tr>
<td>3.3</td>
<td>libxc-0.0.4/xdict/xdict.c File Reference</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Detailed Description</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Function Documentation</td>
</tr>
<tr>
<td>3.3.2.1</td>
<td>xdict_del</td>
</tr>
<tr>
<td>3.3.2.2</td>
<td>xdict_get</td>
</tr>
<tr>
<td>3.3.2.3</td>
<td>xdict_iterate</td>
</tr>
<tr>
<td>3.3.2.4</td>
<td>xdict_map</td>
</tr>
<tr>
<td>3.3.2.5</td>
<td>xdict_new</td>
</tr>
<tr>
<td>3.3.2.6</td>
<td>xdict_set</td>
</tr>
<tr>
<td>3.4</td>
<td>xdict/xdict.c File Reference</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Detailed Description</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Function Documentation</td>
</tr>
<tr>
<td>3.4.2.1</td>
<td>xdict_del</td>
</tr>
<tr>
<td>3.4.2.2</td>
<td>xdict_get</td>
</tr>
<tr>
<td>3.4.2.3</td>
<td>xdict_iterate</td>
</tr>
<tr>
<td>3.4.2.4</td>
<td>xdict_map</td>
</tr>
<tr>
<td>3.4.2.5</td>
<td>xdict_new</td>
</tr>
<tr>
<td>3.4.2.6</td>
<td>xdict_set</td>
</tr>
<tr>
<td>3.5</td>
<td>libxc-0.0.4/xmalloc/xmalloc.c File Reference</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Detailed Description</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Function Documentation</td>
</tr>
<tr>
<td>3.5.2.1</td>
<td>xmalloc</td>
</tr>
<tr>
<td>3.5.2.2</td>
<td>xmalloc_dump</td>
</tr>
<tr>
<td>3.5.2.3</td>
<td>xmalloc_find</td>
</tr>
<tr>
<td>3.5.2.4</td>
<td>xmalloc_free</td>
</tr>
<tr>
<td>3.5.2.5</td>
<td>xmallocrealloc</td>
</tr>
<tr>
<td>3.5.2.6</td>
<td>xmallocreset</td>
</tr>
<tr>
<td>3.5.2.7</td>
<td>xmalloc_set_knob</td>
</tr>
<tr>
<td>3.6</td>
<td>xmalloc/xmalloc.c File Reference</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Detailed Description</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Function Documentation</td>
</tr>
<tr>
<td>3.6.2.1</td>
<td>xmalloc</td>
</tr>
<tr>
<td>3.6.2.2</td>
<td>xmalloc_dump</td>
</tr>
<tr>
<td>3.6.2.3</td>
<td>xmalloc_find</td>
</tr>
<tr>
<td>3.6.2.4</td>
<td>xmalloc_free</td>
</tr>
<tr>
<td>3.6.2.5</td>
<td>xmallocrealloc</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>3.6.2.6</td>
<td>xmalloc_reset</td>
</tr>
<tr>
<td>3.6.2.7</td>
<td>xmalloc_set_knob</td>
</tr>
<tr>
<td>3.7</td>
<td>xshare/xshare.c File Reference</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Detailed Description</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Function Documentation</td>
</tr>
<tr>
<td>3.7.2.1</td>
<td>xshare_close</td>
</tr>
<tr>
<td>3.7.2.2</td>
<td>xshare_errmsg</td>
</tr>
<tr>
<td>3.7.2.3</td>
<td>xshare_error</td>
</tr>
<tr>
<td>3.7.2.4</td>
<td>xshare_open</td>
</tr>
<tr>
<td>3.7.2.5</td>
<td>xshare_symbol</td>
</tr>
<tr>
<td>3.8</td>
<td>xshare/xshare.c File Reference</td>
</tr>
<tr>
<td>3.8.1</td>
<td>Detailed Description</td>
</tr>
<tr>
<td>3.8.2</td>
<td>Function Documentation</td>
</tr>
<tr>
<td>3.8.2.1</td>
<td>xshare_close</td>
</tr>
<tr>
<td>3.8.2.2</td>
<td>xshare_errmsg</td>
</tr>
<tr>
<td>3.8.2.3</td>
<td>xshare_error</td>
</tr>
<tr>
<td>3.8.2.4</td>
<td>xshare_open</td>
</tr>
<tr>
<td>3.8.2.5</td>
<td>xshare_symbol</td>
</tr>
<tr>
<td>3.9</td>
<td>xsock/xsock.c File Reference</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Detailed Description</td>
</tr>
<tr>
<td>3.9.2</td>
<td>Function Documentation</td>
</tr>
<tr>
<td>3.9.2.1</td>
<td>xsock_clear_errno</td>
</tr>
<tr>
<td>3.9.2.2</td>
<td>xsock_close</td>
</tr>
<tr>
<td>3.9.2.3</td>
<td>xsock_connect</td>
</tr>
<tr>
<td>3.9.2.4</td>
<td>xsock_errno</td>
</tr>
<tr>
<td>3.9.2.5</td>
<td>xsock_listen</td>
</tr>
<tr>
<td>3.9.2.6</td>
<td>xsock_read</td>
</tr>
<tr>
<td>3.9.2.7</td>
<td>xsock_strerror</td>
</tr>
<tr>
<td>3.9.2.8</td>
<td>xsock_write</td>
</tr>
<tr>
<td>3.10</td>
<td>xsock/xsock.c File Reference</td>
</tr>
<tr>
<td>3.10.1</td>
<td>Detailed Description</td>
</tr>
<tr>
<td>3.10.2</td>
<td>Function Documentation</td>
</tr>
<tr>
<td>3.10.2.1</td>
<td>xsock_clear_errno</td>
</tr>
<tr>
<td>3.10.2.2</td>
<td>xsock_close</td>
</tr>
<tr>
<td>3.10.2.3</td>
<td>xsock_connect</td>
</tr>
<tr>
<td>3.10.2.4</td>
<td>xsock_errno</td>
</tr>
<tr>
<td>3.10.2.5</td>
<td>xsock_listen</td>
</tr>
<tr>
<td>3.10.2.6</td>
<td>xsock_read</td>
</tr>
<tr>
<td>3.10.2.7</td>
<td>xsock_strerror</td>
</tr>
<tr>
<td>3.10.2.8</td>
<td>xsock_write</td>
</tr>
</tbody>
</table>
3.13.2.2 xtree_apply ................................................. 85
3.13.2.3 xtree_del .................................................. 86
3.13.2.4 xtree_get_child .......................................... 86
3.13.2.5 xtree_get_parent ......................................... 86
3.13.2.6 xtree_get_payload ........................................ 86
3.13.2.7 xtree_map .................................................. 87

3.14 xtree/xtree.c File Reference .................................. 87
3.14.1 Detailed Description ........................................ 88
3.14.2 Function Documentation ..................................... 88
  3.14.2.1 xtree_add_child ......................................... 88
  3.14.2.2 xtree_apply ............................................. 89
  3.14.2.3 xtree_del ................................................ 89
  3.14.2.4 xtree_get_child ......................................... 90
  3.14.2.5 xtree_get_parent ....................................... 90
  3.14.2.6 xtree_get_payload ...................................... 90
  3.14.2.7 xtree_map ................................................ 90

3.15 libxc-0.0.4/xvector/xvector.c File Reference .............. 91
3.15.1 Detailed Description ........................................ 92
3.15.2 Function Documentation ..................................... 92
  3.15.2.1 xvect_dup ................................................ 92
  3.15.2.2 xvector_del_head ....................................... 92
  3.15.2.3 xvector_del_tail ....................................... 93
  3.15.2.4 xvector_find_first ..................................... 93
  3.15.2.5 xvector_find_last ..................................... 93
  3.15.2.6 xvector_free ........................................... 94
  3.15.2.7 xvector_ins_head ....................................... 94
  3.15.2.8 xvector_ins_tail ....................................... 94
  3.15.2.9 xvector_iterate ........................................ 95
  3.15.2.10 xvector_map ........................................... 95
  3.15.2.11 xvector_native ........................................ 96

3.16 xvector/xvector.c File Reference .................................... 96
3.16.1 Detailed Description ........................................ 97
3.16.2 Function Documentation ..................................... 97
  3.16.2.1 xvect_dup ................................................ 97
  3.16.2.2 xvector_del_head ....................................... 97
  3.16.2.3 xvector_del_tail ....................................... 98
  3.16.2.4 xvector_find_first ..................................... 98
  3.16.2.5 xvector_find_last ..................................... 98
  3.16.2.6 xvector_free ........................................... 99
  3.16.2.7 xvector_ins_head ....................................... 99
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>xvector_ins_tail</td>
<td>99</td>
</tr>
<tr>
<td>xvector_iterate</td>
<td>99</td>
</tr>
<tr>
<td>xvector_map</td>
<td>100</td>
</tr>
<tr>
<td>xvector_native</td>
<td>100</td>
</tr>
</tbody>
</table>
Chapter 1

About The Extended C Library: libxc

About the Libxc Library

Introduction

Many developers avoid the C programming language due to the lack of standard functions that are so readily available in other languages. Things such as resizable arrays, common string operations, container data structures and standardised error reporting are missing from the C standard. To be perfectly honest those things aren’t particularly necessary to develop software in C; many of those functions are easily and quickly implemented in portable C anyway.

However, those things can be quite useful. For example, vectors (resizeable arrays) are useful when the developer doesn’t want to re-implement a resizeable array for storing integers, then another one for storing compound structures, then another for storing strings, etc. A single vector data structure that makes it possible to store anything is very useful. This is where libxc comes in. Libxc includes a vector library to store and manipulate elements of vectors (no matter their type). This is the whole point of libxc’s existence - to provide those tiny but necessary things that the standard doesn’t provide, such as vectors, common string operations, dictionary arrays and more, and to provide these things in a cross-platform and portable manner.

There are two main parts to libxc - the data structure functionality and the frequently needed functionality. Both of these are briefly covered below. The API documentation for libxc goes into extensive detail for every function provided, with many of the documentation pages for the functions incorporating an example as well.

This library is available for free download under the BSD license (links below) and feedback is welcome. If you find this useful, please don’t tell me, instead spread the word on all the message boards and forums where someone, for the hundredth time, is asking about a string library for C, or a dictionary implementation for C, or for decent runtime error-reporting libraries for C. Tell them about libxc, and point them in this direction.

Data Structures

Libxc includes vectors, dictionaries, sparse arrays, trees and more. All of these are documented more fully and in extensive detail in the documentation for each module. Documentation is available in HTML, PDF or Manpage formats.

Common Functionality

Libxc includes functionality for runtime error-reporting, automatically prefixing each line of output with source-filename, function-name and line-number. Extra string functionality is implemented, such as for splitting strings, parsing strings, joining strings and useful string utilities.
Memory accounting routines are available as well to help track memory leaks and access violations during de-
bugging and testing. These can be turned on and off during runtime to reduce the processing penalty incurred in
monitoring memory.

Configuration management routines are available - these let the caller parse the command line in a standard manner,
read configuration files and use configuration values to modify program behaviour at runtime.

All of these (and more) have detailed functional documentation and is is available in HTML, PDF or Manpage
formats.

Links

Sourceforce page - http://sourceforge.net/projects/libxc/?source=directory
Homepage - http://www.lelanthran.com/projects/libxc

License

Copyright (c) 2013, Lelanthran Krishna Manickum All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the
following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following
disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the fol-
lowing disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY
EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL
THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPE-
CIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT
OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTI-
ON) HOWEVER 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.

The views and conclusions contained in the software and documentation are those of the authors and should not
be interpreted as representing official policies, either expressed or implied, of the Extended C Library Project.
Chapter 2

File Index

2.1 File List

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

<table>
<thead>
<tr>
<th>Path</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>libxc-0.0.4/xcfg/xcfg.c</td>
<td>Configuration management routines</td>
<td>5</td>
</tr>
<tr>
<td>libxc-0.0.4/xcfg/xcfg.h</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>libxc-0.0.4/xdict/xdict.c</td>
<td>Implementation of dictionary (associative arrays)</td>
<td>24</td>
</tr>
<tr>
<td>libxc-0.0.4/xdict/xdict.h</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>libxc-0.0.4/xerror/xerror.h</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>libxc-0.0.4/xmalloc/xmalloc.c</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td></td>
<td>Extended malloc routines</td>
<td>39</td>
</tr>
<tr>
<td>libxc-0.0.4/xmalloc/xmalloc.h</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>libxc-0.0.4/xshare/dynamic.h</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>libxc-0.0.4/xshare/static.h</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>libxc-0.0.4/xshare/xshare.c</td>
<td></td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>Routines to load dynamic libraries at runtime</td>
<td></td>
</tr>
<tr>
<td>libxc-0.0.4/xshare/xshare.h</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>libxc-0.0.4/xsock/xsock.c</td>
<td>Cross-platform BSD Sockets wrapper</td>
<td>52</td>
</tr>
<tr>
<td>libxc-0.0.4/xsock/xsock.h</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>libxc-0.0.4/xstring/xstring.c</td>
<td></td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>Extra string routines for C</td>
<td></td>
</tr>
<tr>
<td>libxc-0.0.4/xstring/xstring.h</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>libxc-0.0.4/xtree/xtree.c</td>
<td>Create and traverse a tree structure</td>
<td>84</td>
</tr>
<tr>
<td>libxc-0.0.4/xtree/xtree.h</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>libxc-0.0.4/xvector/xvector.h</td>
<td></td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>Implementation of a vector datatype for C</td>
<td></td>
</tr>
<tr>
<td>libxc-0.0.4/xvector/xvector.c</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>libxc-0.0.4/xcfg/xcfg.c</td>
<td>Configuration management routines</td>
<td>15</td>
</tr>
<tr>
<td>libxc-0.0.4/xdict/xdict.c</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td></td>
<td>Implementation of dictionary (associative arrays)</td>
<td>31</td>
</tr>
<tr>
<td>libxc-0.0.4/xdict/xdict.h</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>libxc-0.0.4/xerror/xerror.h</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>libxc-0.0.4/xmalloc/xmalloc.c</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td></td>
<td>Extended malloc routines</td>
<td>42</td>
</tr>
<tr>
<td>libxc-0.0.4/xmalloc/xmalloc.h</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>libxc-0.0.4/xshare/dynamic.h</td>
<td></td>
<td>??</td>
</tr>
<tr>
<td>File</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>xshare/static.h</td>
<td>Routines to load dynamic libraries at runtime</td>
<td>??</td>
</tr>
<tr>
<td>xshare/xshare.c</td>
<td>Cross-platform BSD Sockets wrapper</td>
<td>58</td>
</tr>
<tr>
<td>xshare/xshare.h</td>
<td>Cross-platform BSD Sockets wrapper</td>
<td>??</td>
</tr>
<tr>
<td>xsock/xsock.c</td>
<td>Extra string routines for C</td>
<td>74</td>
</tr>
<tr>
<td>xsock/xsock.h</td>
<td>Extra string routines for C</td>
<td>??</td>
</tr>
<tr>
<td>xstring/xstring.c</td>
<td>Create and traverse a tree structure</td>
<td>87</td>
</tr>
<tr>
<td>xstring/xstring.h</td>
<td>Create and traverse a tree structure</td>
<td>??</td>
</tr>
<tr>
<td>xtree/xtree.c</td>
<td>Implementation of a vector datatype for C</td>
<td>96</td>
</tr>
<tr>
<td>xtree/xtree.h</td>
<td>Implementation of a vector datatype for C</td>
<td>??</td>
</tr>
<tr>
<td>xvector/xvector.c</td>
<td>Implementation of a vector datatype for C</td>
<td>??</td>
</tr>
<tr>
<td>xvector/xvector.h</td>
<td>Implementation of a vector datatype for C</td>
<td>??</td>
</tr>
</tbody>
</table>
Chapter 3

File Documentation

3.1  libxc-0.0.4/xcfg/xcfg.c File Reference

Configuration management routines.

```
#include <stdio.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <limits.h>
#include <math.h>
#include "xdict/xdict.h"
#include "xstring/xstring.h"
#include "xcfg/xcfg.h"
```

Macros

- `#define MAX_CFG_LINE_LENGTH (4096)`

Functions

- `int xcfg_configure (const char *name, const char *doc, const char *defval)`
  
  Sets the documentation and default value for variable name.

- `const char * xcfg_set (const char *name, const char *value, const char *from)`
  
  Sets the value for name to value.

- `const char * xcfg_get (const char *name)`
  
  Gets the value for variable name.

- `int xcfg_get_i (const char *name)`
  
  Gets the value for variable name as an integer.

- `size_t xcfg_get_u (const char *name)`
  
  Gets the value for variable name as an unsigned integer.

- `float xcfg_get_f (const char *name)`
  
  Gets the value for variable name as a float.

- `const char * xcfg_from_env (const char *name)`
  
  Sets the variable name using the environment.

- `void xcfg_shutdown (void)`
  
  Frees all the variables and the resources they use.

- `xcfg_t ** xcfg_get_all (void)`
Returns a list of all the internal variables.

- int xcfg_from_file (char * filename)
  Sets internal configuration variables from given file.

- int xcfg_from_array (char ** argv, char * from)
  Sets internal configuration variables from given array of strings.

- int xcfg_save_config (char * filename)
  Saves the state of all the configuration variables to a config file.

### 3.1.1 Detailed Description

Configuration management routines. A set of functions to provide management of config data for C programs. This library provides routines to get config values from configuration files, the command line and environment variables, while also providing sane defaults.

Configuration is stored globally for all callers and care needs to be exercised when using these functions from different threads. All configuration data is stored as a tuple of Name, Value, Default Value, Documentation and SetFrom.

Name - The name of the configuration variable.
Value - The value that is set for the Named variable.
Default Value - The value that will be used if it is not found on the command-line, nor in a configuration file, nor in an environment variable.
Documentation - The help message that will be displayed to the user when help is requested for this Named variable.
SetFrom - A caller-supplied string describing where this variable got its value from (for example, “Environment Variable”, or “User Prompt”, or “Command Line Argument”, etc). This value can be used to tell the user exactly where a certain value came from (in case a config file overrides their command-line, or vice-versa).

xcfg is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

Author
Lelanthran Krishna Manickum

### 3.1.2 Function Documentation

#### 3.1.2.1 int xcfg.configure ( const char * name, const char * doc, const char * defval )

Sets the documentation and default value for variable name.

Sets the documentation string and the default value to be used for the variable name. If the variable doesn’t exist, it will be created and stored internally with documentation set to doc and default value set to defval.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in name</td>
<td>The variable to effect</td>
</tr>
<tr>
<td>in doc</td>
<td>The documentation string for this variable</td>
</tr>
<tr>
<td>in defval</td>
<td>The default value for this variable; a value that the variable gets if the caller never explicitly sets a value.</td>
</tr>
</tbody>
</table>

Returns

On success 0 is returned. On failure a non-zero integer is returned.

Example

```c
int main (void)
{
```
xcfg_configure("retries",
    "Number of retries to attempt before giving up."
    );
xcfg_configure("destination",
    "Server IP to connect to.",
    "127.0.0.1");

// Here we read in the actual values from a file and set them with
// xcfg_set (name, value, from) calls. If we never find them, the
// default values above will be returned when xcfg_get() is called

xcfg_shutdown (); // Free all resources for the variables
return EXIT_SUCCESS;

3.1.2.2 int xcfg_from_array ( char ** argv, char * from )

Sets internal configuration variables from given array of strings.
Sets internal configuration variables from the array of strings argv. The array argv must be terminated with a NULL pointer, as if it were declared as follows:

char *argv[] = {"--name1=value1", "name2=value2", ..., NULL};

All the variables to be set must be of the form "--name=value". If value is omitted the variable name is assigned an empty non-NULL zero-length value.

A "--" on its own causes the processing of variables to end. Strings in the array argv after the "--" string are ignored.

This function is designed to retrieve values straight out of the argv parameter that is passed to the main() function, so it is safe to pass the argv parameter to main() directly to xcfg_from_array(). This is the easiest way to retrieve command line arguments passed to the program.

Parameters

| in | argv | The array of strings containing the name/value pairs of each variable's value. |
| in | from | The identifier that is used in the SetFrom field of the variable. This need not be unique, but does, in a later part of the program, help the caller determine where a variable got its' value from. |

Returns

On success the number of variables successfully set is returned. On failure a negative number is returned.

See Also

xcfg, xcfg_set() xcfg_from_env(), xcfg_from_file()

Example program

int main (void)
{
    char *options = {
        "--one=1111",
        "--two",
        "--three=3333",
        "--four",
        "--five=5555",
        NULL,
    };
    xcfg_from_array (options, "Internal")
    ...
    xcfg_t ***all = xcfg_get_all (); // Get all the internal variables
File Documentation

```c
xcfg_t **tmp = all; // Keep a pointer to the first one to free it later
while (all && *all) {
    printf("name=%s\n", (*all)->name);
    printf("value=%s\n", (*all)->value);
    printf("doc=%s\n", (*all)->doc);
    printf("from=%s\n", (*all)->from);
    printf("default=%s\n", (*all)->defval);
    printf("==========================\n");
    all++;
}
free (tmp); // Free the returned value (but not each element)
xcfg_shutdown (); // Free all resources for the variables
return EXIT_SUCCESS;
```

The output from the above is:

```
name=one
value='1111'
doc=(null)
from=from c/line
default=(null)

name=two
value='

doc=(null)
from=from c/line
default=(null)

name=three
value='3333'
doc=(null)
from=from c/line
default=(null)

name=four
value='

doc=(null)
from=from c/line
default=(null)
```

3.1.2.3 `const char * xcfg_from_env ( const char * name )`

Sets the variable `name` using the environment.

Sets the value of `name` to a value taken out of the environment using the same name for the environment variable. For example, using "HOME" as the argument (the `name`), this function sets the internal variable to the value that the environment variable "HOME" has.

See Also

`xcfg_set(), xcfg_get()`

Parameters

| in | name | The variable to set |

Returns

On success the value that was set is returned. On failure `NULL` is returned.

Example

```c
int main (void)
```
Sets internal configuration variables from given file.

Sets internal configuration variables from the file specified by `filename`. The file is read a single line at a time, with a maximum line size of 4096 bytes. Each line is of the form "name = value". The name may be composed of spaces (leading and trailing spaces are ignored) and any character except '='. Values may contain anything except newlines (leading and trailing spaces are ignored). Empty lines are ignored. The value is optional and will revert to an empty string if no value is found after the '='. Lines without the delimiter '=' are ignored.

A comment is everything from the first hash character (#) to the end of the line. Hashes are escaped using the two-char sequence "\#". The internal variables that are set (see the main `xcfg` help page) have their `SetFrom` field set to the filename.

### Parameters

- `filename` The filename to read for name/value pairs.

### Returns

On success the number of variables successfully set is returned. On failure a negative number is returned.

### See Also

- `xcfg`, `xcfg_set()`, `xcfg_from_env()`, `xcfg_from_array()`

#### Example configuration file

```plaintext
# Begin small example configuration - Test.cfg
host retries = 3 # Name = "host retries"
example-string = a = b = c # Value = "a = b = c"
check-local= # Value = ""
check-remote # This line gets ignored
withHash = My \# String # Value = "My # String"
# End small example configuration - Test.cfg
```

#### Example program

```c
int main (void)
{
    xcfg_from_file ("Test.cfg");
    ...
    xcfg_t **all = xcfg_get_all (); // Get all the internal variables
    xcfg_t **tmp = all; // Keep a pointer to the first one to free it later
    while (all && *all) {
        printf ("name=%s
", (*all)->name);
        printf ("value=%s
", (*all)->value);
        printf ("doc=%s
", (*all)->doc);
        printf ("from=%s
", (*all)->from);
        printf ("defval=%s
", (*all)->defval);
        printf ("==========================
");
        all++;
    }
    free (tmp); // Free the returned value (but not each element)
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```
The output from the above is:

```c
name=host retries
value='3'
doc=(null)
from=t2
default=(null)

name=example-string
value='a = b = c'
doc=(null)
from=t2
default=(null)

name=check-local
value='

doc=(null)
from=t2
default=(null)

name=withHash
value='My \\# String'
doc=(null)
from=t2
default=(null)
```

3.1.2.5 `const char* xcfg_get ( const char * name )`

Gets the value for variable `name`.

Gets the value for `name`. If the `name` was never set then the value returned is the default value. If a default value does not exist then `NULL` is returned.

See Also

`xcfg_set(), xcfg_get_i(), xcfg_get_u, xcfg_get_f()`

Parameters

| in | name | The variable to read |

Returns

On success the value that was set or the default value if no value was set is returned. On failure `NULL` is returned.

Example

```c
int main (void)
{
    xcfg_set ("retries",
        "Number of retries to attempt before giving up."
        "4");
    // ...
    printf ("Retries is set to %s\n", xcfg_get ("retries");
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```

3.1.2.6 `xcfg_t* xcfg_get_all ( void )`

Returns a list of all the internal variables.
Returns a list of all the internal variables that were set by the caller. The caller has to free the returned array, but not the elements in the returned array (see example).

