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

myList — Documentation

Contents

Library Description

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

Since the Nodes of a List are structs, there is not a single definition; however in the library has been defined a custom type *"myList_type"*, wich consists in a struct composed by a `TYPE` element and a `myList_type` pointer.

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:

- **"myList.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.
- **"myList_type.h":** this is the file wich has to be included via the `#include "myList_type.h"` compiler directive, it contains all the prototypes fixed for the given type.
- **"myList_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/code-C, in the *'libraries/myList'* 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**:

- list_swap( ).
- list_copy( ).
- list_length( ).

IMPLEMENTED TYPES

The library has been implemented in two types:

- **"myList_int":** work using `myList_int` type of Nodes.
- **"myList_float":** work using `myList_float` type of Nodes.

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 *"myList"* library is divided into four categouris, each one containing a different type of functions:

1. **SCAN:** implementation of functions to scan a List from terminal or file.
2. **PRINT:** implementation of functions to print a List to terminal or file.
3. **OPERATIONS:** implementation of usefull functions to use when working with Lists.
4. **UTILITY:** implementation of casual use functions.

---

1. SCAN

```
LIST *list_add_Node(TYPE const value);
```

Allocates a new Node setting the value and the next pointer (equal to **NULL**), then returns a pointer to the Node.

```
LIST *list_scan(size_t const size);
```

Allows to allocate a List of the given dimension, scanning the values from terminal, then returns a pointer to the List.

```
LIST *list_scan_file(char const *path);
```

Allocates a List reading the values from a file, then returns a pointer to the List.

---

2. PRINT

```
void list_print(LIST *list, int const flag_user_interface);
```

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

- **flag = 1** —> `Element i :  Node.value "\n"`.
- **flag = other values** —> `Node.value "\t" Node->next.value ...`.

```
void list_print_file(LIST *list, char const *path, int const flag_user_interface);
```

Prints the List to a file.
The `int const flag_user_interface` is used in order to choose the type of print:

- **flag = 1** —> `Element i:  Node.value "\n"`.
- **flag = other values** —> `Node.value “\n”`.

3. OPERATIONS

```
LIST *list_insert_Head(LIST *list, TYPE const value);
```

Allocates a new Node and inserts it as the Head of the List, then returns a pointer to the Node.

```
void list_insert_Node(LIST *node, TYPE const value);
```

Allocates a new Node and inserts it after the given Node.

```
LIST *list_delete_Head(LIST *list);
```

Deletes the Head of the List, then returns a pointer to the new Head (the next Node).

```
void list_delete_Node(LIST *list, LIST *node);
```

Deletes the given Node from the List.

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

Swaps two elements of the the List; this is done by swapping their values, not the real Nodes.
It is used in order to Sort and Reverse the List.
This function has been implemented in Assembly **x86_64** / **RISC-V64**.

```
void list_sort(LIST *list, size_t const start, size_t const end);
```

Sorts the List using the Bubble Sort Algorithm.

```
void list_reverse(LIST *list, size_t const start, size_t const end);
```

Reverses the List by swapping the elements by opposite pairs.

```
extern LIST *list_copy(LIST *src);
```

Allocates a new List, copying the values from a source List.
This function has been implemented in Assembly **x86_64** / **RISC-V64**.

```
void list_concatenate(LIST *list, LIST *src);
```

Concatenates a source List to the given one.

```
LIST *list_free(LIST *list);
```

Frees the List, then returns a pointer to **NULL**.

4. UTILITY

```
extern size_t list_length(LIST *list);
```

Returns the length of the List.
This function has been implemented in Assembly **x86_64** / **RISC-V64**.

```
LIST *list_find(LIST *list, TYPE const value);
```

Lineary search of the first occurrence of an element in the List.
Returns a pointer to the Node where the value is found.

```
LIST *list_Node(LIST *list, size_t const node);
```

Returns a pointer to the Node of the List in the given position (starts counting from one).

```
void list_fill(LIST *list, size_t const start, size_t const end, TYPE const value);
```

Sets the elements of the List to the given value.