filename: c__attribute__constructor-destructor-multif-20251210.txt
https://www.tutorialspoint.com/attribute-constructor-and-attribute-destructor-syntaxes-in-c-in-tutorials-point

attribute((constructor)) and attribute((destructor)) syntaxes in C in tutorials point ?

Here we will see how to write a code where two functions are present, and one function will be executed before the main function, and another function will be executed after main function. These features are used to do some startup task before executing main, and some clean up task after executing main.

To do this task we have to put attribute for these two functions. When the attribute is constructor attribute, then it will be executed before main(), and when the attribute is destructor type, then it will be executed after main().

We are using GCC functions. The function is __attribute__(). In this case we are using two different options. The Constructor and the Destructor with the __attribute__() function. The syntax __attribute__((constructor)) is used to execute a function when the program starts. and the syntax __attribute__((destructor)) is used to execute the function when main() function is completed. Please go through the example to get better idea.

Example

```
#include <stdio.h>

void before_main() __attribute__((constructor));
void after_main() __attribute__((destructor));

void before_main() {
    printf("This is executed before main.");
}

void after_main() {
    printf("This is executed after main.");
}

main() {
    printf("Inside main");
}
```

Output

```
This is executed before main.
Inside main
This is executed after main.
```

---
https://www.geeksforgeeks.org/c/__attribute__constructor-__attribute__destructor-syntaxes-c/

__attribute__((constructor)) and __attribute__((destructor)) syntaxes in C
02 Jun, 2017

Write two functions in C using GCC compiler, one of which executes before main function and other executes after the main function. GCC specific syntaxes : 1. __attribute__((constructor)) syntax : This particular GCC syntax, when used with a function, executes the same function at the startup of the program, i.e before main() function. 2. __attribute__((destructor)) syntax : This particular GCC syntax, when used with a function, executes the same function just before the program terminates through _exit, i.e after main() function. Explanation : The way constructors and destructors work is that the shared object file contains special sections (.ctors and .dtors on ELF) which contain references to the functions marked with the constructor and destructor attributes, respectively. When the library is loaded/unloaded, the dynamic loader program checks whether such sections exist, and if so, calls the functions referenced therein. Few points regarding these are worth noting : 1. __attribute__((constructor)) runs when a shared library is loaded, typically during program startup. 2. __attribute__((destructor)) runs when the shared library is unloaded, typically at program exit. 3. The two parentheses are presumably to distinguish them from function calls. 4. __attribute__ is a GCC specific syntax;not a function or a macro. Driver code:

```
// C program to demonstrate working of
// __attribute__((constructor)) and
// __attribute__((destructor))
#include<stdio.h>

// Assigning functions to be executed before and
// after main()
void __attribute__((constructor)) calledFirst();
void __attribute__((destructor)) calledLast();

void main() {
    printf("\nI am in main");
}

// This function is assigned to execute before
// main using __attribute__((constructor))
void calledFirst() {
    printf("\nI am called first");
}
```

```
// This function is assigned to execute after
// main using __attribute__((destructor))
void calledLast() {
    printf("\nI am called last");
}

   Output:
I am called first
I am in main
I am called last


---
https://www.codalogic.com/blog/2022/10/07/GCCs-attribute%28%28constructor%29%29

GCC's __attribute__((constructor))
October 2022

   While digging through the workings of QEMU I discovered GCC's __attribute__((constructor)) and
   __attribute__((destructor)). These tell GCC to run the associated function either before entering
   main() or after exiting main().

   They work in both C and C++ so I have written the example below in a C dialect.

   Hopefully the comments in the example explain what is happening well enough.
#include <stdio.h>

// Assigning functions to be executed before and
// after main()

// This function is specified to execute before
// main() using __attribute__((constructor))
void __attribute__((constructor)) before_main() {
    printf( "%s called before main()\n", __PRETTY_FUNCTION__);
}

// Functions can also be specified to execute before or after
// main() in their prototype
void __attribute__((destructor)) after_main();

// Here is main()!
int main() {
    printf( "I am in %s\n", __PRETTY_FUNCTION__);
}

// This function was specified to execute after main()
// using __attribute__((destructor)) in its earlier prototype
void after_main() {
    printf( "%s called after main()\n", __PRETTY_FUNCTION__);
}

// If a prototype specifies __attribute__((constructor)) or
// __attribute__((destructor)) it is optional whether the
// definition also specifies the attribute
void __attribute__((destructor)) another_after_main();
void __attribute__((destructor)) another_after_main() {
    printf( "%s called after main()\n", __PRETTY_FUNCTION__);
}

// BOth attributes can be specified on the same function
void __attribute__((constructor)) __attribute__((destructor))
another_before_and_after_main() {
    printf( "%s called before and after main()\n", __PRETTY_FUNCTION__);
}

// It's possible to specify __attribute__((constructor)) in the
// prototype and __attribute__((destructor)) in the definition
// (or vice versa)
void __attribute__((constructor)) before_and_after_main();
void __attribute__((destructor)) before_and_after_main() {
    printf( "%s called before and after main()\n", __PRETTY_FUNCTION__);
}

   Here is the output:
void before_main() called before main()
void another_before_and_after_main() called before and after main()
void before_and_after_main() called before and after main()
I am in int main()
void before_and_after_main() called before and after main()
void another_before_and_after_main() called before and after main()
void another_after_main() called after main()
void after_main() called after main()

   You can play with the example at: https://godbolt.org/z/cxvTnsz17
```

