C-Library API

Ali Khudiyev

Contents

1 Introduction

C-Library is an API made for C-programmers in order to be able to work with C++ like containers such as vectors, strings and so on. There are four available container types:

1. Vector - dynamic array of a single type
2. String - dynamic array of characters
3. Set - dynamic array of a single type
4. Tuple - dynamic array of multiple types

Each container has its own constructor(s), destructor(s) and "member" functions that are related to that container and some additional related macros to increase the usability. You will find more details in this documentation as well as some coding examples in my github repositoty.

This is a static library which is located in **bin** directory. You can create your projects with no fear of dynamic allocations! You can also implement your ideas more easily by using the **C-Library**; this could be a *csv-parser* or a new *database* project.

1.1 Why 4 of them?

The hardest(implementation-wise) among these four containers is the **vector**. After having a solid vector container, it is relatively easier to implement other three containers. However, after having these four containers, it is usually a lot easier to implement anything you want. That's why I have chosen to implement these four containers for the first release.

Note!
The library has been passed **unit tests** successfully and it has also been tested with **valgrind** to find memory leaks. As the final results, tests have not detected any memory leaks related to the library!
BUT! Feel free to create an issue on github if there is a bug.

2 API Specification

2.1 Function conventions

All function names begin with **"C"+the first letter of container**.
For example,

- **CVector(...)** - constructor for a vector
- **CV_size(...)** - **size()** function for a vector
- **CV_capacity(...)** - **capacity()** function for a vector

2.2 Macro conventions

All macro names begin with **"MC"+the first letter of container**.
For example,

- **MCV_force_push_back(...)** - **push back()** function for a vector
- **MCV_force_insert(...)** - **insert()** function for a vector
- **MCV_emplace_back(...)** - **emplace back()** function for a vector

2.3 Arguments

2.3.1 Function arguments

Arguments are usually required by their memory addresses(pointers).

```
CX_deep_copy(cxxx* container1, const cxxx* container2);
```

Arguments are required by value if they have trailing "**_value**"(i.e. **cxxx container_value**).

2.3.2 Macro arguments

Macro arguments are the same as the function arguments. However, there are some macros that expects variadic arguments. These macros are seperated into two main groups:

1. Macros which take values for container attributes
2. Macros that take executable C statements

Examples for the first group of macros are listed below:

```
MCV_emplace(vec, position, _type, ...)
MCV_emplace_back(vec, _type, ...)
MCV_force_emplace(vec, position, _type, _constructor, ...)
MCV_force_emplace_back(vec, position, _type, _constructor, ...)
```

Examples for the second group of macros are listed below:

```
MCV_For_Each(vec, iterator, ...)
MCV_Enumerate(vec, counter, iterator, ...)
```

2.4 Type definitions

Type	Definition	Return type	Parameters
f_copy_t	Copy constructor pointer	void	void*, const void*, size_t
f_move_t	Move constructor pointer	void	void*, const void*, size_t
f_destructor_t	Destructor pointer	void	void*
f_printer_t	Destructor pointer	const char*	void*
bool_t	unsigned char	-	-

Table 1: Caption

2.5 Constructors

There are 2 constructors for each container:

- **CXxx(...)** - constructor for a heap allocated container object(i.e. **CVector(...)**)
- **CX_init(cxxx* container, ...)** - constructor for a stack allocated container object(i.e. **CV_init(cvector* vec, ...)**)

```
cxxx* CXxx(...);
void CX_init(cxxx* container, ...);
```

2.5.1 Copy Constructors and Copying

When you want to copy a container context to another container there are 2 options:

- **copy(cxxx* container1, const cxxx* container2)** method - standard way of copying
- **deep_copy(cxxx* container1, const cxxx* container2)** method - with the custom copy constructor

Best Practice!
Do not forget that when you copy containers, the destination container's context is destructed. So, always set the destructor before copying!