No parameters are taken.

See Also
xcfg

Returns

On success an array (terminated with a NULL pointer) of xcfg_t pointers is returned. While the caller has to free the array that is returned, the caller should not modify the elements of the array.

Example

```c
int main (void)
{
  xcfg_set ("retries", "Number of retries to attempt before giving up." "4");
  ...
  // Many more xcfg_set and xcfg_configure calls
  xcfg_t **all = xcfg_get_all (); // Get all the internal variables
  xcfg_t **tmp = all; // Keep a pointer to the first one to free it later
  while (all && *all)
  {
    printf ("name=%s
", (*all)->name);
    printf ("value=%s\n", (*all)->value);
    printf ("doc=%s\n", (*all)->doc);
    printf ("from=%s\n", (*all)->from);
    printf ("default=%s\n", (*all)->defval);
    all++;
  }
  free (tmp); // Free the returned value (but not each element)
  xcfg_shutdown (); // Free all resources for the variables
  return EXIT_SUCCESS;
}
```

3.1.2.7 float xcfg_get_f ( const char * name )

Gets the value for variable name as a float.

Gets the value for name, converted to a float. If the name was never set then the value returned is the default value. If a default value does not exist then NAN is returned.

See Also
xcfg_set(), xcfg_get(), xcfg_get_i()

Parameters

| in | name | The variable to read |

Returns

On success the value that was set or the default value if no value was set is returned. On failure NAN is returned.

Example

```c
int main (void)
```
3.1.2.8 int xcfg_get_i (const char * name)

Gets the value for variable name as an integer.

Gets the value for name, converted to an integer. If name was never set then the value returned is the default value. If a default value does not exist then INT_MAX is returned.

See Also
xcfg_set(), xcfg_get()

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>name</th>
<th>The variable to read</th>
</tr>
</thead>
</table>

Returns

On success the value that was set or the default value if no value was set is returned. On failure INT_MAX is returned.

Example

```c
int main (void)
{
    xcfg_set ("retries",
               "Number of retries to attempt before giving up."
               "4");
    // ...
    printf ("Retries is set to %i\n", xcfg_get_i ("retries");
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```

3.1.2.9 size_t xcfg_get_u (const char * name)

Gets the value for variable name as an unsigned integer.

Gets the value for name, converted to an unsigned integer. If name was never set then the value returned is the default value. If a default value does not exist then SIZE_MAX is returned.

See Also
xcfg_set(), xcfg_get(), xcfg_get_i()

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>name</th>
<th>The variable to read</th>
</tr>
</thead>
</table>

Example

```c
int main (void)
{
    xcfg_set ("retries",
               "Number of retries to attempt before giving up."
               "4");
    // ...
    printf ("Retries is set to %u\n", xcfg_get_u ("retries");
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```
Returns

On success the value that was set or the default value if no value was set is returned. On failure SIZE_MAX is returned.

Example

```c
int main (void)
{
    xcfg_set ("retries",
              "Number of retries to attempt before giving up."
              "4");
    // ...
    printf ("Retries is set to %zu\n", xcfg_get_u ("retries");
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```

3.1.2.10 int xcfg_save_config ( char * filename )

Saves the state of all the configuration variables to a config file.

Saves the state of all the configuration variables that are currently stored in the program to a configuration file that can be read in with xcfg_from_file(). This allows the caller to save program configuration state and reload it later, as well as generate a working configuration file from all the values read in thus far.

The variables are saved along with their description and default values (description and default values are saved as comments in the output file).

See Also

xcfg, xcfg_set() xcfg_from_env(), xcfg_from_file(), xcfg_from_array()

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>filename</th>
<th>The filename to save the configuration variables to</th>
</tr>
</thead>
</table>

Returns

On success zero is returned. On failure a negative number is returned.

Example program

```c
int main (void)
{
    char *options = {
        "--one=1111",
        "--two",
        "--three=3333",
        "--four",
        "--",
        "--five=5555",
        NULL,
    };
    xcfg_from_array (options, "Internal")
    ...
    xcfg_save_config ("config.out");
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```
3.1.2.11 const char* xcfg_set ( const char* name, const char* value, const char* from )

Sets the value for name to value.

Sets the value for name to value and the from-field for name to from. The value can later be retrieved with xcfg_get(). The from field can be retrieved along with the variables doc and default-value field.

See Also
xcfg

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>name</th>
<th>The variable to set</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>value</td>
<td>The value to assign to the variable name</td>
</tr>
<tr>
<td>in</td>
<td>from</td>
<td>A caller-specified description of where this variable got its value from.</td>
</tr>
</tbody>
</table>

Returns
On success the value that was set is returned. On failure NULL is returned.

Example

```c
int main (void)
{
    xcfg_set ("retries",  
            "Number of retries to attempt before giving up."
            "4");

    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```

3.1.2.12 void xcfg_shutdown ( void )

Frees all the variables and the resources they use.

Frees all the memory associated with all internal variables and values. This should be called before program end or after the caller is certain that none of the variables will be needed again. After this function returns all the variables that were set with xcfg_set() and friends will cease to exist.

The caller can always call xcfg_set() and friends to store a new set of internal variables. No parameters are taken.

See Also
xcfg

Returns
Nothing.

Example

```c
int main (void)
{
    xcfg_set ("retries",  
            "Number of retries to attempt before giving up."
            "4");

    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```
3.2  xcfg/xcfg.c File Reference

Configuration management routines.

```c
#include <stdio.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <limits.h>
#include <math.h>
#include "xdict/xdict.h"
#include "xstring/xstring.h"
#include "xcfg/xcfg.h"
```

Macros

- `#define MAX_CFG_LINE_LENGTH (4096)`

Functions

- `int xcfg_configure (const char *name, const char *doc, const char *defval)`
  
  Sets the documentation and default value for variable name.

- `const char * xcfg_set (const char *name, const char *value, const char *from)`
  
  Sets the value for variable name.

- `const char * xcfg_get (const char *name)`
  
  Gets the value for variable name.

- `int xcfg_get_i (const char *name)`
  
  Gets the value for variable name as an integer.

- `size_t xcfg_get_u (const char *name)`
  
  Gets the value for variable name as an unsigned integer.

- `float xcfg_get_f (const char *name)`
  
  Gets the value for variable name as a float.

- `const char * xcfg_from_env (const char *name)`
  
  Sets the variable name using the environment.

- `void xcfg_shutdown (void)`
  
  Frees all the variables and the resources they use.

- `xcfg_t ** xcfg_get_all (void)`
  
  Returns a list of all the internal variables.

- `int xcfg_from_file (char *filename)`
  
  Sets internal configuration variables from given file.

- `int xcfg_from_array (char **argv, char *from)`
  
  Sets internal configuration variables from given array of strings.

- `int xcfg_save_config (char *filename)`
  
  Saves the state of all the configuration variables to a config file.

3.2.1  Detailed Description

Configuration management routines. A set of functions to provide management of config data for C programs. This library provides routines to get config values from configuration files, the command line and environment variables, while also providing sane defaults.
Configuration is stored globally for all callers and care needs to be exercised when using these functions from
different threads. All configuration data is stored as a tuple of Name, Value, Default Value, Documentation and
SetFrom.

Name - The name of the configuration variable.
Value - The value that is set for the Named variable.
Default Value - The value that will be used if it is not found on the command-line, nor in a configuration file, nor in an
environment variable.
Documentation - The help message that will be displayed to the user when help is requested for this Named variable.
SetFrom - A caller-supplied string describing where this variable got its value from (for example, "Environment
Variable", or "User Prompt", or "Command Line Argument", etc). This value can be used to tell the user exactly
where a certain value came from (in case a config file overrides their command-line, or vice-versa).

xcfg is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

Author
Lelanthran Krishna Manickum

### 3.2.2 Function Documentation

#### 3.2.2.1 int xcfg_configure ( const char * name, const char * doc, const char * defval )

Sets the documentation and default value for variable name.

Sets the documentation string and the default value to be used for the variable name. If the variable doesn’t exist, it
will be created and stored internally with documentation set to doc and default value set to defval.

**Parameters**

- in name The variable to effect
- in doc The documentation string for this variable
- in defval The default value for this variable; a value that the variable gets if the caller
  never explicitly sets a value.

**Returns**

On success 0 is returned. On failure a non-zero integer is returned.

**Example**

```c
int main (void)
{
    xcfg_configure ("retries", "Number of retries to attempt before giving up." "4");
    xcfg_configure ("destination", "Server IP to connect to.", "127.0.0.1");

    // Here we read in the actual values from a file and set them with
    // xcfg_set (name, value, from) calls. If we never find them, the
    // default values above will be returned when xcfg_get() is called
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```

#### 3.2.2.2 int xcfg_from_array ( char ** argv, char * from )

Sets internal configuration variables from given array of strings.
Sets internal configuration variables from the array of strings argv. The array argv must be terminated with a NULL pointer, as if it were declared as follows:

```c
char *argv[] = {"--name1=value1", "name2=value2", ..., NULL};
```

All the variables to be set must be of the form "--name=value". If value is omitted the variable name is assigned an empty non-NULL zero-length value.

A "--" on its own causes the processing of variables to end. Strings in the array argv after the "--" string are ignored.

This function is designed to retrieve values straight out of the argv parameter that is passed to the main() function, so it is safe to pass the argv parameter to main() directly to xcfg_from_array(). This is the easiest way to retrieve command line arguments passed to the program.

### Parameters

| in | argv | The array of strings containing the name/value pairs of each variable’s value. |
| in | from | The identifier that is used in the SetFrom field of the variable. This need not be unique, but does, in a later part of the program, help the caller determine where a variable got its’ value from. |

### Returns

On success the number of variables successfully set is returned. On failure a negative number is returned.

### See Also

xcfg, xcfg_set(), xcfg_from_env(), xcfg_from_file()

### Example program

```c
int main (void)
{
    char *options = {
        "--one=1111",
        "--two",
        "--three=3333",
        "--four",
        "--",
        "--five=5555",
        NULL,
    };
    xcfg_from_array (options, "Internal")
    ...
    xcfg_t **all = xcfg_get_all (); // Get all the internal variables
    xcfg_t **tmp = all; // Keep a pointer to the first one to free it later
    while (all && *all) {
        printf ("name=%s
", (*all)->name);
        printf ("value=%s
", (*all)->value);
        printf ("doc=%s
", (*all)->doc);
        printf ("from=%s
", (*all)->from);
        printf ("default=%s
", (*all)->defval);
        printf ("=---------------------------------------------\n"
        all++;
    }
    free (tmp); // Free the returned value (but not each element)
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```

The output from the above is:

```
name=one
value='1111'
doc=(null)
```
3.2.2.3 const char * xcfg_from_env ( const char * name )

Sets the variable name using the environment.

Sets the value of name to a value taken out of the environment using the same name for the environment variable. For example, using "HOME" as the argument (the name), this function sets the internal variable to the value that the environment variable "HOME" has.

See Also

xcfg_set(), xcfg_get()

Parameters

| in   | name | The variable to set |

Returns

On success the value that was set is returned. On failure NULL is returned.

Example

```c
int main (void)
{
    xcfg_from_env ("PATH");
    // ...
    printf ("Path is set to %s\n", xcfg_get ("PATH"));
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```

3.2.2.4 int xcfg_from_file ( char * filename )

Sets internal configuration variables from given file.

Sets internal configuration variables from the file specified by filename. The file is read a single line at a time, with a maximum line size of 4096 bytes. Each line is of the form "name = value". The name may be composed of spaces (leading and trailing spaces are ignored) and any character except '='. Values may contain anything except newlines (leading and trailing spaces are ignored). Empty lines are ignored. The value is optional and will revert to an empty string if no value is found after the '='. Lines without the delimiter '=' are ignored.
A comment is everything from the first hash character (#) to the end of the line. Hashes are escaped using the two-char sequence "\#". The internal variables that are set (see the main xcfg help page) have their SetFrom field set to the filename.

### Parameters

<table>
<thead>
<tr>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>filename</code></td>
<td>The filename to read for name/value pairs.</td>
</tr>
</tbody>
</table>

### Returns

On success the number of variables successfully set is returned. On failure a negative number is returned.

### See Also

xcfg, xcfg_set(), xcfg_from_env(), xcfg_from_array()

### Example configuration file

```plaintext
# Begin small example configuration - Test.cfg
host retries = 3 # Name = "host retries"
example-string = a = b = c # Value = "a = b = c"
check-local= # Value = ""
check-remote # This line gets ignored
withHash = My \# String # Value = "My \# String"
# End small example configuration - Test.cfg
```

### Example program

```c
int main (void)
{
    xcfg_from_file ("Test.cfg");
    ...
    xcfg_t ***all = xcfg_get_all (); // Get all the internal variables
    xcfg_t **tmp = all; // Keep a pointer to the first one to free it later
    while (all && *all) {
        printf ("name=%s\n", (*all)->name);
        printf ("value=%s\n", (*all)->value);
        printf ("doc=%s\n", (*all)->doc);
        printf ("from=%s\n", (*all)->from);
        printf ("default=%s\n", (*all)->defval);
        printf (""empty string"\n");
        all++;
    }
    free (tmp); // Free the returned value (but not each element)
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```

The output from the above is:

```
name=host retries
value='3'
doc=(null)
from=t2
default=(null)
==========================
name=example-string
value='a = b = c'
doc=(null)
from=t2
default=(null)
==========================
name=check-local
value=''
doc=(null)
from=t2
```
3.2.2.5 `const char* xcfg_get ( const char * name )`

Gets the value for variable `name`.

Gets the value for `name`. If the `name` was never set then the value returned is the default value. If a default value does not exist then `NULL` is returned.

See Also

`xcfg_set(), xcfg_get_i(), xcfg_get_u, xcfg_get_f()`

Parameters

| in  | name | The variable to read |

Returns

On success the value that was set or the default value if no value was set is returned. On failure `NULL` is returned.

Example

```c
int main (void)
{
    xcfg_set ("retries",
              "Number of retries to attempt before giving up."
             "4");
    // ...
    printf ("Retries is set to %s\n", xcfg_get ("retries"));
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```

3.2.2.6 `xcfg_t** xcfg_get_all ( void )`

Returns a list of all the internal variables.

Returns a list of all the internal variables that were set by the caller. The caller has to free the returned array, but not the elements in the returned array (see example).

No parameters are taken.

See Also

`xcfg`

Returns

On success an array (terminated with a NULL pointer) of xcfg_t pointers is returned. While the caller has to free the array that is returned, the caller should not modify the elements of the array.

Example
int main (void)
{
    xcfg_set ("retries",
              "Number of retries to attempt before giving up."
              "4");
    ...
    // Many more xcfg_set and xcfg_configure calls ...
    xcfg_t **all = xcfg_get_all (); // Get all the internal variables
    xcfg_t **tmp = all; // Keep a pointer to the first one to free it later
    while (all && *all) {
        printf ("name=%s\n", (*all)->name);
        printf ("value=%s\n", (*all)->value);
        printf ("doc=%s\n", (*all)->doc);
        printf ("from=%s\n", (*all)->from);
        printf ("default=%s\n", (*all)->defval);
        all++;
    }
    free (tmp); // Free the returned value (but not each element)
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}

3.2.2.7 float xcfg_get_f ( const char ∗ name )

Gets the value for variable name as a float.

Gets the value for name, converted to a float. If the name was never set then the value returned is the default value. If a default value does not exist then NaN is returned.

See Also
xcfg_set(), xcfg_get(), xcfg_get_i()

Parameters

<table>
<thead>
<tr>
<th></th>
<th>name</th>
<th>The variable to read</th>
</tr>
</thead>
</table>

Returns

On success the value that was set or the default value if no value was set is returned. On failure NaN is returned.

Example

int main (void)
{
    xcfg_set ("alpha",
              "Translucency, between 0.0 and 1.0."
              "0.54231");
    // ...
    printf ("Retries is set to %f\n", xcfg_get_f ("alpha");
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}

3.2.2.8 int xcfg_get_i ( const char ∗ name )

Gets the value for variable name as an integer.

Gets the value for name, converted to an integer. If name was never set then the value returned is the default value. If a default value does not exist then INT_MAX is returned.
See Also
xcfg_set(), xcfg_get()

Parameters

| in | name | The variable to read |

Returns

On success the value that was set or the default value if no value was set is returned. On failure INT_MAX is returned.

Example

```c
int main (void)
{
    xcfg_set ("retries",
    "Number of retries to attempt before giving up."
    "4");
    // ...
    printf ("Retries is set to %i\n", xcfg_get_i ("retries");
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```

3.2.2.9  size_t xcfg_get_u ( const char * name )

Gets the value for variable name as an unsigned integer.

Gets the value for name, converted to an unsigned integer. If name was never set then the value returned is the default value. If a default value does not exist then SIZE_MAX is returned.

See Also
xcfg_set(), xcfg_get(), xcfg_get_i()  

Parameters

| in | name | The variable to read |

Returns

On success the value that was set or the default value if no value was set is returned. On failure SIZE_MAX is returned.

Example

```c
int main (void)
{
    xcfg_set ("retries",
    "Number of retries to attempt before giving up."
    "4");
    // ...
    printf ("Retries is set to %zu\n", xcfg_get_u ("retries");
    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```
3.2.2.10 int xcfg_save_config ( char * filename )

Saves the state of all the configuration variables to a config file.

Saves the state of all the configuration variables that are currently stored in the program to a configuration file that can be read in with xcfg_from_file(). This allows the caller to save program configuration state and reload it later, as well as generate a working configuration file from all the values read in thus far.

The variables are saved along with their description and default values (description and default values are saved as comments in the output file).

See Also

xcfg, xcfg_set(), xcfg_from_env(), xcfg_from_file(), xcfg_from_array()

Parameters

| in  | filename | The filename to save the configuration variables to |

Returns

On success zero is returned. On failure a negative number is returned.

Example program

```c
int main (void)
{
    char *options = {
        "--one=1111",
        "--two",
        "--three=3333",
        "--four",
        "--",
        "--five=5555",
        NULL,
    };
    xcfg_from_array (options, "Internal")
    ...
    xcfg_save_config ("config.out");
    ...
    xcfgShutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```

3.2.2.11 const char* xcfg_set ( const char * name, const char * value, const char * from )

Sets the value for name to value.

Sets the value for name to value and the from-field for name to from. The value can later be retrieved with xcfg_get(). The from field can be retrieved along with the variables doc and default-value field.

See Also

xcfg

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>name</th>
<th>The variable to set</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>value</td>
<td>The value to assign to the variable name</td>
</tr>
<tr>
<td>in</td>
<td>from</td>
<td>A caller-specified description of where this variable got its value from.</td>
</tr>
</tbody>
</table>
Returns

On success the value that was set is returned. On failure NULL is returned.

Example

```c
int main (void)
{
    xcfg_set ("retries",
             "Number of retries to attempt before giving up."
             "4");

    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```

3.2.2.12 void xcfg_shutdown ( void )

Frees all the variables and the resources they use.

Frees all the memory associated with all internal variables and values. This should be called before program end or after the caller is certain that none of the variables will be needed again. After this function returns all the variables that were set with xcfg_set() and friends will cease to exist.

The caller can always call xcfg_set() and friends to store a new set of internal variables. No parameters are taken.

See Also

xcfg

Returns

Nothing.

Example

```c
int main (void)
{
    xcfg_set ("retries",
             "Number of retries to attempt before giving up."
             "4");

    xcfg_shutdown (); // Free all resources for the variables
    return EXIT_SUCCESS;
}
```

3.3 libxc-0.0.4/xdict/xdict.c File Reference

Implementation of dictionary (associative arrays)

