Github: LoreDN
Author: Lorenzo Di Napoli
Repository: https://github.com/LoreDN/C

myArray — Documentation

Contents

Library Description

This library is a collection of some usefull functions to use when working with arrays.

The library is designed to simulate templates, despite pure **C** not having them; this increase the precautions to take when using this library, however it has been designed in order to be as user friendly as possible.

CONTENTS OF THE LIBRARY

The library consists of three types of files:

- **"myArray.h":** the template library, it is the header file where you can find a brief description of how to use the library in practice, together with all the functions prototypes.
- **"myArray_type.h":** this is the file wich has to be included via the `#include "myArray_type.h"` compiler directive, it contains all the prototypes fixed for the given type.
- **"myArray_type.c":** the core of the library, it contains all the functions implementations for the given type and is the file that has to be given to the compiler in order to be able to use the library.

USER FRIENDLY COMPILATION

Since including the correct file can be tedius, on the Github repo https://github.com/LoreDN/C, in the *'libraries/myArray'* subfolder, is already given a **Makefile**, wich automates the compilation.

Everything the user has to do is (eventually) set the wanted compiler, flags and project main file name into the correct variables; the default ones are:

- compiler –> `CC = gcc`
- compiling flags –> `CFLAGS = -Wall`
- project main file name –> `SRC_PROJECT = main.c`

The **Makefile** has been implemented with the scope to be easy to read and eventually modify.

FURTHER IMPLEMENTATIONS

In order to maximize the optimizations, where it is reasonable and usefull in execution time to do it, some functions have been implemented directly in **assembly**: in particular they have been implemented in **Assembly x86_64** and in **Assembly RISC-V64**.

All the compilation process has been hide to the user via the **Makefile**.

When compiling, no other files than the project object should be created; if the compiler detecs important errors and aborts the compilation, you may see the assembly functions object files, however when the compilation is successfull, all this files will be automatically deleted.

The assembly functions are stored in the folder *'assembly'*: when downloading the library, be sure to download the complete folder and not only the *'.h'* and *'.c'* files, otherwhise the assembly functions will not be found and the library can't be used.

If, for any reason, you don't want to use this implementations, but you want to write the functions directly in **C** (as well as other programming languages), it can be done without problems, except for the **Makefile**, which needs to be updated with the new files sources and compilation rules for the alternative implementations.

LIST OF ASSEMBLY IMPLEMENTATIONS

Here is a list of all the functions that have been implemented and optimized directly in **Assembly x86_64** and in **Assembly RISC-V64**:

- array_swap( ).
- array_is_sorted( ).
- array_reverse( ).
- array_max( ).
- array_min( ).
- array_sum( ).
- array_find( ).
- array_fill( ).
- array_copy( ).

IMPLEMENTED TYPES

The library has been implemented in two types:

- **"myArray_int":** work using arrays of integral numbers.
- **"myArray_float":** work using arrays of floating point numbers.

Is possible to use the one library at time or to use both simultaneously, it is sufficientemente to include the correct files when compiling (and add the right definition type for the assembly functions).

However the **Makefile** already sets the compilation, it is sufficient to use the following commands:

- **make main_type:** build the object file of the project.
- **make run_type:** build and run the object file of the project.
- **make clean_type:** remove the object file of the project.

If multiple types are used, be sure to add them all in the commands in the following order: `make command_int_float`.

Functions Documentation

The *"myArray"* library is divided into five categouris, each one containing a different type of functions:

1. **SCAN:** implementation of functions to scan an array from terminal or file.
2. **PRINT:** implementation of functions to print an array to terminal or file.
3. **SORTING:** implementation of the most usefull sorting algorithms.
4. **UTILITY:** implementation of casual use functions.
5. **STATISTIC:** implementation of usefull statistic and mathematical functions for arrays.

---

1. SCAN

```
void array_scan(TYPE *array, size_t const size);
```

Allows to scan an array from terminal (`TYPE *array` must be different from **NULL**).

```
void array_scan_file(size_t *DIM, char* const path);
```

Allocates an array reading its size (first word) from a file, and then scanning it.
It is given a `size_t *DIM` in order to return the size of the newly allocated array.

---

2. PRINT

```
void array_print(TYPE *array, size_t const size, int const flag_user_interface);
```

Prints the array to the terminal.
The `int const flag_user_interface` is used in order to choose the type of print:

- **flag = 1** —> `Element i :  array[ i ] "\n"`.
- **flag = other values** —> `array[ i ] "\t" array[ i + 1 ] ...`.

```
void array_print_file(TYPE *array, size_t const size, char const *path, int const flag_user_interface);
```

Prints the array to a file, the first line is used to print the size.
The `int const flag_user_interface` is used in order to choose the type of print:

- **flag = 1** —> `Size of the array:  size "\n" Element i :  array[ i ] "\n"`.
- **flag = other values** —> `size "\n" array[ i ] “\n”`.

