

gds

Generated by Doxygen 1.8.1.2

Sat Nov 8 2014 20:46:00

Contents

1	Generic Data Structures Library	1
2	Module Index	3
2.1	Modules	3
3	Data Structure Index	5
3.1	Data Structures	5
4	File Index	7
4.1	File List	7
5	Module Documentation	9
5.1	Private functionality for manipulating generic datatypes	9
5.1.1	Detailed Description	10
5.1.2	Typedef Documentation	10
5.1.2.1	gds_cfunc	10
5.1.3	Enumeration Type Documentation	10
5.1.3.1	gds_datatype	10
5.1.4	Function Documentation	10
5.1.4.1	gdt_compare	10
5.1.4.2	gdt_compare_void	11
5.1.4.3	gdt_free	11
5.1.4.4	gdt_get_value	11
5.1.4.5	gdt_reverse_compare_void	11
5.1.4.6	gdt_set_value	12
5.2	Public general generic data structures functionality	13
5.2.1	Detailed Description	13
5.2.2	Enumeration Type Documentation	13
5.2.2.1	gds_option	13
5.2.3	Function Documentation	13
5.2.3.1	gds_assert_quit	13
5.2.3.2	gds_error_quit	13
5.2.3.3	gds_strerror_quit	14

5.3	Public interface to generic list data structure	15
5.3.1	Detailed Description	15
5.3.2	Typedef Documentation	15
5.3.2.1	List	15
5.3.3	Function Documentation	16
5.3.3.1	list_append	16
5.3.3.2	list_create	16
5.3.3.3	list_delete_back	16
5.3.3.4	list_delete_front	16
5.3.3.5	list_delete_index	17
5.3.3.6	list_destroy	17
5.3.3.7	list_element_at_index	17
5.3.3.8	list_find	17
5.3.3.9	list_insert	18
5.3.3.10	list_is_empty	18
5.3.3.11	list_length	18
5.3.3.12	list_prepend	19
5.3.3.13	list_set_element_at_index	19
5.4	Public interface to generic queue data structure	20
5.4.1	Detailed Description	20
5.4.2	Typedef Documentation	20
5.4.2.1	Queue	20
5.4.3	Function Documentation	20
5.4.3.1	queue_capacity	20
5.4.3.2	queue_create	21
5.4.3.3	queue_destroy	21
5.4.3.4	queue_free_space	21
5.4.3.5	queue_is_empty	21
5.4.3.6	queue_is_full	22
5.4.3.7	queue_peek	22
5.4.3.8	queue_pop	22
5.4.3.9	queue_push	23
5.4.3.10	queue_size	23
5.5	Public interface to generic stack data structure	24
5.5.1	Detailed Description	24
5.5.2	Typedef Documentation	24
5.5.2.1	Stack	24
5.5.3	Function Documentation	24
5.5.3.1	stack_capacity	24
5.5.3.2	stack_create	25

5.5.3.3	stack_destroy	25
5.5.3.4	stack_free_space	25
5.5.3.5	stack_is_empty	25
5.5.3.6	stack_is_full	26
5.5.3.7	stack_peek	26
5.5.3.8	stack_pop	26
5.5.3.9	stack_push	27
5.5.3.10	stack_size	27
5.6	Public interface to generic vector data structure.	28
5.6.1	Detailed Description	28
5.6.2	Typedef Documentation	29
5.6.2.1	Vector	29
5.6.3	Function Documentation	29
5.6.3.1	vector_append	29
5.6.3.2	vector_capacity	29
5.6.3.3	vector_create	29
5.6.3.4	vector_delete_back	30
5.6.3.5	vector_delete_front	30
5.6.3.6	vector_delete_index	30
5.6.3.7	vector_destroy	30
5.6.3.8	vector_element_at_index	31
5.6.3.9	vector_find	31
5.6.3.10	vector_free_space	31
5.6.3.11	vector_insert	31
5.6.3.12	vector_is_empty	32
5.6.3.13	vector_length	32
5.6.3.14	vector_prepend	32
5.6.3.15	vector_reverse_sort	33
5.6.3.16	vector_set_element_at_index	33
5.6.3.17	vector_sort	33
6	Data Structure Documentation	35
6.1	gdt_generic_datatype Struct Reference	35
6.1.1	Detailed Description	35
6.1.2	Field Documentation	35
6.1.2.1	c	35
6.1.2.2	compfunc	36
6.1.2.3	d	36
6.1.2.4	data	36
6.1.2.5	i	36

6.1.2.6	l	36
6.1.2.7	ll	36
6.1.2.8	p	36
6.1.2.9	pc	36
6.1.2.10	sc	36
6.1.2.11	st	36
6.1.2.12	type	36
6.1.2.13	uc	36
6.1.2.14	ui	37
6.1.2.15	ul	37
6.1.2.16	ull	37
6.2	list Struct Reference	37
6.2.1	Detailed Description	38
6.2.2	Field Documentation	38
6.2.2.1	compfunc	38
6.2.2.2	exit_on_error	38
6.2.2.3	free_on_destroy	38
6.2.2.4	head	38
6.2.2.5	length	38
6.2.2.6	tail	38
6.2.2.7	type	38
6.3	list_node Struct Reference	39
6.3.1	Detailed Description	39
6.3.2	Field Documentation	39
6.3.2.1	element	39
6.3.2.2	next	39
6.3.2.3	prev	39
6.4	queue Struct Reference	40
6.4.1	Detailed Description	40
6.4.2	Field Documentation	40
6.4.2.1	back	40
6.4.2.2	capacity	40
6.4.2.3	elements	40
6.4.2.4	exit_on_error	40
6.4.2.5	free_on_destroy	41
6.4.2.6	front	41
6.4.2.7	resizable	41
6.4.2.8	size	41
6.4.2.9	type	41
6.5	stack Struct Reference	41

6.5.1	Detailed Description	42
6.5.2	Field Documentation	42
6.5.2.1	capacity	42
6.5.2.2	elements	42
6.5.2.3	exit_on_error	42
6.5.2.4	free_on_destroy	42
6.5.2.5	resizable	42
6.5.2.6	top	42
6.5.2.7	type	42
6.6	vector Struct Reference	43
6.6.1	Detailed Description	43
6.6.2	Field Documentation	43
6.6.2.1	capacity	43
6.6.2.2	compfunc	43
6.6.2.3	elements	43
6.6.2.4	exit_on_error	43
6.6.2.5	free_on_destroy	44
6.6.2.6	length	44
6.6.2.7	type	44
7	File Documentation	45
7.1	gds.dox File Reference	45
7.2	include/private/gds_common.h File Reference	45
7.2.1	Detailed Description	46
7.3	include/private/gdt.dox File Reference	46
7.4	include/private/gdt.h File Reference	46
7.4.1	Detailed Description	48
7.5	include/public/gds_public_types.h File Reference	48
7.5.1	Detailed Description	49
7.6	include/public/gds_util.h File Reference	49
7.6.1	Detailed Description	50
7.7	include/public/general.dox File Reference	50
7.8	include/public/list.dox File Reference	50
7.9	include/public/list.h File Reference	50
7.9.1	Detailed Description	52
7.10	include/public/queue.dox File Reference	52
7.11	include/public/queue.h File Reference	52
7.11.1	Detailed Description	54
7.12	include/public/stack.dox File Reference	54
7.13	include/public/stack.h File Reference	54

7.13.1 Detailed Description	56
7.14 include/public/vector.dox File Reference	56
7.15 include/public/vector.h File Reference	56
7.15.1 Detailed Description	58
7.16 src/gds_util.c File Reference	58
7.17 src/gdt.c File Reference	59
7.17.1 Function Documentation	60
7.17.1.1 gdt_compare_char	60
7.17.1.2 gdt_compare_double	60
7.17.1.3 gdt_compare_int	60
7.17.1.4 gdt_compare_long	60
7.17.1.5 gdt_compare_longlong	60
7.17.1.6 gdt_compare_schar	60
7.17.1.7 gdt_compare_sizet	60
7.17.1.8 gdt_compare_string	60
7.17.1.9 gdt_compare_uchar	60
7.17.1.10 gdt_compare_uint	60
7.17.1.11 gdt_compare_ulong	60
7.17.1.12 gdt_compare_ulonglong	60
7.18 src/list.c File Reference	60
7.18.1 Detailed Description	62
7.18.2 Typedef Documentation	62
7.18.2.1 ListNode	62
7.18.3 Function Documentation	62
7.18.3.1 list_insert_internal	62
7.18.3.2 list_node_at_index	63
7.18.3.3 list_node_create	63
7.18.3.4 list_node_destroy	63
7.19 src/queue.c File Reference	63
7.19.1 Detailed Description	65
7.19.2 Variable Documentation	65
7.19.2.1 GROWTH	65
7.20 src/stack.c File Reference	65
7.20.1 Detailed Description	66
7.20.2 Variable Documentation	66
7.20.2.1 GROWTH	66
7.21 src/vector.c File Reference	67
7.21.1 Detailed Description	68
7.21.2 Function Documentation	68
7.21.2.1 vector_insert_internal	68

7.21.3 Variable Documentation	69
7.21.3.1 GROWTH	69

Chapter 1

Generic Data Structures Library

GDS is a C language generic data structures library.

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Private functionality for manipulating generic datatypes	9
Public general generic data structures functionality	13
Public interface to generic list data structure	15
Public interface to generic queue data structure	20
Public interface to generic stack data structure	24
Public interface to generic vector data structure.	28

Chapter 3

Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

gdt_generic_datatype	
Generic datatype structure	35
list	37
list_node	39
queue	40
stack	41
vector	43

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

include/private/ gds_common.h	
Common internal headers for data structures	45
include/private/ gdt.h	
Interface to generic data element functionality	46
include/public/ gds_public_types.h	
Common public types for generic data structures library	48
include/public/ gds_util.h	
Interface to general utility functions	49
include/public/ list.h	
Interface to generic list data structure	50
include/public/ queue.h	
Interface to generic queue data structure	52
include/public/ stack.h	
Interface to generic stack data structure	54
include/public/ vector.h	
Interface to generic vector data structure	56
src/ gds_util.c	58
src/ gdt.c	59
src/ list.c	
Implementation of generic list data structure	60
src/ queue.c	
Implementation of generic queue data structure	63
src/ stack.c	
Implementation of generic stack data structure	65
src/ vector.c	
Implementation of generic vector data structure	67

Chapter 5

Module Documentation

5.1 Private functionality for manipulating generic datatypes

Data Structures

- struct [gdt_generic_datatype](#)
Generic datatype structure.

Typedefs

- typedef int(* [gds_cfunc](#))(const void *, const void *)
Type definition for comparison function pointer.

Enumerations

- enum [gds_datatype](#) {
 [DATATYPE_CHAR](#), [DATATYPE_UNSIGNED_CHAR](#), [DATATYPE_SIGNED_CHAR](#), [DATATYPE_INT](#),
 [DATATYPE_UNSIGNED_INT](#), [DATATYPE_LONG](#), [DATATYPE_UNSIGNED_LONG](#), [DATATYPE_LONG_](#)-
 [LONG](#),
 [DATATYPE_UNSIGNED_LONG_LONG](#), [DATATYPE_SIZE_T](#), [DATATYPE_DOUBLE](#), [DATATYPE_STRIN-](#)
 [G](#),
 [DATATYPE_POINTER](#) }
Enumeration type for data element type.

Functions

- void [gdt_set_value](#) (struct [gdt_generic_datatype](#) *data, const enum [gds_datatype](#) type, [gds_cfunc](#) cfunc, va_list ap)
Sets the value of a generic datatype.
- void [gdt_get_value](#) (const struct [gdt_generic_datatype](#) *data, void *p)
Gets the value of a generic datatype.
- void [gdt_free](#) (struct [gdt_generic_datatype](#) *data)
Frees memory pointed to by a generic datatype.
- int [gdt_compare](#) (const struct [gdt_generic_datatype](#) *d1, const struct [gdt_generic_datatype](#) *d2)
Compares two generic datatypes.
- int [gdt_compare_void](#) (const void *p1, const void *p2)
Compares two generic datatypes via void pointers.
- int [gdt_reverse_compare_void](#) (const void *p1, const void *p2)
Reverse compares two generic datatypes via void pointers.

5.1.1 Detailed Description

This module implements the mechanism for allowing generic datatypes. Each datatype implements a C `union` containing all the allowable fundamental types. Functions are provided for getting, setting, `free()`ing, and comparing values.

5.1.2 Typedef Documentation

5.1.2.1 `typedef int(* gds_cfunc)(const void *, const void *)`

Type definition for comparison function pointer.

5.1.3 Enumeration Type Documentation

5.1.3.1 `enum gds_datatype`

Enumeration type for data element type.