```c
#include <stdlib.h>
#include <string.h>
#include "xvector/xvector.h"
#include "xdict/xdict.h"
```

Functions

- `xdict_t * xdict_new (int(*cmp)(void *, void *), void(*)(cpy)(void *), void(*)(del)(void *))`

  Create a new dictionary.
### Detailed Description

Implementation of dictionary (associative arrays) Implementation of a dictionary, also known as an associative array, container for C. The caller has to set the hash copy, delete and compare functions when creating the dictionary. If the width of the hash used is less than a pointer width, then the caller can simply set those values to `NULL`, in which case the copy function is bitwise assignment, the delete function is ignored and the compare function is bitwise comparison.

See the examples for each of the functions in this module for more detailed explanation.

xvector is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

**Author**

Lelanthran Krishna Manickum

### Function Documentation

#### 3.3.2.1 void xdict_del ( xdict_t *dict )

Deletes a dictionary and all its entries.

Deletes the specified dictionary/associative array, and frees all the resources that it was using. Note that the caller has to free all the values stored before this function is run to avoid leaking memory.

See the function `xdict_new()` for more information on how this function deletes a dictionary and examples/

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dict</td>
<td>xdict_t</td>
<td>The dictionary to delete</td>
</tr>
</tbody>
</table>

**Returns**

Nothing.

#### 3.3.2.2 void xdict_get ( xdict_t *dict, void *index )

Get the value for a given name/hash in the dictionary.

Retrieves the value of `index` to `element` in the dictionary `dict`.

For names/indices that are of a type that is less than or equal to a pointer width, the caller can simply cast each name on insert to `void *` and use `NULL` for all the `xdict_new()` arguments. If this is done then the caller must cast back when retrieving the name (see example).

For values that are of a type that is less than or equal to a pointer width, the caller can simply cast each value to a `void *` on insert. If this is done then the caller must cast back to the original type when retrieving the value (see example).
Parameters

| in | dict | The dictionary to use |
| in | index | The index identifying the element to return to the caller |

Returns

On success a pointer to the element identified by `index` is returned. On failure or if the element is not found `NULL` is returned.

Example

// The following three static functions are for example two below

static int dict_string_cmp (void *a1, void *a2)
{
    return strcmp ((char *)a1, (char *)a2);
}

static void *dict_string_cpy (void *s)
{
    return xstr_dup (s); // duplicate the string
}

static void dict_string_del (void *s)
{
    if (s) free (s);
}

int main (void)
{
    // First example: using hash entries with each name less
    // than a pointer width
    // All NULL's indicating that datatype for hash is less
    // than a pointer width
    xdict_t *hashmap = xdict_new (NULL, NULL, NULL);

    // Our hashmap will use ints for name and chars for values
    int v1[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    char v2[] = "1234567890";

    // Set all the dictionary entries
    for (size_t i=0; i < sizeof v1/sizeof v1[0]; i++) {
        xdict_set (hashmap, (void *)v1[i], (void *)v2[i]);
    }

    // Check that they are all set properly
    for (size_t i=sizeof v1/sizeof v1[0]; i>0; i--) {
        int index = v1[i-1];
        char value = (char)xdict_get (hashmap, (void *)index);
        printf ("For name=%i, got value '%c'\n", index, value);
    }

    // Free the map and all entries it holds
    xdict_del (hashmap);

    // Second example: Using custom comparison, copy and deletion
    // functions that uses strings as the hash/name and ints as the
    // value
    hashmap = xdict_new (dict_string_cmp,
                        dict_string_cpy,
                        dict_string_del);

    char *v4[] = {"one", "two", "three", "four", "five",
                  "six", "seven");

    // Add in dictionary entries using strings as the hashes/names and
    // ints as the value
    for (size_t i=0; i<sizeof v4/sizeof v4[0]; i++) {
        xdict_set (hashmap, v4[i], (void *)v1[i]);
    }

    // Print it all out to make sure that they got set
    for (size_t i=sizeof v4/sizeof v4[0]; i>0; i--) {
        char *index = v4[i-1];
        int value = (char)xdict_get (hashmap, index);
        printf ("For name=%s, got value '%i'\n", index, value);
3.3.2.3  void xdict_iterate ( xdict_t * dict, void (*)(void *, void *) fptr )

Iterate over all entries in the dictionary.

Iterates over all entries in the dictionary dict, calling the function fptr() for each tuple of name, value in the dictionary.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dict</td>
<td>The dictionary to iterate over</td>
</tr>
<tr>
<td>fptr</td>
<td>The function to call on each name/value pair in the dictionary dict</td>
</tr>
</tbody>
</table>

Returns

Nothing.

Example

// The following three static functions are for example two below

static int dict_string_cmp (void *a1, void *a2)
{  
    return strcmp ((char *)a1, (char *)a2);
}

static void *dict_string_cpy (void *s)
{  
    return xstrDup (s); // duplicate the string
}

static void dict_string_del (void *s)
{  
    if (s) free (s);
}

static void dict_string_int_print (void *n, void *v)
{  
    printf ("name='%s', value='%i'
", (char *)n, (int)v);
}

int main (void)
{  
    hashmap = xdict_new (dict_string_cmp,
                         dict_string_cpy,
                         dict_string_del);

    char *v4[] = {"one", "two", "three", "four", "five", "six", "seven"};
    int v1[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

    // Add in dictionary entries using strings as the hashes/names and
    // ints as the value
    for (size_t i=0; i<sizeof v4/sizeof v4[0]; i++)
        xdict_set (hashmap, v4[i], (void *)v1[i]);

    // Print it all out to make sure that they got set
    xdict_iterate (xtmp, dict_string_int_print);

    // Delete the hash and all it’s entries
    xdict_del (hashmap);

    return EXIT_SUCCESS;
}
3.3.2.4 \texttt{void \*xdict\_map ( xdict\_t \*\texttt{dict}, void \*\texttt{\textdagger}(void \*, void \*) \texttt{fptr} )}

Map a predicate over all the entries in the dictionary.

Iterates over all entries in the dictionary \texttt{dict}, calling the function \texttt{fptr()} for each tuple of \texttt{name}, \texttt{value} in the dictionary and storing the results in a null-terminated array of \texttt{void \*} which is returned to the caller.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{in} \texttt{\textdagger} \texttt{dict}</td>
<td>The dictionary to iterate over</td>
</tr>
<tr>
<td>\texttt{in} \texttt{\textdagger} \texttt{fptr}</td>
<td>The function to call on each name/value pair in the dictionary \texttt{dict}</td>
</tr>
</tbody>
</table>

Returns

An array of \texttt{void \*} containing the results of each invocation of \texttt{fptr()} using the entry's \texttt{name/value} pair.

Example

// The following three static functions are for example two below

\begin{verbatim}
static int dict_string_cmp (void *a1, void *a2)
{
    return strcmp ((char *)a1, (char *)a2);
}

static void *dict_string_cpy (void *s)
{
    return xstr_dup (s); // duplicate the string
}

static void dict_string_del (void *s)
{
    if (s) free (s);
}

static void *dict_string_int_map (void *n, void *v)
{
    return xstr_dup ((char *)n);
}

int main (void)
{
    hashmap = xdict_new (dict_string_cmp,
                        dict_string_cpy,
                        dict_string_del);
    char *v4[] = {"one", "two", "three", "four", "five",
                  "six", "seven"};
    int v1[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    // Add in dictionary entries using strings as the hashes/names and
    // ints as the value
    for (size_t i=0; i<sizeof v4/sizeof v4[0]; i++) {
        xdict_set (hashmap, v4[i], (void *)v1[i]);
    }
    // Print all the names in the dictionary
    char **results = xdict_map (xtmp, dict_string_int_print);
    char **tmp = results;
    while (*tmp) {
        printf ("%s\n", *tmp);
        free (*tmp);
        tmp++;
    }
    // Delete the hash and all it's entries
    xdict_del (hashmap);
    return EXIT_SUCCESS;
\end{verbatim}
3.3.2.5  xdict_t* xdict_new ( int (*)(void*, void*) cmp, void (*)(void*) cpy, void (*)(void*) del )

Create a new dictionary.

Creates a new dictionary. An array of name/value pairs are stored as the dictionary. The three functions cmp, cpy
and del are the functions that the dictionary will use to compare, copy and delete the names in the array. The values
to be stored are caller-allocated pointers.

For names that are of a type that is less than or equal to a pointer width, the caller can simply cast each name on
insert to void* and use NULL for all the xdict_new() arguments. If this is done then the caller must cast back when
retrieving the name (see example).

For values that are of a type that is less than or equal to a pointer width, the caller can simply cast each value to
a void* on insert. If this is done then the caller must cast back to the original type when retrieving the value (see
example).

Parameters

| in | cmp | The comparison function that the dictionary will use to compare the hash/names of any entry in the dictionary. |
| in | cpy | The copying function that the dictionary will use when copying the hash/names of any entry in the dictionary. |
| in | del | The deletion function that the dictionary will use when removing the hash/names of any entry in the dictionary. |

Returns

On success an xdict_t* is returned for subsequent use with xdict_set() and xdict_get(). The caller must free this
value at a later time using xdict_del(). On failure NULL is returned.

Example

// The following three static functions are for example two below

static int dict_string_cmp (void *a1, void *a2)
{
    return strcmp ((char *)a1, (char *)a2);
}

static void *dict_string_cpy (void *s)
{
    return xstr_dup (s); // duplicate the string
}

static void dict_string_del (void *s)
{
    if (s) free (s);
}

int main (void)
{
    // First example: using hash entries with each name less
    // than a pointer width
    // All NULL's indicating that datatype for hash is less
    // than a pointer width
    xdict_t *hashmap = xdict_new (NULL, NULL, NULL);

    // Our hashmap will use ints for name and chars for values
    int v1[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    char v2[] = "1234567890";

    // Set all the dictionary entries
    for (size_t i=0; i < sizeof v1/sizeof v1[0]; i++) {
        xdict_set (hashmap, (void*)v1[i], (void*)v2[i]);
    }

    // Check that they are all set properly
    for (size_t i=sizeof v1/sizeof v1[0]; i>0; i--) {
        int index = v1[i-1];
    }
char value = (char)xdict_get (hashmap, (void *)index);
printf ("For name=%li, got value '%c'
", index, value);
}
// Free the map and all entries it holds
xdict_del (hashmap);

// Second example: Using custom comparison, copy and deletion
// functions that uses strings as the hash/name and ints as the
// value
hashmap = xdict_new (dict_string_cmp,
    dict_string_cpy,
    dict_string_del);
char *v4[] = {"one", "two", "three", "four", "five",
    "six", "seven");
// Add in dictionary entries using strings as the hashes/names and
// ints as the value
for (size_t i=0; i<sizeof v4/sizeof v4[0]; i++) {
    xdict_set (hashmap, v4[i], (void *)v1[i]);
}

// Print it all out to make sure that they got set
for (size_t i=sizeof v4/sizeof v4[0]; i>0; i--) {
    char *index = v4[i-1];
    int value = (char)xdict_get (hashmap, index);
    printf("For name=%s, got value '%i'
", index, value);
}
// Delete the hash and all it's entries
xdict_del (hashmap);
return EXIT_SUCCESS;

3.3.2.6 void xdict_set ( xdict_t *dict, void *index, void *element )

Set the value for a given name/hash in the dictionary.

Sets the value of index to element in the dictionary dict. If the element at index index already exists, it will be reset
to the element given by the caller. A new copy of index is stored in the dictionary.

For names/indices that are of a type that is less than or equal to a pointer width, the caller can simply cast each
name on insert to void * and use NULL for all the xdict_new() arguments. If this is done then the caller must cast
back when retrieving the name (see example).

For values that are of a type that is less than or equal to a pointer width, the caller can simply cast each value to
a void * on insert. If this is done then the caller must cast back to the original type when retrieving the value (see
example).

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>dict</th>
<th>The dictionary to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>index</td>
<td>The index of the entry in the dictionary</td>
</tr>
<tr>
<td>in</td>
<td>element</td>
<td>A pointer to the caller-allocated element to store for entry index in the dictionary</td>
</tr>
<tr>
<td></td>
<td>dict</td>
<td></td>
</tr>
</tbody>
</table>

Returns

On success a pointer to the existing element is returned. On failure NULL is returned.

Example

// The following three static functions are for example two below
static int dict_string_cmp (void *a1, void *a2)
{
    return strcmp ((char *)a1, (char *)a2);
}
static void *dict_string_cpy (void *s) {
    return xstr_dup (s); // duplicate the string
}

static void dict_string_del (void *s) {
    if (s) free (s);
}

int main (void) {
    // First example: using hash entries with each name less
    // than a pointer width
    // All NULL's indicating that datatype for hash is less
    // than a pointer width
    xdict_t *hashmap = xdict_new (NULL, NULL, NULL);

    // Our hashmap will use ints for name and chars for values
    int v1[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    char v2[] = "1234567890";

    // Set all the dictionary entries
    for (size_t i=0; i < sizeof v1/sizeof v1[0]; i++) {
        xdict_set (hashmap, (void *)v1[i], (void *)v2[i]);
    }

    // Check that they are all set properly
    for (size_t i=sizeof v1/sizeof v1[0]; i>0; i--) {
        int index = v1[i-1];
        char value = (char)xdict_get (hashmap, (void *)index);
        printf ("For name=%i, got value '%c'\n", index, value);
    }

    // Free the map and all entries it holds
    xdict_del (hashmap);

    // Second example: Using custom comparison, copy and deletion
    // functions that uses strings as the hash/name and ints as the
    // value
    hashmap = xdict_new (dict_string_cmp,
                        dict_string_cpy,
                        dict_string_del);
    char *v4[] = {"one", "two", "three", "four", "five",
                  "six", "seven"};

    // Add in dictionary entries using strings as the hashes/names and
    // ints as the value
    for (size_t i=0; i < sizeof v4/sizeof v4[0]; i++) {
        xdict_set (hashmap, v4[i], (void *)v1[i]);
    }

    // Print it all out to make sure that they got set
    for (size_t i=sizeof v4/sizeof v4[0]; i>0; i--) {
        char *index = v4[i-1];
        int value = (char)xdict_get (hashmap, index);
        printf ("For name=%s, got value '%i'\n", index, value);
    }

    // Delete the hash and all it's entries
    xdict_del (hashmap);
    return EXIT_SUCCESS;
}
Functions

- xdict_t * xdict_new (int(*cmp)(void *, void *), void *(*cpy)(void *), void(*del)(void *))
  Create a new dictionary.
- void xdict_del (xdict_t *dict)
  Deletes a dictionary and all it's entries.
- void * xdict_set (xdict_t *dict, void *index, void *element)
  Set the value for a given name/hash in the dictionary.
- void * xdict_get (xdict_t *dict, void *index)
  Get the value for a given name/hash in the dictionary.
- void xdict_iterate (xdict_t *dict, void(*fptr)(void *, void *))
  Iterate over all entries in the dictionary.
- void * xdict_map (xdict_t *dict, void(*fptr)(void *, void *))
  Map a predicate over all the entries in the dictionary.

3.4.1 Detailed Description

Implementation of dictionary (associative arrays) Implementation of a dictionary, also known as an associative array, container for C. The caller has to set the hash copy, delete and compare functions when creating the dictionary. If the width of the hash used is less than a pointer width, then the caller can simply set those values to NULL, in which case the copy function is bitwise assignment, the delete function is ignored and the compare function is bitwise comparison.

See the examples for each of the functions in this module for more detailed explanation.

xvector is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

Author

Lelanthran Krishna Manickum

3.4.2 Function Documentation

3.4.2.1 void xdict_del ( xdict_t * dict )

Deletes a dictionary and all it's entries.

Deletes the specified dictionary/associative array, and frees all the resources that it was using. Note that the caller has to free all the values stored before this function is run to avoid leaking memory.

See the function xdict_new() for more information on how this function deletes a dictionary and examples/

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>dict</td>
</tr>
</tbody>
</table>

Returns

Nothing.

3.4.2.2 void * xdict_get ( xdict_t * dict, void * index )

Get the value for a given name/hash in the dictionary.

Retrieves the value of index to element in the dictionary dict.

For names/indices that are of a type that is less than or equal to a pointer width, the caller can simply cast each
name on insert to void* and use NULL for all the xdict_new() arguments. If this is done then the caller must cast back when retrieving the name (see example).

For values that are of a type that is less than or equal to a pointer width, the caller can simply cast each value to a void* on insert. If this is done then the caller must cast back to the original type when retrieving the value (see example).

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>dict</th>
<th>The dictionary to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>index</td>
<td>The index identifying the element to return to the caller</td>
</tr>
</tbody>
</table>

Returns

On success a pointer to the element identified by index is returned. On failure or if the element is not found NULL is returned.

Example

// The following three static functions are for example two below

static int dict_string_cmp (void *a1, void *a2)
{
    return strcmp ((char *)a1, (char *)a2);
}

static void *dict_string_cpy (void *s)
{
    return xstr_dup (s); // duplicate the string
}

static void dict_string_del (void *s)
{
    if (s) free (s);
}

int main (void)
{
    // First example: using hash entries with each name less
    // than a pointer width
    // All NULL's indicating that datatype for hash is less
    // than a pointer width
    xdict_t *hashmap = xdict_new (NULL, NULL, NULL);

    // Our hashmap will use ints for name and chars for values
    int v1[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    char v2[] = "1234567890";

    // Set all the dictionary entries
    for (size_t i=0; i < sizeof v1/sizeof v1[0]; i++) {
        xdict_set (hashmap, (void *)v1[i], (void *)v2[i]);
    }

    // Check that they are all set properly
    for (size_t i=0; i < sizeof v1/sizeof v1[0]; i++) {
        int index = v1[i];
        char value = (char)xdict_get (hashmap, (void *)v1[i]);
        printf ("For name=%d, got value '%c'
", index, value);
    }

    // Free the map and all entries it holds
    xdict_del (hashmap);

    // Second example: Using custom comparison, copy and deletion
    // functions that uses strings as the hash/name and ints as the
    // value
    hashmap = xdict_new (dict_string_cmp,
                        dict_string_cpy,
                        dict_string_del);
    char *v4[] = {"one", "two", "three", "four", "five",
                  "six", "seven"};
    // Add in dictionary entries using strings as the hashes/names and
// ints as the value
for (size_t i=0; i<sizeof v4/sizeof v4[0]; i++) {
    xdict_set (hashmap, v4[i], (void *)v1[i]);
}

// Print it all out to make sure that they got set
for (size_t i=sizeof v4/sizeof v4[0]; i>0; i--) {
    char *index = v4[i-1];
    int value = (char)xdict_get (hashmap, index);
    printf ("For name=%s, got value '%i',\n", index, value);
}

// Delete the hash and all it's entries
xdict_del (hashmap);
return EXIT_SUCCESS;

3.4.2.3 void xdict_iterate ( xdict_t *dict, void(*)(void *, void *) fptr )

Iterate over all entries in the dictionary.

Iterates over all entries in the dictionary dict, calling the function fptr() for each tuple of name, value in the dictionary.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>dict</th>
<th>The dictionary to iterate over</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>fptr</td>
<td>The function to call on each name/value pair in the dictionary dict</td>
</tr>
</tbody>
</table>

Returns

Nothing.

Example

// The following three static functions are for example two below

static int dict_string_cmp (void *a1, void *a2) {
    return strcmp ((char *)a1, (char *)a2);
}

static void *dict_string_cpy (void *s) {
    return xstr_dup (s); // duplicate the string
}

static void dict_string_del (void *s) {
    if (s) free (s);
}

static void dict_string_int_print (void *n, void *v) {
    printf ("name='%s', value='%i',\n", (char *)n, (int)v);
}

int main (void) {
    hashmap = xdict_new (dict_string_cmp,
                         dict_string_cpy,
                         dict_string_del);
    char *v4[] = {"one", "two", "three", "four", "five", "six", "seven"};
    int v1[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    // Add in dictionary entries using strings as the hashes/names and
    // ints as the value
    for (size_t i=0; i<sizeof v4/sizeof v4[0]; i++) {
        xdict_set (hashmap, v4[i], (void *)v1[i]);
    }
}
3.4.2.4 void∗ xdict_map ( xdict_t ∗dict, void ∗(void ∗, void ∗) fptr )

Map a predicate over all the entries in the dictionary.

Iterates over all entries in the dictionary dict, calling the function fptr() for each tuple of name, value in the dictionary and storing the results in a null-terminated array of void ∗ which is returned to the caller.

Parameters

| in | dict | The dictionary to iterate over |
| in | fptr | The function to call on each name/value pair in the dictionary dict |

Returns

An array of void ∗ containing the results of each invocation of fptr() using the entry’s name/value pair.

Example

// The following three static functions are for example two below

static int dict_string_cmp (void ∗a1, void ∗a2)
{ return strcmp ((char ∗)a1, (char ∗)a2); }

static void ∗dict_string_cpy (void ∗s)
{ return xstr_dup (s); // duplicate the string }

static void dict_string_del (void ∗s)
{ if (s) free (s); }

static void ∗dict_string_int_map (void ∗n, void ∗v)
{ return xstr_dup ((char ∗)n); }

int main (void)
{ hashmap = xdict_new (dict_string_cmp,
   dict_string_cpy,
   dict_string_del);
   char ∗v4[] = {"one", "two", "three", "four", "five", "six", "seven"};
   int v1[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};

   // Add in dictionary entries using strings as the hashes/names and
   // ints as the value
   for (size_t i=0; i<sizeof v4/sizeof v4[0]; i++) 
     xdict_set (hashmap, v4[i], (void ∗)v1[i]);

   // Print all the names in the dictionary
   char ∗∗results = xdict_map (xtmp, dict_string_int_print);
   char ∗∗tmp = results;
   while (∗tmp) 

3.4.2.5 \texttt{xdict} 

Create a new dictionary.

Creates a new dictionary. An array of name/value pairs are stored as the dictionary. The three functions \texttt{cmp}, \texttt{cpy} and \texttt{del} are the functions that the dictionary will use to compare, copy and delete the names in the array. The values to be stored are caller-allocated pointers.

For names that are of a type that is less than or equal to a pointer width, the caller can simply cast each name on insert to \texttt{void *} and use \texttt{NULL} for all the \texttt{xdict.new()} arguments. If this is done then the caller must cast back when retrieving the name (see example).

For values that are of a type that is less than or equal to a pointer width, the caller can simply cast each value to a \texttt{void *} on insert. If this is done then the caller must cast back to the original type when retrieving the value (see example).

\begin{center}
\begin{tabular}{|l|l|l|}
\hline
\textbf{Parameter} & \textbf{Name} & \textbf{Description} \\
\hline
\texttt{cmp} & \texttt{cmp} & The comparison function that the dictionary will use to compare the hash/names of any entry in the dictionary. \\
\texttt{cpy} & \texttt{cpy} & The copying function that the dictionary will use when copying the hash/names of any entry in the dictionary. \\
\texttt{del} & \texttt{del} & The deletion function that the dictionary will use when removing the hash/names of any entry in the dictionary. \\
\hline
\end{tabular}
\end{center}

Returns

On success an \texttt{xdict.t *} is returned for subsequent use with \texttt{xdict.set()} and \texttt{xdict.get}. The caller must free this value at a later time using \texttt{xdict.del()}. On failure \texttt{NULL} is returned.

Example

\begin{verbatim}
// The following three static functions are for example two below
static int dict_string_cmp (void *a1, void *a2)
{  return strcmp ((char *)a1, (char *)a2);
}
static void *dict_string_cpy (void *s)
{  return xstr_dup (s); // duplicate the string
}
static void dict_string_del (void *s)
{  if (s) free (s);
}

int main (void)
{  // First example: using hash entries with each name less
   // than a pointer width
   // All NULL's indicating that datatype for hash is less
   // Delete the hash and all it's entries
   xdict_del (hashmap);
   return EXIT_SUCCESS;
}
\end{verbatim}
```c
// than a pointer width
xdict_t *hashmap = xdict_new (NULL, NULL, NULL);

// Our hashmap will use ints for name and chars for values
int v1[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
char v2[] = "1234567890";

// Set all the dictionary entries
for (size_t i=0; i < sizeof v1/sizeof v1[0]; i++) {
    xdict_set (hashmap, (void *)v1[i], (void *)v2[i]);
}
// Check that they are all set properly
for (size_t i=sizeof v1/sizeof v1[0]; i>0; i--) {
    int index = v1[i-1];
    char value = (char)xdict_get (hashmap, (void *)index);
    printf ("For name=%i, got value '%c'\n", index, value);
}
// Free the map and all entries it holds
xdict_del (hashmap);

// Second example: Using custom comparison, copy and deletion
// functions that uses strings as the hash/name and ints as the
// value
hashmap = xdict_new (dict_string_cmp,
    dict_string_cpy,
    dict_string_del);
char *v4[] = {"one", "two", "three", "four", "five",
    "six", "seven");
// Add in dictionary entries using strings as the hashes/names and
// ints as the value
for (size_t i=0; i<sizeof v4/sizeof v4[0]; i++) {
    xdict_set (hashmap, v4[i], (void *)v1[i]);
}
// Print it all out to make sure that they got set
for (size_t i=sizeof v4/sizeof v4[0]; i>0; i--) {
    char *index = v4[i-1];
    int value = (char)xdict_get (hashmap, index);
    printf ("For name=%s, got value '%i'\n", index, value);
}
// Delete the hash and all it’s entries
xdict_del (hashmap);

return EXIT_SUCCESS;
```

### 3.4.2.6 `void* xdict_set ( xdict_t *dict, void *index, void *element )`

Set the value for a given name/hash in the dictionary.

Sets the value of `index` to `element` in the dictionary `dict`. If the element at index `index` already exists, it will be reset to the `element` given by the caller. A new copy of `index` is stored in the dictionary.

For names/indices that are of a type that is less than or equal to a pointer width, the caller can simply cast each name on insert to `void *` and use `NULL` for all the `xdict_new()` arguments. If this is done then the caller must cast back when retrieving the name (see example).

For values that are of a type that is less than or equal to a pointer width, the caller can simply cast each value to a `void *` on insert. If this is done then the caller must cast back to the original type when retrieving the value (see example).

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th><code>dict</code></th>
<th>The dictionary to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td><code>index</code></td>
<td>The index of the entry in the dictionary</td>
</tr>
<tr>
<td>in</td>
<td><code>element</code></td>
<td>A pointer to the caller-allocated element to store for entry <code>index</code> in the dictionary</td>
</tr>
</tbody>
</table>

Generated on Wed Apr 17 2013 18:38:19 for Extended C Library by Doxygen
Returns

On success a pointer to the existing element is returned. On failure NULL is returned.

Example

// The following three static functions are for example two below

static int dict_string_cmp (void *a1, void *a2)
{
    return strcmp ((char *)a1, (char *)a2);
}

static void *dict_string_cpy (void *s)
{
    return xstr_dup (s); // duplicate the string
}

static void dict_string_del (void *s)
{
    if (s) free (s);
}

int main (void)
{
    // First example: using hash entries with each name less
    // than a pointer width
    // All NULL’s indicating that datatype for hash is less
    // than a pointer width
    xdict_t *hashmap = xdict_new (NULL, NULL, NULL);

    // Our hashmap will use ints for name and chars for values
    int v1[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    char v2[] = "1234567890";

    // Set all the dictionary entries
    for (size_t i=0; i < sizeof v1/sizeof v1[0]; i++) {
        xdict_set (hashmap, (void *)v1[i], (void *)v2[i]);
    }

    // Check that they are all set properly
    for (size_t i=sizeof v1/sizeof v1[0]; i>0; i--) {
        int index = v1[i-1];
        char value = (char)xdict_get (hashmap, (void *)index);
        printf("For name=%i, got value '%c'\n", index, value);
    }

    // Free the map and all entries it holds
    xdict_del (hashmap);

    // Second example: Using custom comparison, copy and deletion
    // functions that uses strings as the hash/name and ints as the
    // value
    hashmap = xdict_new (dict_string_cmp,
                        dict_string_cpy,
                        dict_string_del);
    char *v4[] = {"one", "two", "three", "four", "five",
                 "six", "seven");

    // Add in dictionary entries using strings as the hashes/names and
    // ints as the value
    for (size_t i=0; i<sizeof v4/sizeof v4[0]; i++) {
        xdict_set (hashmap, v4[i], (void *)v1[i]);
    }

    // Print it all out to make sure that they got set
    for (size_t i=sizeof v4/sizeof v4[0]; i>0; i--) {
        char *index = v4[i-1];
        int value = (char)xdict_get (hashmap, index);
        printf("For name=%s, got value '%i'\n", index, value);
    }

    // Delete the hash and all it’s entries
    xdict_del (hashmap);

    return EXIT_SUCCESS;
}
3.5 libxc-0.0.4/xmalloc/xmalloc.c File Reference

Extended malloc routines.

