filename: c-cleanup-attribute-02-multif-20251223.txt
https:// stackoverflow.com/questions/34574933/a-good-and-idiomatic-way-to-use-gcc-and-clang-attribute-cleanup-and-point

A good and idiomatic way to use GCC and clang __attribute__((cleanup)) and pointer declarations

I think that GCC extension __attribute__((cleanup)) is a good idea, at least for some cases, but i can't figure out how to use it in a good way. All i'm doing looks still really annoying.

I saw a lot of code doing #define _cleanup_(x) __attribute__((cleanup(x)) just to type less, but it there a way to pass there a standard function like free or closedir, fclose, etc?

As I see I can't just write:

```
__attribute__((cleanup(free))) char *foo = malloc(10);
```

Because the cleanup callback will receive char** pointer, and I have to always write something like:

```
static void free_char(char **ptr) { free(*ptr); }
__cleanup__((free_char)) char *foo = malloc(10);
```

That's pretty annoying, and the most annoying part is to define such cleanup functions for all types you need, because obviously you can't just define it for void **. What is the best way to avoid these things?

***

You can't write __attribute__((cleanup(free))), but you don't need to write a free cleanup function for each type. It's ugly, but you can write this:

```
static void cleanup_free(void *p) {
    free(*(void **) p);
}
```

I first saw this in the systemd codebase.

For other functions you would in general need to a write a wrapper with an extra level of indirection for use with __attribute__((cleanup)). systemd defines a helper macro for this:

```
#define DEFINE_TRIVIAL_CLEANUP_FUNC(type, func)             \
    static inline void func##p(type *p) {                   \
        if (*p)                                             \
        func(*p);                                           \
    }                                                       \
    struct __useless_struct_to_allow_trailing_semicolon__
```

which is used all over the place, e.g.

```
DEFINE_TRIVIAL_CLEANUP_FUNC(FILE*, pclose);

#define _cleanup_pclose_ __attribute__((cleanup(pclosep)))
```

***

There's a library that builds general-purpose smart pointers (unique_ptr and shared_ptr) on top of __attribute__((cleanup)) here: https:// github.com/Snaipe/libcsptr

It allows you to write higher-level code like this:

```
#include <stdio.h>
#include <csptr/smart_ptr.h>
#include <csptr/array.h>

void print_int(void *ptr, void *meta) {
    (void) meta;
    // ptr points to the current element
    // meta points to the array metadata (global to the array), if any.
    printf("%d\n", *(int *) ptr);
}

int main(void) {
    // Destructors for array types are run on every element of the
    // array before destruction.
    smart int *ints = unique_ptr(int[5], {5, 4, 3, 2, 1}, print_int);
    // ints == {5, 4, 3, 2, 1}

    // Smart arrays are length-aware
    for (size_t i = 0; i < array_length(ints); ++i) {
        ints[i] = i + 1;
    }
    // ints == {1, 2, 3, 4, 5}

    return 0;
}
```

As for idiomatic, though? Well the above is certainly close to idiomatic C++. Not C so much. The feature is clearly mainly supported in GCC and Clang because they have C++ compilers as well, so they have the option to make use of the RAII machinery in the C frontend at no extra cost; that doesn't make it a great idea to write C-intended-as-C this way. It kinda relies on a C++ compiler being present despite not actually being used.

If it were me, I'd probably investigate implementing autorelease pools, or something similar that can actually be done in pure C at the language level. Depends how quickly you need your resources to be freed; for memory, you usually can live without immediate cleanup.

---

https:// thephd.dev/c2y-the-defer-technical-specification-its-time-go-go-go

The Defer Technical Specification: It Is Time
March 15, 2025

After the Graz, Austria February 2025 WG14 Meeting, I am now confident in the final status of the defer TS, and it is now time.

... Time to What?

Time for me to write this blog post and prepare everyone for the implementation blitz that needs to happen to make defer a success for the C programming language. If you're smart and hip like Navi who wrote the GCC patch, the maintainer of slimcc who implemented defer from the early spec and found it both easy and helpful, and several others who are super cool and great, you can skip to the (DRAFT) ISO/DIS 25755 - defer Technical Specification and get started! But, for everyone else...

What is defer?

For the big brain 10,000 meter view, defer and the forthcoming TS 25755 is a general-purpose block/scope-based "undo" mechanism that allows you to ensure that no matter what happens a set of behavior (statements) are run. While there are many, many more usages beyond what will be discussed in this article, defer is generally used to cover these cases:

* unlock() of a mutex or other synchronization primitive after a lock();
* free() of memory after a malloc();
* deref() of a reference-counted parameter after a ref() or (shallow) copy() operation;
* rollback on a transaction if something bad happens;

and so, so much more. For C++ people who are going "wait a second, this sounds like destructors!", just go ahead and skip down below and read about the C++ part while ignoring all the stuff in-between about defer and WG14 and voting and consensus and blah blah blah.

For everyone else, we're going to go over some pretty simple examples of defer, using a series of printf's to construct (or fail to construct) a phrase, just to get an idea of how it works. Here's a basic example showing off some of its core properties:

```
#include <stdio.h>

int main () {
    const char* s = "this is not going to appear because it's going to be reassigned";
    defer printf(" bark!\"");
    defer printf("%s", s);

    defer {
        defer printf(" woof");
        printf(" says");
    }
    printf("\"dog");
    s = " woof";
    return 0;
}
```

The output of this program is as follows:

```
$> ./a.out
"dog says woof woof bark!"
```

The following principles become evident:

* The contents of a defer are run at the end of the block that contains it.
    + defer can be nested.
    + The rules for nested defer are the same as normal ones: it executes at the end of its containing block (defer introduces its own block.)
* Multiple defer statements run in reverse lexicographic order.
* defer does not need any braces for simple expression statements, same as for, while, if, etc. constructs.
* defer can have braces to stack multiple statements inside of it, same as for, while, if, etc. constructs.
* defer uses the value of the variable at the time defer is run at the end of the scope, not at the time when the defer statement is encountered.

This forms the core of the defer feature, and the basis by which we can build, compare, and evaluate this new feature.

"Build?" Wait... Are You Just Making This Up Entirely From Scratch?

Thankfully, no. This is something that has been cooked up for a long time by existing implementations in a variety of ways, such as:

* __attribute__((cleanup(func))) void* some_var;, where func takes the address of some_var and gets invoked when some_var's lifetime ends/the scope is finished (Clang, GCC, and SO many more compilers);

* __try/__finally, where the __finally block is invoked on the exit/finish of the __try block (MSVC);
* and, various different library hacks, such as this high-quality defer library and this other library-based library hack.

It has a lot of work and understanding behind it, and a ton of existing practice. Variations of it exist in Apple's MacOS SDK, the C parts of Swift, the Linux Kernel, GTK's g_autoptr (and qemu's Lockable), and so much more. It's also featured in many other languages in exactly the format specified here, including C++ (with RAII), Zig (with defer), and Swift (also as defer, but also a guard feature as well). This, of course, begs the question: if this has so much existing implementations in various styles, and so many years of experience, why is this going into a Technical Specification (or just "TS") rather than directly into the C standard? Well, honestly, there's 2 reasons.

The first reason is that vendors claim they can put it into C and make it globally available â faster than if it's put in the C working draft. Personally, I'm not sure I believe the vendors here; there are many features they have put into C, or even back ported from later versions of C into older versions of C. But, I'm not really at a point in my life that I feel like arguing with the vendors about a boring reskin of feature that's been in C compilers for just under as long as I've been alive, so I'm just going to take their word for it.

The second, more unfortunate, reason is that defer was proposed before I got my hands on it. It was not in a good shape and ready for standardization, and the ideas about what defer should be were somewhat all over the place. Which is fair, because many of the initial papers were exploratory: the problem was that when we had to cut a C23 release, there was a (minor) panic about new features and there was a lot of concentrated effort to try and slim defer down into something ready to go. Going from the wishy-washy status of before that wasn't grounded in existing practice to something material caused the Committee to reject the idea, and state that if it came back it should come back as a TS.

I could argue that this is not fair, because that vote was based off older version of the paper that was not ready and was subject to C23 pressures. The older papers were discussing various ideas like whether to capture variables by value at the point of the defer statement (catastrophic) or whether defer should be stapled to a higher scope / function scope like Go (also catastrophic), and whether writing a for loop would accumulate a (potentially infinite) amount of extra space and allocations to store variables and other data that would be needed to run at the end of the scope (yikes!). None of those shenanigans apply anymore, but we still have to go to a TS, even though it's a mirror-image of how existing practice works (in fact, less powerful than existing practice). Somewhat recently, we took new polls about whether it should go in a TS or whether it should go directly into the IS (International Standard; the working draft basically). There was support and consensus for both, but more consensus for a TS.

It's not really worth fighting about, though, so into a defer TS it goes.

My only worry is that Microsoft is going to do what it usually does and ignore literally everybody else doing things and not do any forward progress with just a defer TS. (As they do with most GNU or Clang or not-Microsoft extensions, some Technical Reports, and some TSs.) So, the only place we'll get experience is in places that already rely pretty heavily on the existence of the compiler feature. But, I'm more than willing to be pleasantly surprised. It could be driven by users demanding Microsoft make some of their C stuff safer through their User Voice / Feature Request submission portal. But, the message from Microsoft since Time Immemorial was always "just write C++", so I can imagine we'll just get the same messaging here, too, and have to wait until defer hits the C Standard before they implement it.

Nevertheless, this TS will be interesting for me. I have several other ideas that should go through a TS process; if I get to watch over the next couple of years that vendors weren't being honest about how quickly they could implement defer in their compilers if only they had a TS to justify it! â that will strongly color my opinion on whether or not any future improvements should use the TS process at all.

So we'll see! In the meantime, however, let's talk about how defer differs from its similarly-named predecessors in other languages.

Scope-based

The central idea behind defer is that, unlike its Go counterpart, defer in C is lexically bound, or "translation-time" only, or "statically scoped". What that means is that defer runs unconditionally at the end of the block or the scope it is bound to based on its lexical position in the order of the program. This gives it well-defined, deterministic behavior that requires no extra storage, no control flow tracking, no clever optimizations to reduce memory footprint, and no additional compiler infrastructure beyond what would normally be the case for typical variable automatic storage duration (i.e., normal-ass variable) lifetime tracking. Here's a tiny example using mtx_t:

```
#include <threads.h>

extern int do_sync_work(int id, mtx_t* m);

int main () {
    mtx_t m = {};
    if (mtx_init(&m, mtx_plain) != thrd_success) {
        return 1;
    }
    // we have successful initialization: destroy this when we're done
    defer mtx_destroy(&m);

    for (int i = 0; i < 12; ++i) {
        if (mtx_lock(&m) != thrd_success) {
            // return exits both the loop and the main() function,
```

```
            // defer block called:
            // - mtx_destroy
            return 1;
        }
        // now that we have succesfully init & locked,
        // make sure unlock is called whenever we leave
        defer mtx_unlock(&m);

        // ...
        // do a bunch of stuff!
        // ...
        if (do_sync_work(i, &m) == 0) {
            // something went wrong: get out of there!
            // return exits both the loop and the main() function,
            // defer blocks called:
            // - mtx_unlock
            // - mtx_destroy
            return 1;
        }

        // re-does the loop, and thus:
        // defer block called:
        // - mtx_unlock
    }

    // defer block called:
    // - mtx_destroy
    return 0;
}
```

The key takeaway from the comment annotations in the above is that: no matter if you early return from the 6th iteration of the for loop, or you bail early because of an error code sometime after the loop:

* if needed, mtx_unlock is always called on m, first;
* and, mtx_destroy is called on m, last.