Enumerator:

```
DATATYPE_CHAR   char
DATATYPE_UNSIGNED_CHAR unsigned char
DATATYPE_SIGNED_CHAR signed char
DATATYPE_INT    int
DATATYPE_UNSIGNED_INT unsigned int
DATATYPE_LONG   long
DATATYPE_UNSIGNED_LONG unsigned long
DATATYPE_LONG_LONG long long
DATATYPE_UNSIGNED_LONG_LONG unsigned long long
DATATYPE_SIZE_T size_t
DATATYPE_DOUBLE double
DATATYPE_STRING char *, string
DATATYPE_POINTER void *
```

5.1.4 Function Documentation

5.1.4.1 `int gdt_compare (const struct gdt_generic_datatype * d1, const struct gdt_generic_datatype * d2)`

Compares two generic datatypes.

Parameters

<i>d1</i>	A pointer to the first generic datatype.
<i>d2</i>	A pointer to the second generic datatype.

Return values

<i>0</i>	The two datatypes are equal.
<i>-1</i>	The first datatype is less than the second datatype.
<i>1</i>	The first datatype is greater than the second datatype.

5.1.4.2 `int gdt_compare_void (const void * p1, const void * p2)`

Compares two generic datatypes via `void` pointers.

This function is suitable for passing to `qsort()`.

Parameters

<i>p1</i>	A pointer to the first generic datatype.
<i>p2</i>	A pointer to the second generic datatype.

Return values

0	The two datatypes are equal.
-1	The first datatype is less than the second datatype.
1	The first datatype is greater than the second datatype.

5.1.4.3 `void gdt_free (struct gdt_generic_datatype * data)`

Frees memory pointed to by a generic datatype.

This function does nothing if the type of the generic datatype set by the last call to `gdt_set_value()` is neither `DATATYPE_STRING` nor `DATATYPE_POINTER`. If the type of the generic datatype is one of these values, the caller is responsible for ensuring that the last value set contains an address on which it is appropriate to call `free()`.

Parameters

<i>data</i>	A pointer to the generic datatype.
-------------	------------------------------------

5.1.4.4 `void gdt_get_value (const struct gdt_generic_datatype * data, void * p)`

Gets the value of a generic datatype.

Parameters

<i>data</i>	A pointer to the generic datatype.
<i>p</i>	A pointer containing the address of an object of type appropriate to the type of the generic datatype set by the last call to <code>gdt_set_value()</code> . This object will be modified to contain the value of the generic datatype.

5.1.4.5 `int gdt_reverse_compare_void (const void * p1, const void * p2)`

Reverse compares two generic datatypes via `void` pointers.

This function is suitable for passing to `qsort()` when the desired behavior is to sort in reverse order.

Parameters

<i>p1</i>	A pointer to the first generic datatype.
<i>p2</i>	A pointer to the second generic datatype.

Return values

0	The two datatypes are equal.
-1	The first datatype is greater than the second datatype.
1	The first datatype is less than the second datatype.

5.1.4.6 void gdt_set_value (struct gdt_generic_datatype * *data*, const enum gds_datatype *type*, gds_cfunc *cfunc*, va_list *ap*)

Sets the value of a generic datatype.

Parameters

<i>data</i>	A pointer to the generic datatype.
<i>type</i>	The type of data for the datatype to contain.
<i>cfunc</i>	A pointer to a comparison function. This is ignored for all types other than DATATYPE_POINTER. For DATATYPE_POINTER, this should contain the address of a function of type <code>int (*)(const void *, const void *)</code> if the datatype will ever need to be compared with another datatype of the same type (e.g. for finding or sorting elements within a data structure). If this functionality is not required, <code>NULL</code> can be provided.
<i>ap</i>	A <code>va_list</code> containing a single argument of the type appropriate to <code>type</code> , containing the value to which to set the generic datatype.

5.2 Public general generic data structures functionality

Enumerations

- enum `gds_option` { `GDS_RESIZABLE` = 1, `GDS_FREE_ON_DESTROY` = 2, `GDS_EXIT_ON_ERROR` = 4 }

Enumeration type for data structure options.

Functions

- void `gds_strerror_quit` (const char *msg,...)
Prints an error message with error number and exits.
- void `gds_error_quit` (const char *msg,...)
Prints an error message exits.
- void `gds_assert_quit` (const char *msg,...)
Prints an error message exits via assert().

5.2.1 Detailed Description

This module contains general functionality used with or by the other data structures, including common creation options, and functions for outputting error messages.

5.2.2 Enumeration Type Documentation

5.2.2.1 enum `gds_option`

Enumeration type for data structure options.

Enumerator:

`GDS_RESIZABLE` Dynamically resizes on demand

`GDS_FREE_ON_DESTROY` Automatically frees pointer members

`GDS_EXIT_ON_ERROR` Exits on error

5.2.3 Function Documentation

5.2.3.1 void `gds_assert_quit` (const char * *msg*, ...)

Prints an error message exits via assert().

This function will do nothing if `NDEBUG` is defined. Otherwise, it behaves in a manner identical to `gds_error_quit()` except it terminates via `assert()`, rather than `exit()`.

Parameters

<i>msg</i>	The format string for the message to print. Format specifiers are the same as the <code>printf()</code> family of functions.
...	Any arguments to the format string.

5.2.3.2 void `gds_error_quit` (const char * *msg*, ...)

Prints an error message exits.

Parameters

<i>msg</i>	The format string for the message to print. Format specifiers are the same as the <code>printf()</code> family of functions.
...	Any arguments to the format string.

5.2.3.3 void gds_strerror_quit (const char * *msg*, ...)

Prints an error message with error number and exits.

This function can be called to print an error message and quit following a function which has indicated failure and has set `errno`. A message containing the error number and a text representation of that error will be printed, following by the message supplied to the function.

Parameters

<i>msg</i>	The format string for the message to print. Format specifiers are the same as the <code>printf()</code> family of functions.
...	Any arguments to the format string.

5.3 Public interface to generic list data structure

Typedefs

- typedef struct [list](#) * [List](#)
Opaque list type definition.

Functions

- [List](#) [list_create](#) (const enum [gds_datatype](#) type, const int opts,...)
Creates a new list.
- void [list_destroy](#) ([List](#) list)
Destroys a list.
- bool [list_append](#) ([List](#) list,...)
Appends a value to the back of a list.
- bool [list_prepend](#) ([List](#) list,...)
Prepends a value to the front of a list.
- bool [list_insert](#) ([List](#) list, const size_t index,...)
Inserts a value into a list.
- bool [list_delete_front](#) ([List](#) list)
Deletes the value at the front of the list.
- bool [list_delete_back](#) ([List](#) list)
Deletes the value at the back of the list.
- bool [list_delete_index](#) ([List](#) list, const size_t index)
Deletes the value at the specified index of the list.
- bool [list_element_at_index](#) ([List](#) list, const size_t index, void *p)
Gets the value at the specified index of the list.
- bool [list_set_element_at_index](#) ([List](#) list, const size_t index,...)
Sets the value at the specified index of the list.
- bool [list_find](#) ([List](#) list, size_t *index,...)
Tests if a value is contained in a list.
- bool [list_is_empty](#) ([List](#) list)
Tests if a list is empty.
- size_t [list_length](#) ([List](#) list)
Returns the length of a list.

5.3.1 Detailed Description

A list is data structure containing a finite ordered collection of values which allows sequential access (compared to a vector, or array, which allows random access).

5.3.2 Typedef Documentation

5.3.2.1 typedef struct list* List

Opaque list type definition.

5.3.3 Function Documentation

5.3.3.1 `bool list_append (List list, ...)`

Appends a value to the back of a list.

Parameters

<i>list</i>	A pointer to the list.
<i>...</i>	The value to append to the end of the list. This should be of a type appropriate to the type set when creating the list.

Return values

<i>true</i>	Success
<i>false</i>	Failure, dynamic memory allocation failed.

5.3.3.2 `List list_create (const enum gds_datatype type, const int opts, ...)`

Creates a new list.

Parameters

<i>type</i>	The datatype for the list.
<i>opts</i>	The following options can be OR'd together: <code>GDS_FREE_ON_DESTROY</code> to automatically <code>free()</code> pointer members when they are deleted or when the list is destroyed; <code>GDS_EXIT_ON_ERROR</code> to print a message to the standard error stream and <code>exit()</code> , rather than returning a failure status.
<i>...</i>	If <i>type</i> is <code>DATATYPE_POINTER</code> , this argument should be a pointer to a comparison function. In all other cases, this argument is not required, and will be ignored if it is provided.

Return values

<i>NULL</i>	List creation failed.
<i>non-NULL</i>	A pointer to the new list.

5.3.3.3 `bool list_delete_back (List list)`

Deletes the value at the back of the list.

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Return values

<i>true</i>	Success
<i>false</i>	Failure, dynamic memory allocation failed.

5.3.3.4 `bool list_delete_front (List list)`

Deletes the value at the front of the list.

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Return values

<i>true</i>	Success
<i>false</i>	Failure, dynamic memory allocation failed.

5.3.3.5 `bool list_delete_index (List list, const size_t index)`

Deletes the value at the specified index of the list.

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the value to delete.

Return values

<i>true</i>	Success
<i>false</i>	Failure, dynamic memory allocation failed or index was out of range.

5.3.3.6 `void list_destroy (List list)`

Destroys a list.

If the `GDS_FREE_ON_DESTROY` option was specified when creating the list, any pointer values still in the list will be `free()`d prior to destruction.

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

5.3.3.7 `bool list_element_at_index (List list, const size_t index, void * p)`

Gets the value at the specified index of the list.

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the value to get.
<i>p</i>	A pointer to an object of a type appropriate to the type set when creating the list. The object at this address will be modified to contain the value at the specified index.

Return values

<i>true</i>	Success
<i>false</i>	Failure, index was out of range.

5.3.3.8 `bool list_find (List list, size_t * index, ...)`

Tests if a value is contained in a list.

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	A pointer to a <code>size_t</code> object which, if the value is contained within the list, will be modified to contain the index of the first occurrence of that value in the list.
...	The value for which to search. This should be of a type appropriate to the type set when creating the list.

Return values

<i>true</i>	The value was found in the list
<i>false</i>	The value was not found in the list

5.3.3.9 `bool list_insert (List list, const size_t index, ...)`

Inserts a value into a list.

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index at which to insert the value.
...	The value to insert into the list. This should be of a type appropriate to the type set when creating the list.

Return values

<i>true</i>	Success
<i>false</i>	Failure, dynamic memory allocation failed or index was out of range.

5.3.3.10 `bool list_is_empty (List list)`

Tests if a list is empty.

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Return values

<i>true</i>	The list is empty
<i>false</i>	The list is not empty

5.3.3.11 `size_t list_length (List list)`

Returns the length of a list.

The length of the list is equivalent to the number of values it contains.

Parameters

<i>list</i>	A pointer to the list.
-------------	------------------------

Returns

The length of the list.

5.3.3.12 `bool list_prepend (List list, ...)`

Prepends a value to the front of a list.

Parameters

<i>list</i>	A pointer to the list.
<i>...</i>	The value to prepend to the start of the list. This should be of a type appropriate to the type set when creating the list.

Return values

<i>true</i>	Success
<i>false</i>	Failure, dynamic memory allocation failed.

5.3.3.13 `bool list_set_element_at_index (List list, const size_t index, ...)`

Sets the value at the specified index of the list.

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the value to set.
<i>...</i>	The value to which to set the specified index of the list. This should be of a type appropriate to the type set when creating the list.

Return values

<i>true</i>	Success
<i>false</i>	Failure, index was out of range.

5.4 Public interface to generic queue data structure

Typedefs

- typedef struct `queue` * `Queue`
Opaque queue type definition.

Functions

- `Queue queue_create` (const size_t capacity, const enum `gds_datatype` type, const int opts)
Creates a new queue.
- void `queue_destroy` (`Queue queue`)
Destroys a queue.
- bool `queue_push` (`Queue queue`,...)
Pushes a value onto the queue.
- bool `queue_pop` (`Queue queue`, void *p)
Pops a value from the queue.
- bool `queue_peek` (`Queue queue`, void *p)
Peeks at the top value of the queue.
- bool `queue_is_full` (`Queue queue`)
Checks whether a queue is full.
- bool `queue_is_empty` (`Queue queue`)
Checks whether a queue is empty.
- size_t `queue_capacity` (`Queue queue`)
Retrieves the current capacity of a queue.
- size_t `queue_size` (`Queue queue`)
Retrieves the current size of a queue.
- size_t `queue_free_space` (`Queue queue`)
Retrieves the free space on a queue.

5.4.1 Detailed Description

A queue is a first-in-first-out (FIFO) data structure. Two fundamental operations are possible. A value can be *pushed* onto the queue, and a value can be *popped* from the queue. By virtue of being a FIFO data structure, pushing and popping happen at opposite ends of the queue. In other words, the value popped will be the first item pushed onto the queue that has not already been popped from it.

5.4.2 Typedef Documentation

5.4.2.1 typedef struct queue* Queue

Opaque queue type definition.

5.4.3 Function Documentation

5.4.3.1 size_t queue_capacity (Queue queue)

Retrieves the current capacity of a queue.

This value can change dynamically if the `GDS_RESIZABLE` option was specified when creating the queue.

Parameters

<i>queue</i>	A pointer to the queue.
--------------	-------------------------

Returns

The capacity of the queue.