---
https://www.apriorit.com/dev-blog/537-using-constructor-attribute-with-ld-preload

Using the GCC Attribute Constructor with LD_PRELOAD
December 11, 2025

Linux has a wide variety of tools that allow you to fully control what's happening. One of them is LD_PRELOAD, which is an environmental variable that allows you to load any library of your choice before anything else. There are a number of LD_PRELOAD tricks that you can use to control and modify software within your environment.

At Apriorit, we specialize in cybersecurity and virtualization and often use hooks for monitoring and system management. We had one case, where we tried to install hooks using the method with the constructor attribute. But when adding hooks for the read method, we encountered a problem where the read method was called earlier than the method with the constructor attribute. As a result, the hooks weren't installed and our application crashed.

This LD_PRELOAD example was born from our search for a solution to this problem. When searching for the solution, we conducted detailed research of the constructor attribute and how to use it. Below you will find our results.

What is the constructor attribute?

The GCC website provides a detailed description of the constructor attribute. The gist is that the constructor attribute works similarly to the destructor attribute, only they do opposite things. The constructor makes it so that a function is called automatically while the execution enters main(). The destructor makes it so that a function is called when exit() is called or when main() has finished. Both of these functions are useful for initializing data that will be used by your program.

To control the order in which constructors and destructors run, you need to provide an integer to define the priority. A destructor with a higher priority number will run before a destructor with a lower number. The opposite is true for constructors - a constructor with a lower number will run earlier.

If you need both a constructor and destructor to handle the same resource, you would usually assign them the same priority. The properties of destructors and constructors are similar to those specified for namespace-scope C++ objects.

How the constructor attribute works

The constructor attribute guarantees that all methods with this attribute will be called before main() but does not guarantee that the method with the attribute will be called before other methods.

Here's a short example that illustrates the behavior of the constructor attribute:

```
ssize_t read(int fd, void *buf, size_t len) {
    printf("read was called\n");
    if (!orig_read) {
        printf("orig_read was not initialized\n");
        return -1;
    }

    return orig_read(fd, buf, len);
}

static __attribute__((constructor))) void init_method2(void) {
    printf("init_method2 was called\n");
    char sym;
    read(0, &sym, sizeof(sym));
}

static __attribute__((constructor)) void init_method(void) {
    printf("init_method was called\n");
    orig_read = dlsym(RTLD_NEXT, "read");
    printf("read was initialized\n");
}
```

Here's the result that you get after launching the application linked with the library from the example above:

```
init_method2 was called
read was called
orig_read was not initialized
init_method was called
read was initialized
```

Setting constructor priorities

In this case, all constructors are called sequentially. When init_method2 is called by the read method (and since the init_method constructor hasn't been called yet), orig_read is not initialized, and thus you'll get this message:

```
orig_read was not initialized
```

In this situation, the problem of launch order can be solved by setting constructor priorities:

```
static __attribute__((constructor (200))) void init_method2(void) {
    printf("init_method2 was called\n");
```

```
    char sym;
    read(0, &sym, sizeof(sym));
}

static __attribute__((constructor (150))) void init_method(void) {
    printf("init_method was called\n");
    orig_read = dlsym(RTLD_NEXT, "read");
    printf("read was initialized\n");
}
```

First, the constructor with the lowest priority number will be called. In this case, it's init_method. You can use numbers higher than 100 to set priorities. Constructor priorities from 0 to 100 are reserved for the implementation.