Notably, the mtx_unlock call only happens if execution is still inside of the for loop, and only happens with exits from that specific scope after defer is passed. This is an important distinction from Go, where every defer is actually "lifted" from its current context and attached to run at the end of the function itself that is around it. This tends to make sense as a "last minute check before a function exits about some error conditions", but it has some devastating consequences for simple code. Take, for example, the following code from above, slightly simplified and modified to make a normal-looking Go program:

```
package main

import (
        "fmt"
        "sync"
    )

var x  = 0

func work(wg *sync.WaitGroup, m *sync.Mutex) {
    defer wg.Done()
        for i := 0; i < 42; i++ {
            m.Lock()
            defer m.Unlock()
            x = x + 1
        }
}

func main() {
    var w sync.WaitGroup
        var m sync.Mutex
        for i := 0; i < 20; i++ {
            w.Add(1)
            go work(&w, &m)
        }
    w.Wait()
    fmt.Println("final value of x", x)
}
```

The output of this program, on Godbolt, is:

```
Killed - processing time exceeded
Program terminated with signal: SIGKILL
Compiler returned: 143
```

Yeah, that's right: it never finishes running. This is because this code deadlocks: the defer call is hoisted to the outside of the for loop in func work. This means that it calls m.Lock(), does the increment, loops around, and then attempts to call m.Lock() again. This is a classic deadlock situation, and one that hits most people often enough in Go that they have to add a little caveat. "Use an immediately invoked function to clamp the defer's reach" is one of those quick caveats:

```
package main
```

```
import (
    "fmt"
    "sync"
)

var x  = 0

func work(wg *sync.WaitGroup, m *sync.Mutex) {
    defer wg.Done()
    for i := 0; i < 42; i++ {
        func() {
            m.Lock()
            defer m.Unlock()
            x = x + 1
        }()
    }
}


func main() {
    var w sync.WaitGroup
    var m sync.Mutex

    for i := 0; i < 20; i++ {
        w.Add(1)
        go work(&w, &m)
    }

    w.Wait()
    fmt.Println("final value of x", x)
}
```

This runs without locking up Godbolt's resource until a SIGKILL. Of course, this is pathological behavior; while it works great for a simple, direct use case ("catch errors and act on them"), it unfortunately results in other problematic behaviors. This is why the version in the defer TS does not cleave strongly to the scope of the function definition (or immediately invoked lambda), but instead directly to the innermost block and its associated scope. This also highlights another important quality of defer that we need when working with a language like C (and also applies to Zig and Swift).

Refer to Variables Directly

Also known as "capture by reference", defer blocks refer to variables in their scope directly (e.g., as if defer captured pointers to everything that was in scope and then automatically dereferenced those pointers so you could just refer to a previous foo directly as foo). This is something that people sometimes struggle with, but the choice is extremely obvious for a lot of both safety and usability reasons. Looking back at the examples above, there would be severe problems if a defer block would copy the m value, so that the lock/unlock paired calls would actually work on different entities. This would be a different kind of messed up that not even Go attempted, and no language should ever try.

When you have an in-line, scope-based, compile-time feature like defer that does not create an "object" and cannot "travel" to different scopes, capturing directly by reference is fine. Referring to variables directly is perfectly fine. You don't need to be careful and worry about captures, or be preemptively careful by capturing things through copying in order to be "safe". defer - unlike RAII objects - can't go anywhere. You don't need to be explicit about how it gets access to things in the local scope, because defer can't leave that scope. This is also a secondary consequence of not following in Go's footsteps; by not scoping it to the function, there's no concerns about whether or not the C-style automatic storage duration variables that are in, say, a for loop or an if statement need to be "lifetime extended" to the whole function's scope.

Direct variable reference and keeping things scope-based does mean that defer does not need to "store" its executions up until the end of the function, nor does it need to record predicates or track branches to know which defer is taken by the end of some arbitrary outer scope. In fact, for any defer block, the model of behavior for the defer TS is pretty much that it takes all the code inside of the defer block and dumps it out onto each and every translation-time (compile-time) exit of that scope. This applies to early return, breaking/continueing out of a loop scope, and also gotoing towards a label.

Oh, even goto?

In general, goto is banned from jumping over a defer or jumping into the sequence of statements in a defer. It can jump back before a defer in that scope. The same goes for trying to use switch, break/continue (with or without a label), and other things. Here's a few examples where things would not compile if you tried it:

```
#include <stdlib.h>

int main () {
    void* p = malloc(1);
    switch (1) {
            defer free(p); // No.
        default:
            defer free(p); // fine
            break;
    }
```

```
    return 0;
}

int main () {
    switch (1) {
        default:
            defer {
                break; // No.
            }
    }
    for (;;) {
        defer {
            break; // No.
        }
    }
    for (;;) {
        defer {
            continue; // No.
        }
    }
    return 0;
}
```

It's also important to be aware that defer that are not reached in terms of execution do not affect the things that come before them. That is, this is a leak still:

```
#include <stdlib.h>

int main () {
    void* p = malloc(1);
    return 0; // scope is exited here, `defer` is unreachable
    defer free(p); // p is leaked!!
}
```

Similar to the bans on break, goto, continue, and similar, return also can't exit a defer block:

```
int main () {
    defer { return 24; } // No.
    return 5;
}
```

Though, if you're an avid user of both __attribute__((cleanup(...))) and __try/__finally, you'll find that some of these restrictions are actually harsher than what is allowed by the mirrored existing practice, today.

Wait.... Existing Practice Can Do WHAT, Now?

The bans written about in the preceding section are a bit of a departure from existing practice. Both __attribute__((cleanup(...))) and __try/__finally the original versions of this present in GCC/Clang/tcc/etc., and MSVC, respectively allowed for some (cursed) uses of goto, pre-empting returns, and more in those implementation-specific kinds of defer.

An MSVC example (with Godbolt):

```
int main () {
    __try {
        return 1;
    }
    __finally {
        return 5;
    }
    // main returns 5 can stack this infinitely
}
```

A GCC example (with Godbolt):

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

int main () {
    __label__ loop_endlessly_and_crash;
    loop_endlessly_and_crash:;
    void horrible_crimes(void* pp) {
        void* p = *(void**)pp;
        printf("before goto...\n");
        goto loop_endlessly_and_crash; // this program never exits successfully or frees memory
        printf("after goto...\n");
        printf("deallocating...\n");
        free(p);
    }
    [[gnu::cleanup(horrible_crimes)]] void* p = malloc(1);
    printf("allocated...\n");
    printf("before label...\n");
    printf("after label...\n");
    return 0;
```

```
}
```

The vast majority of people both inside and outside of the Committee agreed that allowing this directly in defer for the first go-around was Bad and Evil. I also personally agree that I don't like it, though I would actually be okay with relaxing the constraint in the future because even if I don't personally like what I'm seeing from this, I can still write out a tangible, understandable, well-defined behavior for "goto leaves a defer block" or "return is called from within a defer block". The things I won't move on, though, are "goto into a defer block" (which exit of the scope is the goto taking execution to??), or jumping over a defer statement in a given scope: there's no clear, unambiguous, well-defined behavior for that, and it only gets worse with additional control flow.

But, even if you can't return from the TS's deferred block, you still have to be aware of when and how the defer actually runs in relation to the actual expression contained in a return statement or similar scope escape.

defer Timing

Matching existing practice and also C++ destructors, defer is run before the function actually returns but after the computation of the return's value. In a language like this, this is not observable in simple programs. But, in complex programs, this absolutely matters. For example, consider the following code:

```
#include <stddef.h>

extern int important_func_needs_buffer(size_t sz, void* p);
extern int* get_important_buffer(int* p_err, size_t* p_size, int val);
extern void drop_important_buffer(int val, size_t size);

int f (int val) {
    int err = 0;
    size_t size = 0;
    int* p = get_important_buffer(&err, &size, val);
    if (p == nullptr || err != 0) {
        return err;
    }
    defer {
        drop_important_buffer(val, size);
    }
    return important_func_needs_buffer(sizeof(*p) * size, p);
}

int main () {
    if (f(42) == 0) {
        printf("bro definitely cooked. peak.");
        return 0;
    }
    printf("what was bro cooking???");
    return 1;
}
```

There's 2 times in which you can run the defer block and its drop_important_buffer(...) call.
  * before the function returns and before important_func_needs_buffer(...);
  * or, before the function returns but after important_func_needs_buffer(...).

The problem becomes immediately apparent, here: if the defer runs before the expression in the return statement (before important_func_needs_buffer(...)), then you actually drop the buffer before the function has a chance to use it. That's a one-way ticket to a use-after-free, or other extremely security-negative shenanigans. So, the only logical and plausible choice is to run the second option, which is that the defer block runs after the return expression is evaluated but before we leave the function itself.

This does frustrate some people, who want to use defer as a last-minute "return value change" like so:

```
int main (int argc, char* argv[]) {
    int val = 0;
    int* p_val = &val;
    defer {
        if ((argc % 2) == 0) {
            *p_val = 30;
        }
    }
    return val; // returns 0, not 30, even if argc is e.g. 2
   }
```

But I value much more highly compatibility with existing practice (both __try/__finally and __attribute__((cleanup(...)))), compatibility with C++ destructors, and avoiding the absolute security nightmare. If someone wants to evaluate the return expression but still modify the value, they can write a paper or submit feedback to implementations that they want defer { if (whatever) { return ...; } } to be a thing. That way, such a behavior is formalized. And, again, even if I don't personally want to write code like this or see code like this, there's still a detectable, tangible, completely well-defined behavior for what happens if a return is evaluated in a defer. This is also not nearly as complex as e.g. Go's defer, because the defer TS uses a translation-time scoped defer.

It won't result in "dynamically-determined and executed defer causes spooky action at a distance". One would still need to be careful about having nested defers that also overwrite the return, or

subsequent defers that attempt to change the return value. (One would also have to contend that every defer-nested return would need to have its expression evaluated, and potentially discarded, sans optimization to stop it.) Given needing to answer all of these questions, though, it is still icky and I'm glad we don't have to go through with return (or goto or break or continue) within defer statements.

... What About Control Flow Outside of Compilation Time?

Run-time style control flow like longjmp, or similar _Noreturn/[[_Noreturn]]/[[noreturn]]-marked functions, are a-okay if they mimic the above allowed uses of goto. If it jumps out of the function entirely, or jumps into a previous scope but beyond the point where a defer would be, the behavior can end up undefined. That means use of functions like exit, quick_exit, or similar explicitly by the user may leak resources by not executing any currently open defer blocks. This is similar to C++, where calling any of the C standard library exit functions (and, specifically, NOT std::terminate()) means destructors will not get run. The only function that this is not fully true on is thrd_exit, as glibc has built-in behavior where thrd_exit will actually provoke unwinding of thread resources by calling destructors on that thread. (You can then use thrd_exit on the main thread, even in a single-threaded program, as a means to trigger unwinding; this is an implementation detail of glibc, though, and most other C standard libraries don't behave like this.)

The exact wording in the TS and the proposal is that its "unspecified" behavior, but it doesn't actually proscribe any specific set of behaviors that can happen. So, even if we use the "magic" word of "unspecified" for these run-time jumps, the behavior is effectively as bad as undefined behavior because there really isn't any document-provided guarantee about what happens when you run off somewhere with e.g. setjmp/longjmp in these situations. I guess the only thing it prevents is some compiler optimization junkie trying to optimize based on whether or not defer with a run-time jump would trigger undefined behavior, though it's effectively an optimization you can maybe get by only combining defer and one of these run-time jumps. At that point, I'd question what the hell the engineer was doing submitting that kind of "improvement" in the first place to the optimizer, and reject it on the grounds of "Please find something better to do".

But, you never know I guess?

Maybe there would be real gains, but I'm not holding my breath nor making any space for it. But beyond just ignoring dubious weird optimization corners for defer...

Does.... Defer Actually Solve Any Problems, Though?

Believe it or not: yes. I'm not one to waste my time on things with absolutely no real value; there's just too little time and standardization takes too much damn effort to focus on worthless things. Though, if you were to take it from others, you'd hear about how defer complicates the language for not much/no benefit:

> ... The proposal authors show a complex solution to make the code free storage and then show how it can be "simplified" using defer. But it is trivial to centralize cleanup in one function, no new features needed. If I was developing this code for real, I'd take the next step and make it single exit. ...
>
> Victor Yodaiken, "Don't Defer", December 12, 2023

The code Yodaiken is referring to is code contained in the original proposal (the original proposal is being updated in lock-step with the TS), specifically this section. The code in question was offered to me by its author, and I was told to simply / work with the code. So, after a bit of cleanup and checking and review, this is the first-effort defer version of the original code:

```
h_err* h_build_plugins(const char* rootdir, h_build_outfiles outfiles, const h_conf* conf) {
    char* pluginsdir = h_util_path_join(rootdir, H_FILE_PLUGINS);
    if (pluginsdir == NULL)
        return h_err_create(H_ERR_ALLOC, NULL);
    defer free(pluginsdir);
    char* outpluginsdirphp = h_util_path_join (
        rootdir,
        H_FILE_OUTPUT "/" H_FILE_OUT_META "/" H_FILE_OUT_PHP
    );
    if (outpluginsdirphp == NULL) {
        return h_err_create(H_ERR_ALLOC, NULL);
    }
    defer free(outpluginsdirphp);
    char* outpluginsdirmisc = h_util_path_join (
        rootdir,
        H_FILE_OUTPUT "/" H_FILE_OUT_META "/" H_FILE_OUT_MISC
    );
    if (outpluginsdirmisc == NULL) {
        return h_err_create(H_ERR_ALLOC, NULL);
    }
    defer free(outpluginsdirmisc);

