# Vitamin-C | C Language Libraries

Dorian Knight

Created: June 24 2025
Last updated: June 24 2025

## Contents

## 1 Project Overview

Vitamin-C is a set of importable C language libraries that implement data structures and algorithms using dynamic memory allocation. Vitamin-C handles data structure instantiation, manipulation and de-instantiation in the background simplifying workflows and allowing developers to focus on using their data structures instead of debugging them. [Click to view library on GitHub]

## 2 Purpose

Make C language programming more "Pythonic". Python data structure helper functions simplify code development by providing ready-made methods. In the case of arrays, Python provides append, insert, and reverse, to name a few. These functions greatly speed up development time as the developer does not have to painstakingly create their own methods to manipulate their data and instead can worry about using their data to solve complex problems.

## 3 Outcomes

As of v1.0.0, the project supports the following data structures:

- Arrays for integers, floats, doubles and chars

## 4 Skills Used

(a) C language icon

(b) GitHub icon

Figure 1: Skills used in project

The library was built in C for use in any project written in C ranging from embedded systems to operating systems. The project is version controlled on GitHub using semantic versioning [click to view releases].

## 5 Arrays

### 5.1 Quickstart

Use the constructor to create an array struct.

```
struct array* <array name> = array_constructor(<array length>, <array type>);
struct array* quickstartArray = array_constructor(5, CHAR);
```

Supported types include: INTEGER, FLOAT, DOUBLE, CHAR.

When finished with the array you must free the memory that was allocated to your array using the destructor.

```
array_destructor(<array name>);
array_destructor(quickstartArray);
```

Each type of array has specific methods to support data manipulation. Most methods follow the same naming convention: array_(desired action)_(data type) for example, array_get_int.

Integer array structure methods:

- Append: array_push_int(struct array* array, int value)
- Append front: array_push_int_front(struct array* array, int value)
- Get/fetch: array_get_int(struct array* array, int index)
- Put: array_put_int(struct array* array, int index, int value)
- Insert: array_insert_int(struct array* array, int index, int value)

Float array structure methods:

- Append: array_push_float(struct array* array, float value)
- Append front: array_push_float_front(struct array* array, float value)
- Get/fetch: array_get_float(struct array* array, int index)
- Put: array_put_float(struct array* array, int index, float value)
- Insert: array_insert_float(struct array* array, int index, float value)

Double array structure methods:

- Append: array_push_double(struct array* array, double value)
- Append front: array_push_double_front(struct array* array, double value)
- Get/fetch: array_get_double(struct array* array, int index)
- Put: array_put_double(struct array* array, int index, double value)
- Insert: array_insert_double(struct array* array, int index, double value)

Char array structure methods:

- Append: array_push_char(struct array* array, char value)
- Append front: array_push_char_front(struct array* array, char value)
- Get/fetch: array_get_char(struct array* array, int index)
- Put: array_put_char(struct array* array, int index, char value)
- Insert: array_insert_char(struct array* array, int index, char value)

Some methods are common between the different array types.

- Delete: array_delete(struct array* array, int index)
- Pop: array_pop(struct array* array)
- Pop front: array_pop_front(struct array* array)

## 5.2 Array structure

The array structure is a collection of the following attributes:

- list (void pointer)
- list size (int)
- element size (int)
- list type (enum ArrayType)

**List**
The list void pointer points to the first element in the contiguous block of memory that composes the array of data. This pointer must be cast to the appropriate data type before being dereferenced.

Using a void pointer for the array allows the same structure to be used for all supported data types. In C, a pointer's data type must match the value's data type pointed to by the pointer. Using a void pointer gets around the issue of matching the pointer's type to the value's type allowing the array structure to be used for all supported data types.

**List size**
Integer value storing the number of elements contained within the list. The list size variable is used when performing manipulations on the array where knowing where the array starts and ends is critical such as insert, append and pop.

Traditional array implementations require the developer to "cart around" a separate length variable but in Vitamin-C, the length variable is stored and updated automatically within the array structure.

**Element size**
Integer value storing the number of bytes required to store one element of the array. This is used in conjunction with the list size variable when initializing and resizing the array when pushing, popping, and inserting to name a few examples.