Constructor priorities within several interacting libraries

The example described above is far removed from real cases that we encounter in practice, since we can clearly see all dependencies. Let's take a look at a more realistic case where several libraries are interacting. For this, we'll leave only one method with the constructor attribute in the first library.

```
static __attribute__((constructor)) void init_method(void) {
    printf("init_method was called\n");
    orig_read = dlsym(RTLD_NEXT, "read");
    printf("read was initialized\n");
}
```

We'll also add a hook for write to this library that will use the test_func method from another library.

```
ssize_t write(int fd, const void *buf, size_t len) {
    printf("write was called\n");
    test_func();
    if (!orig_write) {
        orig_write = dlsym(RTLD_NEXT, "read");
    }

    return orig_write(fd, buf, len);
}
```

Here's the code of the library that defines test_func:

```
void read_first_byte(int fd) {
    printf("read_first_byte was called\n");
    const size_t size = 1;
    char buf[size];
    int res = read(fd, buf, size);
    if (res < -1) {
        printf("Failed to read from file\n");
        return;
    }
}

void test_func() {
    printf("test_func was called\n");
}

static __attribute__((constructor)) void init_test_lib(void) {
    printf("init_test_lib was called\n");
    read_first_byte(0);
}
```

In this library there's a method with the constructor attribute, with the init_test_lib inside the read_first_byte method using the read call. When running the test app linked to these libraries, you'll get the following result:

```
init_test_lib was called
read_first_byte was called
read was called
orig_read was not initialized
init_method was called
read was initialized
```