    // Check status of rootdir/plugins, returning if it doesn't exist
    {
        int err = h_util_file_err(pluginsdir);
        if (err == ENOENT) {
            return NULL;
        }
        if (err && err != EEXIST) {
            return h_err_from_errno(err, pluginsdir);
        }
    }
```

```
    // Create dirs if they don't exist
    if (mkdir(outpluginsdirphp, 0777) == -1 && errno != EEXIST) {
        return h_err_from_errno(errno, outpluginsdirphp);
    }

    if (mkdir(outpluginsdirmisc, 0777) == -1 && errno != EEXIST) {
        return h_err_from_errno(errno, outpluginsdirmisc);
    }

    // Loop through plugins, building them
    struct dirent** namelist;
    int n = scandir(pluginsdir, &namelist, NULL, alphasort);
    if (n == -1) {
        return h_err_from_errno(errno, namelist);
    }
    defer {
        for (int i = 0; i < n; ++i) {
            free(namelist[i]);
        }
        free(namelist);
    }

    for (int i = 0; i < n; ++i) {
        struct dirent* ent = namelist[i];
        if (ent->d_name[0] == '.') {
            continue;
        }
        char* dirpath = h_util_path_join(pluginsdir, ent->d_name);
        if (dirpath == NULL) {
            return h_err_create(H_ERR_ALLOC, NULL);
        }
        defer free(dirpath);
        char* outdirphp = h_util_path_join(outpluginsdirphp, ent->d_name);
        if (outdirphp == NULL) {
            return h_err_create(H_ERR_ALLOC, NULL);
        }
        defer free(outdirphp);
        char* outdirmisc = h_util_path_join(outpluginsdirmisc, ent->d_name);
        if (outdirmisc == NULL) {
            return h_err_create(H_ERR_ALLOC, NULL);
        }
        defer free(outdirmisc);

        h_err* err;
        err = build_plugin(dirpath, outdirphp, outdirmisc, outfiles, conf);
        if (err) {
            return err;
        }
    }

    return NULL;
}
```

This code has some improvements over the original, insofar that it actually protects against a few leaks that were happening in that general purpose code. Instead of this approach, Yodaiken instead changed it to this:

```
struct plugins {
    char *pluginsdir;
    char *outpluginsdirphp;
    char *outpluginsdirmisc;
    char *dirpath;
    char *outdirphp;
    char *outdirmisc;
    int n;
    struct dirent **namelist;
};

void freeall(struct plugins *x) {
    if (x->pluginsdir)
        free(x->pluginsdir);
    if (x->outpluginsdirphp)
        free(x->outpluginsdirphp);
    if (x->outpluginsdirmisc)
        free(x->outpluginsdirmisc);
    if (x->dirpath)
        free(x->dirpath);
    if (x->outdirphp)
        free(x->outdirphp);
    if (x->outdirmisc)
        free(x->outdirmisc);
    for (int i = 0; i < x->n; i++) {
        free(x->namelist[i]);
    }
}
```

```
h_err *h_build_plugins(const char *rootdir, h_build_outfiles outfiles, const h_conf * conf) {
    struct plugins x = { 0, };
    x.pluginsdir = h_util_path_join(rootdir, H_FILE_PLUGINS);
    if (pluginsdir == NULL)
        return h_err_create(H_ERR_ALLOC, NULL);
    x.outpluginsdirphp = h_util_path_join(rootdir,
        H_FILE_OUTPUT "/" H_FILE_OUT_META
        "/" H_FILE_OUT_PHP);
    if (outpluginsdirphp == NULL) {
        freeall(&x);
        return h_err_create(H_ERR_ALLOC, NULL);
    }
    x.outpluginsdirmisc = h_util_path_join(rootdir,
        H_FILE_OUTPUT "/" H_FILE_OUT_META
        "/" H_FILE_OUT_MISC);
    if (x.outpluginsdirmisc == NULL) {
        freeall(&x);
        return h_err_create(H_ERR_ALLOC, NULL);
    }

    // Check status of rootdir/plugins, returning if it doesn't exist
    {
        int err = h_util_file_err(x.pluginsdir);
        if (err == ENOENT) {
            freeall(&x);
            return NULL;
        }
        if (err && err != EEXIST) {
            freeall(&x);
            return h_err_from_errno(err, x.pluginsdir);
        }
    }

    // Create dirs if they don't exist
    if (mkdir(x.outpluginsdirphp, 0777) == -1 && errno != EEXIST) {
        freeall(&x);
        return h_err_from_errno(errno, x.outpluginsdirphp);
    }
    if (mkdir(outpluginsdirmisc, 0777) == -1 && errno != EEXIST) {
        freeall(&x);
        return h_err_from_errno(errno, outpluginsdirmisc);
    }
    // Loop through plugins, building them
    x.n = scandir(x.pluginsdir, &x.namelist, NULL, alphasort);
    if (n == -1) {
        freeall(&x);
        return h_err_from_errno(errno, x.namelist);
    }
    for (int i = 0; i < n; ++i) {
        struct dirent *ent = namelist[i];
        if (ent->d_name[0] == '.') {
            continue;
        }
        x.dirpath = h_util_path_join(x.pluginsdir, ent->d_name);
        if (dirpath == NULL) {
            freeall(&x);
            return h_err_create(H_ERR_ALLOC, NULL);
        }
        x.outdirphp = h_util_path_join(outpluginsdirphp, ent->d_name);
        if (x.outdirphp == NULL) {
            freeall(&x);
            return h_err_create(H_ERR_ALLOC, NULL);
        }
        x.outdirmisc = h_util_path_join(x.outpluginsdirmisc, ent->d_name);
        if (x.outdirmisc == NULL) {
            freeall(&x);
            return h_err_create(H_ERR_ALLOC, NULL);
        }

        h_err *err;
        err = build_plugin(dirpath, outdirphp, outdirmisc, outfiles, conf);
        if (err) {
            freeall(&x);
            return err;
        }
    }

    freeall(&x);
    return NULL;
}
```

This works too, and one would argue that Yodaiken has done the same as defer but without the new feature or a TS or any shenanigans. But there's a critical part of Yodaiken's argument where his premise falls apart in the example code provided: refactoring. While he states that in "serious" code he would change this to be a single exit, the example code provided is just one that replaces all of the defer or manual frees of the original to instead be freeall. This was not unanticipated by the

proposal he linked to, which not only discusses defer in terms of code savings, but also in terms of vulnerability prevention. And it is exactly that which Yodaiken has fallen into, much like his peers and predecessors who work on large software like the Linux Kernel.

However, one should note that Yodaiken's changes here actually don't account for everything. Inside of the loop, it's not just freeall on error: users need to actually free x.dirpath, x.outdirmisc, and x.outdirphp every single loop. freeall doesn't account for that, so this is actually a downgrade over the defer version (which fixed these problems). It also didn't pull from the correct namelist (it should be x.namelist), but we can just chock that up to a quick blog post from 2 years ago trying to fix some typos.

CVE-2021-3744, and the Truth About Programmers

The problem, that Yodaiken misses in his example code rewrite and his advice to developers, is the same one that the programmers responsible for CVE-2021-3744. You see, much like Yodaiken's rewrite of the code, the function in question here had an object. That object's name was tag. And just like Yodaiken's rewrite, it had a function call like freeall that was meant to be called at the exit point of the function: ccp_dm_free. The problem, of course, is that along one specific error path, in conjunction with other flow control issues, the V5 CCP's tag structure was not being properly freed. That's a leak of (potentially sensitive) information; thankfully, at most it could provoke a Denial of Service, per the original reporter's claims.

This is the exact pitfall that Yodaiken's own code is subject to.

It's not that there isn't a way, in code as plain as C90, to write a function that frees everything. The problem is that in any sufficiently complex system, even with one that has as many eyeballs as bits of the cryptography code in the Linux Kernel, one might not be able to trace all the through-lines for any specifically used data. The function in question for CVE-2021-3744 had exactly what Yodaiken wanted: a single exit point after doing preliminary returns for precondition/invalid checks, goto to a series of laddered cleanup statements for the very end, highly reviewed code, and being developed in as real a context as it gets (the Linux Kernel). But, it still didn't work out.

Thankfully, this CVE is only a 5.5 - denial of service, maybe a bit of information leakage - but it's not the first screwup of this sort. This is only one of hundreds of CVEs that follow the same premise, that have been unearthed over the last 25-summat years of vulnerability tracking. And, most importantly, Yodaiken's code can be changed in the face of defer, in a way that both reduces the number of lines written and does all the same things Yodaiken's code does, but with better future proofing and less potential leaks:

```
struct plugins {
    char *pluginsdir;
    char *outpluginsdirphp;
    char *outpluginsdirmisc;
    char *dirpath;
    char *outdirphp;
    char *outdirmisc;
    int n;
    struct dirent **namelist;
};

void freeall(struct plugins *x) {
    free(x->pluginsdir);
    free(x->outpluginsdirphp);
    free(x->outpluginsdirmisc);
    free(x->dirpath);
    free(x->outdirphp);
    free(x->outdirmisc);
    for (int i = 0; i < x->n; i++) {
        free(x->namelist[i]);
    }
}

void freeloop_all(struct plugins *x) {
    free(x->dirpath);
    free(x->outdirphp);
    free(x->outdirmisc);
    x->dirpath = nullptr;
    x->outdirphp = nullptr;
    x->outdirmisc = nullptr;
}

h_err *h_build_plugins(const char *rootdir, h_build_outfiles outfiles, const h_conf * conf) {
    struct plugins x = { 0, };
    defer freeall(&x);
    x.pluginsdir = h_util_path_join(rootdir, H_FILE_PLUGINS);
    if (pluginsdir == NULL)
        return h_err_create(H_ERR_ALLOC, NULL);
    x.outpluginsdirphp = h_util_path_join(rootdir,
        H_FILE_OUTPUT "/" H_FILE_OUT_META
        "/" H_FILE_OUT_PHP);
    if (outpluginsdirphp == NULL) {
        return h_err_create(H_ERR_ALLOC, NULL);
    }
    x.outpluginsdirmisc = h_util_path_join(rootdir,
        H_FILE_OUTPUT "/" H_FILE_OUT_META
        "/" H_FILE_OUT_MISC);
    if (x.outpluginsdirmisc == NULL) {
```

```
        return h_err_create(H_ERR_ALLOC, NULL);
    }

    // Check status of rootdir/plugins, returning if it doesn't exist
    {
        int err = h_util_file_err(x.pluginsdir);
        if (err == ENOENT) {
            return NULL;
        }
        if (err && err != EEXIST) {
            return h_err_from_errno(err, x.pluginsdir);
        }
    }