**List type**
Enum representing the type of data stored in the array. Possible enum values include:

- INTEGER
- FLOAT
- DOUBLE
- CHAR

This enum is useful when dereferencing the list void pointer inside the array get and put methods. Before performing pointer arithmetic or accessing the value stored at a memory address, the pointer must be cast to the appropriate data type. Having the list type enum provides that information to the library function allowing it to manipulate the data as instructed by the developer.

## 5.3 Instantiation and destructor functions

### 5.3.1 array_constructor

**Input**: int list size, enum ArrayType array type
**Output**: struct array* array
**Process**: Dynamically allocates memory for Array structure and element list according to array type provided in the function arguments. If the array type is invalid, the constructor throws an "ARRAY_TYPE_UNSPECIFIED" error.

### 5.3.2 array_destructor

**Input**: struct array*
**Output**: None
**Process**: Frees dynamically allocated array pointer

## 5.4 Utility functions

The utility functions are not supposed to be directly called by the user, rather, the proper wrapper function found in 5.5 should be called instead. All wrapper functions call specific utility functions to complete the desired operation.

### 5.4.1 arrayErrorHandler

**Input**: int signal
**Output**: None
**Process**: Triggered by "SIGINT". Checks error enum set in watchdog to print error into the terminal console then exits the program passing a signal interrupt.

### 5.4.2 array_push

**Input**: struct array* array void* valuePtr
**Output**: None
**Process**: Reallocates array to increase list size by one. Puts the specified value at the last index of the reallocated array. Raises "ARRAY_TYPE_UNKNOWN" error if array type is not supported.

### 5.4.3 array_push_front

**Input**: struct array* array, void* valuePtr
**Output**: None
**Process**: Reallocate array to increase element count by one. Shift all elements in the array right by one index then place value in the zeroth index.

### 5.4.4 array_get

**Input**: struct array* array, int index
**Output**: void*
**Process**: Checks if requested index is valid, if not, it returns an "ARRAY_OUT_OF_BOUNDS" error. If the requested index is valid, it moves the list pointer to the requested index, casts the list pointer as a void pointer and returns the void pointer to the wrapper function that called this utility function. If the array type is unknown it returns a "ARRAY_TYPE_UNKNOWN" error.

### 5.4.5 array_put

**Input**: struct array* array, int index, void* valuePtr
**Output**: None
**Process**: Checks if requested index is valid, if not, it returns an "ARRAY_OUT_OF_BOUNDS" error. If the index is valid, it moves the list pointer to the requested index and stores the provided value at the requested index. If the array type is unknown it returns a "ARRAY_TYPE_UNKNOWN" error.

### 5.4.6 array_delete

**Input**: struct array* array, int index
**Output**: None
**Process**: Checks if requested index is valid, if not, it returns an "ARRAY_OUT_OF_BOUNDS" error. If the index is valid the function shifts all elements to the right of the specified index one index to the left effectively overwriting the value at the specified index. Once complete, the list size is decremented and the list is resized to be one element smaller. If reallocation fails, a "FAILED_REALLOCATION" error is returned.

### 5.4.7 array_insert

**Input**: struct array* array, int index, void* valuePtr
**Output**: None
**Process**: Checks if requested index is valid, if not, it returns an "ARRAY_OUT_OF_BOUNDS" error. If the index is valid, the function reallocates increases the list size structure attribute and resizes the list to be one element larger. If reallocation fails, a "FAILED_REALLOCATION" error is returned. If reallocation is successful, the list then shifts all elements from the requested index onwards to the right by one element and then puts the desired value at the specified index.

### 5.4.8 array_pop

**Input**: struct array* array
**Output**: None
**Process**: Calls the delete function to remove the last element in the list

### 5.4.9 array_pop_front

**Input**: struct array* array
**Output**: None
**Process** Calls the delete function to remove the zeroth element in the list.

## 5.5 Wrapper Functions

All wrapper functions serve to abstract the value parameter passed by the developer so that a common utility function can be called. Type abstraction is accomplished by assigning the memory address of the value to a void pointer then passing the void pointer into the utility function.

**Integer**

### 5.5.1 array_push_int

**Input**: struct array* array, int value
**Output**: None
**Process**: Abstracts value data type and calls array_push utility function.