```c
#include <stdio.h>
#include <signal.h>
#include <stdbool.h>
#include "xmalloc/xmalloc.h"
```

Macros

- `#define INC_AMOUNT (1024)`

Functions

- `void * xmalloc (const char *file, const char *func, size_t line, size_t size)`
  Allocate memory.
- `void xmalloc_free (void *ptr)`
  Free memory allocated with `xmalloc()`
- `void * xmalloc_realloc (const char *file, const char *func, size_t line, void *ptr, size_t newsize)`
  Reallocate memory.
- `void xmalloc_dump (unsigned char flags)`
  Dump specified accounting information to stdout.
- `void xmalloc_set_knob (int knob, size_t value)`
  Fine-tune `xmalloc` behaviour.
- `void * xmalloc_find (void *ptr)`
  Finds the accounting record for pointer `ptr`.
- `void xmalloc_reset (void)`
  Resets all the `xmalloc` internal accounting structures.

3.5.1 Detailed Description

Extended malloc routines. A set of functions and libraries to wrap around malloc, that will provide a log of the memory allocated. Which file, function and line-number that allocated and how big each allocation was.

`xmalloc` is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

Author

Lelanthran Krishna Manickum

3.5.2 Function Documentation

3.5.2.1 `void * xmalloc ( const char * file, const char * func, size_t line, size_t size )`

Allocate memory.

Allocates the specified amount of memory. Use the macro `XMALLOC (size)` to automatically have the `func, line` and `file` parameter set to the point of invocation.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>size</th>
<th>The amount of memory to allocate in bytes</th>
</tr>
</thead>
</table>

Generated on Wed Apr 17 2013 18:38:19 for Extended C Library by Doxygen
Returns
On success, a newly allocated block of memory of at least size bytes long. On failure NULL is returned. The caller must free the returned pointer on success.

3.5.2.2 void xmalloc_dump ( unsigned char flags )

Dump specified accounting information to stdout.

Dumps all the accounting and statistical information available about xmalloc and memory usage to stdout.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>flags</th>
</tr>
</thead>
</table>
|     | Flags to control the output, that can be logically OR’ed together to print out any combination of available information:
|     | • XMALLOC_DUMP_STATS : Print only aggregate statistics
|     | • XMALLOC_DUMP_BLOCKS : Print only blocklist
|     | • XMALLOC_DUMP_SELF : Print only statistics about xmalloc itself
|     | • XMALLOC_DUMP_ALL : Print all of the above

Returns
Nothing. A side-effect is caused instead (printing information on the screen).

3.5.2.3 void xmalloc_find ( void *ptr )

Finds the accounting record for pointer ptr.
Locates and returns the accounting record for pointer ptr.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ptr</th>
</tr>
</thead>
</table>
|     | The pointer to search for

Returns
On success, the accounting record for the given pointer is returned. On failure or if the record cannot be found, NULL is returned.

3.5.2.4 void xmalloc_free ( void *ptr )

Free memory allocated with xmalloc()

Frees the memory previously allocated with xmalloc(). Use the macro XMALLOC_FREE(ptr) instead of this function directly.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ptr</th>
</tr>
</thead>
</table>
|     | The pointer to the memory to be freed

Returns
Nothing. A side-effect is caused instead (freeing memory)
3.5.2.5 void *xmalloc_realloc ( const char *file, const char *func, size_t line, void *ptr, size_t newsize )

Reallocate memory.

Reallocates the memory block pointed to by ptr to be at least newsize bytes long. Use the macro XMALLOC_REALLOC (ptr, newsize) instead of calling this function directly.

Parameters

| in | ptr | Pointer to the existing block of memory that was allocated with xmalloc() |
| in | newsize | The new size of the memory block in bytes |

Returns

On success, a reallocated block of memory of at least newsize bytes long. On failure NULL is returned. The caller must free the returned pointer on success.

3.5.2.6 void xmalloc_reset ( void )

Resets all the xmalloc internal accounting structures.

Resets all the internal xmalloc accounting and recording structures, and frees all the memory that xmalloc uses to maintain these records. The actual allocated memory is unchanged and unaffected.

Returns

Nothing. A side-effect is caused; all the memory used internally by xmalloc is freed. No other side-effect is implemented.

3.5.2.7 void xmalloc_set_knob ( int knob, size_t value )

Fine-tune xmalloc behaviour.

Fine-tune the behaviour of xmalloc by setting or clearing certain "knobs" in the library at runtime. NOTE WELL: The behaviour of xmalloc changes drastically in response to the knobs at runtime, so only set or clear an option/knob at safe points in your program.

Knobs and values for them are as follows:

- **XMALLOC_KNOB_FLAGS**: The default flags that xmalloc will use to print out information when an exception is detected. Flags are described in xmalloc_dump()
- **XMALLOC_KNOB_INCREMENT**: The amount to increment each allocation ∗ by. The best value is to use what you consider to be the average size ∗ of your allocations throughout the runtime of the program.
- **XMALLOC_KNOB_ENABLE**:
  1. Turn on xmalloc functionality
  2. Turn off xmalloc functionality (xmalloc then acts as a thin wrapper around standard malloc)

Parameters

| in | knob | The knob to modify (see explanation above) |
| in | value | The new value that the knob would take (see explanation above) |
Returns

Nothing. Various side-effects result from the invocation of this function. See above for more information on the side-effects.

### 3.6 xmalloc/xmalloc.c File Reference

Extended malloc routines.

```c
#include <stdio.h>
#include <signal.h>
#include <stdbool.h>
#include "xmalloc/xmalloc.h"
```

#### Macros

- `#define INC_AMOUNT (1024)`

#### Functions

- `void * xmalloc (const char *file, const char *func, size_t line, size_t size)`
  
  Allocate memory.

- `void xmalloc_free (void *ptr)`
  
  Free memory allocated with `xmalloc()`

- `void * xmallocrealloc (const char *file, const char *func, size_t line, void *ptr, size_t newsize)`
  
  Reallocate memory.

- `void xmalloc_dump (unsigned char flags)`
  
  Dump specified accounting information to stdout.

- `void xmalloc_set_knob (int knob, size_t value)`
  
  Fine-tune `xmalloc` behaviour.

- `void * xmalloc_find (void *ptr)`
  
  Finds the accounting record for pointer `ptr`.

- `void xmalloc_reset (void)`
  
  Resets all the `xmalloc` internal accounting structures.

#### 3.6.1 Detailed Description

Extended malloc routines. A set of functions and libraries to wrap around malloc, that will provide a log of the memory allocated. Which file, function and line-number that allocated and how big each allocation was.

`xmalloc` is part of the `libxc` (Extended C Library) and falls under the relevant copyright license in `libxc`.

**Author**

Lelanthran Krishna Manickum

#### 3.6.2 Function Documentation

##### 3.6.2.1 `void * xmalloc ( const char * file, const char * func, size_t line, size_t size )`

Allocate memory.

Allocates the specified amount of memory. Use the macro `XMALLOC (size)` to automatically have the `func`, `line` and `file` parameter set to the point of invocation.
3.6 xmalloc/xmalloc.c File Reference

Parameters

| in  | size | The amount of memory to allocate in bytes |

Returns

On success, a newly allocated block of memory of at least size bytes long. On failure NULL is returned. The caller must free the returned pointer on success.

3.6.2.2 void xmalloc_dump ( unsigned char flags )

Dump specified accounting information to stdout.

Dumps all the accounting and statistical information available about xmalloc and memory usage to stdout.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>flags</th>
<th>Flags to control the output, that can be logically OR'ed together to print out any combination of available information:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>• Xmalloc_DUMP_STATS: Print only aggregate statistics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Xmalloc_DUMP_BLOCKS: Print only blocklist</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Xmalloc_DUMP_SELF: Print only statistics about xmalloc itself</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Xmalloc_DUMP_ALL: Print all of the above</td>
</tr>
</tbody>
</table>

Returns

Nothing. A side-effect is caused instead (printing information on the screen).

3.6.2.3 void xmalloc_find ( void *ptr )

Finds the accounting record for pointer ptr.

Locates and returns the accounting record for pointer ptr.

Parameters

| in  | ptr   | The pointer to search for |

Returns

On success, the accounting record for the given pointer is returned. On failure or if the record cannot be found, NULL is returned.

3.6.2.4 void xmalloc_free ( void *ptr )

Free memory allocated with xmalloc()

Frees the memory previously allocated with xmalloc(). Use the macro Xmalloc_FREE(ptr) instead of this function directly.

Parameters

| in  | ptr   | The pointer to the memory to be freed |
3.6.2.5 void *xmalloc_realloc ( const char *file, const char *func, size_t line, void *ptr, size_t newsize )

Reallocate memory.

Reallocates the memory block pointed to by ptr to be at least newsize bytes long. Use the macro XMALLOC_REALLOC (ptr, newsize) instead of calling this function directly.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>ptr</th>
<th>Pointer to the existing block of memory that was allocated with xmalloc()</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>newsize</td>
<td>The new size of the memory block in bytes</td>
</tr>
</tbody>
</table>

Returns

On success, a reallocated block of memory of at least newsize bytes long. On failure NULL is returned. The caller must free the returned pointer on success.

3.6.2.6 void xmalloc_reset ( void )

Resets all the xmalloc internal accounting structures.

Resets all the internal xmalloc accounting and recording structures, and frees all the memory that xmalloc uses to maintain these records. The actual allocated memory is unchanged and unaffected.

Returns

Nothing. A side-effect is caused; all the memory used internally by xmalloc is freed. No other side-effect is implemented.

3.6.2.7 void xmalloc_set_knob ( int knob, size_t value )

Fine-tune xmalloc behaviour.

Fine-tune the behaviour of xmalloc by setting or clearing certain "knobs" in the library at runtime. NOTE WELL: The behaviour of xmalloc changes drastically in response to the knobs at runtime, so only set or clear an option/knob at safe points in your program.

Knobs and values for them are as follows:

- **XMALLOC_KNOB_FLAGS**: The default flags that xmalloc will use to print out information when an exception is detected. Flags are described in xmalloc_dump()

- **XMALLOC_KNOB_INCREMENT**: The amount to increment each allocation * by. The best value is to use what you consider to be the average size * of your allocations throughout the runtime of the program.

- **XMALLOC_KNOB_ENABLE**:
  1. Turn on xmalloc functionality
  2. Turn off xmalloc functionality (xmalloc then acts as a thin wrapper around standard malloc)

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>knob</th>
<th>The knob to modify (see explanation above)</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>value</td>
<td>The new value that the knob would take (see explanation above)</td>
</tr>
</tbody>
</table>
Returns

Nothing. Various side-effects result from the invocation of this function. See above for more information on the side-effects.

3.7  libxc-0.0.4/xshare/xshare.c File Reference

Routines to load dynamic libraries at runtime.

```c
#include <stdlib.h>
#include <string.h>
#include "xshare/xshare.h"
```

Macros

- `#define MAX_ERROR_MSG (255)`

Functions

- `xshare_library_t xshare_open (char *libname)`
  
  Open a dynamic library.

- `xshare_symbol_t xshare_symbol (xshare_library_t lib, char *symbol)`
  
  Locate a symbol within a dynamic library.

- `void xshare_close (xshare_library_t lib)`
  
  Close a previously opened dynamic library.

- `int xshare_error (void)`
  
  Returns an integer describing the last error.

- `char * xshare_errmsg (void)`
  
  Returns a human-readable string describing the last error.

3.7.1  Detailed Description

Routines to load dynamic libraries at runtime. A set of functions to load dynamic libraries and then locate symbols within those libraries at runtime. Xshare is simply a cross-platform way to load runtime libraries and has been tested on unix variants (Linux, FreeBSD) and Microsoft Windows.

See Also

- `xshare_open()`, `xshare_symbol()`, `xshare_close()`, `xshare_close()`, `xshare_error()`

xshare is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

Author

Lelanthran Krishna Manickum

3.7.2  Function Documentation

3.7.2.1  void xshare_close ( xshare_library_t lib )

Close a previously opened dynamic library.

Closes a previously opened dynamic library that was opened with `xshare_open()`.
Returns

Nothing.

Example:

```c
int main (void)
{
    xshare_library_t ext_lib = xshare_open("libc");
    if (ext_lib==NULL) {
        printf("Could not load the libc library\n");
        return EXIT_FAILURE;
    }
    int (*ext_func) (const char *) = xshare_symbol (ext_lib, "puts");
    if (!ext_func) {
        xshare_close (ext_lib);
        printf("Could not locate the puts function in libc\n");
        return EXIT_FAILURE;
    }
    ext_func("Hello World\n");
    xshare_close (ext_lib);
    return EXIT_SUCCESS;
}
```

3.7.2.2 char\* xshareErrMsg ( void )

Returns a human-readable string describing the last error.

Returns a human-readable string describing the last error that occurred. The actual string will differ from platform to platform but can be used unchanged by the caller to present the user of the program with an error message.

This function is not reentrant and thus multiple calls to any xshare function will overwrite this message. The caller should make a copy of the return value if the value is needed in between calls to any xshare function. The string that is returned will have a maximum length of 256 characters, including the NULL-terminator.

See Also

xshare_open(), xshare_symbol(), xshare_close(), xshare_close(), xshare_error()
3.7.2.3 int xshare_error( void )

Returns an integer describing the last error.

Returns an integer describing the last error that occurred. This value differs from platform to platform and is thus almost totally useless in a program that assumes no knowledge of the underlying platform. Use xshare_errmsg() to get a human-readable description of the error.

See Also

xshare_open(), xshare_symbol(), xshare_close(), xshare_close(), xshare_errmsg()

Returns

A platform-defined integer describing the last error that occurred.

Example:

```c
int main (void)
{
    xshare_library_t ext_lib = xshare_open (*"libc");
    if (ext_lib==NULL) {
        printf (*"Could not load the libc library - %i\b", xshare_error());
        return EXIT_FAILURE;
    }
    int (*ext_func) (const char *) = xshare_symbol (ext_lib, "puts");
    if (!ext_func) {
        printf (*"Locate of 'puts' in libc failed - %i\n", xshare_error());
        xshare_close (ext_lib);
        return EXIT_FAILURE;
    }
    ext_func (*"Hello World\n");
    xshare_close (ext_lib);
    return EXIT_SUCCESS;
}
```

3.7.2.4 xshare_library_t xshare_open ( char * libname )

Open a dynamic library.

Opens the given dynamic library libname for further use with xshare_symbol(). Note that libname is only the filename, not the extension. The library that will be searched for depends on the platform. On unix variants the library that will be searched for is libname.so and on Microsoft Windows the library that will be searched for is libname.dll.

See Also

xshare_symbol(), xshare_close(), xshare_close(), xshare_error()

Parameters

| in | libname | The library to search for |

Returns

On success, a handle to the library is returned. The handle is used for further calls to xshare_symbol() and can be passed to xshare_close() to explicitly unload the library, although the system will unload the library at program exit. On failure (void *)NULL is returned.

Example:

```c
int main (void)
```
{ 
    xshare_library_t ext_lib = xshare_open("libc");
    if (ext_lib==NULL) {
        printf("Could not load the libc library\n");
        return EXIT_FAILURE;
    }
    int (*ext_func) (const char *) = xshare_symbol (ext_lib, "puts");
    if (!ext_func) {
        xshare_close (ext_lib);
        printf("Could not locate the puts function in libc\n");
        return EXIT_FAILURE;
    }
    ext_func("Hello World\n");
    xshare_close (ext_lib);
    return EXIT_SUCCESS;
}

3.7.2.5 xshare_symbol_t xshare_symbol ( xshare_library_t lib, char * symbol )

Locate a symbol within a dynamic library.

Locates the symbol symbol within the dynamic library lib. Note that lib is a handle returned by xshare_open(). The caller is required to ensure that the return value of xshare_symbol() is cast to the correct type before using it.

See Also

xshare_open(), xshare_close(), xshare_close(), xshare_error()

Parameters

| in   | lib     | The handle to the library (previously opened with xshare_open()) |
|      | symbol  | The symbol to locate within the library lib |

Returns

On success, a pointer to the symbol symbol within the library lib is returned. On failure (void *)NULL is returned.

Example:

```c
int main (void)
{
    xshare_library_t ext_lib = xshare_open("libc");
    if (ext_lib==NULL) {
        printf("Could not load the libc library\n");
        return EXIT_FAILURE;
    }
    int (*ext_func) (const char *) = xshare_symbol (ext_lib, "puts");
    if (!ext_func) {
        xshare_close (ext_lib);
        printf("Could not locate the puts function in libc\n");
        return EXIT_FAILURE;
    }
    ext_func("Hello World\n");
    xshare_close (ext_lib);
    return EXIT_SUCCESS;
}
```

3.8 xshare/xshare.c File Reference

Routines to load dynamic libraries at runtime.

```c
#include <stdlib.h>
#include <string.h>
#include "xshare/xshare.h"
```
Macros

- #define MAX_ERROR_MSG (255)

Functions

- xshare_library_t xshare_open (char *libname)
  Open a dynamic library.
- xshare_symbol_t xshare_symbol (xshare_library_t lib, char *symbol)
  Locate a symbol within a dynamic library.
- void xshare_close (xshare_library_t lib)
  Close a previously opened dynamic library.
- int xshare_error (void)
  Returns an integer describing the last error.
- char * xshare_errmsg (void)
  Returns a human-readable string describing the last error.

3.8.1 Detailed Description

Routines to load dynamic libraries at runtime. A set of functions to load dynamic libraries and then locate symbols within those libraries at runtime. Xshare is simply a cross-platform way to load runtime libraries and has been tested on unix variants (Linux, FreeBSD) and Microsoft Windows.

See Also

xshare_open(), xshare_symbol(), xshare_close(), xshare_close(), xshare_error()

Xshare is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

Author

Lelanthran Krishna Manickum

3.8.2 Function Documentation

3.8.2.1 void xshare_close ( xshare_library_t lib )

Close a previously opened dynamic library.
Closes a previously opened dynamic library that was opened with xshare_open().

See Also

xshare_open(), xshare_symbol(), xshare_close(), xshare_error()

Returns

Nothing.

Example:
```c
int main (void)
{
    xshare_library_t ext_lib = xshare_open("libc");
    if (ext_lib==NULL) {
        printf("Could not load the libc library\b");
        return EXIT_FAILURE;
    }
    int (*ext_func) (const char *) = xshare_symbol (ext_lib, "puts");
    if (!ext_func) {
        xshare_close (ext_lib);
        printf("Could not locate the puts function in libc\n");
        return EXIT_FAILURE;
    }
    ext_func("Hello World\n");
    xshare_close (ext_lib);
    return EXIT_SUCCESS;
}

3.8.2.2 char* xshare_errmsg ( void )

Returns a human-readable string describing the last error.

Returns a human-readable string describing the last error that occurred. The actual string will differ from platform
to platform but can be used unchanged by the caller to present the user of the program with an error message.

This function is not reentrant and thus multiple calls to any xshare function will overwrite this message. The caller
should make a copy of the return value if the value is needed in between calls to any xshare function. The string
that is returned will have a maximum length of 256 characters, including the NULL-terminator.

See Also
xshare_open(), xshare_symbol(), xshare_close(), xshare_close(), xshare_error()

Returns
A platform-defined human-readable string describing the last error that occurred.

Example:
```
### Returns

A platform-defined integer describing the last error that occurred.

#### Example:

```c
int main (void)
{
    xshare_library_t ext_lib = xshare_open (*"libc");
    if (ext_lib==NULL) {
        printf (*"Could not load the libc library \b", xshare_error());
        return EXIT_FAILURE;
    }
    int (*ext_func) (const char *) = xshare_symbol (ext_lib, "puts");
    if (!ext_func) {
        printf (*"Locate of \"puts\" in libc failed \b", xshare_error());
        xshare_close (ext_lib);
        return EXIT_FAILURE;
    }
    ext_func (*"Hello World\n");
    xshare_close (ext_lib);
    return EXIT_SUCCESS;
}
```

---

#### xshare/library_t xshare_open ( char *libname )

Open a dynamic library.

Opens the given dynamic library `libname` for further use with `xshare_symbol()`. Note that `libname` is only the filename, not the extension. The library that will be searched for depends on the platform. On Unix variants the library that will be searched for is `libname.so` and on Microsoft Windows the library that will be searched for is `libname.dll`.

See Also

- `xshare_symbol()`, `xshare_close()`, `xshare_close()`, `xshare_error()`

#### Parameters

<table>
<thead>
<tr>
<th>libname</th>
<th>The library to search for</th>
</tr>
</thead>
</table>

#### Returns

On success, a handle to the library is returned. The handle is used for further calls to `xshare_symbol()` and can be passed to `xshare_close()` to explicitly unload the library, although the system will unload the library at program exit. On failure (void *) `NULL` is returned.

#### Example:

```c
int main (void)
{
    xshare_library_t ext_lib = xshare_open (*"libc");
    if (ext_lib==NULL) {
        printf (*"Could not load the libc library\b", xshare_error());
        return EXIT_FAILURE;
    }
    int (*ext_func) (const char *) = xshare_symbol (ext_lib, "puts");
    if (!ext_func) {
        printf (*"Could not locate the puts function in libc\n");
    }
    ext_func (*"Hello World\n");
    xshare_close (ext_lib);
    return EXIT_SUCCESS;
}
```
3.8.2.5 xshare_symbol_t xshare_symbol ( xshare_library_t lib, char * symbol )

Locate a symbol within a dynamic library.

Locates the symbol `symbol` within the dynamic library `lib`. Note that `lib` is a handle returned by `xshare_open()`. The caller is required to ensure that the return value of `xshare_symbol()` is cast to the correct type before using it.

See Also

`xshare_open()`, `xshare_close()`, `xshare_close()`, `xshare_error()`

Parameters

| in | lib | The handle to the library (previously opened with `xshare_open()`) |
| in | symbol | The symbol to locate within the library `lib` |

Returns

On success, a pointer to the symbol `symbol` within the library `lib` is returned. On failure `void *`)NULL is returned.

Example:

```c
int main (void)
{
    xshare_library_t ext_lib = xshare_open (*"libc");
    if (ext_lib==NULL) {
        printf (*"Could not load the libc library\n");
        return EXIT_FAILURE;
    }
    int (*ext_func) (const char *) = xshare_symbol (ext_lib, "puts");
    if (!ext_func) {
        printf (*"Could not locate the puts function in libc\n");
        return EXIT_FAILURE;
    }
    ext_func (*"Hello World\n");
    xshare_close (ext_lib);
    return EXIT_SUCCESS;
}
```

3.9 libxc-0.0.4/xsock/xsock.c File Reference

Cross-platform BSD Sockets wrapper.

