libdynamic Documentation

Release 1.1.0

Fredrik Widlund

Dec 17, 2017

Contents

This is the documentation for libdynamic 1.1.0, last updated Dec 17, 2017.

CHAPTER 1

Introduction

libdynamic is a C library for various dynamic container types. Its main features and design principles are:

- High performance/low overhead
- Simplicity
- Uniformity
- Flexibility

Where appropriate, containers are modelled roughly after the stdc++ equivalents.

libdynamic is licensed under the MIT license; see LICENSE in the source distribution for details.

CHAPTER 2

Contents

2.1 Getting Started

2.1.1 Compiling and installing libdynamic

The libdynamic source is available at https://github.com/fredrikwidlund/libdynamic/releases/download/v1.1.0/libdynamic-1.1.0.tar.gz

Unpack the source tarball and change to the source directory:

```
$ tar xfz libdynamic-1.1.0.tar.gz
$ cd libdynamic-1.1.0
```

The source uses GNU Autotools (autoconf, automake, libtool), so compiling and installing is extremely simple:

```
$ ./configure
$ make
$ make install
```

To run the test suite which requires cmocka and valgrind, invoke:

```
$ make check
```

To change the destination directory (`/usr/local` by default), use the `--prefix=DIR` argument to `./configure`. See `./configure --help` for the list of all possible configuration options.

The command `make check` runs the test suite distributed with libdynamic. This step is not strictly necessary, but it may find possible problems that libdynamic has on your platform. If any problems are found, please report them.

If you obtained the source from a Git repository (or any other source control system), there's no `./configure` script as it's not kept in version control. To create the script, the build system needs to be bootstrapped. There are many ways to do this, but the easiest one is to use the supplied autogen.sh script:

```
$ ./autogen.sh
```

2.1.2 Building the documentation

(This subsection describes how to build the HTML documentation you are currently reading, so it can be safely skipped.)

Documentation is in the `docs/` subdirectory. It's written in reStructuredText with Sphinx annotations. To generate the HTML documentation, invoke:

```
$ make html
```

and point your browser to `doc/_build/html/index.html`. Sphinx 1.0 or newer is required to generate the documentation.

2.1.3 Compiling programs that use libdynamic

libdynamic headers files are included through one C header file, `dynamic.h`, so it's enough to put the line

```
#include <dynamic.h>
```

in the beginning of every source file that uses libdynamic.

There's also just one library to link with, `libdynamic`. libdynamic is built as a static library and should be compiled with LTO (link time optimization) to provide the best performance. Compile and link the program as follows:

```
$ cc -o prog prog.c -flto -fuse-linker-plugin -ldynamic
```

Use of pkg-config is supported and recommended:

```
$ cc -o prog prog.c `pkg-config --cflags --libs libdynamic`
```

2.2 API Reference

2.2.1 Library Version

The libdynamic version uses Semantic Versioning and is of the form *A.B.C*, where *A* is the major version, *B* is the minor version and *C* is the patch version.

When a new release only fixes bugs and doesn't add new features or functionality, the patch version is incremented. When new features are added in a backwards compatible way, the minor version is incremented and the micro version is set to zero. When there are backwards incompatible changes, the major version is incremented and others are set to zero.

The following preprocessor constants specify the current version of the library:

LIBDYNAMIC_VERSION_MAJOR, LIBDYNAMIC_VERSION_MINOR, LIBDYNAMIC_VERSION_PATCH
Integers specifying the major, minor and patch versions, respectively.

LIBDYNAMIC_VERSION A string representation of the current version, e.g. `"1.2.1"`

2.2.2 Design

Bounds checking

Since libdynamic is a low-level and high-performance library, bounds checking is left for the user to implement when and where needed.

Memory allocation

Since gracefully handling memory allocation errors is difficult at best and makes code difficult to optimize libdynamic will exit on memory allocation errors.