5.4.3.2 Queue queue_create (const size_t capacity, const enum gds_datatype type, const int opts)

Creates a new queue.

Parameters

<i>capacity</i>	The initial capacity of the queue.
<i>type</i>	The datatype for the queue.
<i>opts</i>	The following options can be OR'd together: GDS_RESIZABLE to dynamically resize the queue on-demand; GDS_FREE_ON_DESTROY to automatically free() pointer members when they are deleted or when the queue is destroyed; GDS_EXIT_ON_ERROR to print a message to the standard error stream and exit(), rather than returning a failure status.

Return values

<i>NULL</i>	Queue creation failed.
<i>non-NULL</i>	A pointer to the new queue.

5.4.3.3 void queue_destroy (Queue queue)

Destroys a queue.

If the GDS_FREE_ON_DESTROY option was specified when creating the queue, any pointer values still in the queue will be free()d prior to destruction.

Parameters

<i>queue</i>	A pointer to the queue.
--------------	-------------------------

5.4.3.4 size_t queue_free_space (Queue queue)

Retrieves the free space on a queue.

The free space on a queue is equivalent to the capacity of the queue less the size of the queue.

Parameters

<i>queue</i>	A pointer to the queue.
--------------	-------------------------

Returns

The free space on the queue.

5.4.3.5 bool queue_is_empty (Queue queue)

Checks whether a queue is empty.

Parameters

<i>queue</i>	A pointer to the queue.
--------------	-------------------------

Return values

<i>true</i>	Queue is empty
<i>false</i>	Queue is not empty

5.4.3.6 `bool queue_is_full (Queue queue)`

Checks whether a queue is full.

Parameters

<i>queue</i>	A pointer to the queue.
--------------	-------------------------

Return values

<i>true</i>	Queue is full
<i>false</i>	Queue is not full

5.4.3.7 `bool queue_peek (Queue queue, void * p)`

Peeks at the top value of the queue.

This function retrieves the value which would be popped from the queue, without actually popping it.

Parameters

<i>queue</i>	A pointer to the queue.
<i>p</i>	A pointer to an object of a type appropriate to the type set when creating the queue. The object at this address will be modified to contain the value at the top of the queue.

Return values

<i>true</i>	Success
<i>false</i>	Failure, queue is empty.

5.4.3.8 `bool queue_pop (Queue queue, void * p)`

Pops a value from the queue.

Parameters

<i>queue</i>	A pointer to the queue.
<i>p</i>	A pointer to an object of a type appropriate to the type set when creating the queue. The object at this address will be modified to contain the value popped from the queue.

Return values

<i>true</i>	Success
<i>false</i>	Failure, queue is empty.

5.4.3.9 `bool queue_push (Queue queue, ...)`

Pushes a value onto the queue.

Parameters

<i>queue</i>	A pointer to the queue.
<i>...</i>	The value to push onto the queue. This should be of a type appropriate to the type set when creating the queue.

Return values

<i>true</i>	Success
<i>false</i>	Failure, either because the queue is full or, if the <code>GDS_RESIZABLE</code> option was specified when creating the queue, because dynamic memory reallocation failed.

5.4.3.10 `size_t queue_size (Queue queue)`

Retrieves the current size of a queue.

The size of the queue is equivalent to the number of values currently in it.

Parameters

<i>queue</i>	A pointer to the queue.
--------------	-------------------------

Returns

The size of the queue.

5.5 Public interface to generic stack data structure

Typedefs

- typedef struct `stack` * `Stack`
Opaque stack type definition.

Functions

- `Stack stack_create` (const size_t capacity, const enum `gds_datatype` type, const int opts)
Creates a new stack.
- void `stack_destroy` (`Stack stack`)
Destroys a stack.
- bool `stack_push` (`Stack stack`,...)
Pushes a value onto the stack.
- bool `stack_pop` (`Stack stack`, void *p)
Pops a value from the stack.
- bool `stack_peek` (`Stack stack`, void *p)
Peeks at the top value of the stack.
- bool `stack_is_full` (`Stack stack`)
Checks whether a stack is full.
- bool `stack_is_empty` (`Stack stack`)
Checks whether a stack is empty.
- size_t `stack_capacity` (`Stack stack`)
Retrieves the current capacity of a stack.
- size_t `stack_size` (`Stack stack`)
Retrieves the current size of a stack.
- size_t `stack_free_space` (`Stack stack`)
Retrieves the free space on a stack.

5.5.1 Detailed Description

A stack is a last-in-first-out (LIFO) data structure. Two fundamental operations are possible. A value can be *pushed* onto the stack, and a value can be *popped* from the stack. By virtue of being a LIFO data structure, pushing and popping happen at the same end of the stack. In other words, the value popped will be the last item pushed onto the stack that has not already been popped from it.

5.5.2 Typedef Documentation

5.5.2.1 typedef struct `stack`* `Stack`

Opaque stack type definition.

5.5.3 Function Documentation

5.5.3.1 size_t `stack_capacity` (`Stack stack`)

Retrieves the current capacity of a stack.

This value can change dynamically if the `GDS_RESIZABLE` option was specified when creating the stack.

Parameters

<i>stack</i>	A pointer to the stack.
--------------	-------------------------

Returns

The capacity of the stack.

5.5.3.2 Stack stack_create (const size_t capacity, const enum gds_datatype type, const int opts)

Creates a new stack.

Parameters

<i>capacity</i>	The initial capacity of the stack.
<i>type</i>	The datatype for the stack.
<i>opts</i>	The following options can be OR'd together: <code>GDS_RESIZABLE</code> to dynamically resize the stack on-demand; <code>GDS_FREE_ON_DESTROY</code> to automatically <code>free()</code> pointer members when they are deleted or when the stack is destroyed; <code>GDS_EXIT_ON_ERROR</code> to print a message to the standard error stream and <code>exit()</code> , rather than returning a failure status.

Return values

<i>NULL</i>	Stack creation failed.
<i>non-NULL</i>	A pointer to the new stack.

5.5.3.3 void stack_destroy (Stack stack)

Destroys a stack.

If the `GDS_FREE_ON_DESTROY` option was specified when creating the stack, any pointer values still in the stack will be `free()`d prior to destruction.

Parameters

<i>stack</i>	A pointer to the stack.
--------------	-------------------------

5.5.3.4 size_t stack_free_space (Stack stack)

Retrieves the free space on a stack.

The free space on a stack is equivalent to the capacity of the stack less the size of the stack.

Parameters

<i>stack</i>	A pointer to the stack.
--------------	-------------------------

Returns

The free space on the stack.

5.5.3.5 bool stack_is_empty (Stack stack)

Checks whether a stack is empty.

Parameters

<i>stack</i>	A pointer to the stack.
--------------	-------------------------

Return values

<i>true</i>	Stack is empty
<i>false</i>	Stack is not empty

5.5.3.6 `bool stack_is_full (Stack stack)`

Checks whether a stack is full.

Parameters

<i>stack</i>	A pointer to the stack.
--------------	-------------------------

Return values

<i>true</i>	Stack is full
<i>false</i>	Stack is not full

5.5.3.7 `bool stack_peek (Stack stack, void * p)`

Peeks at the top value of the stack.

This function retrieves the value which would be popped from the stack, without actually popping it.

Parameters

<i>stack</i>	A pointer to the stack.
<i>p</i>	A pointer to an object of a type appropriate to the type set when creating the stack. The object at this address will be modified to contain the value at the top of the stack.

Return values

<i>true</i>	Success
<i>false</i>	Failure, stack is empty.

5.5.3.8 `bool stack_pop (Stack stack, void * p)`

Pops a value from the stack.

Parameters

<i>stack</i>	A pointer to the stack.
<i>p</i>	A pointer to an object of a type appropriate to the type set when creating the stack. The object at this address will be modified to contain the value popped from the stack.

Return values

<i>true</i>	Success
<i>false</i>	Failure, stack is empty.

5.5.3.9 `bool stack_push (Stack stack, ...)`

Pushes a value onto the stack.

Parameters

<i>stack</i>	A pointer to the stack.
<i>...</i>	The value to push onto the stack. This should be of a type appropriate to the type set when creating the stack.

Return values

<i>true</i>	Success
<i>false</i>	Failure, either because the stack is full or, if the <code>GDS_RESIZABLE</code> option was specified when creating the stack, because dynamic memory reallocation failed.

5.5.3.10 `size_t stack_size (Stack stack)`

Retrieves the current size of a stack.

The size of the stack is equivalent to the number of values currently in it.

Parameters

<i>stack</i>	A pointer to the stack.
--------------	-------------------------

Returns

The size of the stack.

5.6 Public interface to generic vector data structure.

Typedefs

- typedef struct `vector` * `Vector`
Opaque vector type definition.

Functions

- `Vector vector_create` (const size_t capacity, const enum `gds_datatype` type, const int opts,...)
Creates a new vector.
- void `vector_destroy` (`Vector vector`)
Destroys a vector.
- bool `vector_append` (`Vector vector`,...)
Appends a value to the back of a vector.
- bool `vector_prepend` (`Vector vector`,...)
Prepends a value to the front of a vector.
- bool `vector_insert` (`Vector vector`, const size_t index,...)
Inserts a value into a vector.
- bool `vector_delete_front` (`Vector vector`)
Deletes the value at the front of the vector.
- bool `vector_delete_back` (`Vector vector`)
Deletes the value at the back of the vector.
- bool `vector_delete_index` (`Vector vector`, const size_t index)
Deletes the value at the specified index of the vector.
- bool `vector_element_at_index` (`Vector vector`, const size_t index, void *p)
Gets the value at the specified index of the vector.
- bool `vector_set_element_at_index` (`Vector vector`, const size_t index,...)
Sets the value at the specified index of the vector.
- bool `vector_find` (`Vector vector`, size_t *index,...)
Tests if a value is contained in a vector.
- void `vector_sort` (`Vector vector`)
Sorts a vector in-place, in ascending order.
- void `vector_reverse_sort` (`Vector vector`)
Sorts a vector in-place, in descending order.
- bool `vector_is_empty` (`Vector vector`)
Tests if a vector is empty.
- size_t `vector_length` (`Vector vector`)
Returns the length of a vector.
- size_t `vector_capacity` (`Vector vector`)
Returns the capacity of a vector.
- size_t `vector_free_space` (`Vector vector`)
Returns the free space in a vector.

5.6.1 Detailed Description

A vector (or array) is a data structure containing a finite ordered collection of values which allows random access (compared to a list, which only allows sequential access).

5.6.2 Typedef Documentation

5.6.2.1 typedef struct vector* Vector

Opaque vector type definition.

5.6.3 Function Documentation

5.6.3.1 bool vector_append (Vector vector, ...)

Appends a value to the back of a vector.

Parameters

<i>vector</i>	A pointer to the vector.
...	The value to append to the end of the vector. This should be of a type appropriate to the type set when creating the vector.

Return values

<i>true</i>	Success
<i>false</i>	Failure, dynamic memory allocation failed.

5.6.3.2 size_t vector_capacity (Vector vector)

Returns the capacity of a vector.

The capacity of the vector is equivalent to the number of values it is capable of holding. This value can dynamically change if a vector resizes to append an element at the back of the vector. The capacity does not change when elements are deleted from a vector.

Parameters

<i>vector</i>	A pointer to the vector.
---------------	--------------------------

Returns

The capacity of the vector.

5.6.3.3 Vector vector_create (const size_t capacity, const enum gds_datatype type, const int opts, ...)

Creates a new vector.

Parameters

<i>capacity</i>	The initial capacity for the vector.
<i>type</i>	The datatype for the vector.
<i>opts</i>	The following options can be OR'd together: GDS_FREE_ON_DESTROY to automatically free() pointer members when they are deleted or when the vector is destroyed; GDS_EXIT_ON_ERROR to print a message to the standard error stream and exit(), rather than returning a failure status.
...	If type is DATATYPE_POINTER, this argument should be a pointer to a comparison function. In all other cases, this argument is not required, and will be ignored if it is provided.

Return values

<i>NULL</i>	Vector creation failed.
<i>non-NULL</i>	A pointer to the new vector.

5.6.3.4 `bool vector_delete_back (Vector vector)`

Deletes the value at the back of the vector.

Parameters

<i>vector</i>	A pointer to the vector.
---------------	--------------------------

Return values

<i>true</i>	Success
<i>false</i>	Failure, dynamic memory allocation failed.

5.6.3.5 `bool vector_delete_front (Vector vector)`

Deletes the value at the front of the vector.

Parameters

<i>vector</i>	A pointer to the vector.
---------------	--------------------------

Return values

<i>true</i>	Success
<i>false</i>	Failure, dynamic memory allocation failed.

5.6.3.6 `bool vector_delete_index (Vector vector, const size_t index)`

Deletes the value at the specified index of the vector.

Parameters

<i>vector</i>	A pointer to the vector.
<i>index</i>	The index of the value to delete.

Return values

<i>true</i>	Success
<i>false</i>	Failure, dynamic memory allocation failed or index was out of range.

5.6.3.7 `void vector_destroy (Vector vector)`

Destroys a vector.