```c
#include "xsock/xsock.h"
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include "xerror/xerror.h"
```

Macros

- `#define OUTPUT(...) ;`
• #define TEST_URL ("127.0.0.1")
• #define TEST_PORT (4560)
• #define TEST_TIMEOUT (5)
• #define TEST_CLIENT "This is a test message from client"
• #define TEST_SERVER "This is a test message from server"

Functions

• int xsock_listen (size_t port, size_t timeout)
  Listen for a connection.
• int xsock_connect (char *server, size_t port)
  Connect to the specified address.
• int xsock_close (int fd)
  Close a socket descriptor.
• size_t xsock_write (int fd, void *buf, size_t len)
  Write data to a connected socket.
• size_t xsock_read (int fd, void *buf, size_t len, size_t timeout)
  Read data from given connected and live socket.
• int xsock_clear_errno (void)
  Clear the last error value.
• int xsock_errno (void)
  Retrieve the last error number.
• const char * xsock_strerror (int err)
  Turn the error number given into a human-readable string.
• int xsock_test (bool do_server_test)

3.9.1 Detailed Description

Cross-platform BSD Sockets wrapper. A collection of functions intended to provide easier TCP stream usage within C programs. This module provides limited functionality; callers can connect to a remote address:port, listen on a local port and read/write to connected socket descriptors. Limited though it is, this functionality is enough to perform communications over TCP and/or provide services over TCP.

For more fine-grained functionality, such as non-blocking IO, RAW datagrams, etc, use your platforms’ native sockets implementation directly.

See Also

  xsock_listen(), xsock_connect(), xsock_close(), xsock_write(), xsock_read(), xsock_errno(), xsock_clear_errno(), xsock_strerror()

xsock is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

Author

  Lelanthran Krishna Manickum

3.9.2 Function Documentation

3.9.2.1 int xsock_clear_errno ( void )

Clear the last error value.

Clears the last error that occurred. This function is not reentrant.
See Also

xsock_listen(), xsock_connect(), xsock_close(), xsock_write(), xsock_read(), xsock_errno(), xsock_strerror()

Returns

On success, zero is returned. On error, -1 is returned and the error is unspecified.

3.9.2.2 int xsock_close ( int fd )

Close a socket descriptor.

Closes the socket descriptor fd, which must have been returned from a successful call to xsock_listen() or xsock_close(). On error -1 is returned and the specific error number can be retrieved with xsock_errno().

See Also

xsock_listen(), xsock_connect(), xsock_close(), xsock_write(), xsock_read(), xsock_errno(), xsock_strerror()

Returns

On success, zero is returned. On error, -1 is returned and the specific error number can be retrieved with xsock_errno().

3.9.2.3 int xsock_connect ( char∗ server, size_t port )

Connect to the specified address.

Connect to the specified address server on port port. The address server can be either an IP address or a domain name. A single connection attempt is made to connect to server on port only.

See Also

xsock_listen(), xsock_close(), xsock_write(), xsock_read(), xsock_errno(), xsock_clear_errno(), xsock_strerror()

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>address</th>
<th>The network address/domain to connect to</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>port</td>
<td>The port on which to connect</td>
</tr>
</tbody>
</table>

Returns

On success a connected socket descriptor is returned which can be used with xsock_read(), xsock_write() and xsock_close(). The caller is responsible for closing the socket descriptor with xsock_close(). On error (size_t)-1 is returned and xsock_errno() will return the specific error number.

Example:

```c
int main (void)
{
    #define TEST_ADDRESS (*"10.207.204.12")
    #define TEST_PORT (8080)
    #define TEST_MSG /*Hello World*/
    xsock_clear_errno (); // Connect and send a message
    int fd = xsock_connect (TEST_ADDRESS, TEST_PORT);
    if (fd<0) {
```

}
fprintf (stderr, "Connection refused on %s:$zu\n",
TEST_ADDRESS, TEST_PORT);
fprintf (stderr, "err %i, %s\n", xsock_errno (),
xsock_strerror(xsock_errno()));
return EXIT_FAILURE;
}

// Write the string to the connected socket descriptor
size_t bytes_written = xsock_write (fd, buffer, sizeof buffer);
if (bytes_read==0) {
    printf ("No data written\n");
} else if (bytes_read<0) {
    printf ("Error writing: %s\n", xsock_strerror(xsock_errno()));
} else {
    printf ("Wrote '%zu'\n", bytes);
}
xsock_close (fd);
}

3.9.2.4 int xsock_errno ( void )

Retrieve the last error number.

Returns the number of the last error that occurred. This function is not reentrant and will return the last error that occurred, even if the occurrence was in a different thread.

See Also
    xsock_listen(), xsock_connect(), xsock_close(), xsock_write(), xsock_read(), xsock_clear_errno(), xsock_strerror()

Returns
    The value of the last error that occurred is returned.

3.9.2.5 int xsock_listen ( size_t port, size_t timeout )

Listen for a connection.

Wait no more than timeout seconds for a connection on port port. If a connection is made then the socket descriptor of the incoming connection is returned, otherwise -1 is returned and the error value is set appropriately (see xsock_errno()).

If timeout expires before a connection is made then the return value is zero.

See Also
    xsock_connect(), xsock_close(), xsock_write(), xsock_read(), xsock_errno(), xsock_clear_errno(), xsock_strerror()

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>port</td>
<td>The port to listen on</td>
</tr>
<tr>
<td>timeout</td>
<td>The timeout in seconds to wait for a connection</td>
</tr>
</tbody>
</table>

Returns

On success, the socket descriptor of the incoming connection is returned and can be used with xsock_read(), xsock_write() and xsock_close(). On timeout, zero is returned. On error -1 is returned and xsock_errno() will return the specific error number.

Example:
int main (void)
{
    #define TEST_PORT (8080)
    xsock_clear_errno ();
    // Wait for incoming connection
    int fd = xsock_listen (TEST_PORT, 10);
    if (fd<=0) {
        fprintf (stderr, "No connection received on %u\n", TEST_PORT);
        fprintf (stderr, "err %i, %s\n", xsock_errno (), xsock_strerror(xsock_errno()));
        return EXIT_FAILURE;
    }
    // Read a string from the incoming connection
    char buffer[255]; buffer[255] = 0;
    size_t bytes_read = xsock_read (fd, buffer, sizeof buffer, 5);
    if (bytes_read==0) {
        printf ("Timeout\n");
    } else if (bytes_read<0) {
        printf ("Error reading: %s\n", xsock_strerror(xsock_errno()));
    } else {
        printf ("Read '\%s'\n", buffer);
    }
    xsock_close (fd);
}

3.9.2.6 size_t xsock_read ( int fd, void * buf, size_t len, size_t timeout )

Read data from given connected and live socket.

Read not more than len bytes into buffer buf from socket descriptor fd over the next timeout seconds. Should the timeout expire before the buffer is filled then the number of bytes read in is returned. If no bytes are read in after timeout expires then zero is returned. Should len bytes be read from the socket descriptor then the function returns immediately without waiting for the timeout to expire.

On failure -1 is returned and the specific error number can be retrieved using xsock_errno().

See Also

xsock_listen(), xsock_connect(), xsock_close(), xsock_write(), xsock_errno(), xsock_clear_errno(), xsock_strerror()

Parameters

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>fd</td>
<td>The socket descriptor to read</td>
</tr>
<tr>
<td>in</td>
<td>buf</td>
<td>The buffer to fill with data read in from fd</td>
</tr>
<tr>
<td>in</td>
<td>len</td>
<td>The length of the buffer buf in bytes</td>
</tr>
<tr>
<td>in</td>
<td>timeout</td>
<td>The timeout in seconds to wait for data to become available to read</td>
</tr>
</tbody>
</table>

Returns

On success, the number of actual bytes read from the file descriptor fd is returned. On error (size_t)-1 is returned and xsock_errno() will return the specific error number.

Example:

int main (void)
{
    #define TEST_PORT (8080)
    xsock_clear_errno ();
    // Wait for incoming connection
    int fd = xsock_listen (TEST_PORT, 10);
    if (fd<=0) {
        fprintf (stderr, "No connection received on %u\n", TEST_PORT);
        fprintf (stderr, "err %i, %s\n", xsock_errno (), xsock_strerror(xsock_errno()));
    }
}
return EXIT_FAILURE;
}
// Read a string from the incoming connection
char buffer[255]; buffer[255] = 0;
size_t bytes_read = xsock_read (fd, buffer, sizeof buffer, 5);
if (bytes_read==0) {
    printf ("Timeout\n");
} else if (bytes_read<0) {
    printf ("Error reading: \s\n", xsock_strerror(xsock_errno()));
} else {
    printf ("Read '\s\n", buffer);
}
    xsock_close (fd);

3.9.2.7 const char ∗ xsock_strerror ( int err )

Turn the error number given into a human-readable string.

Changes the error number err into a human-readable string that can be displayed to the user. This function is not reentrant and the caller should make a copy of the return value if the returned string is to be used long after the call to this function is made.

See Also
    xsock_listen(), xsock_connect(), xsock_close(), xsock_write(), xsock_read(), xsock_clear_errno(), xsock_errno()

Returns

A non-modifiable string describing the specified error err in a human-readable manner.

3.9.2.8 size_t xsock_write ( int fd, void ∗ buf, size_t len )

Write data to a connected socket.

Write len bytes of data in buffer buf to socket descriptor fd. The number of bytes actually written is returned. If an error occurs (size_t)-1 is returned and the specific error number can be retrieved with xsock_errno(). The write is performed in blocking mode.

See Also
    xsock_listen(), xsock_connect(), xsock_close(), xsock_read(), xsock_errno(), xsock_clear_errno()

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>fd</th>
<th>The socket descriptor to write to</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>buf</td>
<td>The data to be written</td>
</tr>
<tr>
<td>in</td>
<td>len</td>
<td>The number of bytes to write</td>
</tr>
</tbody>
</table>

Returns

On success the number of bytes actually written to the socket descriptor is returned, or zero if no data is written. On error (size_t)-1 is returned and xsock_errno() will return the specific error number.

Example:

```c
int main (void)
```
```c
#define TEST_ADDRESS "10.207.204.12"
#define TEST_PORT (8080)
#define TEST_MSG ("Hello World")
xsock_clear_errno ();

// Connect and send a message
int fd = xsock_connect (TEST_ADDRESS, TEST_PORT);
if (fd<0) {
    fprintf (stderr, "Connection refused on %s:%zu\n",
             TEST_ADDRESS, TEST_PORT);
    fprintf (stderr, "err %i, %s\n", xsock_errno (),
             xsock_strerror(xsock_errno()));
    return EXIT_FAILURE;
}

// Write the string to the connected socket descriptor
size_t bytes_written = xsock_write (fd, buffer, sizeof buffer);
if (bytes_read==0) {
    printf ("No data written\n");
} else if (bytes_read<0) {
    printf ("Error writing: %s\n", xsock_strerror(xsock_errno()));
} else {
    printf ("Wrote %zu\n", bytes);
}
xsock_close (fd);
}

3.10 xsock/xsock.c File Reference

Cross-platform BSD Sockets wrapper.

```
#include "xsock/xsock.h"
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include "xerror/xerror.h"
```

Macros

- `#define OUTPUT(...)`
- `#define TEST_URL ("127.0.0.1")`
- `#define TEST_PORT (4560)`
- `#define TEST_TIMEOUT (5)`
- `#define TEST_CLIENT "This is a test message from client"`
- `#define TEST_SERVER "This is a test message from server"`

Functions

- `int xsock_listen (size_t port, size_t timeout)`  
  *Listen for a connection.*
- `int xsock_connect (char *server, size_t port)`  
  *Connect to the specified address.*
- `int xsock_close (int fd)`  
  *Close a socket descriptor.*
- `size_t xsock_write (int fd, void *buf, size_t len)`  
  *Write data to a connected socket.*
- `size_t xsock_read (int fd, void *buf, size_t len, size_t timeout)`  
  *Read data from given connected and live socket.*
- `int xsock_clear_errno (void)`
3.10.1 Detailed Description

Cross-platform BSD Sockets wrapper. A collection of functions intended to provide easier TCP stream usage within C programs. This module provides limited functionality; callers can connect to a remote address:port, listen on a local port and read/write to connected socket descriptors. Limited though it is, this functionality is enough to perform communications over TCP and/or provide services over TCP.

For more fine-grained functionality, such as non-blocking IO, RAW datagrams, etc, use your platforms' native sockets implementation directly.

See Also

xsock_listen(), xsock_connect(), xsock_close(), xsock_write(), xsock_read(), xsock_errno(), xsock_clear_errno(), xsock_strerror()

xsock is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

Author

Lelanthran Krishna Manickum

3.10.2 Function Documentation

3.10.2.1 int xsock_clear_errno ( void )

Clear the last error value.

Cleans the last error that occurred. This function is not reentrant.

See Also

xsock_listen(), xsock_connect(), xsock_close(), xsock_write(), xsock_read(), xsock_errno(), xsock_clear_errno(), xsock_strerror()

Returns

On success, zero is returned. On error, -1 is returned and the error is unspecified.

3.10.2.2 int xsock_close ( int fd )

Close a socket descriptor.

Closes the socket descriptor fd, which must have been returned from a successful call to xsock_listen() or xsock_close(). On error -1 is returned and the specific error number can be retrieved with xsock_errno().

See Also

xsock_listen(), xsock_connect(), xsock_close(), xsock_write(), xsock_read(), xsock_errno(), xsock_strerror()

Returns

On success, zero is returned. On error, -1 is returned and the specific error number can be retrieved with xsock_errno().
3.10.2.3 int xsock_connect (char *server, size_t port)

Connect to the specified address.

Connect to the specified address server on port port. The address server can be either an IP address or a domain name. A single connection attempt is made to connect to server on port only.

See Also

xsock_listen(), xsock_close(), xsock_write(), xsock_read(), xsock_errno(), xsock_clear_errno(), xsock_strerror()

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>address</th>
<th>The network address/domain to connect to</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>port</td>
<td>The port on which to connect</td>
</tr>
</tbody>
</table>

Returns

On success a connected socket descriptor is returned which can be used with xsock_read(), xsock_write() and xsock_close(). The caller is responsible for closing the socket descriptor with xsock_close(). On error (size_t)-1 is returned and xsock_errno() will return the specific error number.

Example:

```c
int main (void)
{
    #define TEST_ADDRESS "10.207.204.12"
    #define TEST_PORT (8080)
    #define TEST_MSG "Hello World"
    xsock_clear_errno();
    // Connect and send a message
    int fd = xsock_connect (TEST_ADDRESS, TEST_PORT);
    if (fd<=0) {
        fprintf (stderr, "Connection refused on %s:$zu\n", TEST_ADDRESS, TEST_PORT);
        fprintf (stderr, "err %i, %s\n", xsock_errno (), xsock_strerror(xsock_errno()));
        return EXIT_FAILURE;
    }
    // Write the string to the connected socket descriptor
    size_t bytes_written = xsock_write (fd, buffer, sizeof buffer);
    if (bytes_read==0) {
        printf ("No data written\n");
    } else if (bytes_read<0) {
        printf ("Error writing: %s\n", xsock_strerror(xsock_errno()));
    } else {
        printf ("Wrote '%zu'\n", bytes);
    }
    xsock_close (fd);
}
```

3.10.2.4 int xsock_errno (void)

Retrieve the last error number.

Returns the number of the last error that occurred. This function is not reentrant and will return the last error that occurred, even if the occurrence was in a different thread.

See Also

xsock_listen(), xsock_connect(), xsock_close(), xsock_write(), xsock_read(), xsock_errno(), xsock_clear_errno(), xsock_strerror()
3.10.2.5 `int xsock_listen ( size_t port, size_t timeout )`

Listen for a connection.

Wait no more than `timeout` seconds for a connection on port `port`. If a connection is made then the socket descriptor of the incoming connection is returned, otherwise -1 is returned and the error value is set appropriately (see `xsock_geterrno()`).

If `timeout` expires before a connection is made then the return value is zero.

See Also
- `xsock_connect()`, `xsock_close()`, `xsock_write()`, `xsock_read()`, `xsock_errno()`, `xsock_clear_errno()`, `xsock_strerror()`

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>port</code></td>
<td>The port to listen on</td>
</tr>
<tr>
<td><code>timeout</code></td>
<td>The timeout in seconds to wait for a connection</td>
</tr>
</tbody>
</table>

Returns

On success, the socket descriptor of the incoming connection is returned and can be used with `xsock_read()`, `xsock_write()` and `xsock_close()`. On timeout, zero is returned. On error -1 is returned and `xsock_errno()` will return the specific error number.

Example:

```c
int main (void)
{
    #define TEST_PORT (8080)
    xsock_clear_errno ();
    // Wait for incoming connection
    int fd = xsock_listen (TEST_PORT, 10);
    if (fd<=0) {
        fprintf (stderr, "No connection received on %u\n", TEST_PORT);
        fprintf (stderr, "err %i, %s\n", xsock_errno (),
                  xsock_strerror(xsock_errno()));
        return EXIT_FAILURE;
    }
    // Read a string from the incoming connection
    char buffer[255]; buffer[255] = 0;
    size_t bytes_read = xsock_read (fd, buffer, sizeof buffer, 5);
    if (bytes_read==0) {
        printf ("Timeout\n");
    } else if (bytes_read<0) {
        printf ("Error reading: %s\n", xsock_strerror(xsock_errno()));
    } else {
        printf ("Read '%s'\n", buffer);
    }
    xsock_close (fd);
}
```

3.10.2.6 `size_t xsock_read ( int fd, void * buf, size_t len, size_t timeout )`

Read data from given connected and live socket.

Read not more than `len` bytes into buffer `buf` from socket descriptor `fd` over the next `timeout` seconds. Should the timeout expire before the buffer is filled then the number of bytes read in is returned. If no bytes are read in after
timeout expires then zero is returned. Should len bytes be read from the socket descriptor then the function returns immediately without waiting for the timeout to expire.

On failure -1 is returned and the specific error number can be retrieved using xsock_errno().

See Also

xsock_listen(), xsock_connect(), xsock_close(), xsock_write(), xsock_errno(), xsock_clear_errno(), xsock_strerror()

Parameters

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>in fd</code></td>
<td>The socket descriptor to read</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>in buf</code></td>
<td>The buffer to fill with data read in from <code>fd</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>in len</code></td>
<td>The length of the buffer <code>buf</code> in bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>in timeout</code></td>
<td>The timeout in seconds to wait for data to become available to read</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Returns

On success, the number of actual bytes read from the file descriptor `fd` is returned. On error `(size_t)-1` is returned and `xsock_errno()` will return the specific error number.

Example:

```c
int main (void)
{
    #define TEST_PORT (8080)
    xsock_clear_errno ();
    // Wait for incoming connection
    int fd = xsock_listen (TEST_PORT, 10);
    if (fd<0) {
        fprintf (stderr, "No connection received on %u\n", TEST_PORT);
        fprintf (stderr, "err %i, %s\n", xsock_errno (),
            xsock_strerror(xsock_errno()));
        return EXIT_FAILURE;
    }
    // Read a string from the incoming connection
    char buffer[255]; buffer[255] = 0;
    size_t bytes_read = xsock_read (fd, buffer, sizeof buffer, 5);
    if (bytes_read==0) {
        printf ("Timeout\n");
    } else if (bytes_read<0) {
        printf ("Error reading: %s\n", xsock_strerror(xsock_errno()));
    } else {
        printf ("Read '%s'\n", buffer);
    }
    xsock_close (fd);
}
```

3.10.2.7 `const char* xsock_strerror ( int err )`

Turn the error number given into a human-readable string.

Changes the error number `err` into a human readable string that can be displayed to the user. This function is not reentrant and the caller should make a copy of the return value if the returned string is to be used long after the call to this function is made.

See Also

xsock_listen(), xsock_connect(), xsock_close(), xsock_write(), xsock_read(), xsock_clear_errno(), xsock_strerror()

Returns

A non-modifiable string describing the specified error `err` in a human-readable manner.
3.10.2.8  size_t xsock_write ( int fd, void * buf, size_t len )

Write data to a connected socket.

Write len bytes of data in buffer buf to socket descriptor fd. The number of bytes actually written is returned. If an error occurs (size_t)-1 is returned and the specific error number can be retrieved with xsock_errno(). The write is performed in blocking mode.

See Also

xsock_listen(), xsock_close(), xsock_connect(), xsock_read(), xsock_errno(), xsock_clear_errno(), xsock_strerror()

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>fd</th>
<th>The socket descriptor to write to</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>buf</td>
<td>The data to be written</td>
</tr>
<tr>
<td>in</td>
<td>len</td>
<td>The number of bytes to write</td>
</tr>
</tbody>
</table>

Returns

On success the number of bytes actually written to the socket descriptor is returned, or zero if no data is written. On error (size_t)-1 is returned and xsock_errno() will return the specific error number.

Example:

int main (void)
{
    #define TEST_ADDRESS (*10.207.204.12*)
    #define TEST_PORT (*8080*)
    #define TEST_MSG (*Hello World*)
    xsock_clear_errno ();
    // Connect and send a message
    int fd = xsock_connect (TEST_ADDRESS, TEST_PORT);
    if (fd<=0) {
        fprintf (stderr, "Connection refused on %s:%zu\n",
            TEST_ADDRESS, TEST_PORT);
        fprintf (stderr, "err %i, %s\n", xsock_errno (),
            xsock_strerror(xsock_errno()));
        return EXIT_FAILURE;
    }
    // Write the string to the connected socket descriptor
    size_t bytes_written = xsock_write (fd, buffer, sizeof buffer);
    if (bytes_read==0) {
        printf ("No data written\n");
    } else if (bytes_read<0) {
        printf ("Error writing: %s\n", xsock_strerror(xsock_errno()));
    } else {
        printf ("Wrote \%zu\n", bytes);
    }
    xsock_close (fd);
}

3.11 libxc-0.0.4/xstring/xstring.c File Reference

Extra string routines for C.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include "xstring/xstring.h"
Macros

- `#define DELIMITER ((char)0xff)`

Functions

- `char * xstr_dup (char const *src)`
  
  Duplicate the given null-terminated string.

- `char * xstrncpy (char *dst, char const *src, size_t nchars)`
  
  Safe string copy.

- `char * xstr_cat (char const *first,...)`
  
  Multiple string concatenation.

- `char * xstr_join (char *const array[], char delim)`
  
  Multiple string concatenation (array of strings)

- `char ** xstrSplit (char const *full_string, char const *delim)`
  
  Split a string into multiple substrings.

- `size_t xstr_match_first (const char *needle,...)`
  
  Match a string to a list of parameter strings.

- `size_t xstr_match_first_a (const char *needle, char *const *haystack)`
  
  Find a string in an array of strings.

- `char * xstr_rtrim (char *str)`
  
  Removes all trailing whitespace from a string.

- `char * xstr_ltrim (char *str)`
  
  Removes all leading whitespace from a string.

- `char * xstr_trim (char *str)`
  
  Removes all leading and trailing whitespace from a string.

- `char ** xstr_cpyarray (char **array)`
  
  Copies an array of strings (such as argv)

- `void xstr_delarray (char **array)`
  
  Deletes all resources used by a dynamically allocated array of strings.

- `char * xstr_readfile (char *filename)`
  
  Reads and returns the given file as a single C-style string.

- `bool xstr_file_readable (char *filename)`
  
  Tests if the given file can opened for reading.

- `bool xstr_file_writable (char *filename)`
  
  Tests if the given file can opened for writing.

3.11.1 Detailed Description

Extra string routines for C. xstring is a library of routines to provide easy string manipulation in C. xstring is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

Author

Lelanthran Krishna Manickum
3.11.2 Function Documentation

3.11.2.1 char* xstr_cat ( char const* first, ... )

Multiple string concatenation.
Concatenates all the given strings in the order that they are presented in. The argument list must end with a NULL pointer. The return value is malloc’ed by the function so the caller is responsible for freeing the result.