    // Create dirs if they don't exist
    if (mkdir(x.outpluginsdirphp, 0777) == -1 && errno != EEXIST) {
        return h_err_from_errno(errno, x.outpluginsdirphp);
    }
    if (mkdir(outpluginsdirmisc, 0777) == -1 && errno != EEXIST) {
        return h_err_from_errno(errno, outpluginsdirmisc);
    }
    // Loop through plugins, building them
    x.n = scandir(x.pluginsdir, &x.namelist, NULL, alphasort);
    if (n == -1) {
        return h_err_from_errno(errno, x.namelist);
    }
    for (int i = 0; i < n; ++i) {
        struct dirent *ent = x.namelist[i];
        if (ent->d_name[0] == '.') {
            continue;
        }
        defer freeloop_all(&x);
        x.dirpath = h_util_path_join(x.pluginsdir, ent->d_name);
        if (dirpath == NULL) {
            return h_err_create(H_ERR_ALLOC, NULL);
        }
        x.outdirphp = h_util_path_join(outpluginsdirphp, ent->d_name);
        if (x.outdirphp == NULL) {
            return h_err_create(H_ERR_ALLOC, NULL);
        }
        x.outdirmisc = h_util_path_join(x.outpluginsdirmisc, ent->d_name);
        if (x.outdirmisc == NULL) {
            return h_err_create(H_ERR_ALLOC, NULL);
        }

        h_err *err;
        err = build_plugin(dirpath, outdirphp, outdirmisc, outfiles, conf);
        if (err) {
            return err;
        }
    }

    return NULL;
}
```

As you can see here, we made three just three change to Yodaiken's code here: we use defer freeall(&x) at the very start of the function and delete it everywhere else. We fix the loop part (again) correctly with defer freeloop_all(&x);, which was forgotten in the Yodaiken version. And, to make that possible, we have an additional function of freeloop_all and a modified freeall, to accomodate this. (The removal of the if checks is not necessary, but it should be noted free is one of the very, VERY few functions in the C standard library that's explicitly documented to be a no-op with a null pointer input).

With defer, we no longer need to add a freeall(&x) at every exit point, nor do we need a ladder of gotos cleaning up specific things (in the case where the structure didn't exist and we tried to use a single exit point). We also don't accidentally leak loop resources, too.

It's not that Yodaiken's principle of change wasn't an improvement over the existing code (consolidating the frees), it's just that it simply failed to capture the point of the use of defer: no matter how you exit from this function now (save by using runtime control flow), there is no way to forget to free anything. Nor is there any way to forget to free anything on some specific path. The problems of CVE-2021-3744 and the hundreds of CVEs like it are not really a plausible issue anymore. It means that the C code you write becomes resistant to problems with later changes or refactors: adding additional checks and exits (as we did compared to the original code in the repository, to cover some cases not covered by the original) means a forgotten freeall(&x) doesn't result in a leak.

This is the power of defer in C

Focusing on things that are actually difficult and worth your time is what your talents and efforts are made for. Menial tasks like "did I forget to free this thing or goto the correct cleanup target" are a waste of your time. Even the Linux Kernel is embracing these ideas, because bugs around forgetting to unlock() something or forgetting to free something are awful wastes of everyone's life, from people who have to report 'n' confirm basic resource failures to getting annoying security advisories over fairly mundane failures. We have more interesting code and greater performance gains to be putting our elbow grease into that do not include fiddling with the same basic crud thousands

of times.

This is what the defer TS is supposed to bring for C.

But... What About C++?

For C++ people, MOST (but not all) of defer is covered by destructors (and constructors) and by C++'s object model. The chance of having defer in C++, properly, is less than 0. The authors of C++'s library version of this (scope_guard) have intentionally and deliberately abandoned having this in the C++ standard library, and efforts to revive it (including efforts to revive it to spite defer and tell C to stop using defer) have either gone eerily/swiftly quiet or been abandoned. This does not mean there is no dislike or dissent for defer, just that its C++ compatriots have seemed to mostly calm down and step back from just trying to put raw RAII into C. Not that I would fully object to actually working out an object model and having real RAII, as stated in a previous article and in the rationale of the proposal itself discussing C++ compatibility of defer, certainly not! It's just that everyone who's trying has so far done a rather half-baked job of attempting it, mostly in service of their favorite pet feature rather than as a full, intentional integration of a complete object model that C++ is still working out the extreme edge-case kinks of to this day through Core Working Group issues.

There are also some edge cases where defer is actually better than C++, as mentioned in the rationale of the proposal. For example, exceptions butt up against the very strict noexcept rule for destructors (especially since its not just a rule, but required for standard library objects). This means that using RAII to model defer becomes painful when you intentionally want to use defer or scope_guard as an exception-detection mechanism and a transactional rollback feature. Destructors overwhelming purpose are, furthermore, to make repeatable resource cleanup easy, but in tying it to the object model must store all of the context that is accessible within the object itself so it can be appropriately accessed. Carrying that context can be antithetical to the goals of the given algorithm or procedure, meaning that a lot more effort goes into effective state management and transfer when just having key defer blocks in certain in-line cases would save on both object size and context move/transfer implementation effort. One can get fairly close by having a defer_t<...> templated type in C++ with all move/copy/etc. functions

Destructors can also fall apart in certain specific cases, like in the input and output file streams of C++. Because the destructor needs to finish to completion, cannot throw (per the Standard Library ironclad blanket rules), and must not block or stall (usually), the specification for the C++ standard streams will swallow up any failures to flush the stream when it goes out of scope and the destructor is run. This usually isn't a problem, but I've had to sit in presentations in real life during my C++ Meetup where the engineers gave talks on standard streams (and many of their boost counterparts) making it impossible for them to have high-reliability file operations. They had to build up their own from scratch instead. (I don't think Niall Douglass's (ned13's) Low-Level File IO had made it into Boost by then.)

Nevertheless, while RAII covers the overwhelming majority of use cases (reusable resource and policy), defer stands by itself as something uniquely helpful for the way that C operates. And, in particular, it can help cover real vulnerabilities that happen in C code due to the simple fact that most people are human beings.

Thusly...

The Time is Now

This is the specification for the defer TS. If you are reading this and you are a compiler vendor, beloved patch writer, or even just a compiler hobbyist, the time to implement this is today. Right now. The whole point of a TS and the reason I was forced by previous decisions and discussion out of my control to pick a TS is to obtain deployment experience. Early implementers have already found, recovered, and discovered bugs in their code thanks to defer. There is a wealth of places where using defer will drastically improve the quality of code. Removing a significant chunk of human error as well as reducing risk during refactors or rewrites because someone might forget to add a goto CLEANUP; or a necessary freeThat() call are tangible, real benefits we can do to prevent classes of leaks.

Implement defer. Tell me about it. Tell others about it.

The time is now, before C2Y ships. That's why it's a TS. Whether you gate it behind -fdefer-ts/-fexperimental-defer-ts, or you simply make it part of the base offering without needing extra flags, now is the time. The Committee is starting to constrict and retract heavily from the improvements in C23, and vendors are starting to get skittish again. They want to see serious groundswells in support; you cannot just sit around quietly, hoping that vendors "get the memo" to make fixes or pick up on your frustrations in mailing lists. Go to them. Register on their bug trackers (and look for existing open bugs). E-mail their lists (but search for threads already addressing things). You must be vocal. You must be loud. You must be direct.

You Must Not Be Ignorable.

With: compiler vendors especially the big ones getting more and more serious about telling people to Do It In The Standard Or #$&^! Off (with some exceptions); pressure being applied to have greater and greater consensus in the standard itself making that bar higher and higher; and, vendors and individuals getting more and more pissed off about changes to C jeopardizing their implementation efforts and what they view as the integrity of the C language, extensions and changes are more at risk now than ever. Please. Please, please, prettiest of pleases.

---
https:// news.ycombinator.com/item?id=11305142
rwmj on March 17, 2016

__attribute__((cleanup)) please. It's used by systemd now and is supported by gcc and clang. It makes a real difference to code, avoiding masses of cleanup along error paths.

***
I mentioned it elsewhere, but some equivalent D's Scope Guard statement would be even better:
https:// dlang.org/spec/statement.html#ScopeGuardStatement

***
When you start needing/wanting stuff like that, wouldn't it make sense to just use a C++ compiler and a simple RAII type or two?

***
Really it doesn't. If you add C++ to the mix, you lose the clear relationship with code as written to what runs (is this operator overloaded?), and you have an open invitation for people to attempt to use more and more C++ features, many of which are not well thought through.

Instead of moving to C++, I'd rather allow mixed objects in a more sensible language (eg. linking with objects written in OCaml).

***
I'm not convinced by this argument. It's perfectly possible to stick to a well defined subset of C++ if you're scared by certain features. Just write a coding standards document and enforce it.

Operator overloading is very important for generating the fastest code possible, too. It would be hard to get the benefits of a library like Eigen which can automatically use SSE to optimize maths expressions in C.

***
No. You don't want the vast majority of C++ semantics; especially classes, templates, and different casting rules. Additionally C++ compilers are just slower than C compilers.

***
Agreed. __attribute__((cleanup)) provides a way to do block-scoped cleanup of each variable and eliminate the "tear down in reverse order in each error path" or "tear down in reverse order with goto labels" pattern.

***
I have never seen this before but was just reading about it. It appears to run unconditionally when the scope is exited though: how do you prevent it from tearing down even in the success case?

***
Tearing down in the success case is a feature, not a bug! If you don't want cleanup along the success path (eg if you're returning an allocated string), don't use cleanup on the returned variable.

However that does point to the one problem with attribute((cleanup)). If you consider code like this (using the systemd macros):

```
int f (void) {
    _cleanup_free char *s1;
    _cleanup_free char *s2;
    s1 = strdup ("foo");
    if (!s1) return -1; /* crash */
    s2 = strdup ("bar");
    if (!s2) return -1;
    // ...
    return 0;
}
```

If strdup fails, it'll crash at the place I marked because it will try to free the uninitialized pointer s2.

To get around that you have to initialize everything to NULL (since free(NULL) is legal).

The problem then becomes that you end up "over-initializing" and "over-freeing". It would be nice to have a version of attribute((cleanup)) that would eliminate the call to the cleanup function if the pointer is NULL (or uninitialized?). But then you're relying on the compiler to do some kind of dataflow analysis, which is difficult in a standard (excludes simple compiler implementations), and may not even be possible because of the halting problem.

This is basically the reason why you wouldn't want to use cleanups in kernel code, because performance is paramount there and unnecessary cleanup calls wouldn't be welcome.

***
You could potentially write the "free" function like this:

```
static inline void free(void *ptr) {
    if (!ptr)
        return;
    _free(ptr);
}
```

Then the compiler can easily inline this, omit the NULL check if it knows the pointer can't be NULL, or omit the whole thing if it knows the pointer is NULL.

***
GCC 5 already does that for you (and more), since malloc/free are C standard library functions:

```
  #include <stdlib.h>
```

```
int main() {
    free(NULL);
    char *foo = malloc(42);
    free(foo);
}
```

gcc -O1 compiles it to:

```
  00000000004004f6 <main>:
    4004f6:    b8 00 00 00 00           mov    $0x0,%eax
    4004fb:    c3                       retq
    4004fc:    0f 1f 40 00              nopl   0x0(%rax)
```

***

free(NULL) is already perfectly legal (specced to be a no-op) as noted by the parent. The problem is that `char * s;` is probably _not_ NULL, so you'd try to free some random address on the stack. Of course it is easily fixed with `_cleanup_free char * s = NULL;`, but it's a somewhat difficult error to spot sometimes.

EDIT: I super misread that, my bad.

***

The point is that this informs the compiler of that behavior of free so that it can optimize it away. Basically it gives you some of the advantages you'd get if free were a compiler intrinsic (which it might be, idk).

***

I'm not familiar with using attributes for cleanup, but isn't the problem caused by the predeclaration of the variables? That is, wouldn't this work (and be shorter and simpler)?

```
int f (void) {
    _cleanup_free char *s1 = strdup ("foo");
    if (!s1) return -1;
    _cleanup_free char *s2 = strdup ("bar");
    if (!s2) return -1;
    // ...
    return 0;
}
```

***

OK, I tested it. Yes, this approach worked fine for me in GCC, Clang, and ICC.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define autofree __attribute((cleanup(autofree_func)))
#define PASS 0
#define FAIL 1

void autofree_func(void *ptr_ptr) {
    void *ptr = * (void **) ptr_ptr;
    printf("%s(%p)\n", __func__, ptr);
    free(ptr);
}

int test(int fail1, int fail2) {
    printf("\n%s(%s, %s):\n", __func__,
            fail1 ? "FAIL" : "PASS",
            fail2 ? "FAIL" : "PASS");
    autofree char *s1 = strdup("foo");
    if (fail1) return -1;
    printf("s1: '%s' (%p)\n", s1, s1);
    autofree char *s2 = strdup("bar");
    if (fail2) return -1;
    printf("s2: '%s' (%p)\n", s2, s2);
    return 0;
}

int main(/* int argc, char **argv */) {
    test(PASS, PASS);
    test(PASS, FAIL);
    test(FAIL, FAIL);
    return 0;
}
```

```
nate@skylake$> cc -Wall -Wextra -Wconversion -O3 scoping.c -o scoping

nate@skylake$> ./scoping

test(PASS, PASS):
s1: 'foo' (0x2056010)
s2: 'bar' (0x2056030)
autofree_func(0x2056030)
```

```
autofree_func(0x2056010)

test(PASS, FAIL):
s1: 'foo' (0x2056010)
autofree_func(0x2056030)
autofree_func(0x2056010)

test(FAIL, FAIL):
autofree_func(0x2056010)
```

***

> Tearing down in the success case is a feature, not a bug!

I guess it is if you are thinking of this as C++ RAII destructors, which is indeed one very useful case.

It's not useful in some other cases. Like imagine that you are writing the C equivalent of a constructor. If you initialize half your members and then run out of memory, you want to uninitialize only the members you managed to initialize already. attribute((cleanup)) doesn't help with this case.

***

```
out_error:
dtor(self);
```

***

@haberman, you don't use it for constructing an object you want to return.

***

RAII is the one thing I miss from C++.