If the `GDS_FREE_ON_DESTROY` option was specified when creating the vector, any pointer values still in the vector will be `free()`d prior to destruction.

Parameters

<i>vector</i>	A pointer to the vector.
---------------	--------------------------

5.6.3.8 `bool vector_element_at_index (Vector vector, const size_t index, void * p)`

Gets the value at the specified index of the vector.

Parameters

<i>vector</i>	A pointer to the vector.
<i>index</i>	The index of the value to get.
<i>p</i>	A pointer to an object of a type appropriate to the type set when creating the vector. The object at this address will be modified to contain the value at the specified index.

Return values

<i>true</i>	Success
<i>false</i>	Failure, index was out of range.

5.6.3.9 `bool vector_find (Vector vector, size_t * index, ...)`

Tests if a value is contained in a vector.

Parameters

<i>vector</i>	A pointer to the vector.
<i>index</i>	A pointer to a <code>size_t</code> object which, if the value is contained within the vector, will be modified to contain the index of the first occurrence of that value in the vector.
<i>...</i>	The value for which to search. This should be of a type appropriate to the type set when creating the vector.

Return values

<i>true</i>	The value was found in the vector
<i>false</i>	The value was not found in the vector

5.6.3.10 `size_t vector_free_space (Vector vector)`

Returns the free space in a vector.

The free space in a vector is equivalent to its capacity less its length. The free space can change if a vector dynamically resizes to append an element at the back of the vector, or if elements are deleted from the vector.

Parameters

<i>vector</i>	A pointer to the vector.
---------------	--------------------------

Returns

The free space in the vector.

5.6.3.11 `bool vector_insert (Vector vector, const size_t index, ...)`

Inserts a value into a vector.

Parameters

<i>vector</i>	A pointer to the list.
<i>index</i>	The index at which to insert the value.
...	The value to insert into the vector. This should be of a type appropriate to the type set when creating the vector.

Return values

<i>true</i>	Success
<i>false</i>	Failure, dynamic memory allocation failed or index was out of range.

5.6.3.12 `bool vector_is_empty (Vector vector)`

Tests if a vector is empty.

Parameters

<i>vector</i>	A pointer to the vector.
---------------	--------------------------

Return values

<i>true</i>	The vector is empty
<i>false</i>	The vector is not empty

5.6.3.13 `size_t vector_length (Vector vector)`

Returns the length of a vector.

The length of the vector is equivalent to the number of values it contains. This can be less than the initial capacity, and as low as zero, if elements have been deleted from the vector.

Parameters

<i>vector</i>	A pointer to the vector.
---------------	--------------------------

Returns

The length of the vector.

5.6.3.14 `bool vector_prepend (Vector vector, ...)`

Prepends a value to the front of a vector.

Parameters

<i>vector</i>	A pointer to the vector.
...	The value to prepend to the start of the vector. This should be of a type appropriate to the type set when creating the vector.

Return values

<i>true</i>	Success
<i>false</i>	Failure, dynamic memory allocation failed.

5.6.3.15 void vector_reverse_sort (Vector vector)

Sorts a vector in-place, in descending order.

Parameters

<i>vector</i>	A pointer to the vector.
---------------	--------------------------

5.6.3.16 bool vector_set_element_at_index (Vector vector, const size_t index, ...)

Sets the value at the specified index of the vector.

Parameters

<i>vector</i>	A pointer to the vector.
<i>index</i>	The index of the value to set.
...	The value to which to set the specified index of the vector. This should be of a type appropriate to the type set when creating the vector.

Return values

<i>true</i>	Success
<i>false</i>	Failure, index was out of range.

5.6.3.17 void vector_sort (Vector vector)

Sorts a vector in-place, in ascending order.

Parameters

<i>vector</i>	A pointer to the vector.
---------------	--------------------------

Chapter 6

Data Structure Documentation

6.1 gdt_generic_datatype Struct Reference

Generic datatype structure.

```
#include <gdt.h>
```

Data Fields

- enum [gds_datatype](#) type
 - [gds_cfunc](#) compfunc
 - union {
 - char [c](#)
 - unsigned char [uc](#)
 - signed char [sc](#)
 - int [i](#)
 - unsigned int [ui](#)
 - long [l](#)
 - unsigned long [ul](#)
 - long long int [ll](#)
 - unsigned long long int [ull](#)
 - size_t [st](#)
 - double [d](#)
 - char * [pc](#)
 - void * [p](#)
- } [data](#)

6.1.1 Detailed Description

Generic datatype structure.

6.1.2 Field Documentation

6.1.2.1 char gdt_generic_datatype::c

char

6.1.2.2 gds_cfunc gdt_generic_datatype::compfunc

Comparison function pointer

6.1.2.3 double gdt_generic_datatype::d

double

6.1.2.4 union { ... } gdt_generic_datatype::data

Data union

6.1.2.5 int gdt_generic_datatype::i

int

6.1.2.6 long gdt_generic_datatype::l

long

6.1.2.7 long long int gdt_generic_datatype::ll

long long

6.1.2.8 void* gdt_generic_datatype::p

void *

6.1.2.9 char* gdt_generic_datatype::pc

char *, string

6.1.2.10 signed char gdt_generic_datatype::sc

signed char

6.1.2.11 size_t gdt_generic_datatype::st

size_t

6.1.2.12 enum gds_datatype gdt_generic_datatype::type

Data type

6.1.2.13 unsigned char gdt_generic_datatype::uc

unsigned char

6.1.2.14 unsigned int gdt_generic_datatype::ui

unsigned int

6.1.2.15 unsigned long gdt_generic_datatype::ul

unsigned long

6.1.2.16 unsigned long long int gdt_generic_datatype::ull

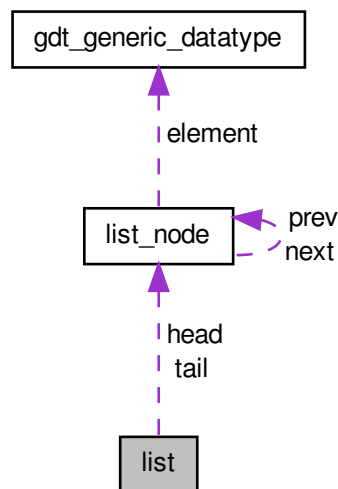
unsigned long long

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

- [include/private/gdt.h](#)

6.2 list Struct Reference

Collaboration diagram for list:



Data Fields

- `size_t` [length](#)
- `enum` [gds_datatype](#) `type`
- `gds_cfunc` `compfunc`
- `struct list_node *` `head`
- `struct list_node *` `tail`
- `bool` [free_on_destroy](#)
- `bool` [exit_on_error](#)

6.2.1 Detailed Description

List structure

6.2.2 Field Documentation

6.2.2.1 `gds_cfunc list::compfunc`

Element comparison function

6.2.2.2 `bool list::exit_on_error`

Exit on error if true

6.2.2.3 `bool list::free_on_destroy`

Free pointer elements on destroy if true

6.2.2.4 `struct list_node* list::head`

Pointer to head of list

6.2.2.5 `size_t list::length`

Length of list

6.2.2.6 `struct list_node* list::tail`

Pointer to tail of list

6.2.2.7 `enum gds_datatype list::type`

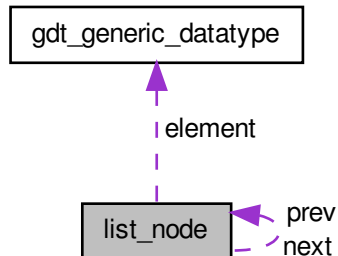
List datatype

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

- [src/list.c](#)

6.3 list_node Struct Reference

Collaboration diagram for list_node:



Data Fields

- struct [gdt_generic_datatype](#) element
- struct [list_node](#) * prev
- struct [list_node](#) * next

6.3.1 Detailed Description

List node structure

6.3.2 Field Documentation

6.3.2.1 struct [gdt_generic_datatype](#) list_node::element

Data element

6.3.2.2 struct [list_node](#)* list_node::next

Pointer to next node

6.3.2.3 struct [list_node](#)* list_node::prev

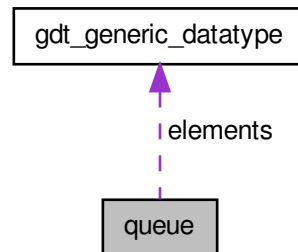
Pointer to previous node

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

- [src/list.c](#)

6.4 queue Struct Reference

Collaboration diagram for queue:



Data Fields

- size_t [front](#)
- size_t [back](#)
- size_t [capacity](#)
- size_t [size](#)
- enum [gds_datatype](#) type
- struct [gdt_generic_datatype](#) * [elements](#)
- bool [resizable](#)
- bool [free_on_destroy](#)
- bool [exit_on_error](#)

6.4.1 Detailed Description

Queue structure

6.4.2 Field Documentation

6.4.2.1 size_t queue::back

Back of queue

6.4.2.2 size_t queue::capacity

Capacity of queue

6.4.2.3 struct gdt_generic_datatype* queue::elements

Pointer to elements

6.4.2.4 bool queue::exit_on_error

Exit on error if true

6.4.2.5 bool queue::free_on_destroy

Free pointer elements on destroy if true

6.4.2.6 size_t queue::front

Front of queue

6.4.2.7 bool queue::resizable

Dynamically resizable if true

6.4.2.8 size_t queue::size

Size of queue

6.4.2.9 enum gds_datatype queue::type

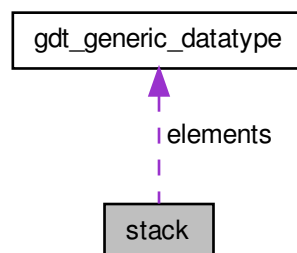
Queue datatype

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

- [src/queue.c](#)

6.5 stack Struct Reference

Collaboration diagram for stack:



Data Fields

- size_t [top](#)
- size_t [capacity](#)
- enum [gds_datatype](#) [type](#)
- struct [gdt_generic_datatype](#) * [elements](#)
- bool [resizable](#)
- bool [free_on_destroy](#)
- bool [exit_on_error](#)

6.5.1 Detailed Description

Stack structure

6.5.2 Field Documentation

6.5.2.1 `size_t stack::capacity`

Stack capacity

6.5.2.2 `struct gdt_generic_datatype* stack::elements`

Pointer to elements

6.5.2.3 `bool stack::exit_on_error`

Exit on error if true

6.5.2.4 `bool stack::free_on_destroy`

Free pointer elements on destroy if true

6.5.2.5 `bool stack::resizable`

Dynamically resizable if true

6.5.2.6 `size_t stack::top`

Top of stack

6.5.2.7 `enum gds_datatype stack::type`

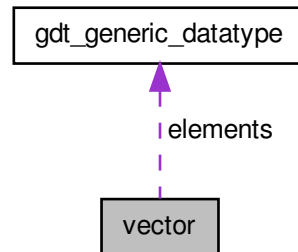
Stack datatype

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

- [src/stack.c](#)

6.6 vector Struct Reference

Collaboration diagram for vector:



Data Fields

- `size_t` [length](#)
- `size_t` [capacity](#)
- `enum` [gds_datatype](#) `type`
- `struct` [gdt_generic_datatype](#) * `elements`
- `int`(* [compfunc](#))(const void *, const void *)
- `bool` [free_on_destroy](#)
- `bool` [exit_on_error](#)

6.6.1 Detailed Description

Vector structure

6.6.2 Field Documentation

6.6.2.1 `size_t` `vector::capacity`

Vector capacity

6.6.2.2 `int`(* `vector::compfunc`)(const void *, const void *)

Compare function

6.6.2.3 `struct` `gdt_generic_datatype`* `vector::elements`

Pointer to elements

6.6.2.4 `bool` `vector::exit_on_error`

Exit on error if true

6.6.2.5 `bool vector::free_on_destroy`

Free pointer elements on destroy if true

6.6.2.6 `size_t vector::length`

Vector length

6.6.2.7 `enum gds_datatype vector::type`

Vector datatype

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

- [src/vector.c](#)

Chapter 7

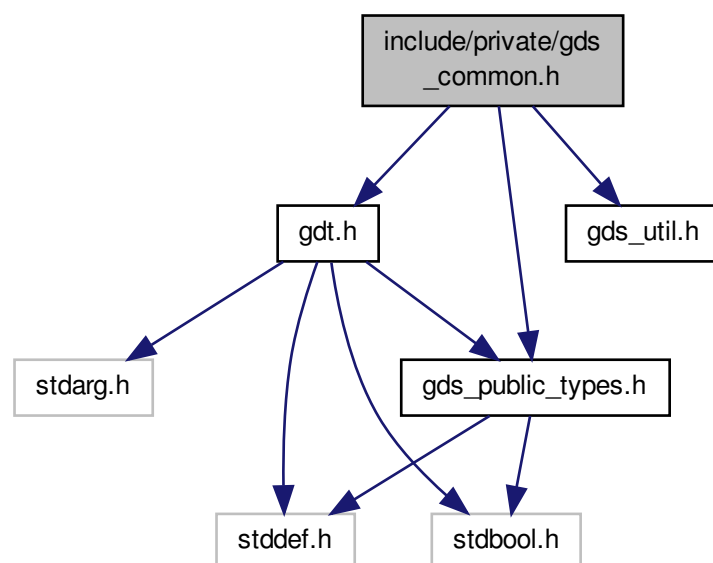
File Documentation

7.1 gds.dox File Reference

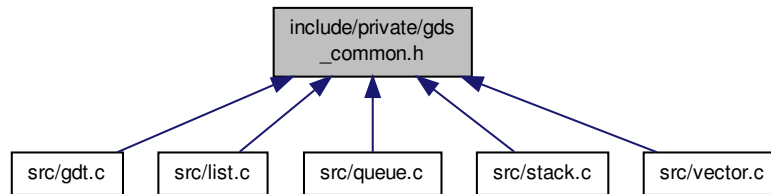
7.2 include/private/gds_common.h File Reference

Common internal headers for data structures.

```
#include "gds_public_types.h"  
#include "gdt.h"  
#include "gds_util.h"  
Include dependency graph for gds_common.h:
```



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



7.2.1 Detailed Description

Common internal headers for data structures.