2.2.3 Buffer

A buffer object represents raw memory that is dynamically increased when data is inserted. The amount of memory actually allocated will grow exponentially to allow for amortized constant time appends.

buffer
This data structure represents the buffer object.

void **buffer_construct** (*buffer *buffer*)
Constructs an empty *buffer*.

void **buffer_destruct** (*buffer *buffer*)
Releases all resources used by the *buffer*.

size_t **buffer_size** (*buffer *buffer*)
Returns the size of the memory contained in the *buffer*.

size_t **buffer_capacity** (*buffer *buffer*)
Returns the amount of memory allocated for the *buffer*.

void **buffer_reserve** (*buffer *buffer*, size_t *size*)
Ensure that the *buffer* capacity is at least *size* bytes large.

void **buffer_resize** (*buffer *buffer*, size_t *size*)
Set the buffer size of *buffer* to be *size*. If the buffer is enlarged the added buffer data is undefined.

void **buffer_compact** (*buffer *buffer*)
Reduces the amount of allocated memory in the *buffer* to match the current buffer size.

void **buffer_insert** (*buffer *buffer*, size_t *position*, void **data*, size_t *size*)
Inserts *data* with a given *size* into the given *position* of the *buffer*

void **buffer_insert_fill** (*buffer *buffer*, size_t *postion*, size_t *count*, void **data*, size_t *size*)
Inserts *count* copies of *data* with a given *size* into the given *position* of the *buffer*

void **buffer_erase** (*buffer *buffer*, size_t *position*, size_t *size*)
Removes *size* bytes from the data in the *buffer* at the given *position*.

void **buffer_clear** (*buffer *buffer*)
Clears the *buffer* of all content.

void ***buffer_data** (*buffer *buffer*)
Returns a pointer the the content of the *buffer*.

2.2.4 List

Lists are sequence containers that allow constant time insert and erase operations anywhere within the sequence, and iteration in both directions.

List containers are implemented as doubly-linked lists; Doubly linked lists can store each of the elements they contain in different and unrelated storage locations. The ordering is kept internally by the association to each element of a link to the element preceding it and a link to the element following it.

Lists are modelled roughly after the C++ list counterpart.

list
    This data structure represents the list object.

void **list_release_callback** (void *)
    This type defines a function reference to a user defined callback that release resources associated with an object

int **list_compare_callback** (void **first*, void **second*)
    This type defines a function reference to a user defined callback that compares the *first* and the *second* object, and returns a negative value if *first* is smaller, a positive value if *first* is larger, and 0 if they are the same.

void **list_construct** (*list* **list*)
    Constructs an empty *list*.

void **list_destruct** (*list* **list*, *list_release_callback* **release*)
    Releases all resources used by the *list*. If object has resources that needs to be released the *release* callback optionally can be defined.

void ***list_next** (void **object*)
    Returns a pointer to the next object after *object*.

void ***list_previous** (void **object*)
    Returns a pointer to the previous object before *object*.

int **list_empty** (*list* **list*)
    Returns 1 if the *list* is empty.

void ***list_front** (*list* **list*)
    Returns a pointer to the first object in *list*.

void ***list_back** (*list* **list*)
    Returns a pointer to the last object in *list*.

void ***list_end** (*list* **list*)
    Returns a pointer to the watch dog object at the end of the *list*.

void ***list_push_front** (*list* **list*, void **object*, size_t *size*)
    Copies the contents of *object* of size *size* to the front of the *list*.

void ***list_push_back** (*list* **list*, void **object*, size_t *size*)
    Copies the contents of *object* of size *size* to the back of the *list*.

void **list_insert** (void **list_object*, void **object*, size_t *size*)
    Copies the contents of *object* of size *size* before *list_object*.

void **list_erase** (void **object*, *list_release_callback* **release*)
    Deletes *object* from the list. If the object has resources that needs to be released the *release* callback optionally can be defined.

void **list_clear** (*list* **list*, *list_release_callback* **release*)
    Deletes all objects from *list*. If the objects has resources that needs to be released the *release* callback optionally can be defined.

void ***list_find** (*list* **list*, *list_compare_callback* **compare*, void **object*)
: Finds an object in *list* where the contents are the same as for *object*. The callback function *compare* needs to be defined accordingly.

2.2.5 Vector

Vectors are sequence containers representing arrays that can change in size. Vectors are modelled roughly after the C++ vector counterpart.

Just like arrays, vectors use contiguous storage locations for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container.

Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it. This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container.

Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Reallocations only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with amortized constant time complexity.

Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

vector
: This data structure represents the vector object.

void **vector_release_callback** (void *)
: This type defines a function reference to a user defined callback that release resources associated with an object

void **vector_construct** (*vector* **vector*, size_t *size*)
: Constructs an empty *vector* for elements of the given *size*.

void **vector_destruct** (*vector* **vector*, *vector_release_callback* **release*)
: Releases all resources used by the *vector*, optionally calling *release* to release resources associated with each object.

size_t **vector_size** (*vector* **vector*)
: Returns the size of the memory contained in the *vector*.

size_t **vector_capacity** (*vector* **vector*)
: Returns the amount of memory allocated for the *vector*.

int **vector_empty** (*vector* **vector*)
: Returns 1 if the *vector* contains no elements.

void **vector_reserve** (*vector* **vector*, size_t *size*)
: Ensure that the *vector* capacity is at least *size* elements.

void **vector_shrink_to_fit** (*vector* **vector*)
: Reduces the amount of allocated memory in the *vector* to match the current vector size.

void ***vector_at** (*vector* **vector*, size_t *position*)
: Returns a pointer to the element in the given *position* in the *vector*.

void ***vector_front** (*vector* **vector*)
: Returns a pointer to the first element in the *vector*.

void ***vector_back** (*vector* *vector)
Returns a pointer to the last element in the *vector*.

void ***vector_data** (*vector* *vector)
Returns a direct pointer to the memory array used internally by the *vector* to store its owned elements.

Because elements in the vector are guaranteed to be stored in contiguous storage locations in the same order as represented by the vector, the pointer retrieved can be offset to access any element in the array.

void **vector_insert** (*vector* *vector, size_t *position*, void **object*)
Inserts the *object* into the *vector* at the given *position*.

void **vector_insert_range** (*vector* *vector, size_t *position*, void **first*, void **last*)
Inserts a range of sequential objects, specified by giving the *first* and *last* object, into the *vector* at the given *position*.

void **vector_insert_fill** (*vector* *vector, size_t *position*, size_t *count*, void **object*)
Inserts *count* copies of the *object* into the *vector* at the given *position*.

vector_erase (*vector* *vector, size_t *position*, *vector_release_callback* **release*)
Removes the element in the given *position* in the *vector*, optionally calling *release* to release resources associated with the object.

vector_erase_range (*vector* *vector, size_t *first*, size_t *last*, *vector_release_callback* **release*)
Removes the elements from the *vector* starting at the given *first* position and ending before the *last* position, optionally calling *release* to release resources associated with each object.

The element at the *last* position is not removed.

void **vector_push_back** (*vector* *vector, void **object*)
Appends the *object* to the end of the *vector*.

void **vector_pop_back** (*vector* *vector)
Removes the last element of the *vector*.

void **vector_clear** (*vector* *vector, *vector_release_callback* **release*)
Clears the *vector* of all elements, optionally calling *release* to release resources associated with each object.

2.2.6 String

Strings are objects that represent sequences of characters. String objects are modelled roughly after the C++ string counterpart.

string
This data structure represents the string object.

void **string_construct** (*string* *string)
Constructs an empty *string*.

void **string_destruct** (*string* *string)
Releases all resources used by the *string*.

size_t **string_length** (*string* *string)
Returns the length of the *string*.

size_t **string_capacity** (*string* *string)
Returns the amount of memory allocated for the *string*.

int **string_empty** (*string* *string)
Returns 1 if the *string* is empty.

void **string_reserve** (*string* **string*, size_t *size*)
Ensures that the allocated memory for the *string* is at least *size* bytes.

void **string_shrink_to_fit** (*string* **string*)
Reduces the amount of allocated memory in the *string* to match the current string length.

void **string_insert** (*string* **string*, size_t *position*, char **characters*)
Insert null-terminated *characters* into the *string* at the given *position*.

void **string_insert_buffer** (*string* **string*, size_t *position*, char **data*, size_t *size*)
Insert *data* of the given *size* into the *string* at the given *position*.

void **string_prepend** (*string* **string*, char **characters*)
Prepend null-terminated *characters* onto the *string*.

void **string_append** (*string* **string*, char **characters*)
Append null-terminated *characters* onto the *string*.

void **string_erase** (*string* **string*, size_t *position*, size_t *size*)
Remove *size* number of characters from the *string* at the given *position*.

void **string_replace** (*string* **string*, size_t *position*, size_t *size*, char **characters*)
Replace the portion of the *string* that begins at *position* and spans *size* positions with null-terminated *characters*.

void **string_replace_all** (*string* **string*, char **find*, char **sub*)
Replace all occurances of *find* with *sub*.

void **string_clear** (*string* **string*)
Empty the *string*.

char ***string_data** (*string* **string*)
Return null-terminated characters corresponding to the content of *string*.

ssize_t **string_find** (*string* **string*, char **find*, size_t *position*)
Find the first position of *find* in the *string* starting at the given *position*.