In order to fully understand what's going on, you can use LD_DEBUG=all and check what the loader does:

```
23486:
23486: calling init: /home/user/constructor/build-test_lib/libtest_lib.so
23486:
23486: symbol=puts;  lookup in file=./constructor_test [0]
23486: symbol=puts;  lookup in file=/home/user/constructor/bin/Debug/libconstructor.so [0]
23486: symbol=puts;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
23486: binding file /home/user/constructor/build-test_lib/libtest_lib.so [0] to /lib/x86_64-linux-gnu/libc.so.6
 [0]: normal symbol `puts' [GLIBC_2.2.5]
23486: symbol=_dl_find_dso_for_object;  lookup in file=./constructor_test [0]
23486: symbol=_dl_find_dso_for_object;  lookup in file=/home/user/constructor/bin/Debug/libconstructor.so [0]
23486: symbol=_dl_find_dso_for_object;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
```

```
23486: symbol=_dl_find_dso_for_object;  lookup in file=/lib/x86_64-linux-gnu/libdl.so.2 [0]
23486: symbol=_dl_find_dso_for_object;  lookup in file=/home/user/constructor/build-test_lib/libtest_lib.so [0]
23486: symbol=_dl_find_dso_for_object;  lookup in file=/lib64/ld-linux-x86-64.so.2 [0]
23486: binding file /lib/x86_64-linux-gnu/libc.so.6 [0] to /lib64/ld-linux-x86-64.so.2 [0]: normal symbol `_dl_
find_dso_for_object' [GLIBC_PRIVATE]init_test_lib was called
23486: symbol=read_first_byte;  lookup in file=./constructor_test [0]
23486: symbol=read_first_byte;  lookup in file=/home/user/constructor/bin/Debug/libconstructor.so [0]
23486: symbol=read_first_byte;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
23486: symbol=read_first_byte;  lookup in file=/lib/x86_64-linux-gnu/libdl.so.2 [0]
23486: symbol=read_first_byte;  lookup in file=/home/user/constructor/build-test_lib/libtest_lib.so [0]
23486: binding file /home/user/constructor/build-test_lib/libtest_lib.so [0] to /home/user/constructor/build-te
st_lib/libtest_lib.so [0]: normal symbol `read_first_byte'read_first_byte was called
23486: symbol=read;  lookup in file=./constructor_test [0]
23486: symbol=read;  lookup in file=/home/user/constructor/bin/Debug/libconstructor.so [0]
23486: binding file /home/user/constructor/build-test_lib/libtest_lib.so [0] to /home/user/constructor/bin/Debu
g/libconstructor.so [0]: normal symbol `read' [GLIBC_2.2.5]
23486: symbol=puts;  lookup in file=./constructor_test [0]
23486: symbol=puts;  lookup in file=/home/user/constructor/bin/Debug/libconstructor.so [0]
23486: symbol=puts;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
23486: binding file /home/user/constructor/bin/Debug/libconstructor.so [0] to /lib/x86_64-linux-gnu/libc.so.6 [
0]: normal symbol `puts' [GLIBC_2.2.5]read was called orig_read was not initialized
23486:
23486: calling init: /lib/x86_64-linux-gnu/libdl.so.2
23486:
23486:
23486: calling init: /home/user/constructor/bin/Debug/libconstructor.so
23486: init_method was called
23486: symbol=dlsym;  lookup in file=./constructor_test [0]
23486: symbol=dlsym;  lookup in file=/home/user/constructor/bin/Debug/libconstructor.so [0]
23486: symbol=dlsym;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
23486: symbol=dlsym;  lookup in file=/lib/x86_64-linux-gnu/libdl.so.2 [0]
23486: binding file /home/user/constructor/bin/Debug/libconstructor.so [0] to /lib/x86_64-linux-gnu/libdl.so.2
[0]: normal symbol `dlsym' [GLIBC_2.2.5]
23486: symbol=_dl_sym;  lookup in file=./constructor_test [0]
23486: symbol=_dl_sym;  lookup in file=/home/user/constructor/bin/Debug/libconstructor.so [0]
23486: symbol=_dl_sym;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
23486: binding file /lib/x86_64-linux-gnu/libdl.so.2 [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol
`_dl_sym' [GLIBC_PRIVATE]
23486: symbol=read;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
23486: binding file /home/user/constructor/bin/Debug/libconstructor.so [0] to /lib/x86_64-linux-gnu/libc.so.6 [
0]: normal symbol `read' read was initialized
23486: symbol=__libc_start_main;  lookup in file=./constructor_test [0]
23486: symbol=__libc_start_main;  lookup in file=/home/user/constructor/bin/Debug/libconstructor.so [0]
23486: symbol=__libc_start_main;  lookup in file=/lib/x86_64-linux-gnu/libc.so.6 [0]
23486: binding file ./constructor_test [0] to /lib/x86_64-linux-gnu/libc.so.6 [0]: normal symbol `__libc_start_
main' [GLIBC_2.2.5]
23486:
23486: initialize program: ./constructor_test
23486:
```

---

https://mcuoneclipse.com/2023/10/09/global-constructors-and-destructors-with-c-not-c/

Global Constructors and Destructors with C (not C++)
October 9, 2023

In the OOP world, global objects get initialized with a constructor and destroyed at the end with a destructor. Interestingly, the GNU gcc has attributes to mark functions as constructor and destructors, which can greatly simply system startup and shutdown, even if not using C++.
C Function marked as Constructor and called before main()

With the GNU gcc compiler, I can mark functions with an attribute, so they get called before entering main() or after exit of main(). The attribute works both in C and C++, but it especially useful in C to initialize modules in an automated way.

Global Constructors and Destructors

In C++, when having global objects or variables, then they need to be constructed automatically at program startup, and destructed at program shutdown.

Consider the following example:

```
class Car {
    private: int price;
    public:
        Car(void) { // constructor
            price = 1000; // initial base price for every car
        }
        ~Car(void) { // destructor
            price = -1;
        }
        void SetPrice(int price) { // setter method
            this->price = price;
        }
};
```

```
// global variables
static Car c; // global object, shall be initialized by startup (constructor)
static int i; // initialized by startup (zero-out)
static int j = 0x1234; // initialized by startup (copy-down)
```

Here the program startup will call the constructor of Car c and initialize it that way, in a similar way as the startup zero-out and startup copy-down initializes variables in C. The difference is that zero-out and copy-down simply initializes the value/memory, where the constructor and destructor are function calls. All this will be handled by the startup code, and the initialization happens before calling main().

The idea is to use that concept of calling function calls before main() for normal C modules, to have them initialized and de-initialized, so I don't have to do this manually in my code.