Author

Paul Griffiths

Copyright

Copyright 2014 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

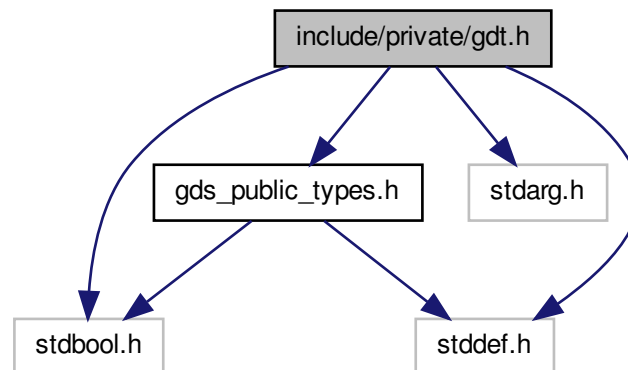
7.3 include/private/gdt.dox File Reference

7.4 include/private/gdt.h File Reference

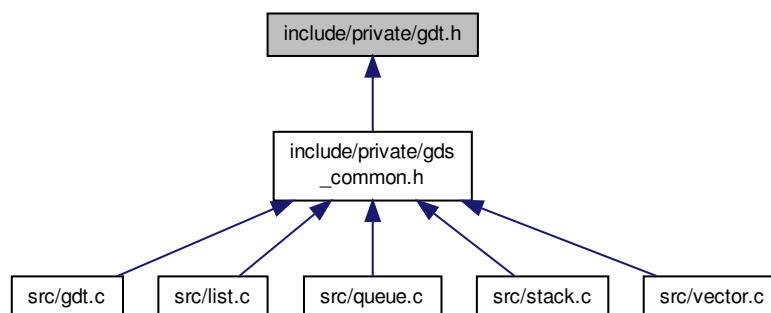
Interface to generic data element functionality.

```
#include <stdbool.h>
#include <stddef.h>
#include <stdarg.h>
#include "gds_public_types.h"
```


Include dependency graph for gdt.h:



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



Data Structures

- struct [gdt_generic_datatype](#)
Generic datatype structure.

Functions

- void [gdt_set_value](#) (struct [gdt_generic_datatype](#) *data, const enum [gds_datatype](#) type, [gds_cfunc](#) cfunc, va_list ap)
Sets the value of a generic datatype.
- void [gdt_get_value](#) (const struct [gdt_generic_datatype](#) *data, void *p)
Gets the value of a generic datatype.
- void [gdt_free](#) (struct [gdt_generic_datatype](#) *data)
Frees memory pointed to by a generic datatype.

- int [gdt_compare](#) (const struct [gdt_generic_datatype](#) *d1, const struct [gdt_generic_datatype](#) *d2)

Compares two generic datatypes.

- int [gdt_compare_void](#) (const void *p1, const void *p2)

Compares two generic datatypes via void pointers.

- int [gdt_reverse_compare_void](#) (const void *p1, const void *p2)

Reverse compares two generic datatypes via void pointers.

7.4.1 Detailed Description

Interface to generic data element functionality.

Author

Paul Griffiths

Copyright

Copyright 2014 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

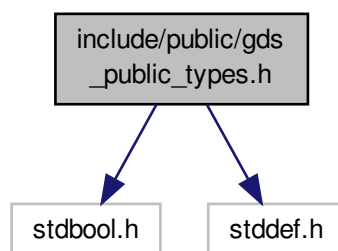
7.5 include/public/gds_public_types.h File Reference

Common public types for generic data structures library.

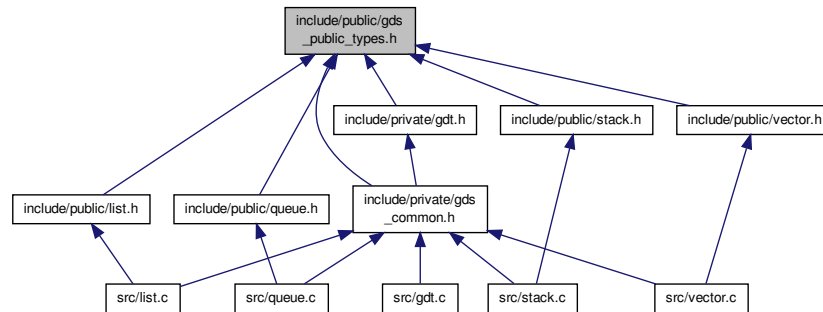
```
#include <stdbool.h>
```

```
#include <stddef.h>
```

Include dependency graph for gds_public_types.h:



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



Typedefs

- typedef int(* [gds_cfunc](#))(const void *, const void *)

Type definition for comparison function pointer.

Enumerations

- enum [gds_option](#) { [GDS_RESIZABLE](#) = 1, [GDS_FREE_ON_DESTROY](#) = 2, [GDS_EXIT_ON_ERROR](#) = 4 }

Enumeration type for data structure options.

- enum [gds_datatype](#) { [DATATYPE_CHAR](#), [DATATYPE_UNSIGNED_CHAR](#), [DATATYPE_SIGNED_CHAR](#), [DATATYPE_INT](#), [DATATYPE_UNSIGNED_INT](#), [DATATYPE_LONG](#), [DATATYPE_UNSIGNED_LONG](#), [DATATYPE_LONG_LONG](#), [DATATYPE_UNSIGNED_LONG_LONG](#), [DATATYPE_SIZE_T](#), [DATATYPE_DOUBLE](#), [DATATYPE_STRING](#), [DATATYPE_POINTER](#) }

Enumeration type for data element type.

7.5.1 Detailed Description

Common public types for generic data structures library.

Author

Paul Griffiths

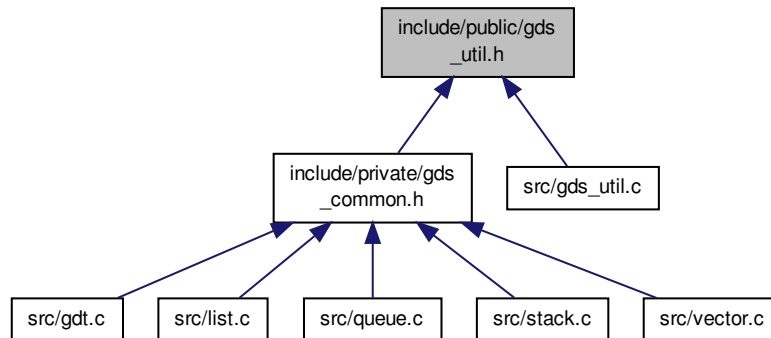
Copyright

Copyright 2014 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

7.6 include/public/gds_util.h File Reference

Interface to general utility functions.

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



Functions

- void [gds_strerror_quit](#) (const char *msg,...)
Prints an error message with error number and exits.
- void [gds_error_quit](#) (const char *msg,...)
Prints an error message exits.
- void [gds_assert_quit](#) (const char *msg,...)
Prints an error message exits via assert().

7.6.1 Detailed Description

Interface to general utility functions.

Author

Paul Griffiths

Copyright

Copyright 2014 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

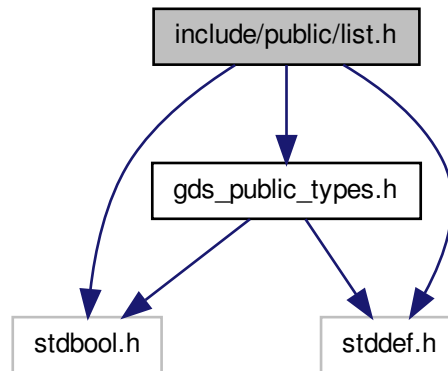
7.7 include/public/general.dox File Reference

7.8 include/public/list.dox File Reference

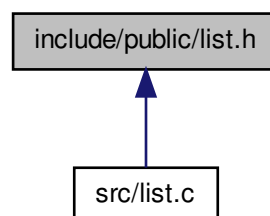
7.9 include/public/list.h File Reference

Interface to generic list data structure.

```
#include <stdbool.h>
#include <stddef.h>
#include "gds_public_types.h"
Include dependency graph for list.h:
```



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



Typedefs

- typedef struct `list` * `List`
Opaque list type definition.

Functions

- `List` `list_create` (const enum `gds_datatype` type, const int opts,...)
Creates a new list.
- void `list_destroy` (`List` list)
Destroys a list.
- bool `list_append` (`List` list,...)
Appends a value to the back of a list.

- `bool list_prepend (List list,...)`
Prepends a value to the front of a list.
- `bool list_insert (List list, const size_t index,...)`
Inserts a value into a list.
- `bool list_delete_front (List list)`
Deletes the value at the front of the list.
- `bool list_delete_back (List list)`
Deletes the value at the back of the list.
- `bool list_delete_index (List list, const size_t index)`
Deletes the value at the specified index of the list.
- `bool list_element_at_index (List list, const size_t index, void *p)`
Gets the value at the specified index of the list.
- `bool list_set_element_at_index (List list, const size_t index,...)`
Sets the value at the specified index of the list.
- `bool list_find (List list, size_t *index,...)`
Tests if a value is contained in a list.
- `bool list_is_empty (List list)`
Tests if a list is empty.
- `size_t list_length (List list)`
Returns the length of a list.

7.9.1 Detailed Description

Interface to generic list data structure. The list is implemented as a double-ended, double-linked list.

Author

Paul Griffiths

Copyright

Copyright 2014 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

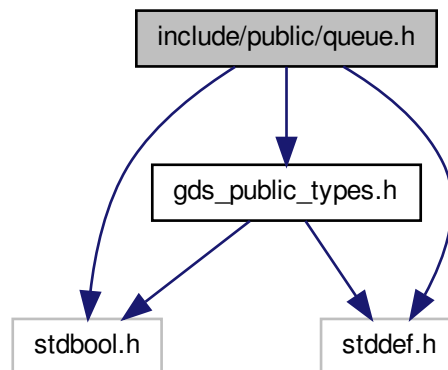
7.10 include/public/queue.dox File Reference

7.11 include/public/queue.h File Reference

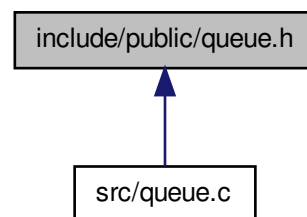
Interface to generic queue data structure.

```
#include <stdbool.h>
#include <stddef.h>
#include "gds_public_types.h"
```

Include dependency graph for queue.h:



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



Typedefs

- typedef struct [queue](#) * [Queue](#)
Opaque queue type definition.

Functions

- [Queue queue_create](#) (const [size_t](#) capacity, const enum [gds_datatype](#) type, const int opts)
Creates a new queue.
- void [queue_destroy](#) ([Queue](#) queue)
Destroys a queue.
- bool [queue_push](#) ([Queue](#) queue,...)
Pushes a value onto the queue.
- bool [queue_pop](#) ([Queue](#) queue, void *p)
Pops a value from the queue.

- bool `queue_peek` (`Queue queue`, void *p)

Peeks at the top value of the queue.

- bool `queue_is_full` (`Queue queue`)

Checks whether a queue is full.

- bool `queue_is_empty` (`Queue queue`)

Checks whether a queue is empty.

- size_t `queue_capacity` (`Queue queue`)

Retrieves the current capacity of a queue.

- size_t `queue_size` (`Queue queue`)

Retrieves the current size of a queue.

- size_t `queue_free_space` (`Queue queue`)

Retrieves the free space on a queue.

7.11.1 Detailed Description

Interface to generic queue data structure.