Parameters

| in | first | The mandatory first string in the list of strings to concatenate. The list of arguments to this function must end with a NULL pointer. |

Returns

On success, a newly allocated block of memory which contains all the given strings concatenated into a single large string. On failure NULL is returned. The caller must free the returned pointer on success.

```c
int main (void) {
    char t1[] = "Hello ";
    char *t2 = "World, ";
    char *result = xstr_cat (t1, t2, "everyone ", "\n", NULL);
    // result now contains the string "Hello World, everyone 
"
    printf ("%s", result);
    // Caller responsible for freeing the return from xstr_cat()
    free (result);
    return EXIT_SUCCESS;
}
```

3.11.2.2 char** xstr_cpyarray ( char ** array )

Copies an array of strings (such as argv)

Returns a copy of array, which must be an array of strings terminated with a NULL pointer. The array array must be of the same form as

```c
char *array[] = {"one", "two", ..., NULL};
```

The returned array is malloced (and thus must be freed by the caller) and each element in the returned array is also malloced (and thus must be freed by the caller). The caller should use xstr_delarray() to free the returned value.

See Also

xstr_delarray()

Parameters

| in | array | The array to make a copy of. |

Returns

On success a copy of the array array is returned. The return value is malloced and must be freed by the caller. Each element in the returned array is also malloced and must be freed by the caller. The caller should use xstr_delarray() to free the returned array. On any failure, NULL is returned.

Example: Copy the command line arguments and print the copy.

```c
int main (int argc, char **argv) {
    char t1[] = "Hello ";
    char *t2 = "World, ";
    char *result = xstr_cat (t1, t2, "everyone ", "\n", NULL);
    // result now contains the string "Hello World, everyone 
"
    printf ("%s", result);
    // Caller responsible for freeing the return from xstr_cat()
    free (result);
    return EXIT_SUCCESS;
}
```
{ char **argv_cpy = xstr_cpyarray (argv);
  argc = argc;
  for (size_t i=0; argv_cpy[i]; i++) {
    printf (%zu: '%s'


3.11.2.3 void xstr_delarray ( char ** array )

Deletes all resources used by a dynamically allocated array of strings.

Deletes the array array, which must be an array of strings. The array must be terminated with a NULL pointer.

The array array is deleted by first iterating through all the elements and freeing each one until the terminating NULL element is found, after which the array array itself is freed.

For example, making a copy of the command line arguments in argv would result in the structure that xstr_delarray() will delete and free.

See Also

    xstr_cpyarray()

Parameters

| in       | array | The array to delete |

Returns

    Nothing

Example: Copy the command line arguments and print the copy.

int main (int argc, char **argv)
{
  char **argv_cpy = xstr_cpyarray (argv);
  argc = argc;
  for (size_t i=0; argv_cpy[i]; i++) {
    printf (%zu: '%s'

3.11.2.4 char * xstr_dup ( char const * src )

Duplicate the given null-terminated string.

Duplicates the given string by allocating space using malloc and copying the given string into the newly allocated space. The caller is responsible for freeing the return value.

Parameters

| in       | src    | The source string to be duplicated. |
Returns

On success, a newly allocated block of memory which contains the src string. On failure NULL is returned. The caller must free the returned pointer on success.

3.11.2.5 bool xstr_file_readable ( char * filename )

Tests if the given file can be opened for reading.

Attempts to open the file filename for reading; file filename is non-destructively opened and then closed. No changes are made to the file filename.

If the file specified by filename doesn't exist or is otherwise unreadable (permissions might not allow, for example), then false is returned. If the file was successfully opened and closed, true is returned.

NOTE: Many of the libxc functions which require a filename as an argument will attempt to open the given filename and return failure if the attempt to open the file was unsuccessful. A well-behaved program will use xstr_file_readable() to first ensure that the file exists and can be read before calling such functions.

See Also

xstr_readfile(), xstr_file_writable()

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>filename</th>
<th>The filename to test for reading</th>
</tr>
</thead>
</table>

Returns

If the file filename is successfully opened for reading then true is returned. If any errors occurred that prevented the successful opening and closing of the file filename for reading then false is returned.

Example:

```c
int main (void)
{
  if (xstr_file_readable("input.txt")) {
    printf("file input.txt is readable\n");
  } else {
    printf("file input.txt is not readable\n");
  }
  ...
  return EXIT_SUCCESS;
}
```

3.11.2.6 bool xstr_file_writable ( char * filename )

Tests if the given file can be opened for writing.

Attempts to open the file filename for writing; file filename is non-destructively opened and then closed. No changes are made to the file filename.

If the file specified by filename can neither be created for writing nor opened for writing (permissions might not allow writing, for example), then false is returned. If the file was successfully opened and closed, true is returned and the file is left unchanged.

NOTE: Before writing any file, a well-behaved program will first use xstr_file_readable() to check if the file already exists. If it does then the user should be prompted to confirm that they are sure that they want to overwrite the existing file.
See Also

`xstr_readfile()`, `xstr_file_readable()`

Parameters

| in         | `filename` | The filename to test for writing |

Returns

If the file `filename` is successfully opened for writing then true is returned. If any errors occurred that prevented the successful opening and closing of the file `filename` for writing then false is returned.

Example:

```c
int main (void)
{
    if (xstr_file_writable ("output.txt")) {
        printf ("file output.txt is writable\n");
    } else {
        printf ("file output.txt is not writable\n");
    }
    ...
    return EXIT_SUCCESS;
}
```

3.11.2.7 `char * xstr_join ( char *const array[], char delim )`

Multiple string concatenation (array of strings)

Concatenates all the given strings in the array `array` in the order that they are found in the array `array`, optionally separated with delimiter `delim`. The final element of the array `array` must be a `NULL` pointer because `xstr_join()` doesn’t receive the length of the array. The final NULL pointer in the array signifies the end of the array to `xstr_join()`, thus `xstr_join()` will stop concatenation upon reaching the first element in the array that evaluates to NULL.

The return value is malloc’ed by the function so the caller is responsible for freeing the result.

Parameters

| in         | `array` | An array of char pointers, representing the array of strings that are to be concatenated. The last element of this array must be a NULL pointer. |
| in         | `delim` | Optional delimiter for the resulting string. If non-zero, this character is used to separate each of the strings from `array` in the result. |

Returns

On success, a newly allocated block of memory which contains all the given strings concatenated into a single large string, optionally separated by `delim`. On failure NULL is returned. The caller must free the returned pointer on success.

```c
int main (void)
{
    char *a_of_s[] = {"One ", "Two ", "Three ", "Four ", "Five ", NULL};
    char *result = xstr_join (a_of_s, 0);
    printf ("a_of_s = '%s'
", result);
    // outputs: a_of_s = 'One Two Three Four Five'
    free (result);

    result = xstr_join (a_of_s, ':');
    printf ("a_of_s = '%s'
", result);
    // outputs: a_of_s = 'One :Two :Three :Four :Five'
    free (result);
```
3.11.2.8 char* xstr_ltrim ( char * str )

Removes all leading whitespace from a string.

Removes all leading whitespace from the given string str. The string str is modified in place and no copies are made. The string str is not reallocated.

When str is empty the string str is returned unchanged. When str is NULL, NULL is returned. Should no leading whitespace be found then the string str is returned unchanged. Should the string str be nothing but whitespace, then str is made into an empty string and returned.

See Also

xstr_rtrim(), xstr_trim()

Parameters

| in, out | str | The string to trim off the leading whitespace from. |

Returns

The str that was passed into the function is returned, with any leading whitespace found in it removed.

Example:

```c
int main (void)
{
    char *test_string = xstr_dup (" \t \n testing");
xstr_ltrim (test_string);
printf ("--%s--\n", test_string); // Outputs '--testing--'
free (test_string);
return EXIT_SUCCESS;
}
```

3.11.2.9 size_t xstr_match_first ( const char * needle, ... )

Match a string to a list of parameter strings.

Given a string, needle, and a list of strings to examine, xstr_match_first() will determine which of the strings in (...) match needle, and return the position of that string in (...)

For example the caller can use xstr_match_first() in a switch statement that decides based on strings which decision path to take (see example below). The argument list (...) must be terminated with a NULL pointer.

Note that xstr_match_first() uses strcmp to match needle and thus will incur much overhead. It is recommended that xstr_match_first() not be used in loops with many iterations.

See Also

xstr_match_first_a()

Parameters

| in | needle | The string to match all the others against |
| in | ... | The set of strings to search for needle, terminated with a NULL |
Returns

On success the index (starting from 0) of the string in (...) that matches needle is returned. On failure or failure to match, (size_t)-1 is returned.

Example:

```c
int main (void)
{
    char input[25];
    memset (input, 0, sizeof input);
    printf ("Enter command to execute: \n");
    fgets (input, sizeof input - 1, stdin);
    switch (xstr_match_first (input, "play",
                               "pause",
                               "restart",
                               "stop",
                               NULL))
    {
        case 0: // take action for "play"; break
        case 1: // take action for "pause"; break
        case 2: // take action for "restart"; break
        case 3: // take action for "stop"; break
        case (size_t)-1:
            // Unknown input
        default:
    }
    return EXIT_SUCCESS;
}
```

3.11.2.10 size_t xstr_match_first_a (const char *needle, char **const haystack)

Find a string in an array of strings.

Given a string, needle, and array haystack of strings to examine, xstr_match_first_a() will determine which of the strings in haystack match needle, and return the position of that string in haystack.

For example the caller can use xstr_match_first_a() in a switch statement that decides based on strings which decision path to take (see example below). The array haystack must be terminated with a NULL pointer.

Note that xstr_match_first_a() uses strcmp to match needle and thus will incur much overhead. It is recommended that xstr_match_first_a() not be used in loops with many iterations.

See Also

xstr_match_first()

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>needle</th>
<th>The string to match all the others against</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>haystack</td>
<td>The array of strings to search for needle, terminated with a NULL pointer</td>
</tr>
</tbody>
</table>

Returns

On success the index (starting from 0) of the string in haystack that matches needle is returned. On failure or failure to match, (size_t)-1 is returned.

Example:

```c
int main (void)
{
    char *commands[] = { "play", "pause", "restart", "stop", NULL, }; char input[25];
    memset (input, 0, sizeof input);
    printf ("Enter command to execute: \n");
```
fgets (input, sizeof input - 1, stdin);
switch (xstr_match_first_a (input, commands))
{
    case 0: // take action for "play"; break;
    case 1: // take action for "pause"; break;
    case 2: // take action for "restart"; break;
    case 3: // take action for "stop"; break;
    case (size_t)-1:
        default:    // Unknown input
            break;
    }
return EXIT_SUCCESS;

3.11.2.11  char xstrncpy ( char * dst, char const * src, size_t nchars )

Safe string copy.
Safely copy the src string to the location pointed to by dst, copying at most nchars - 1 characters. The copied string is properly null-terminated.

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>out</th>
<th>dst</th>
<th>The destination to copy src to. The caller must ensure that at least nchars characters will fit into dst.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>src</td>
<td>The source string to be copied.</td>
<td></td>
</tr>
<tr>
<td>in</td>
<td>nchars</td>
<td>The number of characters that will fit into dst.</td>
<td></td>
</tr>
</tbody>
</table>

Returns
On success dst is returned and dst will contain a copy of src of up to nchars - 1 characters. On failure NULL is returned.

Example:

```c
int main (void)
{
    char t1[] = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
    char *t2 = "Hello World!";
    char *result = xstrncpy (t1, t2, sizeof t1);
    // result now contains the string "Hello World!"
    return EXIT_SUCCESS;
}
```

3.11.2.12  char xstr_readfile ( char * filename )

Reads and returns the given file as a single C-style string.
Reads the file filename and returns the entire file contents as a single C-style NULL-terminated string. The caller must ensure that the returned NULL-terminated string is freed.
If the file specified by filename doesn't exist or is otherwise unreadable (permissions might not allow, for example), then NULL is returned.

See Also
xstr_file_readable()

Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>in</th>
<th>filename</th>
<th>The filename to read and return as a single string</th>
</tr>
</thead>
</table>

Generated on Wed Apr 17 2013 18:38:19 for Extended C Library by Doxygen
Returns

A single string, terminated with a NULL character, that contains the contents of the file filename

Example: Program that prints its own source

```c
int main (void)
{
    char *myself = xstr_readfile ("main.c");
    if (myself) {
        printf ("Read in %zu bytes in main.c\n", strlen (myself));
        printf ("%s\n", myself);
        free (myself);
    }
    return EXIT_SUCCESS;
}
```

3.11.2.13 char * xstr_rtrim ( char * str )

Removes all trailing whitespace from a string.

Removes all trailing whitespace from the given string str. The string str is modified in place and no copies are made. The string str is not reallocated.

When str is empty the string str is returned unchanged. When str is NULL, NULL is returned. Should no trailing whitespace be found then the string str is returned unchanged. Should the string str be nothing but whitespace, then str is made into an empty string and returned.

See Also

xstr_ltrim(), xstr_trim()

Parameters

| in, out | str | The string to trim off the trailing whitespace from. |

Returns

The str that was passed into the function is returned, with any trailing whitespace found in it removed.

Example:

```c
int main (void)
{
    char *test_string = xstr_dup ("testing \t \n ");
    xstr_rtrim (test_string);
    printf ("--%s--\n", test_string); // Outputs '--testing--'
    free (test_string);
    return EXIT_SUCCESS;
}
```

3.11.2.14 char** xstr_split ( char const * full_string, char const * delim )

Split a string into multiple substrings.

Splits the string full_string into multiple substrings using delim as the set of characters that delimit the substrings in the original string. Any delimiting character that appears in the string that is not a delimiter should be escaped.

For example, given full_string = "this,is,a,string:that.is.delimited" and the set of delimiters as delim = ",:;", xstr_split() will return the set of strings "this", "is", "a", "string", "that", "is" and "delimited". If full_string ends with a delimiter before the terminating NULL, then the final empty field is omitted.
The caller is responsible for freeing the result (See section "RETURNS" and the example below for more information).

### Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>full_string</th>
<th>A null-terminated C string</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>delim</td>
<td>The set of delimiters to use, in the form of a null-terminated C string</td>
</tr>
</tbody>
</table>

### Returns

On success, a newly allocated array which contains all the substrings found, with the final element of the array being NULL. Each substring in the array is also newly allocated. The caller is responsible for freeing each substring and for freeing the array itself. On failure, NULL is returned.

```c
int main (void) {
    char *mf_string = "\.this is a:string\,that?is\'delimited..";
    char *delims = ":\.,';";
    char **result = xstr_split (mf_string, delims);
    // Print out each element of result, and then free it
    for (size_t i=0; result && result[i]; i++) {
        printf ("substr='%s'\n", result[i]);
        free (result[i]);
    }
    // Free the entire array
    if (result) free (result);
    return EXIT_SUCCESS;
}
```

### 3.11.2.15 char ∗ xstr_trim ( char ∗ str )

Removes all leading and trailing whitespace from a string.

Removes all leading and trailing whitespace from the given string str. The string str is modified in place and no copies are made. The string str is not reallocated.

When str is empty the string str is returned unchanged. When str is NULL, NULL is returned. Should neither leading whitespace nor trailing whitespace be found then the string str is returned unchanged. Should the string str be nothing but whitespace, then str is made into an empty string and returned.

See Also

xstr_rtrim(), xstr_trim()

### Parameters

| in,out | str | The string to trim off the leading and trailing whitespace from. |

### Returns

The str that was passed into the function is returned, with any leading and trailing whitespace found in it removed.

**Example:**

```c
int main (void) {
    char *test_string = xstr_dup (" \t \n testing \t \n ");
    xstr_trim (test_string);
    printf ("--%s--\n", test_string); // Outputs '--testing--'
    free (test_string);
}
```
3.12 xstring/xstring.c File Reference

Extra string routines for C.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include <ctype.h>
#include "xstring/xstring.h"
```

Macros

- `#define DELIMITER ((char)0xff)`

Functions

- `char * xstr_dup (char const *src)`
  Duplicate the given null-terminated string.
- `char * xstrncpy (char *dst, char const *src, size_t nchars)`
  Safe string copy.
- `char * xstr_cat (char const *first,....)`
  Multiple string concatenation.
- `char * xstr_join (char *const array[], char delim)`
  Multiple string concatenation (array of strings)
- `size_t xstr_split (char const *full_string, char const *delim)`
  Split a string into multiple substrings.
- `size_t xstr_match_first (const char *needle,....)`
  Match a string to a list of parameter strings.
- `size_t xstr_match_first_a (const char *needle, char *const *haystack)`
  Find a string in an array of strings.
- `char * xstr_rtrim (char *str)`
  Removes all trailing whitespace from a string.
- `char * xstr_ltrim (char *str)`
  Removes all leading whitespace from a string.
- `char * xstr_trim (char *str)`
  Removes all leading and trailing whitespace from a string.
- `char ** xstr_cpyarray (char **array)`
  Copies an array of strings (such as argv)
- `void xstr_delarray (char **array)`
  Deletes all resources used by a dynamically allocated array of strings.
- `char * xstr_readfile (char *filename)`
  Reads and returns the given file as a single C-style string.
- `bool xstr_file_readable (char *filename)`
  Tests if the given file can opened for reading.
- `bool xstr_file_writable (char *filename)`
  Tests if the given file can opened for writing.
3.12 Detailed Description

Extra string routines for C. xstring is a library of routines to provide easy string manipulation in C. xstring is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

Author

Lelanthran Krishna Manickum

3.12.2 Function Documentation

3.12.2.1 char ∗ xstr_cat ( char const ∗ first, ... )

Multiple string concatenation.

Concatenates all the given strings in the order that they are presented in. The argument list must end with a NULL pointer. The return value is malloc'ed by the function so the caller is responsible for freeing the result.

Parameters

| in  | first | The mandatory first string in the list of strings to concatenate. The list of arguments to this function must end with a NULL pointer. |

Returns

On success, a newly allocated block of memory which contains all the given strings concatenated into a single large string. On failure NULL is returned. The caller must free the returned pointer on success.

```c
int main (void)
{
    char t1[] = "Hello ";
    char *t2 = "World,";
    char *result = xstr_cat (t1, t2, "everyone ", "\n", NULL);
    // result now contains the string "Hello World, everyone \n"
    printf ("%s", result);
    // Caller responsible for freeing the return from xstr_cat()
    free (result);
    return EXIT_SUCCESS;
}
```

3.12.2.2 char ∗∗ xstr_cpyarray ( char ∗∗ array )

Copies an array of strings (such as argv)

Returns a copy of array, which must be an array of strings terminated with a NULL pointer. The array array must be of the same form as

```c
char *array[] = {"one", "two", ..., NULL};
```

The returned array is malloced (and thus must be freed by the caller) and each element in the returned array is also malloced (and thus must be freed by the caller). The caller should use xstr_delarray() to free the returned value.

See Also

xstr_delarray()

Parameters

| in  | array | The array to make a copy of. |

Generated on Wed Apr 17 2013 18:38:19 for Extended C Library by Doxygen
Returns

On success a copy of the array `array` is returned. The return value is malloced and must be freed by the caller. Each element in the returned array is also malloced and must be freed by the caller. The caller should use `xstr_delarray()` to free the returned array. On any failure, NULL is returned.

Example: Copy the command line arguments and print the copy.

```c
int main (int argc, char **argv)
{
    char **argv_cpy = xstr_cpyarray (argv);
    argc = argc;
    for (size_t i=0; argv_cpy[i]; i++) {
        printf ("%zu: '%s'\n", i, argv_cpy[i]);
    }
    xstr_delarray (argv_cpy);
    return EXIT_SUCCESS;
}
```

3.12.2.3 `void xstr_delarray ( char ** array )`

Deletes all resources used by a dynamically allocated array of strings.

Deletes the array `array`, which must be an array of strings. The array must be terminated with a NULL pointer.

The `array` `array` is deleted by first iterating through all the elements and freeing each one until the terminating NULL element is found, after which the array `array` itself is freed.

For example, making a copy of the command line arguments in `argv` would result in the structure that `xstr_delarray()` will delete and free.

See Also

`xstr_cpyarray()`

Parameters

| in | array | The array to delete |

Returns

Nothing

Example: Copy the command line arguments and print the copy.

```c
int main (int argc, char **argv)
{
    char **argv_cpy = xstr_cpyarray (argv);
    argc = argc;
    for (size_t i=0; argv_cpy[i]; i++) {
        printf ("%zu: '%s'\n", i, argv_cpy[i]);
    }
    xstr_delarray (argv_cpy);
    return EXIT_SUCCESS;
}
```

3.12.2.4 `char* xstr_dup ( char const * src )`

Duplicate the given null-terminated string.

Duplicates the given string by allocating space using malloc and copying the given string into the newly allocated space. The caller is responsible for freeing the return value.
Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>src</td>
<td>The source string to be duplicated.</td>
</tr>
</tbody>
</table>

Returns

On success, a newly allocated block of memory which contains the src string. On failure NULL is returned. The caller must free the returned pointer on success.

3.12.2.5 bool xstr_file_readable (char * filename)

Tests if the given file can opened for reading.

Attempts to open the file filename for reading; file filename is non-destructively opened and then closed. No changes are made to the file filename.

If the file specified by filename doesn’t exist or is otherwise unreadable (permissions might not allow, for example), then false is returned. If the file was successfully opened and closed, true is returned.

NOTE: Many of the libxc functions which require a filename as an argument will attempt to open the given filename and return failure if the attempt to open the file was unsuccessful. A well-behaved program will use xstr_file_readable() to first ensure that the file exists and can be read before calling such functions.

See Also

xstr_readfile(), xstr_file_writable()

Parameters

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>filename</td>
<td>The filename to test for reading</td>
</tr>
</tbody>
</table>

Returns

If the file filename is successfully opened for reading then true is returned. If any errors occurred that prevented the successful opening and closing of the file filename for reading then false is returned.

Example:

```c
int main (void)
{
  if (xstr_file_readable("input.txt")) {
    printf("file input.txt is readable\n");
  } else {
    printf("file input.txt is not readable\n");
  }
  ...
  return EXIT_SUCCESS;
}
```

3.12.2.6 bool xstr_file_writable (char * filename)

Tests if the given file can opened for writing.

Attempts to open the file filename for writing; file filename is non-destructively opened and then closed. No changes are made to the file filename.

If the file specified by filename can neither be created for writing nor opened for writing (permissions might not allow writing, for example), then false is returned. If the file was successfully opened and closed, true is returned and the file is left unchanged.
NOTE: Before writing any file, a well-behaved program will first use `xstr_file_readable()` to check if the file already exists. If it does then the user should be prompted to confirm that they are sure that they want to overwrite the existing file.

See Also

- `xstr_readfile()`, `xstr_file_readable()`

Parameters

| in | filename | The filename to test for writing |

Returns

If the file `filename` is successfully opened for writing then `true` is returned. If any errors occurred that prevented the successful opening and closing of the file `filename` for writing then `false` is returned.

Example:

```c
int main (void)
{
    if (xstr_file_writable ("output.txt")) {
        printf ("file output.txt is writable\n");
    } else {
        printf ("file output.txt is not writable\n");
    }
    ...
    return EXIT_SUCCESS;
}
```

3.12.2.7 `char *xstr_join ( char *const array[], char delim )`

Multiple string concatenation (array of strings)

Concatenates all the given strings in the array `array` in the order that they are found in the array `array`, optionally separated with delimiter `delim`. The final element of the array `array` must be a `NULL` pointer because `xstr_join()` doesn’t receive the length of the the array. The final `NULL` pointer in the array signifies the end of the array to `xstr_join()`, thus `xstr_join()` will stop concatenation upon reaching the first element in the array that evaluates to `NULL`.

The return value is malloc’ed by the function so the caller is responsible for freeing the result.

Parameters

| in | array | An array of char pointers, representing the array of strings that are to be concatenated. The last element of this array must be a NULL pointer. |

| in | delim | Optional delimiter for the resulting string. If non-zero, this character is used to separate each of the strings from `array` in the result. |

Returns

On success, a newly allocated block of memory which contains all the given strings concatenated into a single large string, optionally separated by `delim`. On failure `NULL` is returned. The caller must free the returned pointer on success.

