C Language Constructors and Destructors with GCC

Constructors and Destructors are special functions. These are one of the features provided by an Object Oriented Programming language. Constructors and Destructors are defined inside an object class. When an object is instantiated, ie. defined of or dynamically allocated of that class type, the Constructor function of that class is executed automatically. There might be many constructors of which the correct implementation is automatically selected by the compiler. When this object is destroyed or deallocated, the Destructor function is automatically executed. For example when the scope of the object has finished or the object was dynamically allocated and now being freed. The Constructors and the Destructors are generally contains initialization and cleanup codes respectively required by an object to operate correctly. Because these functions are automatically invoked by the compiler therefore the programmer freed from the headache of calling them manually.

There is no such thing called 'constructors' and 'destructors' in C programming language or in structured languages, although there is no boundaries on defining such functions which act like them. You need to make functions which act like the constructors and destructors and then call them manually.

The GCC constructor and destructor attributes

GCC has attributes with which you can tell the compiler about how a lot of things should be handled by the compiler. Among such attributes the below function attributes are used to define constructors and destructors in C language. These would only work under GCC. As there is no *objects* and *class* approach in C the working of these functions are not like C++ or other OOP language constructor and destructors. With this feature, the functions defined as constructor function would be executed before the function `main` starts to execute, and the destructor would be executed after the `main` has finished execution. The GCC function attributes to define constructors and destructors are as follows:

```
__attribute__((constructor))
__attribute__((destructor))
__attribute__((constructor (PRIORITY)))
__attribute__((destructor (PRIORITY)))
```

For example, to declare a function named `begin ()` as a constructor, and `end ()` as a destructor, you need to tell `gcc` about these functions through the following declaration.

```c
void begin (void) __attribute__((constructor));
void end (void) __attribute__((destructor));
```

An alternate way to flag a function as a C constructor or destructor can also be done at the time of the function definition.

```
__attribute__((constructor)) void begin (void)
{
 /* Function Body */
}
__attribute__((destructor)) void end (void)
{
 /* Function Body */
}
```

After declaring the functions as constructors and destructors as above, `gcc` will automatically call `begin ()` before calling `main ()` and call `end ()` after leaving main or after the execution of `exit ()` function. The following sample code demonstrates the feature.

```
#include <stdio.h>

void begin (void) __attribute__((constructor));
void end (void) __attribute__((destructor));

int main (void)
{
  printf ("\nInside main ()");
}

void begin (void)
{
  printf ("\nIn begin ()");
}

void end (void)
{
  printf ("\nIn end ()\n");
}
```

Execution of this code will come up with an output which clearly shows how the functions were executed.

```
In begin ()
Inside main ()
In end ()
```

Multiple Constructors and Destructors

Multiple constructors and destructors can be defined and can be automatically executed depending upon their priority. In this case the syntax is `__attribute__((constructor (PRIORITY)))` and `__attribute__((destructor (PRIORITY)))`. In this case the function prototypes would look like.

```c
void begin_0 (void) __attribute__((constructor (101)));
void end_0 (void) __attribute__((destructor (101)));

void begin_1 (void) __attribute__((constructor (102)));
void end_1 (void) __attribute__((destructor (102)));

void begin_2 (void) __attribute__((constructor (103)));
void end_2 (void) __attribute__((destructor (103)));
```

The constructors with *lower priority* value would be executed first. The destructors with *higher priority* value would be executed first. So the constructors would be called in the sequence: `begin_0`, `begin_1 ()`, `begin_2 ()` . and the destructors are called in the sequence `end_2 ()`, `end_1 ()`, `end_0 ()`. Note the LIFO execution sequence of the constructors and destructors depending on the priority values.

The sample code below demonstrates this

```c
#include <stdio.h>

void begin_0 (void) __attribute__((constructor (101)));
void end_0 (void) __attribute__((destructor (101)));

void begin_1 (void) __attribute__((constructor (102)));
void end_1 (void) __attribute__((destructor (102)));

void begin_2 (void) __attribute__((constructor (103)));
void end_2 (void) __attribute__((destructor (103)));

int main (void)
{
  printf ("\nInside main ()");
}

void begin_0 (void)
{
  printf ("\nIn begin_0 ()");
}

void end_0 (void)
{
  printf ("\nIn end_0 ()");
}

void begin_1 (void)
{
  printf ("\nIn begin_1 ()");
}

void end_1 (void)
{
  printf ("\nIn end_1 ()");
}

void begin_2 (void)
{
  printf ("\nIn begin_2 ()");
}

void end_2 (void)
{
  printf ("\nIn end_2 ()");
}
```

The output is as below:

```
In begin_0 ()
In begin_1 ()
In begin_2 ()
Inside main ()
In end_2 ()
In end_1 ()
In end_0 ()
```

Note that, when compiling with priority values between 0 and 100 (inclusive), `gcc` would throw you warnings that the priority values from 0 to 100 are reserved for implementation, so these values might be used internally that we might not know. So it is better to use values out of this range. The value of the priority does not depend, instead the relative values of the priority is the determinant of the sequence of execution.

Note that the function `main ()` is not the first function/code block to execute in your code there are a lot of code already executed before `main` starts to execute. The function `main` is the user's code entry point, but the program entry point is not the `main` function. There is a startup function which prepares the environment for the execution. The startup functions first call the functions declared as constructors and then calls the `main`. When `main` returns the control to the startup function it then calls those functions which you have declared as the destructors. There are separate sections in the executable *.ctors* and *.dtors* which hold these functions. (not discussed here).

As there is no class object creation in C language does not have such features like in C++ or other OOP languages but this feature can bring some flexibility by calling the functions automatically on execution and termination of the code which you needed to do inside the main. For example one use may be like, you have a dynamically allocated global variable, might point to a linked list head or an array, or a file descriptor which you can allocate inside a constructor. If some error is encountered you can immediately call `exit ()` or the program terminates normally, depending on the error code you can make cleanup in the destructors.

Another good use is probably calling the initialization functions of some library, a bunch of which needs to be called in each program. You can make a separate file with the constructor files calling properly the library functions (probably operating on globals) and calling the library cleanup functions in the destructors. Whenever you make a program what you need to do is to compile your code with this file containing the constructors and destructors and forget about calling the initialization and cleanup functions in the main program. But doing this will make your code unportable, as it would only work under GCC.