Author

Paul Griffiths

Copyright

Copyright 2014 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

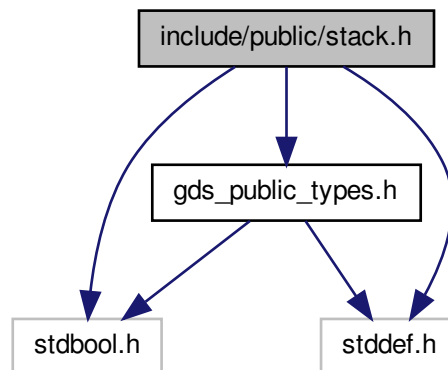
7.12 include/public/stack.dox File Reference

7.13 include/public/stack.h File Reference

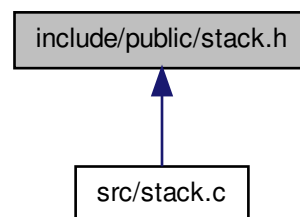
Interface to generic stack data structure.

```
#include <stdbool.h>
#include <stddef.h>
#include "gds_public_types.h"
```


Include dependency graph for stack.h:



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



Typedefs

- typedef struct [stack](#) * [Stack](#)
Opaque stack type definition.

Functions

- [Stack](#) [stack_create](#) (const size_t capacity, const enum [gds_datatype](#) type, const int opts)
Creates a new stack.
- void [stack_destroy](#) ([Stack](#) stack)
Destroys a stack.
- bool [stack_push](#) ([Stack](#) stack,...)
Pushes a value onto the stack.
- bool [stack_pop](#) ([Stack](#) stack, void *p)
Pops a value from the stack.

- bool `stack_peek` (`Stack stack`, void *p)

Peeks at the top value of the stack.

- bool `stack_is_full` (`Stack stack`)

Checks whether a stack is full.

- bool `stack_is_empty` (`Stack stack`)

Checks whether a stack is empty.

- size_t `stack_capacity` (`Stack stack`)

Retrieves the current capacity of a stack.

- size_t `stack_size` (`Stack stack`)

Retrieves the current size of a stack.

- size_t `stack_free_space` (`Stack stack`)

Retrieves the free space on a stack.

7.13.1 Detailed Description

Interface to generic stack data structure.

Author

Paul Griffiths

Copyright

Copyright 2014 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

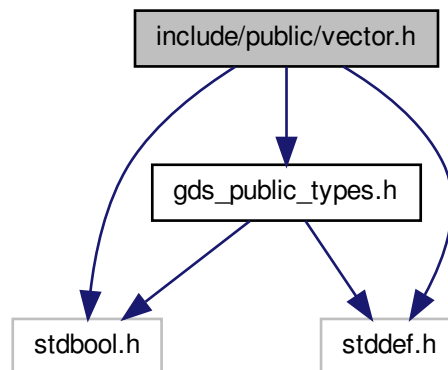
7.14 include/public/vector.dox File Reference

7.15 include/public/vector.h File Reference

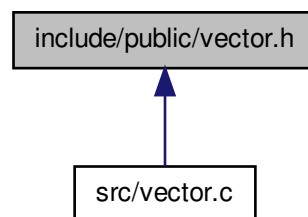
Interface to generic vector data structure.

```
#include <stdbool.h>
#include <stddef.h>
#include "gds_public_types.h"
```

Include dependency graph for vector.h:



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



Typedefs

- typedef struct `vector` * `Vector`
Opaque vector type definition.

Functions

- `Vector vector_create` (const size_t capacity, const enum `gds_datatype` type, const int opts,...)
Creates a new vector.
- void `vector_destroy` (`Vector vector`)
Destroys a vector.
- bool `vector_append` (`Vector vector`,...)
Appends a value to the back of a vector.
- bool `vector_prepend` (`Vector vector`,...)
Prepends a value to the front of a vector.

- bool `vector_insert` (`Vector vector`, const size_t index,...)
Inserts a value into a vector.
- bool `vector_delete_front` (`Vector vector`)
Deletes the value at the front of the vector.
- bool `vector_delete_back` (`Vector vector`)
Deletes the value at the back of the vector.
- bool `vector_delete_index` (`Vector vector`, const size_t index)
Deletes the value at the specified index of the vector.
- bool `vector_element_at_index` (`Vector vector`, const size_t index, void *p)
Gets the value at the specified index of the vector.
- bool `vector_set_element_at_index` (`Vector vector`, const size_t index,...)
Sets the value at the specified index of the vector.
- bool `vector_find` (`Vector vector`, size_t *index,...)
Tests if a value is contained in a vector.
- void `vector_sort` (`Vector vector`)
Sorts a vector in-place, in ascending order.
- void `vector_reverse_sort` (`Vector vector`)
Sorts a vector in-place, in descending order.
- bool `vector_is_empty` (`Vector vector`)
Tests if a vector is empty.
- size_t `vector_length` (`Vector vector`)
Returns the length of a vector.
- size_t `vector_capacity` (`Vector vector`)
Returns the capacity of a vector.
- size_t `vector_free_space` (`Vector vector`)
Returns the free space in a vector.

7.15.1 Detailed Description

Interface to generic vector data structure.

Author

Paul Griffiths

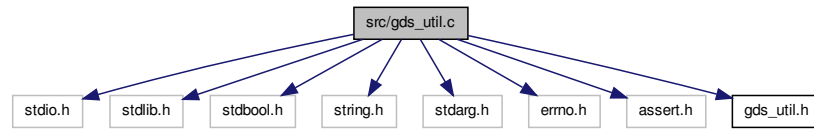
Copyright

Copyright 2014 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

7.16 src/gds_util.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>
#include <stdarg.h>
#include <errno.h>
#include <assert.h>
#include "gds_util.h"
```

Include dependency graph for gds_util.c:



Functions

- void [gds_strerror_quit](#) (const char *msg,...)
Prints an error message with error number and exits.
- void [gds_error_quit](#) (const char *msg,...)
Prints an error message exits.
- void [gds_assert_quit](#) (const char *msg,...)
Prints an error message exits via assert().

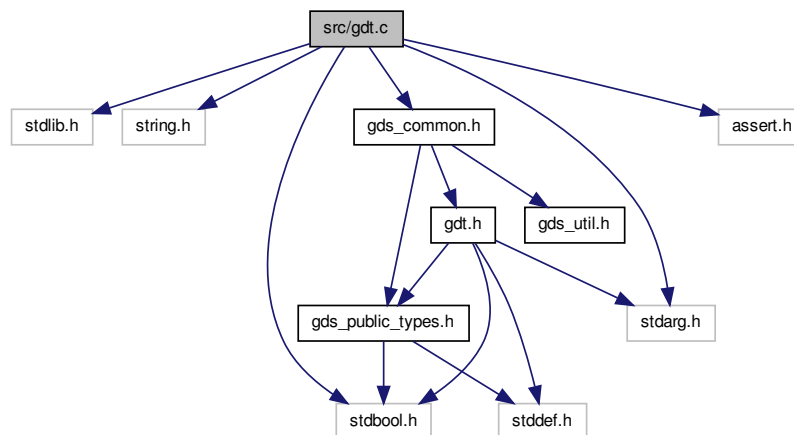
7.17 src/gdt.c File Reference

```

#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <assert.h>
#include <stdarg.h>
#include "gds_common.h"

```

Include dependency graph for gdt.c:



Functions

- static int [gdt_compare_char](#) (const void *p1, const void *p2)
- static int [gdt_compare_uchar](#) (const void *p1, const void *p2)

- static int [gdt_compare_schar](#) (const void *p1, const void *p2)
- static int [gdt_compare_int](#) (const void *p1, const void *p2)
- static int [gdt_compare_uint](#) (const void *p1, const void *p2)
- static int [gdt_compare_long](#) (const void *p1, const void *p2)
- static int [gdt_compare_ulong](#) (const void *p1, const void *p2)
- static int [gdt_compare_longlong](#) (const void *p1, const void *p2)
- static int [gdt_compare_ulonglong](#) (const void *p1, const void *p2)
- static int [gdt_compare_sizet](#) (const void *p1, const void *p2)
- static int [gdt_compare_double](#) (const void *p1, const void *p2)
- static int [gdt_compare_string](#) (const void *p1, const void *p2)
- void [gdt_set_value](#) (struct [gdt_generic_datatype](#) *data, const enum [gds_datatype](#) type, [gds_cfunc](#) cfunc, va_list ap)
Sets the value of a generic datatype.
- void [gdt_get_value](#) (const struct [gdt_generic_datatype](#) *data, void *p)
Gets the value of a generic datatype.
- void [gdt_free](#) (struct [gdt_generic_datatype](#) *data)
Frees memory pointed to by a generic datatype.
- int [gdt_compare](#) (const struct [gdt_generic_datatype](#) *d1, const struct [gdt_generic_datatype](#) *d2)
Compares two generic datatypes.
- int [gdt_compare_void](#) (const void *p1, const void *p2)
Compares two generic datatypes via void pointers.
- int [gdt_reverse_compare_void](#) (const void *p1, const void *p2)
Reverse compares two generic datatypes via void pointers.

7.17.1 Function Documentation

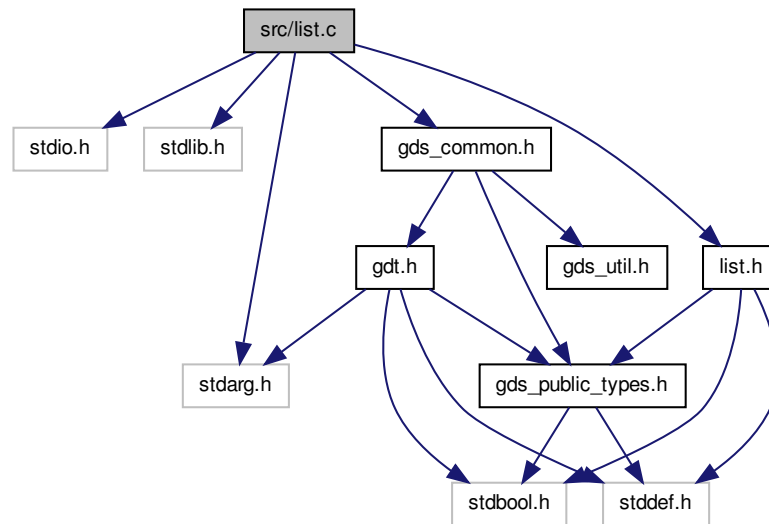
- 7.17.1.1 static int [gdt_compare_char](#) (const void * *p1*, const void * *p2*) [static]
- 7.17.1.2 static int [gdt_compare_double](#) (const void * *p1*, const void * *p2*) [static]
- 7.17.1.3 static int [gdt_compare_int](#) (const void * *p1*, const void * *p2*) [static]
- 7.17.1.4 static int [gdt_compare_long](#) (const void * *p1*, const void * *p2*) [static]
- 7.17.1.5 static int [gdt_compare_longlong](#) (const void * *p1*, const void * *p2*) [static]
- 7.17.1.6 static int [gdt_compare_schar](#) (const void * *p1*, const void * *p2*) [static]
- 7.17.1.7 static int [gdt_compare_sizet](#) (const void * *p1*, const void * *p2*) [static]
- 7.17.1.8 static int [gdt_compare_string](#) (const void * *p1*, const void * *p2*) [static]
- 7.17.1.9 static int [gdt_compare_uchar](#) (const void * *p1*, const void * *p2*) [static]
- 7.17.1.10 static int [gdt_compare_uint](#) (const void * *p1*, const void * *p2*) [static]
- 7.17.1.11 static int [gdt_compare_ulong](#) (const void * *p1*, const void * *p2*) [static]
- 7.17.1.12 static int [gdt_compare_ulonglong](#) (const void * *p1*, const void * *p2*) [static]

7.18 src/list.c File Reference

Implementation of generic list data structure.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "gds_common.h"
#include "list.h"
```

Include dependency graph for list.c:



Data Structures

- struct [list_node](#)
- struct [list](#)

Typedefs

- typedef struct [list_node](#) * [ListNode](#)

Functions

- static [ListNode](#) [list_node_create](#) ([List](#) list, va_list ap)
Private function to create list node.
- static void [list_node_destroy](#) ([List](#) list, [ListNode](#) node)
Destroys a list node.
- static [ListNode](#) [list_node_at_index](#) ([List](#) list, const size_t index)
Private function to return the node at a specified index.
- static bool [list_insert_internal](#) ([List](#) list, [ListNode](#) node, const size_t index)
Private function to insert a node into a list.
- [List](#) [list_create](#) (const enum [gds_datatype](#) type, const int opts,...)
Creates a new list.
- void [list_destroy](#) ([List](#) list)
Destroys a list.

- `bool list_append (List list,...)`
Appends a value to the back of a list.
- `bool list_prepend (List list,...)`
Prepends a value to the front of a list.
- `bool list_insert (List list, const size_t index,...)`
Inserts a value into a list.
- `bool list_delete_index (List list, const size_t index)`
Deletes the value at the specified index of the list.
- `bool list_delete_front (List list)`
Deletes the value at the front of the list.
- `bool list_delete_back (List list)`
Deletes the value at the back of the list.
- `bool list_element_at_index (List list, const size_t index, void *p)`
Gets the value at the specified index of the list.
- `bool list_set_element_at_index (List list, const size_t index,...)`
Sets the value at the specified index of the list.
- `bool list_find (List list, size_t *index,...)`
Tests if a value is contained in a list.
- `bool list_is_empty (List list)`
Tests if a list is empty.
- `size_t list_length (List list)`
Returns the length of a list.