```c
int main (void)
{
    char *a_of_s[] = {"One ", "Two ", "Three ", "Four ", "Five ", NULL};
    char *result = xstr_join (a_of_s, 0);
    printf ("a_of_s = \'\%s\'\n", result);
    // outputs: a_of_s = 'One Two Three Four Five'
```
free (result);
result = xstr_join (a_of_s, ':');
printf ("a_of_s = '%s'
", result);
// outputs:  a_of_s = 'One :Two :Three :Four :Five'
free (result);
return EXIT_SUCCESS;
}

3.12.2.8 char ∗ xstr_ltrim ( char ∗ str )

Removes all leading whitespace from a string.

Removes all leading whitespace from the given string str. The string str is modified in place and no copies are made. The string str is not reallocated. When str is empty the string str is returned unchanged. When str is NULL, NULL is returned. Should no leading whitespace be found then the string str is returned unchanged. Should the string str be nothing but whitespace, then str is made into an empty string and returned.

See Also
xstr_rtrim(), xstr_trim()

Parameters

| in, out | str | The string to trim off the leading whitespace from. |

Returns

The str that was passed into the function is returned, with any leading whitespace found in it removed.

Example:

```c
int main (void)
{
    char ∗ test_string = xstr_dup (" \t \n testing");
xstr_ltrim (test_string);
printf ("--%s--\n", test_string); // Outputs '--testing--'
free (test_string);
return EXIT_SUCCESS;
}
```

3.12.2.9 size_t xstr_match_first ( const char ∗ needle, ... )

Match a string to a list of parameter strings.

Given a string, needle, and a list of strings to examine, xstr_match_first() will determine which of the strings in (...) match needle, and return the position of that string in (....)

For example the caller can use xstr_match_first() in a switch statement that decides based on strings which decision path to take (see example below). The argument list (...) must be terminated with a NULL pointer.

Note that xstr_match_first() uses strcmp to match needle and thus will incur much overhead. It is recommended that xstr_match_first() not be used in loops with many iterations.

See Also
xstr_match_first_a()
Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>needle</th>
<th>The string to match all the others against</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>haystack</td>
<td>The set of strings to search for needle, terminated with a NULL</td>
</tr>
</tbody>
</table>

Returns

On success the index (starting from 0) of the string in (...) that matches needle is returned. On failure or failure to match, (size_t)-1 is returned.

Example:

```c
int main (void)
{
    char input[25];
    memset (input, 0, sizeof input);
    printf ("Enter command to execute: \n");
    fgets (input, sizeof input - 1, stdin);
    switch (xstr_match_first (input, "play",
                              "pause",
                              "restart",
                              "stop",
                              NULL))
    {
    case 0: // take action for "play"; break
    case 1: // take action for "pause"; break
    case 2: // take action for "restart"; break
    case 3: // take action for "stop"; break
    case (size_t)-1:
        default: // Unknown input
            return EXIT_SUCCESS;
    }
}
```

3.12.2.10 size_t xstr_match_first_a ( const char * needle, char const *const * haystack )

Find a string in an array of strings.

Given a string, needle, and array haystack of strings to examine, xstr_match_first() will determine which of the strings in haystack match needle, and return the position of that string in haystack.

For example the caller can use xstr_match_first_a() in a switch statement that decides based on strings which decision path to take (see example below). The array haystack must be terminated with a NULL pointer.

Note that xstr_match_first_a() uses strcmp to match needle and thus will incur much overhead. It is recommended that xstr_match_first_a() not be used in loops with many iterations.

See Also

xstr_match_first()
int main (void)
{
    char *commands[] = { "play", "pause", "restart", "stop", NULL, };  
    char input[25];
    memset (input, 0, sizeof input);
    printf ("Enter command to execute: 
");
    fgets (input, sizeof input - 1, stdin);
    switch (xstr_match_first_a (input, commands))
    {
        case 0: // take action for "play";
            break;
        case 1: // take action for "pause";
            break;
        case 2: // take action for "restart";
            break;
        case 3: // take action for "stop";
            break;
        case (size_t)-1:
            default:
            // Unknown input
            break;
    }
    return EXIT_SUCCESS;
}

3.12.2.11 char *xstrncpy(char *dst, char const *src, size_t nchars) 

Safe string copy.

Safely copy the src string to the location pointed to by dst, copying at most nchars - 1 characters. The copied string is properly null-terminated.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>out dst</td>
<td>The destination to copy src to. The caller must ensure that at least nchars characters will fit into dst.</td>
</tr>
<tr>
<td>in src</td>
<td>The source string to be copied.</td>
</tr>
<tr>
<td>in nchars</td>
<td>The number of characters that will fit into dst.</td>
</tr>
</tbody>
</table>

Returns

On success dst is returned and dst will contain a copy of src of up to nchars - 1 characters. On failure NULL is returned.

Example:

int main (void)
{
    char t1[] = "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx";
    char *t2 = "Hello World!";
    char *result = xstrncpy (t1, t2, sizeof t1);
    // result now contains the string "Hello World!"
    return EXIT_SUCCESS;
}

3.12.2.12 char *xstr_readfile(char *filename)

Reads and returns the given file as a single C-style string.

Reads the file filename and returns the entire file contents as a single C-style NULL-terminated string. The caller must ensure that the returned NULL-terminated string is freed.

If the file specified by filename doesn’t exist or is otherwise unreadable (permissions might not allow, for example), then NULL is returned.
See Also:

- `xstr_file_readable()`

Parameters:

| in   | filename | The filename to read and return as a single string |

Returns:

A single string, terminated with a `NULL` character, that contains the contents of the file `filename`

**Example:** Program that prints its own source

```c
int main (void)
{
    char *myself = xstr_readfile ("main.c");
    if (myself) {
        printf ("Read in %zu bytes in main.c\n", strlen (myself));
        printf ("%s\n", myself);
        free (myself);
    } return EXIT_SUCCESS;
}
```

3.12.2.13 char ∗xstr_rtrim ( char ∗str )

Removes all trailing whitespace from a string.

Removes all trailing whitespace from the given string `str`. The string `str` is modified in place and no copies are made. The string `str` is not reallocated.

When `str` is empty the string `str` is returned unchanged. When `str` is `NULL`, `NULL` is returned. Should no trailing whitespace be found then the string `str` is returned unchanged. Should the string `str` be nothing but whitespace, then `str` is made into an empty string and returned.

See Also:

- `xstr_ltrim()`, `xstr_trim()`

Parameters:

| in,out | str | The string to trim off the trailing whitespace from. |

Returns:

The `str` that was passed into the function is returned, with any trailing whitespace found in it removed.

**Example:**

```c
int main (void)
{
    char *test_string = xstr_dup ("testing \t \n ");
    xstr_rtrim (test_string);
    printf ("--%s--\n", test_string); // Outputs '--testing--'
    free (test_string);
    return EXIT_SUCCESS;
}
```
3.12.2.14 char** xstr_split ( char const * full_string, char const * delim )

Splits the string into multiple substrings.

Splits the string `full_string` into multiple substrings using `delim` as the set of characters that delimit the substrings in the original string. Any delimiter character that appears in the string that is not a delimiter should be escaped.

For example, given `full_string = "this,is,a,string:that.is.delimited"` and the set of delimiters as `delim = ":,\."` `xstr_split()` will return the set of strings "this", "is", "a", "string", "that", "is" and "delimited". If `full_string` ends with a delimiter before the terminating NULL, then the final empty field is omitted.

The caller is responsible for freeing the result (See section "RETURNS" and the example below for more information).

Parameters

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>full_string</td>
<td>A null-terminated C string</td>
</tr>
<tr>
<td>in</td>
<td>delim</td>
<td>The set of delimiters to use, in the form of a null-terminated C string</td>
</tr>
</tbody>
</table>

Returns

On success, a newly allocated array which contains all the substrings found, with the final element of the array being NULL. Each substring in the array is also newly allocated. The caller is responsible for freeing each substring and for freeing the array itself. On failure, NULL is returned.

```c
int main (void)
{
    char *mf_string = ".this is.a:string\,that?is\'delimited..";
    char *delims = ":,\.";
    char **result = xstr_split (mf_string, delims);
    // Print out each element of result, and then free it
    for (size_t i=0; result && result[i]; i++) {
        printf ("substr='%s'\n", result[i]);
        free (result[i]);
    }
    // Free the entire array
    if (result) free (result);
    return EXIT_SUCCESS;
}
```

3.12.2.15 char* xstr_trim ( char * str )

Removes all leading and trailing whitespace from a string.

Removes all leading and trailing whitespace from the given string `str`. The string `str` is modified in place and no copies are made. The string `str` is not reallocated.

When `str` is empty the string `str` is returned unchanged. When `str` is NULL, NULL is returned. Should neither leading whitespace nor trailing whitespace be found then the string `str` is returned unchanged. Should the string `str` be nothing but whitespace, then `str` is made into an empty string and returned.

See Also

`xstr_rtrim()`, `xstr_trim()

Parameters

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>in,out</td>
<td>str</td>
<td>The string to trim off the leading and trailing whitespace from.</td>
</tr>
</tbody>
</table>
Returns

The string that was passed into the function is returned, with any leading and trailing whitespace found in it removed.

Example:

```c
int main (void)
{
    char *test_string = xstr_dup (" \t \n testing \t \n ");
    xstr_trim (test_string);
    printf ("--%s--\n", test_string); // Outputs ‘--testing--’
    free (test_string);
    return EXIT_SUCCESS;
}
```

3.13 libxc-0.0.4/xtree/xtree.c File Reference

Create and traverse a tree structure.

```c
#include "xvector/xvector.h"
#include "xtree/xtree.h"
```

Functions

- `xtree_t * xtree_add_child (xtree_t *parent, void *element)`
  - Create and add a node to a parent node.
- `void * xtree_del (xtree_t *node)`
  - Deletes a node and all its children.
- `xtree_t * xtree_get_child (xtree_t *node, size_t i)`
  - Returns a child node of the specified node.
- `xtree_t * xtree_get_parent (xtree_t *node)`
  - Retrieve the parent of this node.
- `xtree_t * xtree_get_payload (xtree_t *node)`
  - Retrieves the payload stored in this node.
- `void xtree_apply (xtree_t *node, void (*)(void *, void *), void *extra)`
  - Traverse the tree, applying a function to each node.
- `xtree_t * xtree_map (xtree_t *node, void (*)(void *, void *), void *extra)`
  - Traverse the tree, applying a function to each node and storing the result.

3.13.1 Detailed Description

Create and traverse a tree structure. A C library to create and traverse multi-children trees. Each child node points to a single parent node and each parent node may have multiple child nodes. Child nodes are stored in an ordered list for each parent, in the order that they are created. Each node whether parent or child may also store a payload - the element of data that is to be stored in the tree.

xtree is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

Author

Lelanthran Krishna Manickum
3.13.2 Function Documentation

3.13.2.1 xtree_t* xtree_add_child ( xtree_t * parent, void * element )

Create and add a node to a parent node.

Creates and returns a new node, with payload as the payload and parent as the parent. If parent is NULL then the node that is created and returned is a root node of a tree and can be used as a parent for subsequent xtree_add_child() invocations.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>parent</th>
<th>The parent of the node to be created. If parent is NULL then the returned node is a toplevel node of a new tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>element</td>
<td>The payload to be stored in the newly created node</td>
</tr>
</tbody>
</table>

Returns

On success a newly created node with payload set to payload and parent set to parent is returned. On failure NULL is returned.

3.13.2.2 void xtree_apply ( xtree_t * node, void(*)(void *, void *) func, void * extra )

Traverse the tree, applying a function to each node.

The tree node is traversed and on each visit to a node the function func is applied on the payload of the node. The tree is traversed bottom up, one branch at a time starting with the first stored child on each parent.

The function func should accept two arguments and return nothing. The first argument to func is the payload of the node and the second argument is the caller supplied extra argument. In this way, using the extra argument, the caller has flexibility to write predicate functions for side-effects.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>node</th>
<th>The tree to traverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>func</td>
<td>The function to apply to each node’s payload as the tree is traversed. The first argument to func is the node’s payload data and the second argument is the extra argument supplied to xtree_apply()</td>
</tr>
<tr>
<td>in</td>
<td>extra</td>
<td>An extra argument that forms the second argument when func is called on a node’s payload (the first argument to func is the node’s payload data)</td>
</tr>
</tbody>
</table>

Returns

Nothing.

Example:

```c
// Assuming that the tree is being used to store char pointers
static void print_payload (void *p, void *f)
{
    char *payload = p;
    FILE *outf = f;
    fprintf (outf, "%s\n", payload);
}

int main (void)
{
    xtree_t *tree_of_strings;
    ...
    // Print out entire tree
    xtree_apply (tree_of_strings, print_payload, stdout);
```
void xtree_del ( xtree_t * node )

Deletes a node and all its children.

Deletes the node *node*, together with all its children. All resources used by the node and its children other than the payload data is freed.

**Parameters**

| in  | node | The node to delete |

**Returns**

On success the payload of the deleted node is returned (the payload of the node's children are discarded). It might be useful to the caller to receive the last payload value of the node before it was freed. On failure *NULL* is returned.

xtree_t* xtree_get_child ( xtree_t * node, size_t i )

Returns a child node of the specified node.

Returns the *i*th child of this node (children in a node are stored in the order that they are created).

**Parameters**

| in  | node | The node that contains the child that the caller is asking for. |
| in  | i    | The index of the child within the *node* specified above |

**Returns**

On success the *i*th child of this node is returned. On failure *NULL* is returned.

xtree_t* xtree_get_parent ( xtree_t * node )

Retrieves the parent of this node.

Returns the parent node of *node*, if it exists. If *node* or its parent does not exist, returns *NULL* instead.

**Parameters**

| in  | node | The node to examine for a parent |

**Returns**

*NULL* is returned if either *node* or its parent does not exist, in all other cases the parent of the given node *node* is returned.

xtree_t* xtree_get_payload ( xtree_t * node )

Retrieves the payload stored in this node.

Retrieves the payload stored in this node. If neither the node nor the payload exist, *NULL* is returned.
Parameters

| in | node | The node to examine for a payload |

Returns

NULL is returned if either the node node or its payload doesn’t exist. In all other cases the payload is returned.

3.13.2.7 xtree\textunderscore t\ast xtree\textunderscore map ( xtree\textunderscore t * node, void *(*)(void *, void *) func, void * extra )

Traverse the tree, applying a function to each node and storing the result.

The tree node is traversed and on each visit to a node in the tree the function func is applied on the payload of the node. The tree is traversed bottom up, one branch at a time starting with the first stored child on each parent. The result of each invocation of func is stored in a tree which is returned to the caller.

The function func should accept two arguments and return a void pointer (void *). The first argument to func is the payload of the node and the second argument is the caller supplied extra argument. In this way, using the extra argument, the caller has the flexibility to write predicate functions. For example, to find a certain value within the tree the caller can supply func as a comparison function and extra as the value to be compared against.

Parameters

| in | node | The tree to traverse |
| in | func | The function to apply to each node's payload as the tree is traversed. The first argument to func is the node's payload data and the second argument is the extra argument supplied to xtree_map(). |
| in | extra | An extra argument that forms the second argument when func is called on a node's payload (the first argument to func is the node's payload data) |

Returns

On success, a tree identical in structure to the original tree but with the payload for each node being the result of func(payload,extra). Any branch in the tree that experienced failure is pruned from the result.

Example:

// Assuming that the tree is being used to store char pointers
static void find_payload (void *p, void *n)
{
    char *payload = p,
        *needle = n;
    return (void *)(strcmp (payload, needle));
}

int main (void)
{
    xtree\textunderscore t *tree_of_strings;
    ...
    xtree\textunderscore t *hw = xtree\textunderscore map (tree_of_strings, find_payload, "Hello World");
    // hw now has all the results of all the strcmp for each node in
    // the original tree_of_strings
    ...
    return EXIT_SUCCESS;
}
Functions

- `xtree_t * xtree_add_child (xtree_t *parent, void *element)`
  Create and add a node to a parent node.

- `void * xtree_del (xtree_t *node)`
  Deletes a node and all its children.

- `xtree_t * xtree_get_child (xtree_t *node, size_t i)`
  Returns a child node of the specified node.

- `xtree_t * xtree_get_parent (xtree_t *node)`
  Retrieve the parent of this node.

- `xtree_t * xtree_get_payload (xtree_t *node)`
  Retrieves the payload stored in this node.

- `void xtree_apply (xtree_t *node, void(*)(void *, void *), void *extra)`
  Traverse the tree, applying a function to each node.

- `xtree_t * xtree_map (xtree_t *node, void *(*)(void *, void *), void *extra)`
  Traverse the tree, applying a function to each node and storing the result.

3.14.1 Detailed Description

Create and traverse a tree structure. A C library to create and traverse multi-children trees. Each child node points to a single parent node and each parent node may have multiple child nodes. Child nodes are stored in an ordered list for each parent, in the order that they are created. Each node whether parent or child may also store a payload - the element of data that is to be stored in the tree.

`xtree` is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

Author

Lelanthran Krishna Manickum

3.14.2 Function Documentation

3.14.2.1 `xtree_t * xtree_add_child (xtree_t * parent, void * element)`

Create and add a node to a parent node.

Creates and returns a new node, with `payload` as the payload and `parent` as the parent. If `parent` is `NULL` then the node that is created and returned is a root node of a tree and can be used as a parent for subsequent `xtree_add_child()` invocations.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th><code>parent</code></th>
<th>The parent of the node to be created. If parent is <code>NULL</code> then the returned node is a toplevel node of a new tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td><code>element</code></td>
<td>The payload to be stored in the newly created node</td>
</tr>
</tbody>
</table>

Returns

On success a newly created node with payload set to `payload` and parent set to `parent` is returned. On failure `NULL` is returned.
3.14.2.2 void xtree_apply ( xtree_t *node, void (*)(void *, void *) func, void *extra )

Traverse the tree, applying a function to each node.

The tree node is traversed and on each visit to a node the function func is applied on the payload of the node. The
tree is traversed bottom up, one branch at a time starting with the first stored child on each parent.

The function func should accept two arguments and return nothing. The first argument to func is the payload of
the node and the second argument is the caller supplied extra argument. In this way, using the extra argument, the
caller has flexibility to write predicate functions for side-effects.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>node</th>
<th>The tree to traverse</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>func</td>
<td>The function to apply to each node's payload as the tree is traversed. The first argument to func is the node's payload data and the second argument is the extra argument supplied to xtree_apply()</td>
</tr>
<tr>
<td>in</td>
<td>extra</td>
<td>An extra argument that forms the second argument when func is called on a node's payload (the first argument to func is the node's payload data)</td>
</tr>
</tbody>
</table>

Returns

Nothing.

Example:

// Assuming that the tree is being used to store char pointers
static void print_payload (void *p, void *f) {
  char *payload = p;
  FILE *outf = f;
  fprintf(outf, "%s\n", payload);
}

int main (void) {
  xtree_t *tree_of_strings;
  ... // Print out entire tree
  xtree_apply (tree_of_strings, print_payload, stdout);
  return EXIT_SUCCESS;
}

3.14.2.3 void xtree_del ( xtree_t *node )

Deletes a node and all it's children.

Deletes the node node, together with all it's children. All resources used by the node and it's children other than the
payload data is freed.

Parameters

| in  | node | The node to delete |

Returns

On success the payload of the deleted node is returned (the payload of the node's children are discarded). It
might be useful to the caller to receive the last payload value of the node before it was freed. On failure NULL
is returned.
3.14.2.4  

`xtree_t* xtree_get_child ( xtree_t* node, size_t i )`

Returns a child node of the specified node. 
Returns the \textit{i}’th child of this node (children in a node are stored in the order that they are created).

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>node</th>
<th>The node that contains the child that the caller is asking for.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>\textit{i}</td>
<td>The index of the child within the \textit{node} specified above</td>
</tr>
</tbody>
</table>

**Returns**

On success the \textit{i}’th child of this node is returned. On failure \textit{NULL} is returned.

3.14.2.5  

`xtree_t* xtree_get_parent ( xtree_t* node )`

Retrieve the parent of this node.
Returns the parent node of \textit{node}, if it exists. If \textit{node} or it’s parent does not exist, returns \textit{NULL} instead.

**Parameters**

| in          | node | The node to examine for a parent |

**Returns**

\textit{NULL} is returned if either \textit{node} or it’s parent does not exist, in all other cases the parent of the given node \textit{node} is returned.

3.14.2.6  

`xtree_t* xtree_get_payload ( xtree_t* node )`

Retrieves the payload stored in this node.
Retrieves the payload stored in this node. If neither the node nor the payload exist, \textit{NULL} is returned.

**Parameters**

| in          | node | The node to examine for a payload |

**Returns**

\textit{NULL} is returned if either the node \textit{node} or it’s payload doesn’t exist. In all other cases the payload is returned.

3.14.2.7  

`xtree_t* xtree_map ( xtree_t* node, void*(void*, void*) func, void* extra )`

Traverse the tree, applying a function to each node and storing the result.

The tree \textit{node} is traversed and on each visit to a node in the tree the function \textit{func} is applied on the payload of the node. The tree is traversed bottom up, one branch at a time starting with the first stored child on each parent. The result of each invocation of \textit{func} is stored in a tree which is returned to the caller.

The function \textit{func} should accept two arguments and return a void pointer (\textit{void*}). The first argument to \textit{func} is the payload of the node and the second argument is the caller supplied \textit{extra} argument. In this way, using the \textit{extra} argument, the caller has the flexibility to write predicate functions. For example, to find a certain value within the tree the caller can supply \textit{func} as a comparison function and \textit{extra} as the value to be compared against.
Parameters

|  in  |  node  | The tree to traverse |
|  in  |  func  | The function to apply to each node's payload as the tree is traversed. The first argument to func is the node's payload data and the second argument is the extra argument supplied to xtree_map(). |
|  in  |  extra  | An extra argument that forms the second argument when func is called on a node's payload (the first argument to func is the node's payload data) |

Returns

On success, a tree identical in structure to the original tree but with the payload for each node being the result of func(payload, extra). Any branch in the tree that experienced failure is pruned from the result.

Example:

```c
// Assuming that the tree is being used to store char pointers
static void find_payload (void *p, void *n)
{
    char *payload = p,
    *needle = n;
    return (void *)(strcmp (payload, needle));
}

int main (void)
{
    xtree_t *tree_of_strings;
    ...
    xtree_t *hw = xtree_map (tree_of_strings, find_payload, "Hello World");
    // hw now has all the results of all the strcmp for each node in
    // the original tree_of_strings
    ...
    return EXIT_SUCCESS;
}
```

3.15  libxc-0.0.4/xvector/xvector.c File Reference

Implementation of a vector datatype for C.

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "xvector/xvector.h"
```

Macros

- `#define DIAGS(...) fprintf (stderr, __VA_ARGS__)`

Functions

- `xvector_t * xvector_ins_tail (xvector_t *array, void *element)`
  
  Append element to the vector.

- `xvector_t * xvector_ins_head (xvector_t *array, void *element)`
  
  Insert element at first position in the vector.

- `void * xvector_del_tail (xvector_t *array)`
  
  Remove the last element of the vector.

- `void * xvector_del_head (xvector_t *array)`

Generated on Wed Apr 17 2013 18:38:19 for Extended C Library by Doxygen
Remove the first element of the vector.

- void xvector_free (xvector_t *array)
  Free the vector.
- void xvector_iterate (xvector_t *array, void(*fptr)(void *))
  Iterate over each element of the vector.
- void ** xvector_find_first (xvector_t *array, void *needle, int(*predicate)(void *, void *))
  Find the first occurrence of needle in array.
- void ** xvector_find_last (xvector_t *array, void *needle, int(*predicate)(void *, void *))
  Find the last occurrence of needle in array.
- xvector_t * xvector_map (xvector_t *array, void *(*predicate)(void *, void *), void *value)
  Maps the given function across all elements of the vector.
- void * xvector_native (xvector_t *array)
  Return a normal array of elements from the vector.
- void * xvect_dup (void *pod, size_t len)
  Return a copy of the given Plain Old Data element.

Variables

- size_t init_size = 5
- size_t inc_size = 2

3.15.1 Detailed Description

Implementation of a vector datatype for C. A vector datatype for C that allows growable arrays. (Shrinking not supported yet). The container itself is thread-safe and needs to be linked with pthreads. Note that xvector only stores pointers; no deep copying is done of the actual pointers.

xvector is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

Author

Lelanthran Krishna Manickum

3.15.2 Function Documentation

3.15.2.1 void xvect_dup (void *pod, size_t len)

Return a copy of the given Plain Old Data element.