GCC Constructor and Destructor Attributes

The solution is to take advantage of special GNU gcc compiler attributes.

From the gcc user manual:

> constructor
> destructor
> The constructor attribute causes the function to be called automatically before execution enters main (). Similarly, the destructor attribute causes the function to be called automatically after main () has completed or exit () has been called. Functions with these attributes are useful for initializing data that will be used implicitly during the execution of the program.
> https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Function-Attributes.html

With this, I can mark my initialization and de-initialization routines with the attributes, for example:

```
__attribute((destructor)) void STEPPER_Deinit(void) {
    ...
}

__attribute((constructor)) void STEPPER_Init(void) {
    ...
}
```

I usually have Deinit() functions in my driver, in case I need to shut-down a peripheral and initialize it. As embedded systems usually 'never end', the Deinit() and as such the destructor might not be needed and used.

It is possible to mark a function both as constructor and destructor. It can be mixed with the prototype/declaration and definition, and vice versa:

```
__attribute((constructor)) void CalledAsConstructerAndDestructor(void)
__attribute((destructor)) void CalledAsConstructerAndDestructor(void) {
    /* called both in constructing and destructing phase */
}
```

Order of Calls

In many cases, you might want to have control over the order of calls. It is possible to give the destructor and constructor a 'priority', with lower numbers the higher urgency, for example:

```
__attribute__ ((constructor(101))) void myInit(void) {
    ...
}
```

Values from 0 to 100 are reserved and used for example by the gcov. So any number from 101- 65535 can be used. If no value/argument is provided, then they get a lower priority than the ones with the number.

But VS Code Intellisense will flag the arguments as an error which can be ignored (see bug report).

```
attribute "constructor" does not take arguments C/C++(1094)
```

To suppress the error, the following can be used:

```
#if __INTELLISENSE__
    #pragma diag_suppress 1094
#endif
```

Linker File

To have the init and de-init called by the startup code, I have to make sure the objects are linked and listed in a special section. Below is the added section placement for constructors and destructors:

```
.text : ALIGN(4)
    {
        FILL(0xff)
        __vectors_start__ = ABSOLUTE(.) ;
        KEEP(*(.isr_vector))
        /* Global Section Table */
        . = ALIGN(4) ;
        __section_table_start = .;
        __data_section_table = .;
```

```
        LONG(LOADADDR(.data));
        LONG(    ADDR(.data));
        LONG(  SIZEOF(.data));
        LONG(LOADADDR(.data_RAM2));
        LONG(    ADDR(.data_RAM2));
        LONG(  SIZEOF(.data_RAM2));
        __data_section_table_end = .;
        __bss_section_table = .;
        LONG(    ADDR(.bss));
        LONG(  SIZEOF(.bss));
        LONG(    ADDR(.bss_RAM2));
        LONG(  SIZEOF(.bss_RAM2));
        __bss_section_table_end = .;
        __section_table_end = . ;
        /* End of Global Section Table */

        *(.after_vectors*)

        /* Kinetis Flash Configuration data */
        . = 0x400 ;
        PROVIDE(__FLASH_CONFIG_START__ = .) ;
        KEEP(*(.FlashConfig))
        PROVIDE(__FLASH_CONFIG_END__ = .) ;
        ASSERT(!(__FLASH_CONFIG_START__ == __FLASH_CONFIG_END__), "Linker Flash Config Support Enabled, but no
.FlashConfig section provided within application");
        /* End of Kinetis Flash Configuration data */

        *(.text*)
        *(.rodata .rodata.* .constdata .constdata.*)
        . = ALIGN(4);
        /*---------------------------------------------------------------------------- */
        /* Constructors and Destructors */
        . = ALIGN(4);
        KEEP(*(.init))

        . = ALIGN(4);
        __preinit_array_start = .;
        KEEP (*(.preinit_array))
        __preinit_array_end = .;

        . = ALIGN(4);
        __init_array_start = .;
        KEEP (*(SORT(.init_array.*)))
        KEEP (*(.init_array))
        __init_array_end = .;

        KEEP(*(.fini));

        . = ALIGN(4);
        KEEP (*crtbegin.o(.ctors))
        KEEP (*(EXCLUDE_FILE (*crtend.o) .ctors))
        KEEP (*(SORT(.ctors.*)))
        KEEP (*crtend.o(.ctors))

        . = ALIGN(4);
        KEEP (*crtbegin.o(.dtors))
        KEEP (*(EXCLUDE_FILE (*crtend.o) .dtors))
        KEEP (*(SORT(.dtors.*)))
        KEEP (*crtend.o(.dtors))
        . = ALIGN(4);
        /* End Constructors and Destructors */
        /*---------------------------------------------------------------------------- */
        /*---------------------------------------------------------------------------- */
    } > PROGRAM_FLASH
```

The same linker placement is used for C++ constructors and destructors too.