Standard copy. Standard copy is available for all containers, however, this method of copying stuff might not be the best when you are dealing with pointers. Since it simply copies by value, there might appear a couple of pointers pointing to the same memory address. Consider the following example:

```
typedef struct{
    int a, *b;
}My_Class;

void MyClass_destruct(void* ptr){
    free(((My_Class*)ptr)->b);
}
```

```
int main(){

    cvector vec1, vec2;

    // Constructing stack allocated objects
    CV_init(&vec1, sizeof(My_Class), 1);
    CV_init(&vec2, sizeof(My_Class), 1);

    // Setting destructors
    CV_set_destructor(&vec1, MyClass_destruct);
    CV_set_destructor(&vec2, MyClass_destruct);

    My_Class obj1, obj2;

    obj1.a = 0;
    obj1.b = malloc(sizeof(int)); *obj1.b = 1;

    obj2.a = 2;
    obj2.b = malloc(sizeof(int)); *obj2.b = 3;

    CV_push_back(&vec1, &obj1);
    CV_push_back(&vec2, &obj2);

    // The value a in vec1 is obj1.a,
    //    The pointer b in vec1 is obj1.b.
    // The value a in vec2 is obj2.a,
    //    The pointer b in vec2 is obj2.b.

    CV_copy(&vec1, &vec2);

    // The value a in vec1 is obj2.a,
    //    The pointer b in vec1 is obj2.b.
    // The value a in vec2 is obj2.a,
    //    The pointer b in vec2 is obj2.b.

    CV_destruct(&vec1); // Ok, freed obj2.b
    CV_destruct(&vec2); // Error, cannot free obj2.b again

    // Segmentation fault due to trying to free
    //    the pointer obj2.b twice!

    return 0;
}
```

Copy constructor. Copy constructors are to copy class instances in a user-defined manner. By default, a copy constructor of any container object is set to **NULL**, which means, you cannot use **deep_copy** method until the copy constructor is set.

The copy constructor has to follow the following convention:

```
void copy(void* dest, const void* src, size_t n_byte);
```

After creating custom copy constructor you have to set it with the following function:

```
void set_deep_copy(cxxx* container,
                   void (*copy)(void*, const void*, size_t));
```

Best Practice!
It is a good practice to set your custom copy constructors if your class instance contains pointer(s).

```
typedef struct{
   int a, *b;
}My_Class;

void MyClass_destruct(void* ptr){
   free(((My_Class*)ptr)->b);
}

void MyClass_copy(void* dest, void* src, size_t n_byte){
    for(size_t i=0; i<n_byte/sizeof(My_Class); ++i){
        memcpy(&((My_Class*)dest)[i].a,
               &((My_Class*)src)[i].a,
               sizeof(int));

        int* ptr = (int*)malloc(sizeof(int));
        memcpy(ptr, ((My_Class*)src)[i].b, sizeof(int));
        memcpy(&((My_Class*)dest)[i].b, &ptr, sizeof(int*));
    }
}

int main(){

   cvector vec1, vec2;

   // Constructing stack allocated objects
   CV_init(&vec1, sizeof(My_Class), 1);
   CV_init(&vec2, sizeof(My_Class), 1);

   // Setting copy constructors
   CV_set_deep_copy(&vec1, MyClass_copy);
   CV_set_deep_copy(&vec2, MyClass_copy);

   // Setting destructors
   CV_set_destructor(&vec1, MyClass_destruct);
   CV_set_destructor(&vec2, MyClass_destruct);

   My_Class obj1, obj2;

   obj1.a = 0;
   obj1.b = malloc(sizeof(int)); *obj1.b = 1;

   obj2.a = 2;
```

```
    obj2.b = malloc(sizeof(int)); *obj2.b = 3;

    CV_push_back(&vec1, &obj1);
    CV_push_back(&vec2, &obj2);

    // The value a in vec1 is obj1.a,
    //    The pointer b in vec1 is obj1.b.
    // The value a in vec2 is obj2.a,
    //    The pointer b in vec2 is obj2.b.

    CV_deep_copy(&vec1, &vec2);
    // While using deep_copy it is enough to
    //    set copy constructor of the destination object

    // The value a in vec1 is obj2.a,
    //    The pointer b in vec1 is a new memory address(ptr)
    //       which points to *obj2.b.
    // The value a in vec2 is obj2.a,
    //    The pointer b in vec2 is obj2.b.

    CV_destruct(&vec1); // Ok, freed allocated memory address(ptr)
    CV_destruct(&vec2); // Ok, freed obj2.b

    return 0;
}
```