### 5.5.2 array_get_int

**Input**: struct array* array, int index
**Output**: int
**Process**: Calls array_get utility function which returns a void pointer. Casts the void pointer as an integer pointer then dereferences the integer pointer to get the value stored at the requested index. Returns the dereferenced integer.

### 5.5.3 array_put_int

**Input**: struct array* array, int index, int value
**Output**: None
**Process**: Abstracts the value data type then calls the array_put utility function.

### 5.5.4 array_push_int_front

**Input**: struct array* array, int value
**Output**: None
**Process**: Abstracts the value data type then calls the array_push_front utility function.

### 5.5.5 array_insert_int

**Input**: struct array*, int index, int value
**Output**: None
**Process**: Abstracts the value data type then calls the array_insert utility function.

**Float**

### 5.5.6 array_push_float

**Input**: struct array* array, float value
**Output**: None
**Process**: Abstracts value data type and calls array_push utility function.

### 5.5.7 array_get_float

**Input**: struct array* array, int index
**Output**: float
**Process**: Calls array_get utility function which returns a void pointer. Casts the void pointer as a float pointer then dereferences the float pointer to get the value stored at the requested index. Returns the dereferenced float.

### 5.5.8 array_put_float

**Input**: struct array* array, int index, float value
**Output**: None
**Process**: Abstracts the value data type then calls the array_put utility function.

### 5.5.9 array_push_float_front

**Input**: struct array* array, float value
**Output**: None
**Process**: Abstracts the value data type then calls the array_push_front utility function.

### 5.5.10 array_insert_float

**Input**: struct array*, int index, float value
**Output**: None
**Process**: Abstracts the value data type then calls the array_insert utility function.

**Double**

### 5.5.11 array_push_double

**Input**: struct array* array, double value
**Output**: None
**Process**: Abstracts value data type and calls array_push utility function.

### 5.5.12 array_get_double

**Input**: struct array* array, int index
**Output**: double
**Process**: Calls array_get utility function which returns a void pointer. Casts the void pointer as a double pointer then dereferences the double pointer to get the value stored at the requested index. Returns the dereferenced double.

### 5.5.13 array_put_double

**Input**: struct array* array, int index, double value
**Output**: None
**Process**: Abstracts the value data type then calls the array_put utility function.

### 5.5.14 array_push_double_front

**Input**: struct array* array, double value
**Output**: None
**Process**: Abstracts the value data type then calls the array_push_front utility function.

### 5.5.15 array_insert_double

**Input**: struct array*, int index, double value
**Output**: None
**Process**: Abstracts the value data type then calls the array_insert utility function.

**Char**

#### 5.5.16 array_push_char

**Input**: struct array* array, char value
**Output**: None
**Process**: Abstracts value data type and calls array_push utility function.

#### 5.5.17 array_get_char

**Input**: struct array* array, int index
**Output**: char
**Process**: Calls array_get utility function which returns a void pointer. Casts the void pointer as a char pointer then dereferences the char pointer to get the value stored at the requested index. Returns the dereferenced char.

#### 5.5.18 array_put_char

**Input**: struct array* array, int index, char value
**Output**: None
**Process**: Abstracts the value data type then calls the array_put utility function.

#### 5.5.19 array_push_char_front

**Input**: struct array* array, char value
**Output**: None
**Process**: Abstracts the value data type then calls the array_push_front utility function.

#### 5.5.20 array_insert_char

**Input**: struct array*, int index, char value
**Output**: None
**Process**: Abstracts the value data type then calls the array_insert utility function.

## 5.6 Validation & Verification

Verification and validation took place concurrently with library development. All unit tests are stored within the "Unit_tests" folder.

## 5.7 Error handling

While working with dynamically allocated arrays it is possible that the library runs into errors that it can't recover from. In that instance, the library raises a signal interrupt (SIGINT) which is handled by the array error handler. The possible critical errors that have been accounted for are the following:

- Index out of bounds
- Failed to allocate memory block for new array
- Failed to reallocate array to new memory block
- Desired operation requires knowledge of array type but array type is unknown
- Array constructor invoked but array type is unspecified

The array handler checks for the aforementioned errors within the watchdog to determine how to proceed. As of v1.0.0 the error handler prints out a custom error message and exits the program by invoking exit passing SIGINT.