---

https:// echorand.me/site/notes/articles/c_cleanup/cleanup_attribute_c.html

Using the __cleanup__ variable attribute in GCC

GCC's C compiler allows you to define various variable attributes. One of them is the cleanup attribute (which you can also write as __cleanup__) which allows you to define a function to be called when the variable goes out of scope (for example, before returning from a function). This is useful, for example to never forget to close a file or freeing the memory you may have allocated. Next up is a demo example defining this attribute on an integer variable (which obviously has no practical value). I am using gcc (GCC) 4.7.2 20121109 on Fedora 18.

Demo

The next code listing declares an integer variable, avar with the cleanup attribute set such that the function clean_up is called before main() returns (Line no. 27).

```
# include <stdio.h>

/* Demo code showing the usage of the cleanup variable
   attribute. See:http:// gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html
*/

/* cleanup function
   the argument is a int * to accept the address
   to the final value
*/

void clean_up(int *final_value) {
    printf("Cleaning up\n");
    printf("Final value: %d\n",*final_value);

}

int main(int argc, char **argv) {
    /*  declare cleanup attribute along with initiliazation
        Without the cleanup attribute, this is equivalent
        to:
        int avar = 1;
    */

    int avar __attribute__ ((__cleanup__(clean_up))) = 1;
    avar = 5;

    return 0;
}
```

The clean_up function above accepts an argument which is an integer pointer. This is a pointer to the integer variable avar for which this function is called due to the __cleanup__ attribute being set. When you compile and execute the program, you should see the following output:

```
$> gcc -Wall cleanup_attribute_demo.c
$> ./a.out
Cleaning up
Final value: 5
```

Download [https:// echorand.me/site/notes/_downloads/cleanup_attribute_demo.c]cleanup_attribute_demo.c.

Next, I will present a hopefull more useful example.

Cleaning up temporary files

In your programs, you may need to create one or more temporary files for some reason. Most likely, you would want to remove them after your program exits. Defining a __cleanup__ attribute on the FILE * variable (assuming stream I/O) and setting it to an appopriate cleanup function sounds like something which could be put to good use. We don't have to manually call the cleanup function.

Here is the program:

```
/* Demo code showing the usage of the cleanup variable
   attribute. See:http:// gcc.gnu.org/onlinedocs/gcc/Variable-Attributes.html
*/

/* Defines two cleanup functions to close and delete a temporary file
   and free a buffer
*/

# include <stdlib.h>
# include <stdio.h>

# define TMP_FILE "/tmp/tmp.file"

void free_buffer(char **buffer) {
    printf("Freeing buffer\n");
    free(*buffer);
}

void cleanup_file(FILE **fp) {
    printf("Closing file\n");
    fclose(*fp);

    printf("Deleting the file\n");
    remove(TMP_FILE);
}

int main(int argc, char **argv) {
    char *buffer __attribute__ ((__cleanup__(free_buffer))) = malloc(20);
    FILE *fp __attribute__ ((__cleanup__(cleanup_file)));

    fp = fopen(TMP_FILE, "w+");

    if (fp != NULL)
        fprintf(fp, "%s", "Alinewithnospaces");

    fflush(fp);
    fseek(fp, 0L, SEEK_SET);
    fscanf(fp, "%s", buffer);
    printf("%s\n", buffer);

    return 0;
}
```

The above program creates a temporary file in the location specified by TMP_FILE, writes a line of text with no spaces, resets the file pointer to the beginning and reads it back. In line no.32, I declare a variable fp of type FILE* and define the __cleanup__ attribute such that the function cleanup__file will be called upon the return of the main() function. This function closes the file and also deletes it from the file system. When you run your program, you should see the following output:

```
Alinewithnospaces
Closing file
Deleting the file
Freeing buffer
```

If you check the existence of the file specified by TMP_FILE, you will see that it doesn't exist. Note how I also use define the __cleanup__ attribute on the variable, buffer to automatically free memory as well.

Download [https:// echorand.me/site/notes/_downloads/cleanup_tempfile.c]cleanup_tempfile.c.

Resources

I first came across this attribute while reading Understanding and Using C Pointers (Link to my review) where the author discusses RAII. Take a look at how the RAII_VARIABLE macro is defined using the __cleanup__ attribute (In case you didn't know, ## is concatenation in C macros).
Please enable JavaScript to view the comments powered by Disqus.

---

https:// world.hey.com/basic70/cleaner-c-code-using-cleanup-8d505d3c

Cleaner C code using cleanup()

August 7, 2025

Recently, I found the cleanup(func) construct in C. This is an attribute that can be set on variables, and means that when the variable leaves its current scope, the given function is called. It is therefore somewhat similar to a destructor, even though it is set on variables and not on

types.

My C functions currently often have the following structure.

```
T1 func1(T2 arg1, ...) {
    T3* data = T3_create_data(arg1, ...); // step 1
                                          // process 'data' in some way... // step 2
    T1 result = data->result; // step 3
    T3_release(data); // step 4
    return result; // step 5
}
```

As I write steps 1 and 4 first when creating a new function, I basically never have any memory leaks. However, step 3 is annoying, as I must repeat type name T1. Also, if the processing must be terminated at some point during step 2, the code still needs to reach step 4 so the resources used by 'data' can be released. This requires either a "goto theEnd;" and a label at the beginning of step 3 or 4, or an ever increasing number of indentations and more complex logic. Neither one is particularly pretty. It typically also forces 'result' to be non-const, which is annoying.
By using cleanup(), the code becomes clearer, and the temporary variable in step 3 can be removed.

```
T1 func2(T2 arg1, ...) {
    T3* attribute((cleanup(T3_release))) data = // step 4
        T3_create_data(arg1, ...); // step 1
                                   // process 'data' in some way... // step 2
    return data->result; // step 3+5
}
```

When browsing through the code it's indeed a bit more difficult to find the 'T3_release' here. Still, the function that allocates data and the one that releases it are mentioned right next to each other. Also, the rest of the code is free to return from the function at any point.
A proper destructor set on the type would handle the case when 'data' is a variable on the stack, and therefore absolutely will die at the end of the block. However, the cleanup() construct also handles variables on the heap, and makes it obvious whether the data is temporary or will stay alive referenced by some other (global) variable. Potentially this could also be used by the compiler to allocate this data in a special section of the heap, for reduced memory fragmentation and/or increased speed.
In my code, sometimes step 1 takes a lock, which is then released in step 4. With some preprocessing magic I can now define the convenient block_lock() macro, as below. By using __COUNTER__ the same block can be protected by multiple locks (which of course must always be taken in the same order) without any problems.

```
#define mutex_merge(a,b) a##b
#define mutex_varname(a) mutex_merge(mutex_tmpvar_,a)
#define block_lock(lock) \
  mutex_t* __attribute((cleanup(mutex_unlock_ptr))) mutex_varname(__COUNTER__) = lock; \
  mutex_lock(lock)
```

In the function pattern above, steps 1+4 now simply become "block_lock(lockVariable);", and then the rest of the block is protected from other threads. A fun exception is when step 5 actually does something more than just a return, and that something must run without the lock already being taken.
Update: There are at least two important situations to watch out for when using this construct.
First, when the function is still using goto. Using goto is fine as such, you just cannot jump over the declaration of a variable that uses the cleanup() attribute. The result is that the variable will never be initialized, but the cleanup code will still be run when the block ends.
The second case is when a variable is declared within a switch statement. If this is between a pair of curly braces, fine. Otherwise the cleanup code will always be run when the switch statement ends, regardless of where it entered.

---

https:// hackerbikepacker.com/kernel-auto-cleanup-1

Linux Kernel Development - Automatic Cleanup 1/2
Jun 3, 2024

One of the most common criticisms of the C programming language is that dynamically allocated objects are not automatically released. And those who say this are right: memory leaks are a very common issue in C code, including the Linux kernel. Does that mean that C is useless, and the whole kernel should be rewritten in Rust as soon as possible? Definitely not, and even though some code is being rewritten in Rust, the great majority of the new code added with every release is still in C, and that will not change any soon. Instead, we should try to mitigate current pitfalls with new solutions... or simply start using the existing ones, like the Linux kernel recently did.

1. Background: underutilized cleanup compiler attribute
   Note: this section paraphrases/plagiarizes/summarizes some code from include/linux/cleanup.h as well as this article by Jonathan Corbet on LWN.net, which I strongly recommend. Here I will just digest the key points and add some code snippets for complete noobs :wink:

   Both GCC and Clang support a variable attribute called cleanup, which adds a "callback" that runs when the variable goes out of scope. If you never used compiler attributes, it is as simple as placing __attribute__ with the required attribute inside brackets right after the variable declaration. The cleanup attribute expects a function that takes a pointer to the variable type:

```
void cleanup_function(int *foo) {}
int foo __attibute (__cleanup__(cleanup_fuction));
```

Whenever foo goes out of scope, cleanup_function will be called. Here is a very simple example you can try and tweak:

```
/* cleanup.c */
#include <stdio.h>
#include <stdlib.h>

void cleanup_function(int *foo) {
    printf("foo: I do! The answer is %d.\n", *foo);
}

int main() {
    int foo __attribute ((__cleanup__(cleanup_function))) = 42;

    printf("What's the answer to everything?\n");
    printf("main: I don't know, bye!\n");

    return 0;
}
```

Let's compile and run:

```
$> gcc -o cleanup cleanup.c
$> ./cleanup
What's the answer to everything?
main: I don't know, bye!
foo: I do! The answer is 42.
```

If you can print a message, you can do more interesting stuff like freeing allocated memory, unlocking mutexes, and so on. I bet you are starting to grasp what we are aiming to achieve with this attribute.

Is that new magic? Well, the cleanup attribute exists in GCC since v3.3, which was released in 2003! Ok, then we have been using it in the Linux kernel for decades, right? Not really. It was first introduced in 2023 by Peter Zijlstra. Why? I don't know. Maybe no one thought about it before, maybe the community did not like the approach... If someone knows, please leave a comment.

You could argue that adding the attribute to every variable that requires some cleanup does not look beautiful, and it is laborious. Yet, I still believe it should be used way more often in C projects that already use compiler extensions. And as we will see in a bit, there are some ways to save a few strokes, especially in the long run.

In case you are wondering why I talked about the cleanup attribute, but then I used __cleanup__ instead: the underscores are optional, and they can save you from collisions in case an existing macro is called cleanup. You can read more about attribute syntax in the official documentation.

The next section is my attempt to help you understand how this attribute is used in the Linux kernel to automatically release memory, and what happens under the hood. Why would you want to know what happens under the hood? Because if you don't really know what you are doing, you will probably introduce bugs, and you will never be able to extend the mechanism to other use cases. As you will see in a bit, you can easily mess up, and new use cases for this mechanism will arise every now and then. Bear in mind that this feature is rather new in the kernel, and new macros are under review to extend its usage. Bleeding-edge development with 20-year-old compiler attributes!

2. Walkthrough: the __free() macro step by step

The cleanup attribute is really cool, and we are about to see an example where we will use it to free dynamically allocated memory. But typing the whole __attribute ((__cleanup__(cleanup_function))) is too tedious.

Instead, we could define a much shorter macro to call the cleanup function. I know that macros look like black magic for beginners, but don't worry: they are usually nothing more than syntactic sugar for more complex code.

For example, we could save some typing with a macro like the following __free():

```
// wherever __free() is used, replace it with this long expression
#define __free(func)    __attribute ((__cleanup__(func)))
```

Let's use our new macro to automatically free memory when a variable goes out of scope:

```
/* free1.c */
#include <stdio.h>
#include <stdlib.h>

#define __free(func)    __attribute ((__cleanup__(func)))

void cleanup_function(int **foo) {
    printf("Avoiding a memory leak ;)\n");
    free(*foo);
}

int main() {
    int *foo __free(cleanup_function) = malloc(10*sizeof(*foo));

    // Some (probably buggy ;)) code

    return 0;
```

```
}
```

I said before that clang also supports the cleanup attribute, didn't I? Let's see:

```
$> clang -o free1 free1.c
$> ./free1
Avoiding a memory leak ;)
```

Awesome! Are we done? Well, we still have to pass the cleanup function to our macro. Letting everyone use their own cleanup function for a given type does not make much sense, let alone in a huge project like the Linux kernel. Furthermore, no one wants to memorize the name of the cleanup function for every type. Offering a simple API that hides the cleanup mechanism would be more efficient. The API user could simply call __free(type), and the rest would be transparent. Let's add a new macro to generate the cleanup functions according to the type:

```
#define DEFINE_FREE(_name, _type, _free) \
        static inline void __free_##_name(void *p) {_type _T = *(_type *)p; _free; }
```

Wow, wow! What was that? Is that really new syntactic sugar? Yes, it is. Every time you use the DEFINE_FREE() macro, you have to pass a name to generate the cleanup function, the variable type, and the free mechanism. The name could be anything, but the variable type sounds reasonable, so we end up with a cleanup function called __free_type() like __free_int() or __free_foo(). We have used the handy ## preprocessing operator for that. The type is obviously the variable type we want to free, and the free mechanism is the code we want to execute in the cleanup function.