2.5.2 Move constructors

No move constructors are available for now!

2.6 Destructors

There can be several types of destructors depending on a container, however, there are always two main destructors for each one of them:

- **delete(cxxx* container)** method (i.e. **CV_delete(cvector* vec)**)
- **destruct(void* container)** method (i.e. **CV_destruct(void* vec)**)

Be careful!
Destructors do not delete(free) heap-allocated container object pointer. You have to always free such pointer by yourself!

delete() destructor. This is a simple destructor and its job is to free user data in a container without bothering user-allocated pointers. This is the main reason that it not recommended to use this destructor if you have your own custom allocations.

Be careful!
If you have a library-independent heap-allocated variable then this function will not free the allocated memory of that variable.

```c
typedef struct{
   int a, *b;
}My_Class;

int main(){

   cvector vec;

   // Constructing stack allocated object
   CV_init(&vec, sizeof(My_Class), 1);

   My_Class obj;

   obj.a = 0;
   obj.b = malloc(sizeof(int)); *obj.b = 1;

   CV_push_back(&vec, &obj1);

   CV_delete(&vec);
   // Memory leak: 4 bytes are lost

   free(obj.b);
   // After manual free, no memory leak!

   return 0;
}
```

User-defined destructors and destruct(). You can store your own data types or structs in provided containers easily. However, when you want to delete them from memory, it can be a bit harsh to do it all by yourself. This is why there is a concept of user-defined destructors. You should create your own destructor function for your each data type(i.e. your custom struct) and then set it with the library function.

The destructor has to follow the following convention:

```c
void destructor(void* container);
```

After creating custom destructor you have to set it with the following function:

```c
void set_destructor(cxxx* container,
                    void (*destructor)(void*));
```

The **destruct** destructor calls user's custom destructor function if set.

Best Practice!
IT IS USUALLY BEST PRACTICE TO SET YOUR CUSTOM DESTRUCTOR, AND ESPECIALLY, IF YOU HAVE YOUR OWN LIBRARY INDEPENDENT HEAP-ALLOCATED VARIABLES.

```c
typedef struct{
   int a, *b;
}My_Class;

void MyClass_destruct(void* ptr){
```

```
    free(((My_Class*)ptr)->b);
}

int main(){

    cvector vec;

    // Constructing stack allocated object
    CV_init(&vec, sizeof(My_Class), 1);

    // Setting the custom destructor
    CV_set_destructor(&vec, MyClass_destruct);

    My_Class obj;

    obj.a = 0;
    obj.b = malloc(sizeof(int)); *obj.b = 1;

    CV_push_back(&vec, &obj1);

    CV_destruct(&vec);
    // No memory leak!

    return 0;
}
```

2.7 Member functions and macros

There are several functions and macros implemented to be able to work with the containers easily. Almost all of the standard C++ container library functions have been implemented from here.

2.7.1 Functions

Visit here for more details!

2.7.2 Macros

There are several macros to help the programmer to get through some tough situations.

```
MCX_For_Each(cxxx, iterator, ...)
MCX_Enumerate(cxxx, position, iterator, ...)
MCX_str(cxxx, position)
```

For Each macro. Given the **container pointer**(cxxx) and the **iterator**(pointer of valid type) this macro iterates through the container elements. Current element in the container can be accessed by the **iterator** pointer.