Startup Code

In the Startup code I need to make sure that the constructors and destructors get called. In a C++ system, this should be already implemented. In a C application, I have to make sure two functions in the C library get called.

First I have to add two prototypes:

```
extern void __libc_init_array(void); /* call constructors */
extern void __libc_fini_array(void); /* call destructors */
```

Then I need to make sure they get called before and after main():
Constructor and Destructor Calls around main()

Library Runtime

If you get an error during linking with
undefined reference to `_init'

then you need to check your linker file, because you have to add the crt*.o runtime support files which perform the initialization and de-initialization. Below is what I have in my linker files:

```
GROUP (
    "libc.a"
    "libgcc.a"
    "libm.a"
    "libc_nano.a"
    /* startup with constructor/destructor support: */
    "crti.o"
    "crtn.o"
    "crtbegin.o"
    "crtend.o"
)
```

Summary
With this, the constructors get called before main, and the destructors get called if I would leave main.

In summary, I have now an elegant way to call initialization and de-initialization in my application.

---
https://blog.timac.org/2016/0716-constructor-and-destructor-attributes/

constructor and destructor attributes
Jul 16, 2016

GCC (and Clang) supports constructor and destructor attributes:
`__attribute__((constructor))`
`__attribute__((destructor))`

Description
A function marked with the `__attribute__((constructor))` attribute will be called automatically before your main() function is called. Similarly a function marked with the `__attribute__((destructor))` attribute will be called automatically after your main() function returns.

You can find the GCC documentation [https://gcc.gnu.org/onlinedocs/gcc-3.0.1/gcc_5.html]here:

> constructor
> destructor
> The constructor attribute causes the function to be called automatically before execution enters main (). Similarly, the destructor attribute causes the function to be called automatically after main () has completed or exit () has been called. Functions with these attributes are useful for initializing data that will be used implicitly during the execution of the program.
>
> These attributes are not currently implemented for Objective C.

Note: The GCC documentation tells that these attributes are not implemented for Objective-C. However this seems to work as expected with my tests using Clang 'clang-703.0.31' from Xcode 7.3.1.

Example
Here is an example of C code to demonstrate these attributes:

```
//
// To compile:
// clang -o constructor constructor.c
//

#include <stdio.h>

void constructor() __attribute__((constructor));
void destructor() __attribute__((destructor));

int main() {
    printf ("main called\n");
    return 0;
}

void constructor() {
    printf ("constructor called\n");
}

void destructor() {
    printf ("destructor called\n");
}
```

When running this application, you will see the following output logs as you would expect:

```
constructor called
main called
destructor called
```

How does it work under the hood?
When you mark functions with these attributes, the compiler will create in your binary the sections called `__mod_init_func` for the constructors and `__mod_term_func` for the destructors. These sections contain the list of function pointers. You can use the excellent MachOView to see these sections:

When your application is launched, dyld will call the constructors before your main() function is called. This is handled by the following dyld function:

```
void ImageLoaderMachO::doModInitFunctions(const LinkContext& context)
```

The destructors are handled by the dyld function:

```
void ImageLoaderMachO::doTermination(const LinkContext& context)
```

Since dyld is open source you can look at the implementation in the file ImageLoaderMachO.cpp of dyld. The source code for macOS 10.11.4 is available here.

Example of use

* The first obvious usage is to be able to initialize some global variables with a constructor and do some cleanup with a destructor. It could be used to initialize some libraries too.
* Another usage is code injection. In a previous post 'Simple code injection using DYLD_INSERT_LIBRARIES' I wrote code to replace some methods with an Objective-C +(void)load class method. Using a constructor would allow to inject code earlier in the process.
* A constructor attribute could be used to implement a software protection. You could encrypt your executable with a custom encryption and use a constructor function to decrypt the binary just before it is loaded.