Some beginners might have found a different kind of black magic in the macro we just defined: void pointers. But again, don't worry. They are useful and actually not that complex. We will use a void pointer to pass any pointer (e.g. a pointer to the variable type), and avoid cumbersome double pointers like the int **foo we used in the last example. Give it a try with void *foo instead, and everything should work like before.

Now that we have our DEFINE_FREE() macro, we can adapt our previous __free() macro to get the generated cleanup function instead:

```
#define __free(_name)   __attribute ((__cleanup__(__free_##_name)))
```

And now, another example, this time with cleanup functions for int * and a slightly more complex type strcut foo *:

```
/* free2.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Some more complex type
struct foo {
    char *name;
};

static inline void foo_cleaner(struct foo *foo) {
    printf("Bye %s\n", foo->name);
    free(foo->name);
    free(foo);
}

/************** OUR "API" internals **************/
#define DEFINE_FREE(_name, _type, _free) \
        static inline void __free_##_name(void *p) {_type _T = *(_type *)p; _free; }

// Our cleaner for int *
DEFINE_FREE(int, int *, printf("Bye dynamic int = %d\n", *_T); free(_T))

// Our cleaner for struct foo *
DEFINE_FREE(foo, struct foo *, if(_T) foo_cleaner(_T))
/*************************************************/

/******************* OUR "API"  *******************/
#define __free(_name)   __attribute ((__cleanup__(__free_##_name)))
/*************************************************/

int main() {
    int *pint __free(int) = malloc(sizeof(*pint));
    struct foo *pfoo __free(foo) = malloc(sizeof(*pfoo));

    *pint = 42;
    pfoo->name = strdup("Javier Carrasco");

    // Some code

    return 0;
}
```

This time we have programmed the cleanup in two different ways: directly in the DEFINE_FREE() macro for int * types (a simple free), and as a separate function for struct foo *, because a bit more code was needed. Alright, let's see if it works:

```
$> clang -o free2 free2.c
$> ./free2
Bye Javier Carrasco
Bye dynamic int = 42
```

Note that the order in which the cleanup functions are executed is not random. In fact, they are executed in the reverse order in which the variables with the cleanup attribute are declared in the scope. This is not relevant here, but it's something to keep in mind.

If you understood everything I told you in this section, congratulations: the macros I used were taken from the Linux kernel*, and you don't have to learn the same thing twice. If you are still trying to understand how attributes and macros work, don't panic: a second, more thorough read will definitely help. Just don't give up!

* The __free() macro from the kernel is not exactly like mine. Instead, an intermediate step is used to get the compiler-specific syntax for the cleanup attribute, increasing flexibility. What you will actually find in the kernel is #define __free(_name) __cleanup(__free_##_name), where __cleanup(__free_##_name) is defined as follows for Clang (in compiler-clang.h): #define __cleanup(func) __maybe_unused __attribute__((__cleanup__(func)))

3. Return valid memory, but keep on using auto cleanup!
What if the allocated memory is required later in the code? Will the cleanup attribute free memory anyway? Of course, it will, as soon as the variable goes out of scope. Does that mean that we can't use that attribute in that case? No, someone has thought about it already.

We can use the cleanup feature like we did until now to automatically free memory in the error paths, and use a specific macro to return the pointer instead of freeing it. Yet another macro? Yes, and we will see a few more in the next chapter. Get used to them and stop complaining, because they are here to stay, and actually they are really useful.

We have two simple macros at our disposal: no_free_ptr() and return_ptr(). The first one uses a second pointer to store the memory address, setting the original pointer to NULL*. Why? Because the cleanup is going to step in anyway at the end of the variable's scope. We are not inhibiting the cleanup, at least not explicitly, but the original variable (pointer) will be NULL and nothing will be done. The second variable (pointer) does not use the cleanup attribute, and the memory will not be freed. The return_ptr() macro is just a convenient return no_free_ptr(p).

If you have a function that returns a pointer to some memory, but only if nothing goes wrong, a simple return_ptr(your_pointer) at the end will be enough, and the error paths where the memory should be freed will be covered by the __free() like before. Of course, the caller will be then in charge of freeing the memory when it is no longer needed. If you forget the return_ptr() macro, you will be returning freed memory, which is definitely bad.

no_free_ptr() can be used on its own, and you will find several examples in the kernel. For example this patch uses that macro because memdup_user() frees memory itself if something goes wrong and returns a PTR_ERR() (error pointer). In that case, you don't want to free the error pointer (big mistake), but propagate the error without giving up automatic deallocation otherwise.

Update: as Harshit Mogalapalli pointed out (thanks for your feedback!), there's this recent patch by Dan Carpenter that fixes such potential misuse for objects that are directly released with kfree or kvfree by checking too if the pointer is an error pointer (if (!IS_ERR_OR_NULL(_T) instead of if(_T)). As you can see, this area is under heavy development!

* Technically, no_free_ptr() uses a second macro for it, called __get_and_null_ptr(). This is where the real black magic happens by means of two GCC extensions: ({}) and __auto_type. The first extension is used to define an expression that "returns" a value (__val), and the second one, more obvious, increases flexibility to work with different pointers. This second macro is only used internally, and as I mentioned, it just moves the content of the pointer with automatic cleanup to a new variable. You will find __get_and_null_ptr() in include/linux/cleanup.h as well.

```
#define __get_and_null_ptr(p)           \
    ({ __auto_type __ptr = &(p);        \
       __auto_type __val = *__ptr;      \
       *__ptr = NULL; __val; })
```

The Linux kernel uses GCC extensions A LOT.

4. Initialize your variables, and fear any "goto"
Some mistake I have seen several times when beginners use this feature for the first time is variable declaration without initialization (think of RAII in other programming languages like C++). What does that mean? Look at the following snippet:

```
{
    struct foo *pfoo __free(foo);
    goto fail;
    pfoo = malloc(sizeof(*pfoo));
fail:
    return -1;
}
```

That goto fail is jumping over the variable initialization! When the end of the scope is reached, some random address will be freed! We should have initialized the variable when we declared it. If there was no value for the initialization, we should initialize it to NULL. A more subtle case is the following:

```
/* fail.c */

#include <stdio.h>
```

```
#include <stdlib.h>

void cleanup_function(int *foo) {
    printf("I should not run, or at least print 42... %d.\n", *foo);
}

int main() {
    goto fail;
    int foo __attribute ((__cleanup__(cleanup_function))) = 42;
fail:
    return 0;
}
```

Apparently, we have initialized the variable. But it should not matter, because the execution will jump over it, right? Unfortunately, this is not how things work. Let's compile and run:

```
$> gcc -o fail fail.c
$> ./fail

I should not run, or at least print 42... 0.
# NEVER assume that uninitialized variables are set to 0!
```

Shit! What happened? The goto does not change the scope/reach of the variable (the block where it is declared), but the initialization does take place where we initialize them. Therefore, declaring a variable that uses the cleanup attribute will basically force us to refactor any previous goto in the same scope, and no new goto instructions should be added before the variable declaration. Thus, declaring them at the beginning of the scope and setting them to NULL when the desired initialization is not possible (because the value is still not available at that point) could avoid such problems.

By the way, that issue was obvious to me because clangd complained about it:
[img]clangd goto error I wonder why LSPs are not mandatory...

The error message is a bit cryptic, but if we compile the same program with clang, we will get more feedback:

```
$> clang -o fail fail.c
fail.c:12:2: error: cannot jump from this goto statement to its label
        goto fail;
        ^
fail.c:13:6: note: jump bypasses initialization of variable with __attribute__((cleanup))
        int foo __attribute ((__cleanup__(cleanup_function))) = 42;
            ^
1 error generated.
```

Am I suggesting that clang (16.0.6) is better than gcc (13.2.0)? In this particular case... yes, by far! :laughing:

This would have not happened if we had used a return instead of a goto. Why? Because the compiler would have reached the return without finding the variable declaration, and it would have not been included in that branch. Moral: if you are using a goto to reach a label that only returns a value, simply return the value.

Jumping around is fun when you are a kid, but good programmers try to hold back. Use goto when you really need it, but be careful, or you will twist your ankle!

5. Why Rust then?
   If we can use those macros to automate cleanups, we are making C completely safe! Let's remove Rust from the kernel and keep on improving our beloved C. Well, there are still many other scenarios where nasty bugs can be (very) easily programmed in C. Ask your favorite chat AI: you will get a huge list of possible mistakes.

   Furthermore, the cleanup attribute and the macros based on it can also be misused, and I will give you a couple of real examples... first of all, I believe that refactoring stable code is not always a good idea, even to increase code safety "for future modifications". We have had a case in the LFX Mentorship where a mentee refactored some code to add the __free() macro without adding the required return_ptr(), and that mistake kept the kernel from booting. The patch had to be reverted, and several people had to get involved to solve the issue as soon as possible. This is how bad refactoring existing code can go. No one notices until it's too late and shit hits the fan. Don't you believe me? Then take a look at this recent fix for a regression while refactoring some code in the IIO subsystem to include the cleanup feature. This time, an experienced kernel developer (way, way more experienced than me) introduced the bug, and apparently, this regression could even cause skin burns! :fire: Moral of the story: be cautious because anyone can mess up, test your patch as much as you can, and favor new code or real bugs to introduce this feature until you master it.

   On the other hand, I am sure that there are still many bugs hidden in the code that could be avoided with this feature. Actually, I fixed myself this memory leak in a driver from the cpufreq subsystem by adding a simple __free()*. It will take a while until everyone gets used to it, but all things considered, I would say that introducing the cleanup attribute was a good idea.

   Anyway, competition is always good (like having GCC and Clang competing to become your preferred C compiler), Rust has been a reality in many successful projects for years, and (at least in theory) it could make kernel development more appealing for new generations. So let's keep on increasing safety for C while letting Rust find its own way into the kernel.

   * If you find a similar bug, consider splitting the fix into two patches: one with the "traditional" fix (e.g. adding the missing kfree()), and a second one with the cleanup attribute. This feature is

not available in many stable kernels, and the first patch will be easier to backport.

6. Why 1/2?

This article is getting a bit long, and there are still many macros and use cases I did not mention: classes (yes, classes in C!), scoped loops, usage with mutexes... Some of that stuff is so new in the kernel that there is still not much real code to use as an example, but fortunately enough to learn the key concepts and stay up-to-date when it comes to Linux kernel development.

That's all for today. Please send me a message if you find any inaccuracy, and I will fix it as soon as possible. Stay tuned and don't miss the next chapter, whenever I get it finished...

[**1][https:// hackerbikepacker.com/kernel-auto-cleanup-2]Next -->

[**1]https:// hackerbikepacker.com/kernel-auto-cleanup-2

Linux Kernel Development - Automatic Cleanup 2/2
Jun 17, 2024

This second and last episode about the automatic cleanup mechanisms in the Linux kernel covers the concept of classes, macros being used to automatically release resources like mutexes, and the ongoing work to increase its coverage. If you know nothing or very little about the cleanup compiler attribute and how it works, please take a look at the first episode, and then get back to this one.

The goal is to ensure that even beginners with only basic knowledge of C are not left behind, while also providing useful information for experienced developers who may not be familiar with the topics covered in this article. I have included very basic examples for the beginners, and slightly more advanced mechanisms for the experienced developers. Please pick whatever you find useful for your level, and if you notice anything I could improve to make things clearer, please provide your feedback!

1. Classes in the kernel

Don't get scared (or too excited), the Linux kernel has not adopted C++. Classes in the Linux kernel are not exactly what you might know from other programming languages: they are built upon the concepts we have seen so far to increase resource management automation.

Think of them as objects that provide a constructor and a destructor. If you know nothing about classes, constructors, and destructors, it does not matter: we are simply going to define a new type (typedef in C) out of an existing one, and create an automatic initializer for new objects of that type (the constructor). The destructor is the function for the automatic cleanup we have already learned. I will use a rather loose terminology throughout this article to make sure that any pure C programmer understands everything at first (or maybe second) glance.

At the moment of writing, classes don't support fancy features you might know from other programming languages, and they might never do, so keep calm and continue thinking in C terminology.