7.18.1 Detailed Description

Implementation of generic list data structure. The list is implemented as a double-ended, double-linked list.

Author

Paul Griffiths

Copyright

Copyright 2014 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

7.18.2 Typedef Documentation

7.18.2.1 `typedef struct list_node * ListNode`

List node structure

7.18.3 Function Documentation

7.18.3.1 `static bool list_insert_internal (List list, ListNode node, const size_t index)` [static]

Private function to insert a node into a list.

Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node to insert.
<i>index</i>	The index at which to insert.

Return values

<i>true</i>	Success
<i>false</i>	Failure, index out of range

7.18.3.2 static ListNode list_node_at_index (List *list*, const size_t *index*) [static]

Private function to return the node at a specified index.

Parameters

<i>list</i>	A pointer to the list.
<i>index</i>	The index of the requested node.

Return values

<i>NULL</i>	Failure, index out of range
<i>non-NULL</i>	A pointer to the node at the specified index

7.18.3.3 static ListNode list_node_create (List *list*, va_list *ap*) [static]

Private function to create list node.

Parameters

<i>list</i>	A pointer to the list.
<i>ap</i>	A <i>va_list</i> containing the data value for the node. This should be of a type appropriate to the type set when creating the list.

Return values

<i>NULL</i>	Failure, dynamic memory allocation failed
<i>non-NULL</i>	A pointer to the new node

7.18.3.4 static void list_node_destroy (List *list*, ListNode *node*) [static]

Destroys a list node.

If the `GDS_FREE_ON_DESTROY` option was specified when creating the list, any pointer values still in the list will be `free()`d prior to destruction.

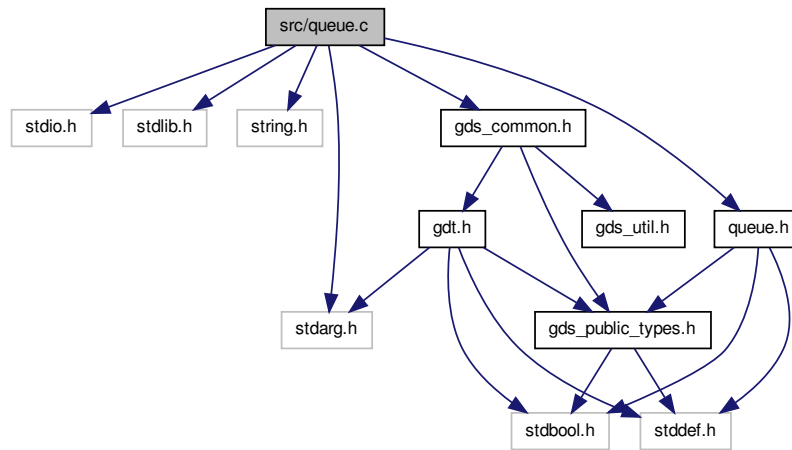
Parameters

<i>list</i>	A pointer to the list.
<i>node</i>	A pointer to the node.

7.19 src/queue.c File Reference

Implementation of generic queue data structure.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include "gds_common.h"
#include "queue.h"
Include dependency graph for queue.c:
```



Data Structures

- struct [queue](#)

Functions

- [Queue queue_create](#) (const size_t capacity, const enum [gds_datatype](#) type, const int opts)
Creates a new queue.
- void [queue_destroy](#) (Queue queue)
Destroys a queue.
- bool [queue_push](#) (Queue queue,...)
Pushes a value onto the queue.
- bool [queue_pop](#) (Queue queue, void *p)
Pops a value from the queue.
- bool [queue_peek](#) (Queue queue, void *p)
Peeks at the top value of the queue.
- bool [queue_is_full](#) (Queue queue)
Checks whether a queue is full.
- bool [queue_is_empty](#) (Queue queue)
Checks whether a queue is empty.
- size_t [queue_capacity](#) (Queue queue)
Retrieves the current capacity of a queue.
- size_t [queue_free_space](#) (Queue queue)
Retrieves the free space on a queue.
- size_t [queue_size](#) (Queue queue)
Retrieves the current size of a queue.

Variables

- static const size_t `GROWTH` = 2

7.19.1 Detailed Description

Implementation of generic queue data structure.

Author

Paul Griffiths

Copyright

Copyright 2014 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

7.19.2 Variable Documentation

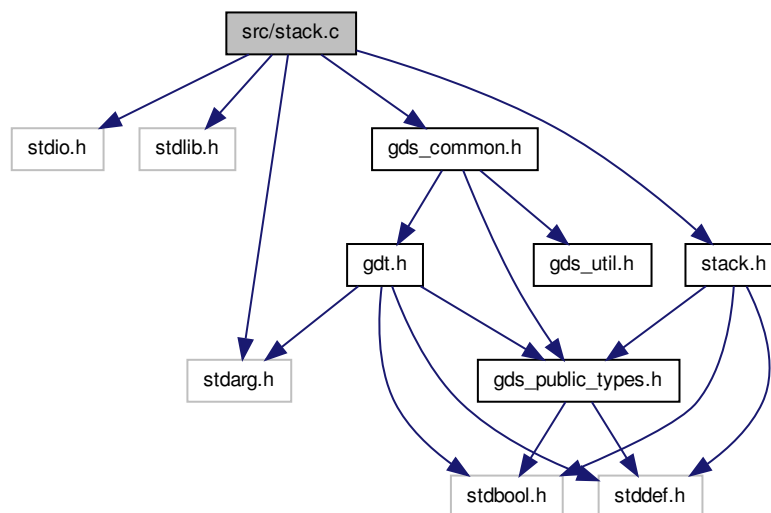
7.19.2.1 `const size_t GROWTH = 2` `[static]`

Growth factor for dynamic memory allocation

7.20 src/stack.c File Reference

Implementation of generic stack data structure.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "gds_common.h"
#include "stack.h"
Include dependency graph for stack.c:
```



Data Structures

- struct [stack](#)

Functions

- [Stack stack_create](#) (const size_t capacity, const enum [gds_datatype](#) type, const int opts)
Creates a new stack.
- void [stack_destroy](#) (Stack stack)
Destroys a stack.
- bool [stack_push](#) (Stack stack,...)
Pushes a value onto the stack.
- bool [stack_pop](#) (Stack stack, void *p)
Pops a value from the stack.
- bool [stack_peek](#) (Stack stack, void *p)
Peeks at the top value of the stack.
- bool [stack_is_full](#) (Stack stack)
Checks whether a stack is full.
- bool [stack_is_empty](#) (Stack stack)
Checks whether a stack is empty.
- size_t [stack_capacity](#) (Stack stack)
Retrieves the current capacity of a stack.
- size_t [stack_free_space](#) (Stack stack)
Retrieves the free space on a stack.
- size_t [stack_size](#) (Stack stack)
Retrieves the current size of a stack.

Variables

- static const size_t [GROWTH](#) = 2

7.20.1 Detailed Description

Implementation of generic stack data structure.

Author

Paul Griffiths

Copyright

Copyright 2014 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

7.20.2 Variable Documentation

7.20.2.1 const size_t GROWTH = 2 [static]

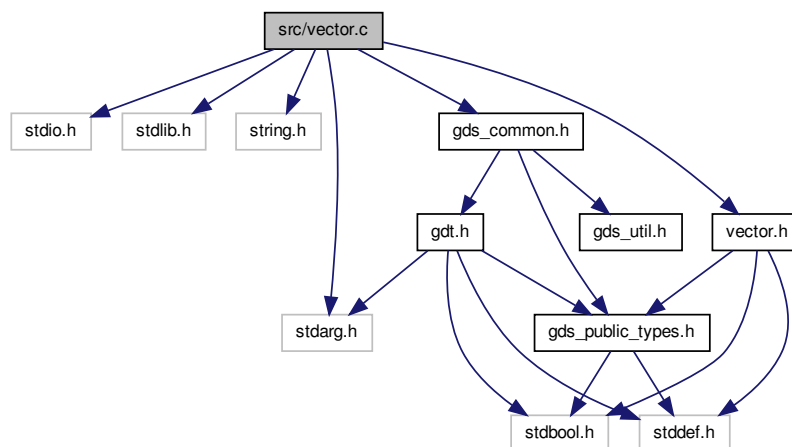
Growth factor for dynamic memory allocation