3. SORTING

```
extern void array_swap(TYPE *a, TYPE *b);
```

Swaps two elements of the the array, used in various sorting algorithms.
This function has been implemented in Assembly **x86_64** / **RISC-V64**.

```
size_t array_partition(TYPE *array, size_t const start, size_t const end);
```

Partitions the given array in two parts and returns the pivot:

- all elements on the left side are <= **pivot**.
- all elements on the right side are >= **pivot**.

Used in Quick Sort, it follows the Lomuto scheme —> **starting pivot = end**.

```
void array_heapify(TYPE *array, size_t const start, size_t const size, size_t const root);
```

Heapify an array from a root, swapping it with the biggest element, used in Heap Sort.

```
void array_merge(TYPE *array, size_t const start, size_t const middle, size_t const end);
```

Merge two parts of the array (identified by `size_t middle`), sorting them during the merging process, used in Merge Sort and Block Sort.

```
void array_insertion_sort(TYPE *array, size_t const start, size_t const end);
```

Sorts the array comparing each element with the previous one, and if it is needed swaps them.

```
void array_selection_sort(TYPE *array, size_t const start, size_t const end);
```

Sorts the array comparing the elements and sorting the minimun one every iteration.

```
void array_bubble_sort(TYPE *array, size_t const start, size_t const end);
```

Sorts the array comparing the elements and sorting the maximum one every iteration.

```
void array_quick_sort(TYPE *array, size_t const start, size_t const end);
```

Sorts the array partitioning it recursively.
For definition, on average is very efficient, but it is not stable.

```
void array_heap_sort(TYPE *array, size_t const start, size_t const end);
```

Sorts the array heapifying it an then sorting the root at every iteration.

```
void array_merge_sort(TYPE *array, size_t const start, size_t const end);
```

Sorts the array by subdividing and then merging it back together.

```
void array_block_sort(TYPE *array, size_t const start, size_t const end);
```

Sorts the array by diving it in blocks, sorting them (using the previously documented Insertion Sort), and then merging them back together.

```
void array_counting_sort(TYPE *array, size_t const start, size_t const end, int const flag_stable);
```

Calculates the maximum and minimun values of the array, then builds an histogram based on the integral part of each element, finally rewrites the array following the histogram datas.
The `int const flag_stable` is set in order to choose the type of implementation:

- **flag = 1 —> stable implementation:** it uses more memory but preserves the order of the elements with the same values.
- **flag = other values** —> the array is totally rewritten, losing the original elements order (lighter and faster but not recommanded when is problematic to lose original order info).

This sorting algorithm works with all numeric types of array, but returns an array containing only the integral part of each elements, due to the nature of the histogram (when using floating point numbers it is impossible to do not lose the floating point informations).

```
extern int array_is_sorted(TYPE *array, size_t const start, size_t const end);
```

Checks if an array is sorted by comparing each element with the following one.
This function has been implemented in Assembly **x86_64** / **RISC-V64**.

```
extern void array_reverse(TYPE *array, size_t const start, size_t const end);
```

Reverses the array by swapping the elements by opposite pairs.
This function has been implemented in Assembly **x86_64** / **RISC-V64**.

4. UTILITY

```
extern int array_find(TYPE *array, size_t const start, size_t const end, TYPE const value);
```

Lineary search of the first occurrence of an element in the array.
This function has been implemented in Assembly **x86_64** / **RISC-V64**.

```
int array_binary_search(TYPE *array, size_t const start, size_t const end, TYPE const value);
```

Binary search of the first occurrence of an element in the array.
This method of search works only if the array is sorted.

```
extern void array_fill(TYPE *array, size_t const start, size_t const end, TYPE const value);
```

Sets the elements of the array to the given value.
This function has been implemented in Assembly **x86_64** / **RISC-V64**.

```
extern void array_copy(TYPE *array, size_t const start, size_t const end, TYPE *src, size_t const src_idx);
```

Copies the values from a source array to the given one.
This function has been implemented in Assembly **x86_64** / **RISC-V64**.

5. STATISTIC

```
extern TYPE array_max(TYPE *array, size_t const start, size_t const end);
```

Gets the maximum value from the array.
This function has been implemented in Assembly **x86_64** / **RISC-V64**.

```
extern TYPE array_min(TYPE *array, size_t const start, size_t const end);
```

Gets the minimum value from the array.
This function has been implemented in Assembly **x86_64** / **RISC-V64**.

```
extern TYPE array_sum(TYPE *array, size_t const start, size_t const end);
```

Gets the sum of the elements of the array.
This function has been implemented in Assembly **x86_64** / **RISC-V64**.

```
float array_average(TYPE *array, size_t const start, size_t const end);
```

Calculates the average of the elements of the array.

```
int *array_histogram(TYPE *array, size_t const start, size_t const end, size_t const min, size_t const max);
```

Builds the histogram of the array.
Due to how the histogram works, this function will always build the histogram approximating to the integral part of the values.