filename: c-cleanup-attribute-01-20251222.txt
https://omeranson.github.io/blog/2022/06/12/cleanup-attribute-in-C

Cleanup Attribute in C

12 Jun 2022

In this blog post I explore __attribute__((cleanup(...))). I discuss what it does, how it does it, why use it, performance considerations, and finish by saying it's absolutely fantastic.

Introduction

Let's have a show of hands. How many here forgot to close a file? Or a socket? Or free a pointer? Or unlock a mutex?

How many did it on function exit, but not with that annoying return in the middle? Well, not me! I use smart pointers. Except in C. Where they don't exist. Which sucks.

Problem Description

There are a lot of cases where you acquire something at the beginning of a function, such as a file, a socket, a mutex, or some memory, and have to release it when you are done with it. Usually at the end of the function, or more specifically, at the end of the block.

As developers, this is one more thing to remember. Worse still, it has to be remembered at every exit point. So if you break out of your loop, or return in the middle of the function, you have to remember to clean up there as well.

Needless to say, this is usually forgotten. And then you have to understand why your programme is deadlocked, and you are sad, and you cry into your coffee, and then your coffee is all salty.

Solution

gcc and clang support the cleanup attribute. The attribute is applied to variables. It instructs the compiler to run a function when the variable goes out of scope. The function is called with a pointer to the variable.

Usage

Let's see it in an example:

```
void cleanup_free(void *p_) {
    void **p = (void **)p_;
    free(*p);
}

void f(const char *str) {
    __attribute__((cleanup(cleanup_free))) char *str_copy = strdup(str);
    // Do something with str_copy
    // ...
    // cleanup_free(&str_copy) is called here
}
```

The above code is equivalent to:

```
void f(const char *str) {
    __attribute__((cleanup(cleanup_free))) char * str_copy = strdup(str);
    // Do something with str_copy
    // ...
    free(str_copy);
}
```

Note that we inlined cleanup_free and removed the redundant dereference. We will see further down that compiler optimisation does this.

The neat thing about this is that it works wherever you go out of scope:

```
void f(const char *str) {
    if (strncmp(str, "/", 1) != 0) {
        __attribute__((cleanup(cleanup_free))) char * str_copy = strdup(str);
        // Do something with str_copy
        // ...
        // cleanup_free(&str_copy) is called here
    }
    // str_copy was already cleaned up by this point
}
```

Also works when you return early. Yes, this is a contrived example:

```
void f(const char *str) {
    __attribute__((cleanup(cleanup_free))) char * str_copy = strdup(str);
    if (strncmp(str, "/", 1) != 0) {
        // cleanup_free(&str_copy) is called here
        return;
    }
    // Do something with str_copy
    // ...
    // cleanup_free(&str_copy) is called here
}
```

This can also be written better, and still works as expected:

```
void f(const char *str) {
    if (strncmp(str, "/", 1) != 0) {
        return;
    }
    __attribute__((cleanup(cleanup_free))) char * str_copy = strdup(str);
    // Do something with str_copy
    // ...
    // cleanup_free(&str_copy) is called here
}
```

Note that we have declared str_copy late. In this case, cleanup_free isn't called in the first return.

Use Cases

There are a few use-cases I can think of off the top of my head.

Smart Pointers

I like smart pointers. They take care of freeing up unused resources automatically exactly when I no longer use it. They show exactly who owns my object.

But they are missing from C. So let's add them. Note this will be a Very simplistic implementation. For starters, we'll only implement unique_pointer.

A unique_pointer has two main function:

* Assign
* Release

And it calls the cleanup function when it is done. Let's see how to implement:

```
#define SMART_POINTER(type, name, cleanup_func) __attribute__((cleanup(cleanup_func))) type *name
#define SMART_POINTER_ASSIGN(name, value)  do { cleanup_func(&name); name = value; } while (0)
#define SMART_POINTER_RELEASE(name)  do { name = NULL; } while (0)
```

Here I assume that if the value of the smart pointer is NULL, then the cleanup function is a no-op.

Let's see it in use:

```
void cleanup_fclose(void* fpp) {
    FILE *fp = *((FILE **)fpp);
    if (fp) {
        fclose(fp);
    }
}

int read_file(const char *filename, char *buffer, int buflen) {
    SMART_POINTER(FILE, fp, cleanup_fclose) = fopen(filename, "r");
    if (fp == NULL) {
        return -1;
    }
    return fread(buffer, 1, buflen, fp);
}
```

On my environment this works. We can now go even further. We have smart pointers!

```
char *read_file(const char *filename) {
    SMART_POINTER(FILE, fp, cleanup_fclose) = fopen(filename, "r");
    if (fp == NULL) {
        return NULL;
    }
    char *buffer = calloc(1024, 1);
    return fread(buffer, 1, 1024, fp);
}

int main() {
    SMART_POINTER(char, buffer, cleanup_free) = read_file("/etc/fstab");
    printf("Read file: %s\n", buffer);
    return 0;
}
```

Sadly, we still need to declare buffer as a smart pointer. On the bright side, running this with valgrind says: All heap blocks were freed - no leaks are possible!

Entry/Exit logs

Many times, I want to add a log when my function enters and exits. Well now I can! Within limits.

```
void print_const_string_p(void *p) {
    char * log = *((char **)p);
    fprintf(stderr, "%s\n", log);
}

#define ENTRY_EXIT_LOG(entrylog, exitlog) \
    fprintf(stderr, "%s\n", entrylog); \
    __attribute__((cleanup(print_const_string_p))) const char *exitlogstr = exitlog

void f() {
```

```
    ENTRY_EXIT_LOG("Enter", "Exit");
}
```

Adding variables, especially to the exit log, is difficult, but not impossible. It is left as an exercise to the reader. Because I'm lazy.

Performance Consideration

I'll start by saying that I abhor premature optimisation. In general, write your code, and then run preformance tests on it and fix whatever is slowing you down. In my experience, saving on function calls is rarely (but sadly not always not) the way to go.

Having said that, let's look at the effects of using the cleanup attribute.

We will use a simple test function, read_file from above. We will have a version without the use of the cleanup attribute to the left, and to the right the same function using the cleanup attribute. A comparsion of the code is displayed here:
Topology configuration classes

The following is the object dump of both functions when compiled without optimisation:
Topology configuration classes

You'll note the main difference is that cleanup_fclose is called instead of fclose, and that the address of fp is taken by adding a lea instruction. Additionally the test method changed by using mov and test when the cleanup attribute is used, as opposed to cmpq. I don't know how to explain that.

However, let me convince you it doesn't matter. With a normal level of optimisation (-O2), the compiled code is identical:
- Topology configuration classes
- Meaning the differences were optimised out.
- Seeing as the resulting functions are identical, an empirical test is not performed.

Conclusion

We have seen the cleanup attribute. It allows writing cleaner, safer code. It has some disadvantages we did not discuss, such as that it is not in the standard.

I think it's fantastic. One of the things I like most about C++ and Rust is that the destructor is called when a variable goes out of scope. In my opinion this method is much better than the classical grabage collection that comes with e.g., Java.

And now it's available in C!