Enumerate macro. Given the **container pointer**(cxxx), **counter** and the **iterator**(pointer of valid type) this macro iterates through the container elements. Current element in the container can be accessed by the **iterator** pointer and its index is stored in the provided **counter** variable.

str macro and printers. When you have complicated data types(i.e. vector of custom struct), you might want to be able to print the elements of your type easily. It means, you might not want to deal with the process of printing each element every time by accessing it in the container and the dereference it(if needed) or maybe specify its printable section on your own. That's why there is a concept of user-defined printers in the library.

You can define your own printer function by respecting the following convention:

```
const char* (*printer)(void* data);
```

Then you can set it by using the following function:

```
void CX_set_printer(cxxx* container,
                    const char* (*printer)(void*));
```

Finally, you can use the following macro to get the printable "string" of your data type:

```
MCX_str(cxxx, position)
```

This macro returns **const char***, so, you can use it inside the **printf()** function as follows:

```
printf("%s", MCX_str(containter, 0));
```

Best Practice!
It is better to use library **printer buffer** rather than your own one. To use it you can use **CPrinter_get_buffer()** and **CPrinter_set_buffer(const char* buffer, size_t size).**

Remember!
You do not need to delete **printer buffer** explicitly by using **CPrinter_delete()** function unless you have not set your custom printer to a container. If set, containers tend to delete that buffer by themselves.

3 Usage and examples

In this section, you will go through the main steps which have to be followed while using the API library. The correct usage of a "C container" has to follow the steps below:

1. Construct the container
2. Do operations
3. Destruct the container

Container construction. This step requires the container to be initialized properly. You can always check the header file and try to initialize your container object on your own, however, custom initialization may lead to bugs or crashes. So, it is not recommend action to take and if you want to do it then do it on your own risk!

Be Careful!
EACH CONTAINER OBJECT HAS TO BE CONSTRUCTED BY EITHER STACK-CONSTRUCTOR OR HEAP-CONSTRUCTOR.

Operations. There are several "member functions" for each container.

Container destruction. The last step when you do not need a container anymore is to delete/destruct it. You can use several destructors including your own custom ones.

If it is a heap allocated container you are trying to delete you also have to delete its pointer on the heap manually by calling **free(void* ptr)** function of C.

Be Careful!
IF YOU HAVE A HEAP-ALLOCATED CONTAINER OBJECT THEN YOU HAVE TO FREE ITS POINTER ADDRESS MANUALLY WITH THE STANDARD **free(void*** FUNCTION.

3.1 Examples

```
#include <cpplib.h>
#include <stdio.h>

typedef struct{
    int ID;
    cstring name;
}Person;

void Person_destruct(void* ptr){
    CS_destruct(&((Person*)ptr)->name);
}

const char* Person_printer(void* data){
    Person* person = (Person*)data;
    char buffer[100];

    sprintf(buffer, "%d: %s",
```

```
            person->ID, CS_c_str(&person->name));
    CPrinter_set_buffer(buffer, 100);

    return CPrinter_get_buffer();
}

int main(){
    cvector* vec = CVector(sizeof(Person), 1);
    CV_set_destructor(vec, Person_destruct);
    CV_set_printer(vec, Person_printer);

    for(int i=0; i<5; ++i){
        char buffer[100];
        scanf("%s", buffer);

        Person person;
        person.ID = i;
        CS_init(&person.name, 1);
        CS_append(&person.name, buffer, 100);

        CV_push_back(vec, &person);
    }

    size_t i;
    Person* iterator;
    MCV_Enumerate(vec, i, iterator,
        printf("Iteration %zu | %s\n", i, MCV_str(vec, i));
    );

    CV_destruct(vec);
    free(vec);

    return 0;
}
```

See more **examples** from here.