7.21 src/vector.c File Reference

Implementation of generic vector data structure.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdarg.h>
#include "gds_common.h"
#include "vector.h"
```

Include dependency graph for vector.c:



Data Structures

- struct [vector](#)

Functions

- static bool [vector_insert_internal](#) ([Vector](#) vector, const size_t index, va_list ap)
Private function to insert a vector element.
- [Vector](#) [vector_create](#) (const size_t capacity, const enum [gds_datatype](#) type, const int opts,...)
Creates a new vector.
- void [vector_destroy](#) ([Vector](#) vector)
Destroys a vector.
- bool [vector_append](#) ([Vector](#) vector,...)
Appends a value to the back of a vector.
- bool [vector_prepend](#) ([Vector](#) vector,...)
Prepends a value to the front of a vector.
- bool [vector_insert](#) ([Vector](#) vector, const size_t index,...)
Inserts a value into a vector.
- bool [vector_delete_index](#) ([Vector](#) vector, const size_t index)
Deletes the value at the specified index of the vector.
- bool [vector_delete_front](#) ([Vector](#) vector)
Deletes the value at the front of the vector.

- bool `vector_delete_back` (`Vector vector`)
Deletes the value at the back of the vector.
- bool `vector_element_at_index` (`Vector vector`, const size_t `index`, void *`p`)
Gets the value at the specified index of the vector.
- bool `vector_set_element_at_index` (`Vector vector`, const size_t `index`,...)
Sets the value at the specified index of the vector.
- bool `vector_find` (`Vector vector`, size_t *`index`,...)
Tests if a value is contained in a vector.
- void `vector_sort` (`Vector vector`)
Sorts a vector in-place, in ascending order.
- void `vector_reverse_sort` (`Vector vector`)
Sorts a vector in-place, in descending order.
- bool `vector_is_empty` (`Vector vector`)
Tests if a vector is empty.
- size_t `vector_length` (`Vector vector`)
Returns the length of a vector.
- size_t `vector_capacity` (`Vector vector`)
Returns the capacity of a vector.
- size_t `vector_free_space` (`Vector vector`)
Returns the free space in a vector.

Variables

- static const size_t `GROWTH` = 2

7.21.1 Detailed Description

Implementation of generic vector data structure.

Author

Paul Griffiths

Copyright

Copyright 2014 Paul Griffiths. Distributed under the terms of the GNU General Public License. <http://www.gnu.org/licenses/>

7.21.2 Function Documentation

7.21.2.1 static bool `vector_insert_internal` (`Vector vector`, const size_t `index`, va_list `ap`) [static]

Private function to insert a vector element.

Parameters

<code>vector</code>	A pointer to the vector.
<code>index</code>	The index at which to insert.
<code>ap</code>	A <code>va_list</code> containing the value to be inserted. This should be of a type appropriate to the type set when creating the vector.

Return values

<i>true</i>	Success
<i>false</i>	Failure, dynamic reallocation failed or index out of range.

7.21.3 Variable Documentation

7.21.3.1 `const size_t GROWTH = 2` `[static]`

Growth factor for dynamic memory allocation

Index

- back
 - queue, [40](#)
- c
 - gdt_generic_datatype, [35](#)
- capacity
 - queue, [40](#)
 - stack, [42](#)
 - vector, [43](#)
- compfunc
 - gdt_generic_datatype, [35](#)
 - list, [38](#)
 - vector, [43](#)
- d
 - gdt_generic_datatype, [36](#)
- DATATYPE_CHAR
 - Private functionality for manipulating generic datatypes, [10](#)
- DATATYPE_DOUBLE
 - Private functionality for manipulating generic datatypes, [10](#)
- DATATYPE_INT
 - Private functionality for manipulating generic datatypes, [10](#)
- DATATYPE_LONG
 - Private functionality for manipulating generic datatypes, [10](#)
- DATATYPE_LONG_LONG
 - Private functionality for manipulating generic datatypes, [10](#)
- DATATYPE_POINTER
 - Private functionality for manipulating generic datatypes, [10](#)
- DATATYPE_SIGNED_CHAR
 - Private functionality for manipulating generic datatypes, [10](#)
- DATATYPE_SIZE_T
 - Private functionality for manipulating generic datatypes, [10](#)
- DATATYPE_STRING
 - Private functionality for manipulating generic datatypes, [10](#)
- DATATYPE_UNSIGNED_CHAR
 - Private functionality for manipulating generic datatypes, [10](#)
- DATATYPE_UNSIGNED_INT
 - Private functionality for manipulating generic datatypes, [10](#)
- DATATYPE_UNSIGNED_LONG
 - Private functionality for manipulating generic datatypes, [10](#)
- DATATYPE_UNSIGNED_LONG_LONG
 - Private functionality for manipulating generic datatypes, [10](#)
- data
 - gdt_generic_datatype, [36](#)
- element
 - list_node, [39](#)
- elements
 - queue, [40](#)
 - stack, [42](#)
 - vector, [43](#)
- exit_on_error
 - list, [38](#)
 - queue, [40](#)
 - stack, [42](#)
 - vector, [43](#)
- free_on_destroy
 - list, [38](#)
 - queue, [40](#)
 - stack, [42](#)
 - vector, [43](#)
- front
 - queue, [41](#)
- GDS_EXIT_ON_ERROR
 - Public general generic data structures functionality, [13](#)
- GDS_FREE_ON_DESTROY
 - Public general generic data structures functionality, [13](#)
- GDS_RESIZABLE
 - Public general generic data structures functionality, [13](#)
- GROWTH
 - queue.c, [65](#)
 - stack.c, [66](#)
 - vector.c, [69](#)
- gds.dox, [45](#)
- gds_assert_quit
 - Public general generic data structures functionality, [13](#)
- gds_cfunc
 - Private functionality for manipulating generic datatypes, [10](#)
- gds_datatype

- Private functionality for manipulating generic datatypes, 10
- `gds_error_quit`
 - Public general generic data structures functionality, 13
- `gds_option`
 - Public general generic data structures functionality, 13
- `gds_strerror_quit`
 - Public general generic data structures functionality, 14
- `gdt.c`
 - `gdt_compare_char`, 60
 - `gdt_compare_double`, 60
 - `gdt_compare_int`, 60
 - `gdt_compare_long`, 60
 - `gdt_compare_longlong`, 60
 - `gdt_compare_schar`, 60
 - `gdt_compare_sizet`, 60
 - `gdt_compare_string`, 60
 - `gdt_compare_uchar`, 60
 - `gdt_compare_uint`, 60
 - `gdt_compare_ulong`, 60
 - `gdt_compare_ulonglong`, 60
- `gdt_compare`
 - Private functionality for manipulating generic datatypes, 10
- `gdt_compare_char`
 - `gdt.c`, 60
- `gdt_compare_double`
 - `gdt.c`, 60
- `gdt_compare_int`
 - `gdt.c`, 60
- `gdt_compare_long`
 - `gdt.c`, 60
- `gdt_compare_longlong`
 - `gdt.c`, 60
- `gdt_compare_schar`
 - `gdt.c`, 60
- `gdt_compare_sizet`
 - `gdt.c`, 60
- `gdt_compare_string`
 - `gdt.c`, 60
- `gdt_compare_uchar`
 - `gdt.c`, 60
- `gdt_compare_uint`
 - `gdt.c`, 60
- `gdt_compare_ulong`
 - `gdt.c`, 60
- `gdt_compare_ulonglong`
 - `gdt.c`, 60
- `gdt_compare_void`
 - Private functionality for manipulating generic datatypes, 10
- `gdt_free`
 - Private functionality for manipulating generic datatypes, 11
- `gdt_generic_datatype`, 35
 - `c`, 35
 - `compfunc`, 35
 - `d`, 36
 - `data`, 36
 - `i`, 36
 - `l`, 36
 - `ll`, 36
 - `p`, 36
 - `pc`, 36
 - `sc`, 36
 - `st`, 36
 - `type`, 36
 - `uc`, 36
 - `ui`, 36
 - `ul`, 37
 - `ull`, 37
- `gdt_get_value`
 - Private functionality for manipulating generic datatypes, 11
- `gdt_reverse_compare_void`
 - Private functionality for manipulating generic datatypes, 11
- `gdt_set_value`
 - Private functionality for manipulating generic datatypes, 12
- `head`
 - `list`, 38
- `i`
 - `gdt_generic_datatype`, 36
- `include/private/gds_common.h`, 45
- `include/private/gdt.dox`, 46
- `include/private/gdt.h`, 46
- `include/public/gds_public_types.h`, 48
- `include/public/gds_util.h`, 49
- `include/public/general.dox`, 50
- `include/public/list.dox`, 50
- `include/public/list.h`, 50
- `include/public/queue.dox`, 52
- `include/public/queue.h`, 52
- `include/public/stack.dox`, 54
- `include/public/stack.h`, 54
- `include/public/vector.dox`, 56
- `include/public/vector.h`, 56
- `l`
 - `gdt_generic_datatype`, 36
- `length`
 - `list`, 38
 - `vector`, 44
- `List`
 - Public interface to generic list data structure, 15
- `list`, 37
 - `compfunc`, 38
 - `exit_on_error`, 38
 - `free_on_destroy`, 38
 - `head`, 38
 - `length`, 38

- tail, 38
- type, 38
- list.c
 - list_insert_internal, 62
 - list_node_at_index, 63
 - list_node_create, 63
 - list_node_destroy, 63
 - ListNode, 62
- list_append
 - Public interface to generic list data structure, 16
- list_create
 - Public interface to generic list data structure, 16
- list_delete_back
 - Public interface to generic list data structure, 16
- list_delete_front
 - Public interface to generic list data structure, 16
- list_delete_index
 - Public interface to generic list data structure, 17
- list_destroy
 - Public interface to generic list data structure, 17
- list_element_at_index
 - Public interface to generic list data structure, 17
- list_find
 - Public interface to generic list data structure, 17
- list_insert
 - Public interface to generic list data structure, 18
- list_insert_internal
 - list.c, 62
- list_is_empty
 - Public interface to generic list data structure, 18
- list_length
 - Public interface to generic list data structure, 18
- list_node, 39
 - element, 39
 - next, 39
 - prev, 39
- list_node_at_index
 - list.c, 63
- list_node_create
 - list.c, 63
- list_node_destroy
 - list.c, 63
- list_prepend
 - Public interface to generic list data structure, 18
- list_set_element_at_index
 - Public interface to generic list data structure, 19
- ListNode
 - list.c, 62
- ll
 - gdt_generic_datatype, 36
- next
 - list_node, 39
- p
 - gdt_generic_datatype, 36
- pc
 - gdt_generic_datatype, 36
- prev
 - list_node, 39
- Private functionality for manipulating generic datatypes, 9
 - DATATYPE_CHAR, 10
 - DATATYPE_DOUBLE, 10
 - DATATYPE_INT, 10
 - DATATYPE_LONG, 10
 - DATATYPE_LONG_LONG, 10
 - DATATYPE_POINTER, 10
 - DATATYPE_SIGNED_CHAR, 10
 - DATATYPE_SIZE_T, 10
 - DATATYPE_STRING, 10
 - DATATYPE_UNSIGNED_CHAR, 10
 - DATATYPE_UNSIGNED_INT, 10
 - DATATYPE_UNSIGNED_LONG, 10
 - DATATYPE_UNSIGNED_LONG_LONG, 10
 - gds_cfunc, 10
 - gds_datatype, 10
 - gdt_compare, 10
 - gdt_compare_void, 10
 - gdt_free, 11
 - gdt_get_value, 11
 - gdt_reverse_compare_void, 11
 - gdt_set_value, 12
- Public general generic data structures functionality, 13
 - GDS_EXIT_ON_ERROR, 13
 - GDS_FREE_ON_DESTROY, 13
 - GDS_RESIZABLE, 13
 - gds_assert_quit, 13
 - gds_error_quit, 13
 - gds_option, 13
 - gds_strerror_quit, 14
- Public interface to generic list data structure, 15
 - List, 15
 - list_append, 16
 - list_create, 16
 - list_delete_back, 16
 - list_delete_front, 16
 - list_delete_index, 17
 - list_destroy, 17
 - list_element_at_index, 17
 - list_find, 17
 - list_insert, 18
 - list_is_empty, 18
 - list_length, 18
 - list_prepend, 18
 - list_set_element_at_index, 19
- Public interface to generic queue data structure, 20
 - Queue, 20
 - queue_capacity, 20
 - queue_create, 21
 - queue_destroy, 21
 - queue_free_space, 21
 - queue_is_empty, 21
 - queue_is_full, 22
 - queue_peek, 22
 - queue_pop, 22
 - queue_push, 22

- queue_size, 23
- Public interface to generic stack data structure, 24
 - Stack, 24
 - stack_capacity, 24
 - stack_create, 25
 - stack_destroy, 25
 - stack_free_space, 25
 - stack_is_empty, 25
 - stack_is_full, 26
 - stack_peek, 26
 - stack_pop, 26
 - stack_push, 26
 - stack_size, 27
- Public interface to generic vector data structure., 28
 - Vector, 29
 - vector_append, 29
 - vector_capacity, 29
 - vector_create, 29
 - vector_delete_back, 30
 - vector_delete_front, 30
 - vector_delete_index, 30
 - vector_destroy, 30
 - vector_element_at_index, 31
 - vector_find, 31
 - vector_free_space, 31
 - vector_insert, 31
 - vector_is_empty, 32
 - vector_length, 32
 - vector_prepend, 32
 - vector_reverse_sort, 32
 - vector_set_element_at_index, 33
 - vector_sort, 33
- Queue
 - Public interface to generic queue data structure, 20
- queue, 40
 - back, 40
 - capacity, 40
 - elements, 40
 - exit_on_error, 40
 - free_on_destroy, 40
 - front, 41
 - resizable, 41
 - size, 41
 - type, 41
- queue.c
 - GROWTH, 65
- queue_capacity
 - Public interface to generic queue data structure, 20
- queue_create
 - Public interface to generic queue data structure, 21
- queue_destroy
 - Public interface to generic queue data structure, 21
- queue_free_space
 - Public interface to generic queue data structure, 21
- queue_is_empty
 - Public interface to generic queue data structure, 21
- queue_is_full
 - Public interface to generic queue data structure, 22
- queue_peek
 - Public interface to generic queue data structure, 22
- queue_pop
 - Public interface to generic queue data structure, 22
- queue_push
 - Public interface to generic queue data structure, 22
- queue_size
 - Public interface to generic queue data structure, 23
- resizable
 - queue, 41
 - stack, 42
- sc
 - gdt_generic_datatype, 36
- size
 - queue, 41
- src/gds_util.c, 58
- src/gdt.c, 59
- src/list.c, 60
- src/queue.c, 63
- src/stack.c, 65
- src/vector.c, 67
- st
 - gdt_generic_datatype, 36
- Stack
 - Public interface to generic stack data structure, 24
- stack, 41
 - capacity, 42
 - elements, 42
 - exit_on_error, 42
 - free_on_destroy, 42
 - resizable, 42
 - top, 42
 - type, 42
- stack.c
 - GROWTH, 66
- stack_capacity
 - Public interface to generic stack data structure, 24
- stack_create
 - Public interface to generic stack data structure, 25
- stack_destroy
 - Public interface to generic stack data structure, 25
- stack_free_space
 - Public interface to generic stack data structure, 25
- stack_is_empty
 - Public interface to generic stack data structure, 25
- stack_is_full
 - Public interface to generic stack data structure, 26
- stack_peek
 - Public interface to generic stack data structure, 26
- stack_pop
 - Public interface to generic stack data structure, 26
- stack_push
 - Public interface to generic stack data structure, 26
- stack_size
 - Public interface to generic stack data structure, 27
- tail

- list, [38](#)
- top
 - stack, [42](#)
- type
 - gdt_generic_datatype, [36](#)
 - list, [38](#)
 - queue, [41](#)
 - stack, [42](#)
 - vector, [44](#)
- uc
 - gdt_generic_datatype, [36](#)
- ui
 - gdt_generic_datatype, [36](#)
- ul
 - gdt_generic_datatype, [37](#)
- ull
 - gdt_generic_datatype, [37](#)
- Vector
 - Public interface to generic vector data structure., [29](#)
- vector, [43](#)
 - capacity, [43](#)
 - compfunc, [43](#)
 - elements, [43](#)
 - exit_on_error, [43](#)
 - free_on_destroy, [43](#)
 - length, [44](#)
 - type, [44](#)
- vector.c
 - GROWTH, [69](#)
 - vector_insert_internal, [68](#)
- vector_append
 - Public interface to generic vector data structure., [29](#)
- vector_capacity
 - Public interface to generic vector data structure., [29](#)
- vector_create
 - Public interface to generic vector data structure., [29](#)
- vector_delete_back
 - Public interface to generic vector data structure., [30](#)
- vector_delete_front
 - Public interface to generic vector data structure., [30](#)
- vector_delete_index
 - Public interface to generic vector data structure., [30](#)
- vector_destroy
 - Public interface to generic vector data structure., [30](#)
- vector_element_at_index
 - Public interface to generic vector data structure., [31](#)
- vector_find
 - Public interface to generic vector data structure., [31](#)
- vector_free_space
 - Public interface to generic vector data structure., [31](#)
- vector_insert
 - Public interface to generic vector data structure., [31](#)
- vector_insert_internal
 - vector.c, [68](#)
- vector_is_empty
 - Public interface to generic vector data structure., [32](#)
- vector_length
 - Public interface to generic vector data structure., [32](#)
- vector_prepend
 - Public interface to generic vector data structure., [32](#)
- vector_reverse_sort
 - Public interface to generic vector data structure., [32](#)
- vector_set_element_at_index
 - Public interface to generic vector data structure., [33](#)
- vector_sort
 - Public interface to generic vector data structure., [33](#)