If no position can be found the function will return -1.

int **string_compare** (*string* **one*, *string* **two*)
Returns 1 if string *one* and string *two* contain the same characters.

void **string_split** (*string* **string*, char **delimiters*, *vector* **vector*)
Splits the *string* at any character specified in *delimiters* into a *vector* of strings. Empty parts are not included in the result. *vector* should point at allocated but uninitialized memory before being supplied to the function.

2.2.7 Map

Maps are associative containers that store elements formed by the combination of a key value and a mapped value, and which allows for fast retrieval of individual elements based on their keys. Map objects are modelled roughly after the C++ unordered_map counterpart.

For performance reasons some support callbacks need to be included in various calls rather than as map properties.

size_t **map_hash_callback** (void **element*)
The *map_hash_callback* function should return a hash value of the key of the element.

int **map_equal_callback** (void **element1*, void **element2*)
The *map_equal_callback* function is called with a pointer to two elements, *element1* and *element2*, and should return 1 if the elements are equal.

void **map_set_callback** (void **destination*, void **source*)
The *map_set_callback* function is called with a pointer to a *source* element from where the element data is read, and a *destination* element where the data is written.

void **map_release_callback** (void **element*)
The *map_release_callback* function is called with a pointer a map element when it is removed from the map.

map
This data structure represents the map object.

void **map_construct** (*map* **map*, size_t *element_size*, void **element_empty*, int (**set*)(void *, void *))
Constructs an empty *map*, where each element containing the key and value is of the size *element_size*, and *element_empty* corresponds to an empty element.

void **map_destruct** (*map* **map*, int (**equal*)(void *, void *), void (*release)(void *))
Releases all resources used by the *map*. The *release* callback can be NULL, and if so *equal* is not required.

size_t **map_size** (*map* **map*)
Returns the number of elements in the *map*.

void **map_reserve** (*map* **map*, size_t *size*, size_t (**hash*)(void *), int (*equal)(void *, void *), int (*set)(void *, void *))
Reserves space in the *map* for *size* number of elements.

void ***map_element_empty** (*map* **map*)
Returns the defined empty element of the *map*.

void ***map_at** (*map* **map*, void **element*, size_t (**hash*)(void *), int (*equal)(void *, void *))
Returns a pointer to the element in the *map* that has a key that cornensponds to the key in *element*. If the key is not found a pointer to an empty element is returned.

void **map_insert** (*map* **map*, void **element*, size_t (**hash*)(void *), int (*equal)(void *, void *), int (*set)(void *, void *), void (*release)(void *))
Insert an *element* into the *map*. If the key of the element already exists in the map the element will be released.

void **map_erase** (*map* **map*, void **element*, size_t (**hash*)(void *), int (*equal)(void *, void *), int (*set)(void *, void *), void (*release)(void *))
Removes an *element* from the *map*.

void **map_clear** (*map* **map*, int (**equal*)(void *, void *), int (*)(void *set, void *), void (*release)(void *))
Clears the *map* of all the elements.

2.2.8 Hash

A few hash function are included in libdynamic.

uint64_t **hash_data** (void **data*, size_t *size*)
Returns a 64-bit hash of *size* bytes of memory pointed to by *data*. The library uses a C port of Google Farmhash.

uint64_t **hash_string** (char **string*)
Returns a 64-bit hash of the null-terminated *string*.

2.3 Changes in libdynamic

2.3.1 Version 1.0

Released 2017-01-03

- Initial release

2.3.2 Version 1.1

Released 2017-12-17

- New features:
    - List type
    - More uniform interfaces

CHAPTER 3

Indices and Tables

- genindex
- search

Index

B

H

L

M

S

V