Makes and returns a copy of the given pod (Plain Old Data) which is no more than len length in bytes. This is simply an alternative to memcpy(), and it is suggested that the caller use that function instead.

Parameters

<table>
<thead>
<tr>
<th>in</th>
<th>pod</th>
<th>The address of the existing data object</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>len</td>
<td>The length of the above data object</td>
</tr>
</tbody>
</table>

Returns

On success an exact bitwise copy of the contents at pod of length len is returned. On failure NULL is returned.

3.15.2.2 void xvector_del_head (xvector_t *array)

Remove the first element of the vector.
Removes the first element of the vector, shrinking the vector if necessary. All the elements of the vector are shifted one position to the left; the second element becomes the new first element, the third element becomes the new second element, etc.

Parameters

| in | array | The vector from which to remove the first element. If the vector is NULL then a NULL is returned and the vector remains unchanged. |

Returns

On success a pointer to the removed element is returned and the vector will have one less element, with all the remaining elements being shifted one position to the left. On failure NULL is returned.

3.15.2.3 void* xvector_del_tail ( xvector_t * array )

Remove the last element of the vector.

Removes the last element of the vector, shrinking the vector if necessary. The position of the other elements of the vector remain unchanged.

Parameters

| in | array | The vector from which to remove the last element. If the vector is NULL then a NULL is returned and the vector remains unchanged. |

Returns

On success a pointer to the removed element is returned and the vector will have one less element. On failure NULL is returned.

3.15.2.4 void** xvector_find_first ( xvector_t * array, void * needle, int(*)(void *, void *) predicate )

Find the first occurrence of needle in array.

Uses the predicate function predicate() to compare each element to needle, starting at element 0. The first match causes the search to end. The function pointer predicate has to take two arguments, A and B, both void pointers, and must return 0 if they match, 1 if A > B and -1 if A < B.

Parameters

| in | array | The vector in which to iterate the search over. If the vector is NULL then no action is taken and xvector_find_first returns NULL. |
| in | needle | The pointer to the search object (see above) |
| in | predicate | The comparison function |

Returns

On success the first match in all the elements (as determined by the predicate function) is returned. On failure NULL is returned.

3.15.2.5 void** xvector_find_last ( xvector_t * array, void * needle, int(*)(void *, void *) predicate )

Find the last occurrence of needle in array.
Uses the predicate function `predicate()` to compare each element to `needle`, starting at the last element and working backward. The first match when searching from the end causes the search to end. The function pointer `predicate` has to take two arguments, `A` and `B`, both void pointers, and must return 0 if they match, 1 if `A > B` and -1 if `A < B`.

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>array</th>
<th>The vector in which to iterate the search over. If the vector is <code>NULL</code> then no action is taken and <code>xvector_find_last</code> returns <code>NULL</code>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>needle</td>
<td>The pointer to the search object (see above)</td>
</tr>
<tr>
<td>in</td>
<td>predicate</td>
<td>The comparison function</td>
</tr>
</tbody>
</table>

**Returns**

On success the last match in all the elements (as determined by the `predicate` function) is returned. On failure `NULL` is returned.

### 3.15.2.6 void xvector_free ( xvector_t * array )

Free the vector.

Free the vector. None of the elements themselves are freed, only the container holding them; the caller is responsible for ensuring that they still have a reference to use to free the elements of the vector.

**Parameters**

| in | array | The vector to free. |

**Returns**

Nothing

### 3.15.2.7 xvector_t * xvector_ins_head ( xvector_t * array, void * element )

Insert element at first position in the vector.

Inserts the given element to the vector at the first position, growing the vector if necessary. `element` must be a pointer width or less in size.

**Parameters**

<table>
<thead>
<tr>
<th>in</th>
<th>array</th>
<th>The vector into which to insert the element. If the vector is <code>NULL</code> then a new vector is created</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>element</td>
<td>The pointer to insert into the vector</td>
</tr>
</tbody>
</table>

**Returns**

On success a `vector_t *` is returned and the old value of `array` is not valid. The `element` is guaranteed to be the first element of the returned vector. The vector must be freed with `xvector_free()`. On failure `NULL` is returned.

### 3.15.2.8 xvector_t * xvector_ins_tail ( xvector_t * array, void * element )

Append element to the vector.

Appends the given element to the vector, growing the vector if necessary. `element` must be a pointer width or less in size.
Parameters

| in | array | The vector to which to add the element. If the vector is NULL then a new vector is created |
| in | element | The pointer to add to the vector |

Returns

On success a vector, t * is returned and the old value of array is not valid. The element is guaranteed to be the last element of the returned vector. The vector must be freed with \texttt{xvector_free()}. On failure NULL is returned.

3.15.2.9 \textbf{void} \texttt{xvector_iterate ( xvector_t \ast \ array, void(\ast)(\ast) fptr )}

Iterate over each element of the vector.

Iterate over each element of the vector in the order that they are stored (i.e. left to right) and on each iteration apply the function \texttt{f()} using the element of the vector as an argument.

Logically, the following pseudocode applies:

\begin{verbatim}
foreach element from vector do
  call function fptr(element)
done
\end{verbatim}

Parameters

| in | array | The vector to iterate over. If the vector is NULL then no action is taken and \texttt{xvector_iterate} returns immediately |
| in | fptr | The predicate function to apply on every single element of \texttt{array} |

Returns

Nothing

3.15.2.10 \textbf{xvector_t} \ast \texttt{xvector_map ( xvector_t \ast \ array, void(\ast)(\ast) predicate, void \ast value )}

Maps the given function across all elements of the vector.

Iterates over the given vector \texttt{array} in order of position from lowest to highest, and for each element invokes the \texttt{predicate} function using the element as the first argument and the \texttt{value} as the second argument to the \texttt{predicate} function. The return value of all the invocations are stored in a vector which is returned to the caller.

Parameters

| in | array | The vector over which the function will map. If the vector is NULL then no action is taken and \texttt{xvector_map} returns NULL. |
| in | array | The vector to map the function func across |
| in | predicate | The function that gets mapped across all elements |
| in | value | The value that gets used as the second argument to the predicate function (each element of the array is used as the first argument) |

Returns

On success a vector containing all the return values from the mapping is returned to the caller. The caller is responsible for freeing the returned vector. On failure, the error component of the vector is set to true and the possibly incomplete vector is returned to the caller.
3.15.2.11  void xvector_native ( xvector_t * array )

Return a normal array of elements from the vector.
The elements of the vector (all pointers) are copied to an array of void pointers and returned to the caller. The
returned array is NULL-terminated, in that the final element (which is not part of the vector) is a NULL pointer. The
caller must free the returned array.

Parameters

| in | array | The vector to convert to plain array form. If the vector array is NULL, then NULL is returned and no action is taken |

Returns

On success an array of void pointers is returned, with the array being terminated by a NULL pointer. The caller
must free this array.

3.16  xvector/xvector.c File Reference

Implementation of a vector datatype for C.

#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "xvector/xvector.h"

Macros

• #define DIAGS(...) fprintf (stderr, __VA_ARGS__)

Functions

• xvector_t * xvector_ins_tail (xvector_t *array, void *element)
  Append element to the vector.
• xvector_t * xvector_ins_head (xvector_t *array, void *element)
  Insert element at first position in the vector.
• void * xvector_del_tail (xvector_t *array)
  Remove the last element of the vector.
• void * xvector_del_head (xvector_t *array)
  Remove the first element of the vector.
• void xvector_free (xvector_t *array)
  Free the vector.
• void xvector_iterate (xvector_t *array, void(*fptr)(void *))
  Iterate over each element of the vector.
• void ** xvector_find_first (xvector_t *array, void *needle, int(*predicate)(void *, void *))
  Find the first occurrence of needle in array.
• void ** xvector_find_last (xvector_t *array, void *needle, int(*predicate)(void *, void *))
  Find the last occurrence of needle in array.
• xvector_t * xvector_map (xvector_t *array, void *(*predicate)(void *, void *), void *value)
  Maps the given function across all elements of the vector.
• void * xvector_native (xvector_t *array)
Return a normal array of elements from the vector.

- void * xvect_dup (void *pod, size_t len)
  Return a copy of the given Plain Old Data element.

Variables

- size_t init_size = 5
- size_t inc_size = 2

3.16.1 Detailed Description

Implementation of a vector datatype for C. A vector datatype for C that allows growable arrays. (Shrinking not supported yet). The container itself is thread-safe and needs to be linked with pthreads. Note that xvector only stores pointers; no deep copying is done of the actual pointers.

xvector is part of the libxc (Extended C Library) and falls under the relevant copyright license in libxc.

Author

Lelanthran Krishna Manickum

3.16.2 Function Documentation

3.16.2.1 void* xvect_dup ( void * pod, size_t len )

Return a copy of the given Plain Old Data element.

Makes and returns a copy of the given pod (Plain Old Data) which is no more than len length in bytes. This is simply an alternative to memcpy(), and it is suggested that the caller use that function instead.

Parameters

| in | pod | The address of the existing data object |
| in | len | The length of the above data object |

Returns

On success an exact bitwise copy of the contents at pod of length len is returned. On failure NULL is returned.

3.16.2.2 void* xvector_del_head ( xvector_t *array )

Remove the first element of the vector.

Removes the first element of the vector, shrinking the vector if necessary. All the elements of the vector are shifted one position to the left; the second element becomes the new first element, the third element becomes the new second element, etc.

Parameters

| in | array | The vector from which to remove the first element. If the vector is NULL then a NULL is returned and the vector remains unchanged. |

Returns

On success a pointer to the removed element is returned and the vector will have one less element, with all the remaining elements being shifted one position to the left. On failure NULL is returned.
3.16.2.3 void* xvector_del_tail ( xvector_t * array )

Remove the last element of the vector. Removes the last element of the vector, shrinking the vector if necessary. The position of the other elements of the vector remain unchanged.

Parameters

| in | array | The vector from which to remove the last element. If the vector is NULL then a NULL is returned and the vector remains unchanged. |

Returns

On success a pointer to the removed element is returned and the vector will have one less element. On failure NULL is returned.

3.16.2.4 void** xvector_find_first ( xvector_t * array, void * needle, int(*)(void *, void *) predicate )

Find the first occurrence of needle in array.
Uses the predicate function predicate() to compare each element to needle, starting at element 0. The first match causes the search to end. The function pointer predicate has to take two arguments, A and B, both void pointers, and must return 0 if they match, 1 if A > B and -1 if A < B.

Parameters

| in | array | The vector in which to iterate the search over. If the vector is NULL then no action is taken and xvector_find_first returns NULL. |
| in | needle | The pointer to the search object (see above) |
| in | predicate | The comparison function |

Returns

On success the first match in all the elements (as determined by the predicate function) is returned. On failure NULL is returned.

3.16.2.5 void** xvector_find_last ( xvector_t * array, void * needle, int(*)(void *, void *) predicate )

Find the last occurrence of needle in array.
Uses the predicate function predicate() to compare each element to needle, starting at the last element and working backward. The first match when searching from the end causes the search to end. The function pointer predicate has to take two arguments, A and B, both void pointers, and must return 0 if they match, 1 if A > B and -1 if A < B.

Parameters

| in | array | The vector in which to iterate the search over. If the vector is NULL then no action is taken and xvector_find_last returns NULL. |
| in | needle | The pointer to the search object (see above) |
| in | predicate | The comparison function |

Returns

On success the last match in all the elements (as determined by the predicate function) is returned. On failure NULL is returned.
3.16.2.6  void xvector_free ( xvector_t * array )

Free the vector.
Free the vector. None of the elements themselves are freed, only the container holding them; the caller is responsible for ensuring that they still have a reference to use to free the elements of the vector.

Parameters

| in  | array | The vector to free. |

Returns

Nothing

3.16.2.7  xvector_t* xvector_ins_head ( xvector_t * array, void * element )

Insert element at first position in the vector.
Inserts the given element to the vector at the first position, growing the vector if necessary. element must be a pointer width or less in size.

Parameters

| in  | array | The vector into which to insert the element. If the vector is NULL then a new vector is created |
| in  | element | The pointer to insert into the vector |

Returns

On success a vector_t* is returned and the old value of array is not valid. The element is guaranteed to be the first element of the returned vector. The vector must be freed with xvector_free(). On failure NULL is returned.

3.16.2.8  xvector_t* xvector_ins_tail ( xvector_t * array, void * element )

Append element to the vector.
Appends the given element to the vector, growing the vector if necessary. element must be a pointer width or less in size.

Parameters

| in  | array | The vector to which to add the element. If the vector is NULL then a new vector is created |
| in  | element | The pointer to add to the vector |

Returns

On success a vector_t* is returned and the old value of array is not valid. The element is guaranteed to be the last element of the returned vector. The vector must be freed with xvector_free(). On failure NULL is returned.

3.16.2.9  void xvector_iterate ( xvector_t * array, void(*)(void *) fptr )

Iterate over each element of the vector.
Iterate over each element of the vector in the order that they are stored (i.e. left to right) and on each iteration apply the function f() using the element of the vector as an argument.
Logically, the following pseudocode applies:

```c
foreach element from vector do
    call function fptr(element)
done
```

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array</code></td>
<td>The vector to iterate over. If the vector is <code>NULL</code> then no action is taken and <code>xvector_iterate</code> returns immediately.</td>
</tr>
<tr>
<td><code>fptr</code></td>
<td>The predicate function to apply on every single element of array.</td>
</tr>
</tbody>
</table>

Returns

Nothing

3.16.2.10 `xvector_t* xvector_map ( xvector_t* array, void (*)(void*, void*) predicate, void* value )`

Maps the given function across all elements of the vector.

Iterates over the given vector `array` in order of position from lowest to highest, and for each element invokes the `predicate` function using the element as the first argument and the `value` as the second argument to the `predicate` function. The return value of all the invocations are stored in a vector which is returned to the caller.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array</code></td>
<td>The vector over which the function will map. If the vector is <code>NULL</code> then no action is taken and <code>xvector_map</code> returns <code>NULL</code>.</td>
</tr>
<tr>
<td><code>predicate</code></td>
<td>The function that gets mapped across all elements.</td>
</tr>
<tr>
<td><code>value</code></td>
<td>The value that gets used as the second argument to the predicate function (each element of the array is used as the first argument).</td>
</tr>
</tbody>
</table>

Returns

On success a vector containing all the return values from the mapping is returned to the caller. The caller is responsible for freeing the returned vector. On failure, the error component of the vector is set to true and the possibly incomplete vector is returned to the caller.

3.16.2.11 `void* xvector_native ( xvector_t* array )`

Return a normal array of elements from the vector.

The elements of the vector (all pointers) are copied to an array of `void` pointers and returned to the caller. The returned array is `NULL`-terminated, in that the final element (which is not part of the vector) is a `NULL` pointer. The caller must free the returned array.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>array</code></td>
<td>The vector to convert to plain array form. If the vector <code>array</code> is <code>NULL</code>, then <code>NULL</code> is returned and no action is taken.</td>
</tr>
</tbody>
</table>

Returns

On success an array of `void` pointers is returned, with the array being terminated by a NULL pointer. The caller must free this array.
Index

libxc-0.0.4/xcfg/xcfg.c, 5
  xcfg_configure, 6
  xcfg_from_array, 7
  xcfg_from_env, 8
  xcfg_from_file, 9
  xcfg_get, 10
  xcfg_get_all, 10
  xcfg_get_f, 11
  xcfg_get_i, 12
  xcfg_get_u, 12
  xcfg_save_config, 13
  xcfg_set, 13
  xcfg_shutdown, 14

libxc-0.0.4/xdict/xdict.c, 24
  xdict_del, 25
  xdict_get, 25
  xdict_iterate, 27
  xdict_map, 27
  xdict_new, 28
  xdict_set, 30

libxc-0.0.4/xmalloc/xmalloc.c, 39
  xmalloc, 39
  xmalloc_dump, 40
  xmalloc_find, 40
  xmalloc_free, 40
  xmalloc_realloc, 40
  xmalloc_reset, 41
  xmalloc_set_knob, 41

libxc-0.0.4/xshare/xshare.c, 45
  xshare_close, 45
  xshare_errormsg, 46
  xshare_error, 46
  xshare_open, 47
  xshare_symbol, 48

libxc-0.0.4/xsock/xsock.c, 52
  xsock_clear_errno, 53
  xsock_close, 54
  xsock_connect, 54
  xsock_errno, 55
  xsock_error, 56
  xsock_strerror, 57
  xsock_write, 57

libxc-0.0.4/xstring/xstring.c, 63
  xstr_cat, 65
  xstr_cpyarray, 65
  xstr_delarray, 66
  xstr_dup, 66
  xstr_file_readable, 67
  xstr_file_writable, 67
  xstr_join, 68
  xstr_ltrim, 69
  xstr_match_first, 69
  xstr_match_first_a, 70
  xstrncpy, 71
  xstr_readfile, 71
  xstr_rtrim, 72
  xstr_split, 72
  xstr_trim, 73

libxc-0.0.4/xtree/xtree.c, 84
  xtree_add_child, 85
  xtree_apply, 85
  xtree_del, 86
  xtree_get_child, 86
  xtree_get_parent, 86
  xtree_get_payload, 86
  xtree_map, 87

libxc-0.0.4/xvector/xvector.c, 91
  xvect_dup, 92
  xvector_del_head, 92
  xvector_del_tail, 93
  xvector_find_first, 93
  xvector_find_last, 93
  xvector_free, 94
  xvector_ins_head, 94
  xvector_ins_tail, 94
  xvector_iterate, 95
  xvector_map, 95
  xvector_native, 95

xcfg/xcfg.c, 15
  xcfg_configure, 16
  xcfg_from_array, 16
  xcfg_from_env, 18
  xcfg_from_file, 18
  xcfg_get, 20
  xcfg_get_all, 20
  xcfg_get_f, 21
  xcfg_get_i, 21
  xcfg_get_u, 22
  xcfg_save_config, 22
  xcfg_set, 23
  xcfg_shutdown, 24
  xcfg_configure
    libxc-0.0.4/xcfg/xcfg.c, 6
    xcfg/xcfg.c, 16
  xcfg_from_array
    libxc-0.0.4/xcfg/xcfg.c, 7
    xcfg/xcfg.c, 16
xcfg_from_env
  libxc-0.0.4/xcfg/xcfg.c, 8
  xcfg/xcfg.c, 18
xcfg_from_file
  libxc-0.0.4/xcfg/xcfg.c, 9
  xcfg/xcfg.c, 18
xcfg_get
  libxc-0.0.4/xcfg/xcfg.c, 10
  xcfg/xcfg.c, 20
xcfg_get_all
  libxc-0.0.4/xcfg/xcfg.c, 10
  xcfg/xcfg.c, 20
xcfg_get_f
  libxc-0.0.4/xcfg/xcfg.c, 11
  xcfg/xcfg.c, 21
xcfg_get_i
  libxc-0.0.4/xcfg/xcfg.c, 12
  xcfg/xcfg.c, 21
xcfg_get_u
  libxc-0.0.4/xcfg/xcfg.c, 12
  xcfg/xcfg.c, 22
xcfg_save_config
  libxc-0.0.4/xcfg/xcfg.c, 13
  xcfg/xcfg.c, 22
xcfg_set
  libxc-0.0.4/xcfg/xcfg.c, 13
  xcfg/xcfg.c, 23
xcfg_shutdown
  libxc-0.0.4/xcfg/xcfg.c, 14
  xcfg/xcfg.c, 24
xdict/xdict.c, 31
  xdict_del, 32
  xdict_get, 32
  xdict_iterate, 34
  xdict_map, 35
  xdict_new, 36
  xdict_set, 37
xdict_del
  libxc-0.0.4/xdict/xdict.c, 25
  xdict/xdict.c, 32
xdict_get
  libxc-0.0.4/xdict/xdict.c, 25
  xdict/xdict.c, 32
xdict_iterate
  libxc-0.0.4/xdict/xdict.c, 27
  xdict/xdict.c, 34
xdict_map
  libxc-0.0.4/xdict/xdict.c, 27
  xdict/xdict.c, 35
xdict_new
  libxc-0.0.4/xdict/xdict.c, 28
  xdict/xdict.c, 36
xdict_set
  libxc-0.0.4/xdict/xdict.c, 30
  xdict/xdict.c, 37
xmalloc
  libxc-0.0.4/xmalloc/xmalloc.c, 39
  xmalloc/xmalloc.c, 42
xmalloc/xmalloc.c, 42
  xmalloc, 42
  xmalloc_dump, 43
  xmalloc_find, 43
  xmalloc_free, 43
  xmalloc_realloc, 44
  xmalloc_reset, 44
  xmalloc_set_knob, 44
xmalloc_dump
  libxc-0.0.4/xmalloc/xmalloc.c, 40
  xmalloc/xmalloc.c, 43
xmalloc_find
  libxc-0.0.4/xmalloc/xmalloc.c, 40
  xmalloc/xmalloc.c, 43
xmalloc_free
  libxc-0.0.4/xmalloc/xmalloc.c, 40
  xmalloc/xmalloc.c, 43
xmalloc_realloc
  libxc-0.0.4/xmalloc/xmalloc.c, 40
  xmalloc/xmalloc.c, 44
xmalloc_reset
  libxc-0.0.4/xmalloc/xmalloc.c, 41
  xmalloc/xmalloc.c, 44
xmalloc_set_knob
  libxc-0.0.4/xmalloc/xmalloc.c, 41
  xmalloc/xmalloc.c, 44
xshare/xshare.c, 48
  xshare_close, 49
  xshare_errmsg, 50
  xshare_error, 50
  xshare_open, 51
  xshare_symbol, 52
xshare_close
  libxc-0.0.4/xshare/xshare.c, 45
  xshare/xshare.c, 49
xshare_errmsg
  libxc-0.0.4/xshare/xshare.c, 46
  xshare/xshare.c, 50
xshare_error
  libxc-0.0.4/xshare/xshare.c, 46
  xshare/xshare.c, 50
xshare_open
  libxc-0.0.4/xshare/xshare.c, 47
  xshare/xshare.c, 51
xshare_symbol
  libxc-0.0.4/xshare/xshare.c, 48
  xshare/xshare.c, 52
xsock/xsock.c, 58
  xsock_clear_errno, 59
  xsock_close, 59
  xsock_connect, 59
  xsock_errno, 60
  xsock_listen, 61
  xsock_read, 61
  xsock_strerror, 62
  xsock_write, 62
xsock_clear_errno
  libxc-0.0.4/xsock/xsock.c, 53
xvect_dup, 97
xvector_del_head, 97
xvector_del_tail, 97
xvector_find_first, 98
xvector_find_last, 98
xvector_free, 98
xvector_ins_head, 99
xvector_ins_tail, 99
xvector_iterate, 99
xvector_map, 100
xvector_native, 100
xvector_del_head
  libxc-0.0.4/xvector/xvector.c, 92
  xvector/xvector.c, 97
xvector_del_tail
  libxc-0.0.4/xvector/xvector.c, 93
  xvector/xvector.c, 97
xvector_find_first
  libxc-0.0.4/xvector/xvector.c, 93
  xvector/xvector.c, 98
xvector_find_last
  libxc-0.0.4/xvector/xvector.c, 93
  xvector/xvector.c, 98
xvector_free
  libxc-0.0.4/xvector/xvector.c, 94
  xvector/xvector.c, 98
xvector_ins_head
  libxc-0.0.4/xvector/xvector.c, 94
  xvector/xvector.c, 99
xvector_ins_tail
  libxc-0.0.4/xvector/xvector.c, 94
  xvector/xvector.c, 99
xvector_iterate
  libxc-0.0.4/xvector/xvector.c, 95
  xvector/xvector.c, 99
xvector_map
  libxc-0.0.4/xvector/xvector.c, 95
  xvector/xvector.c, 100
xvector_native
  libxc-0.0.4/xvector/xvector.c, 95
  xvector/xvector.c, 100