How do we define a class, and how do we declare objects of the class? Once again, the macros defined in include/linux/cleanup.h are the way to go. In particular, we are going to analyze DEFINE_CLASS and CLASS:

```
// Macro to define classes:
#define DEFINE_CLASS(_name, _type, _exit, _init, _init_args...)         \
typedef _type class_##_name##_t;                                        \
static inline void class_##_name##_destructor(_type *p)                 \
{ _type _T = *p; _exit; }                                               \
static inline _type class_##_name##_constructor(_init_args)             \
{ _type t = _init; return t; }

// Macro to declare an object of the class:
#define CLASS(_name, var)                                               \
        class_##_name##_t var __cleanup(class_##_name##_destructor) =   \
                class_##_name##_constructor
```

Before you give up: the macros are easier than they look, and in the end they do nothing more than defining a class and instantiating objects of that class. Not only that, you will find the following trivial example in the same header file, which I am going to digest for you:

```
DEFINE_CLASS(fdget, struct fd, fdput(_T), fdget(fd), int fd)

CLASS(fdget, f)(fd);
if (!f.file)
    return -EBADF;

 // use 'f' without concern
```

This example shows how to use a class to automatically manage file references, but it could be any other resource you would like to get and then release, like a mutex (more about it in the next section). fdget is the class name, struct fd is the type we are going to use for our typedef, _exit is the destructor, _init the constructor, and _init_args... the arguments we want to pass to the constructor.

In this case, a class named fdget is defined, which is nothing more than a struct fd (some struct defined somewhere, it doesn't matter) that calls fdget() with int fd as the argument when an object

of the class is created. When the object (variable) goes out of scope, fdput() is automatically called.

If you are still lost, I got you. Let's expand the CLASS macro based on what was passed to DEFINE_CLASS by simply replacing the parameters in the macro definitions with the ones used for the example:

```
/* DEFINE_CLASS(fdget, struct fd, fdput(_T), fdget(fd), int fd) expands to: */
// 1. typedef:
typedef struct fd class_fdget_t;

// 2. destructor:
static inline void class_fdget_destructor(struct fd *p) { struct fd _T = *p; fdput(_T); }

// 3. constructor:
static inline struct fd class_fdget_constructor(int fd) { struct fd t = fdget(fd); return t; }

/* CLASS(fdget, f)(fd) expands to: */
class_fdget_t f __cleanup(class_fdget_destructor) = class_fdget_constructor(fd);
// That looks similar to what we saw in the first episode, doesn't it?
// variable declaration + cleanup macro, and some initialization.
```

We have simply declared an object called f of the fdget class, whose type is class_fdget_t (basically a struct fd with new superpowers), by means of the CLASS macro. We have initialized f with our constructor that received fd as the argument for the initialization. The fd file reference will be automatically released when f goes out of scope with the mechanism we already know (here the destructor function), so we don't have to worry about forgetting a call to fdput() every time f goes out of scope.

In principle, you will seldom (if ever) call DEFINE_CLASS yourself. The class definition is only made once, and then all users can simply use CLASS to declare the objects and used them as required.

Still not clear? I am running out of ideas, but I may have one more for total noobs. Let's rewrite the program from the first episode to include the new macros and hence support classes. Feel free to copy the code and experiment with it:

```
/* class.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct foo {
    char *name;
};

static inline void foo_cleaner(struct foo *foo) {
    printf("Bye %s\n", foo->name);
    free(foo->name);
    free(foo);
}

static inline struct foo *foo_builder(const char *name) {
    struct foo *foo = malloc(sizeof(*foo));

    foo->name = strdup(name);
    printf("Hello %s\n", foo->name);

    return foo;
}

/************** OUR "API" internals **************/
// Macro to define classes:
// 1. typedef
// 2. destructor
// 3. constructor
#define DEFINE_CLASS(_name, _type, _exit, _init, _init_args...)         \
typedef _type class_##_name##_t;                                        \
static inline void class_##_name##_destructor(_type *p)                 \
{ _type _T = *p; _exit; }                                               \
static inline _type class_##_name##_constructor(_init_args)             \
{ _type t = _init; return t; }

// Definition of the foo class
DEFINE_CLASS(foo, struct foo *,  foo_cleaner(_T), foo_builder(name), const char *name)
// __free() is used in the CLASS macro to set the destructor
#define __free(_name)   __attribute ((__cleanup__(_name)))

/****************** OUR "API"  ******************/
// Macro to declare an object of the class:
#define CLASS(_name, var)                                               \
        class_##_name##_t var __free(class_##_name##_destructor) =      \
                class_##_name##_constructor
/************************************************/

int main() {
        CLASS(foo, f)("Javier Carrasco");
```

```
        printf("Nice to meet you, %s\n", f->name);

        return 0;
}
```

Now let's compile and run the program. I will use Valgrind to make sure that there are no memory leaks:

```
$> clang -g -o class class.c
$> valgrind ./class
==8111== Memcheck, a memory error detector
==8111== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==8111== Using Valgrind-3.21.0 and LibVEX; rerun with -h for copyright info
==8111== Command: ./class
==8111==
Hello Javier Carrasco
Nice to meet you, Javier Carrasco
Bye Javier Carrasco
==8111==
==8111== HEAP SUMMARY:
==8111==     in use at exit: 0 bytes in 0 blocks
==8111==   total heap usage: 3 allocs, 3 frees, 1,048 bytes allocated
==8111==
==8111== All heap blocks were freed -- no leaks are possible
==8111==
==8111== For lists of detected and suppressed errors, rerun with: -s
==8111== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

No memory leaks, 0 errors, and we got the expected output. Awesome!

By the way, why did Valgrind find 3 allocs and 3 frees when there are only two calls to malloc() and two calls to free()? I am sure that you can figure it out :wink:

(Leave this your second reading if you are struggling) If you open the cleanup header file, you will find another macro between DEFINE_CLASS and CLASS called EXTEND_CLASS. It is used to get a new class with its own constructor out of an existing class, and at the moment of writing is not used out of cleanup.h. That does not mean that it is dead code, as we will see in the next section, when conditional mutexes are created out of blocking mutexes.

Great, now that we know how classes work in the Linux kernel, let's see a fairly common case: the mutex class.

2. Automatic mutex handling

Although it may have great potential, you will not find many real cases where DEFINE_CLASS is used in the kernel (there are currently 6 calls outside cleanup.h). Nonetheless, its usage for mutex handling is becoming pretty common in several subsystems, even though the macros are not called like we did in the previous section. Surprise, surprise: there are some more macros in cleanup.h, and the ones we are going to see now are specific for mutexes. Don't panic, they are simple wrappers around the macros we just saw:

   1. How do we define a new mutex (from now on, guard) class? With DEFINE_GUARD:

```
#define DEFINE_GUARD(_name, _type, _lock, _unlock) \
DEFINE_CLASS(_name, _type, if (_T) { _unlock; }, ({ _lock; _T; }), _type _T); \
static inline void * class_##_name##_lock_ptr(class_##_name##_t *_T) \
{ return *_T; }
```

      Once again, you will seldom define a class for mutexes yourself. There will probably be one already that suits your needs. And if not, you will only have to use DEFINE_GUARD once for the required type.

   2. How do we declare a new object of the mutex class? With guard:

```
#define guard(_name) \
 CLASS(_name, __UNIQUE_ID(guard))
```

   As you might have guessed, there is already a class definition for the regular mutex struct mutex (in include/linux/mutex.h). That means that we can simplify the use of that mutex for a given scope with a single line like this: guard(mutex)(&my_mutex);, which will expand to what we know: a variable declaration + cleanup macro, and some initialization. In this case, the initialization is a mutex lock, and the cleanup is a mutex unlock. Exactly what we want.

   You will find several examples in the kernel, for example in the IIO subsystem, where I used this mechanism for the hdc3020 and veml6075 drivers. Not releasing a taken mutex is terrible, and not that difficult to forget if the scope has multiple exit points (return, goto). The guard macro is short and simple, it works as expected, and having a reliable mechanism to avoid deadlocks gives peace of mind. By the way, you will find other macros that automate cleanups in that subsystem, because as we will see later, the maintainer is the author of some of them.

   Before we move to the next section, I would like to mention another guard macro that you will find in the kernel, and is really nice: scoped_guard. It works like guard, but instead of applying for the current scope, it only applies for the immediate compound statement i.e. the next instruction or instructions within curly brackets:

```
scoped_guard(mutex)(&my_mutex)
```

```
    // do something

scoped_guard(mutex)(&my_mutex) {
        // do a bunch of things
}
```

Often you want to release the mutex as soon as the resource is available again, and there is no need to wait until the end of the current scope is reached. There has already been some refactoring to use scoped guards where the regular ones were used. Be wise and use the right tool for the job!

(Leave this for your second reading if you are struggling) DEFINE_COND_GUARD is a wrapper around EXTEND_CLASS to define conditional mutexes, which are out of the scope of this article. In a nutshell, and simplifying things, they return if the lock operation did not work instead of sleeping until the operation is permitted. There is also the scoped_cond_guard macro to work with scoped conditional mutexes, so you have multiple choices. Moreover, some subsystems use their own wrappers for specific tasks they carry out on a regular basis. But once you understand the basic macros, everything else is just syntactic sugar.

It might be worth mentioning that there are some additional macros (DEFINE_LOCK_GUARD_*, you will find them at the end of cleanup.h) to create guards with types for locks that don't have a type themselves (e.g. RCU, where DEFINE_LOCK_GUARD_0, that receives no type argument, is used for that purpose. See rcupdate.h), or require a pointer to access/manipulate some data for the locking/unlocking (e.g. interrupt flags in spin_lock_irqsave).

3. Other cleanup macros in the Linux kernel

If you liked the scoped_guard we saw in the previous section, you are going to enjoy the following macros. It is fairly common in the Linux kernel that the lifetime of an object (whatever the object is, think of some allocated resources) is given by a refcount (to count references to the object). When that refcount reaches zero, no one needs the object anymore, and it can be released. It is often the case, that an iterator (e.g. a pointer) is used in a loop to iterate over a series of objects, incrementing the object's refcount as long as it is in use to ensure its availability, and decrementing it at the end of the iteration. That sounds like a good fit for some kind of custom-scoped magic, doesn't it?

A good example could be the iteration over nodes from a device tree. I have already talked about device trees in other articles like this one, so please check it out if you know absolutely nothing about them. I try to make the explanation generic anyway, so you can keep on reading.

The refcount handling in loops has been automated long time ago by keeping the refcount of the current object greater than zero as long as it is in use, repeating that operation until there are no more objects to iterate over, but there is a trick: that automation only works as long as the end of the loop is reached. But often we don't want to iterate over all possible nodes, right? We could have found the one we were looking for, or even want to exit after some error occurs.

In those cases, the object's refcount must be manually decremented to reach its original value before the iteration, and avoid a memory leak by preventing the object's refcount to ever be zero (e.g. after removing a module). Manual intervention is always dangerous. Actually, I have fixed multiple bugs in such loops while writing this article by just looking for examples, like this one, this one, or this one. Even though a wrong refcount handling does not directly lead to a leak as long as the object stays in use, what will happen with that object in the future is often difficult to control and predict. By now you should be convinced that when used with care, the cleanup attribute is a great addition to the kernel.

The new macros to handle such iterative tasks have the same name as the old ones, plus the _scoped suffix. Their implementation is also similar, and by now you should understand the difference without any explanation from my side. Let's see for example device_for_each_child_node() and device_for_each_child_node_scoped(), which was recently added by Jonathan Cameron (IIO maintainer):

```
#define device_for_each_child_node(dev, child)                          \
        for (child = device_get_next_child_node(dev, NULL); child;      \
             child = device_get_next_child_node(dev, child))

#define device_for_each_child_node_scoped(dev, child)                   \
        for (struct fwnode_handle *child __free(fwnode_handle) =        \
                device_get_next_child_node(dev, NULL);                  \
             child; child = device_get_next_child_node(dev, child))
```

Exactly, whatever is going on within that loop, the __free() macro is the key to automatically release that fwnode_handle child after every iteration, no matter what that child is. As I said, it has something to do with inner nodes of some device from a device tree, but the important thing is to understand the pattern. Of course, there must be a DEFINE_FREE for the fwnode_handle to define the cleanup function, but that is also something we have already learned in the first episode.

Let's see how you can easily mess up with the non-scoped version of these macros, and how easy the fixes with the scoped macros can be (not always, though... case by case!). In this case, I used for_each_child_node_scoped() to fix this memory leak in the sun50i cpufreq driver:

```
static bool dt_has_supported_hw(void) {
    bool has_opp_supported_hw = false;
    struct device *cpu_dev;

    cpu_dev = get_cpu_device(0);
    if (!cpu_dev)
        return false;

    struct device_node *np __free(device_node) = dev_pm_opp_of_get_opp_desc_node(cpu_dev);
```

```
    if (!np)
        return false;

    for_each_child_of_node(np, opp) {
        if (of_find_property(opp, "opp-supported-hw", NULL)) {
            has_opp_supported_hw = true;
            break; // early exit with no fwnode_handle_put(opp) to decrement refcount!
        }
    }

    return has_opp_supported_hw;
}
```

Attach _scoped to for_each_child_of_node, and the bug is fixed. And if another early exit is added in the future, we are still safe. Is that not great?

4. Ongoing work

Even though the scoped versions of a number of such macros are already available in the mainline kernel, there are still others that only have the non-scoped variant. They are waiting for a use case where it makes sense (e.g. fixing a bug where manual release is missing), and therefore the number of members of the _scoped family will probably grow in the next months/years. Similarly, there are many objects that don't have their own __free() or an associated class, and I bet that some of them could profit from that as well.

I sent myself this patch (still being discussed, mainly for other reasons than the cleanup mechanism itself) a few days ago to fix a bug in the hwmon ltc2992 driver where the current fwnode_for_each_available_child_node is not correctly used. The people who were involved in the design and implementation of the automatic cleanup mechanisms paved the path, and now we can profit from it to add new use cases like this one. Props to them!

The Linux Kernel Mentorship Program from the Linux Foundation is (among many others) also pushing the use of the __free() macro forward by refactoring existing code where error paths increase the risk of missing a memory deallocator. This task was proposed by Julia Lawall during a mentoring session about Coccinelle, and several patches have already been accepted. I must admit that some subsystems were more enthusiastic than others, but in general they are making their way to the kernel.

To be honest, I understand why some maintainers are pushing back. I refactored some easy code myself to be able to review my mentees' work, and since then I restricted myself to adding the feature to new code or where it fixes real bugs and the improvement is undeniable. Some refactoring where a bunch of goto jumps vanishes is in my opinion a good patch to apply, but we have already seen that there are many things to consider to avoid regressions. A few maintainers still don't feel comfortable with the auto cleanup stuff (some were completely unaware of it), and they could miss some potential regressions when reviewing patches. If you want to do some refactoring anyway, please be very careful, and make sure you understand the internals. I hope my articles helped a bit!

Apart from the ongoing work I mentioned, I suppose there is much more being discussed in multiple mailing lists. I will let you investigate on your own. Maybe you are even working on some extensions I don't know... I would love to hear about it!

Hopefully the ongoing work will continue increasing code quality and above all, delivering a better kernel to the end user.

---

https:// github.com/taeber/cwith/issues/1

Perhaps consider using __attribute__((cleanup()) on GCC? #1

GCC (& Clang) have a variable attribute that runs a function when a variable goes out of scope.

This allows for writing a with that handles return statements and such properly.

A quick example of the underlying concept:

```
#include "stdio.h"

void write_samples(const char *path, int numSamples, const float *samples) {
    // Mild annoyance - cleanup takes a pointer to the variable.
    // Which you need to plumb through if the cleanup function takes it by value.
    void cleanup(FILE **f) {fclose(*f);}

    // nested functions are another GNUism. In C++11 or later you could use a lambda instead.
    FILE *fp __attribute__((cleanup(cleanup))) = fopen(path, "wb");
    fprintf(fp, "%d\n", numSamples);
    if (ferror(fp))
        return;
    for (int i = 0; i < numSamples; i++)
        fprintf(fp, "%f\n", samples[i]);
}
```

(In C++ replace the nested function with a lambda.)
In C++ this even handles exceptions sanely (read: runs the cleanup as part of exception handling).

---

https:// nibblestew.blogspot.com/2016/07/comparing-gcc-c-cleanup-attribute-with.html

Comparing GCC C cleanup attribute with C++ RAII
Wednesday, July 20, 2016

I recently got into a Twitter discussion about the GCC cleanup extension in C vs plain old C++ (and Rust, which does roughly the same). There were claims that the former is roughly the same as the latter. Let's examine this with a simple example. Let's start with the following C code.

```
Res *func() {
    Res *r = get_resource(); /* always succeeds for simplicity. */
    if(do_something(r)) {
        return r;
    } else {
        deallocate(r);
        return NULL;
    }
}
```

This is straightforward: get a resource, do something with it and depending on outcome either return the resource or deallocate it and return NULL. This requires the developer to track the life cycle of r manually. This is prone to human errors such as forgetting the deallocate call, especially when more code is added to the function.

This code is trivial, but still representative of real world code. Usually you would have many things and resources going on in a function like this but for simplicity we omit all those. The question now becomes, how would one make this function truly reliable using GCC cleanup extension. By reliability we mean that the compiler must handle all cases and the developer must not need to write any manual management code.

The obvious approach is this:

```
Res *func() {
    Res *r __attribute__((cleanup(cleanfunc)));
    r = get_resource();
    if(do_something(r)) {
        return r;
    } else {
        return NULL;
    }
}
```

This does not work, though. The cleanup function is called always before function exit, so if do_something returns true, the function returns a pointer to a freed resource. Oops. To make it work you would need to do this:

```
Res *func() {
    Res *r __attribute__((cleanup(cleanfunc)));
    r = get_resource();
    if(do_something(r)) {
        Res *r2 = r;
        r = NULL;
        return r2;
    } else {
        return NULL;
    }
}
```

This is pointless manual work which is easy to get wrong, thus violating reliability requirements. The only other option is to put the deallocator inside the else block. That does not work either, because it just replaces a call to deallocate with a manually written setup to call the deallocator upon scope exit (which happens immediately).

This is unavoidable with the cleanup attribute. It is always called. It can not automatically handle the case where the life cycle of a resource is conditional. Even if it were possible to tell GCC not to call the cleanup function, that call must be written by hand. Conditional life cycles always require manual work, thus making it unreliable (as in: a human needs to analyze and write code to fix the issue).

In C++ this would look (roughly) like the following.

```
std::unique_ptr<Res> func() {
    std::unique_ptr<Res> r(new Resource());
    if(do_something(r)) {
        return r;
    } else {
        return nullptr; // or throw an exception if you prefer
    }
}
```

In this case the resource is always handled properly. It is not deallocated in the true branch but is deallocated in the false branch. No manual life cycle code is needed. Adding new code to this function is simple because the developer does not need to care about the life cycle of r, the compiler will enforce proper usage and does it more reliably than any human being.

Conclusions

GCC's cleanup attribute is a great tool for improving reliability to C. It should be used it whenever possible (note that it is not supported on MSVC so code that needs to be portable can't use it). However the cleanup attribute is not as easy to use or reliable as C++'s RAII primitives.

---
https:// sources.debian.org/src/linux/6.12.38-1/include/linux/compiler_attributes.h/

File: compiler_attributes.h

package info (click to toggle)
linux 6.12.38-1
    * links: PTS, VCS
    * area: main
    * in suites: trixie
    * size: 1,675,544 kB
    * sloc: ansic: 25,916,840; asm: 269,589; sh: 136,393; python: 65,219; makefile: 55,714; perl:
    37,750; xml: 19,284; cpp: 5,894; yacc: 4,927; lex: 2,939; awk: 1,594; sed: 28; ruby: 25

```
/* SPDX-License-Identifier: GPL-2.0 */
#ifndef __LINUX_COMPILER_ATTRIBUTES_H
#define __LINUX_COMPILER_ATTRIBUTES_H

/*
 * The attributes in this file are unconditionally defined and they directly
 * map to compiler attribute(s), unless one of the compilers does not support
 * the attribute. In that case, __has_attribute is used to check for support
 * and the reason is stated in its comment ("Optional: ...").
 *
 * Any other "attributes" (i.e. those that depend on a configuration option,
 * on a compiler, on an architecture, on plugins, on other attributes...)
 * should be defined elsewhere (e.g. compiler_types.h or compiler-*.h).
 * The intention is to keep this file as simple as possible, as well as
 * compiler- and version-agnostic (e.g. avoiding GCC_VERSION checks).
 *
 * This file is meant to be sorted (by actual attribute name,
 * not by #define identifier). Use the __attribute__((__name__)) syntax
 * (i.e. with underscores) to avoid future collisions with other macros.
 * Provide links to the documentation of each supported compiler, if it exists.
 */

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-alias-function-attribute
 */
#define __alias(symbol)                 __attribute__((__alias__(#symbol)))

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-aligned-function-attribute
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Type-Attributes.html#index-aligned-type-attribute
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html#index-aligned-variable-attribute
 */
#define __aligned(x)                    __attribute__((__aligned__(x)))
#define __aligned_largest               __attribute__((__aligned__))

/*
 * Note: do not use this directly. Instead, use __alloc_size() since it is conditionally
 * available and includes other attributes. For GCC < 9.1, __alloc_size__ gets undefined
 * in compiler-gcc.h, due to misbehaviors.
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-alloc_005fsize-function-att
ribute
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#alloc-size
 */
#define __alloc_size__(x, ...)          __attribute__((__alloc_size__(x, ## __VA_ARGS__)))

/*
 * Note: users of __always_inline currently do not write "inline" themselves,
 * which seems to be required by gcc to apply the attribute according
 * to its docs (and also "warning: always_inline function might not be
 * inlinable [-Wattributes]" is emitted).
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-always_005finline-function-
attribute
 * clang: mentioned
 */
#define __always_inline                 inline __attribute__((__always_inline__))

/*
 * The second argument is optional (default 0), so we use a variadic macro
 * to make the shorthand.
 *
 * Beware: Do not apply this to functions which may return
 * ERR_PTRs. Also, it is probably unwise to apply it to functions
 * returning extra information in the low bits (but in that case the
 * compiler should see some alignment anyway, when the return value is
 * massaged by 'flags = ptr & 3; ptr &= ~3;').
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-assume_005faligned-function
-attribute
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#assume-aligned
 */
```

```
#define __assume_aligned(a, ...)        __attribute__((__assume_aligned__(a, ## __VA_ARGS__)))

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html#index-cleanup-variable-attribute
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#cleanup
 */
#define __cleanup(func)                 __attribute__((__cleanup__(func)))

/*
 * Note the long name.
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-const-function-attribute
 */
#define __attribute_const__             __attribute__((__const__))

/*
 * Optional: only supported since gcc >= 9
 * Optional: not supported by clang
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-copy-function-attribute
 */
#if __has_attribute(__copy__)
# define __copy(symbol)                 __attribute__((__copy__(symbol)))
#else
# define __copy(symbol)
#endif

/*
 * Optional: not supported by gcc
 * Optional: only supported since clang >= 14.0
 *
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#diagnose_as_builtin
 */
#if __has_attribute(__diagnose_as_builtin__)
# define __diagnose_as(builtin...)      __attribute__((__diagnose_as_builtin__(builtin)))
#else
# define __diagnose_as(builtin...)
#endif

/*
 * Don't. Just don't. See commit 771c035372a0 ("deprecate the '__deprecated'
 * attribute warnings entirely and for good") for more information.
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-deprecated-function-attribu
te
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Type-Attributes.html#index-deprecated-type-attribute
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html#index-deprecated-variable-attribu
te
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Enumerator-Attributes.html#index-deprecated-enumerator-attribute
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#deprecated
 */
#define __deprecated

/*
 * Optional: not supported by clang
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Type-Attributes.html#index-designated_005finit-type-attrib
ute
 */
#if __has_attribute(__designated_init__)
# define __designated_init              __attribute__((__designated_init__))
#else
# define __designated_init
#endif

/*
 * Optional: only supported since clang >= 14.0
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-error-function-attribute
 */
#if __has_attribute(__error__)
# define __compiletime_error(msg)       __attribute__((__error__(msg)))
#else
# define __compiletime_error(msg)
#endif

/*
 * Optional: not supported by clang
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-externally_005fvisible-func
tion-attribute
 */
#if __has_attribute(__externally_visible__)
# define __visible                      __attribute__((__externally_visible__))
#else
# define __visible
```

```
#endif

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-format-function-attribute
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#format
 */
#define __printf(a, b)                  __attribute__((__format__(printf, a, b)))
#define __scanf(a, b)                   __attribute__((__format__(scanf, a, b)))

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-gnu_005finline-function-att
ribute
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#gnu-inline
 */
#define __gnu_inline                    __attribute__((__gnu_inline__))

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-malloc-function-attribute
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#malloc
 */
#define __malloc                        __attribute__((__malloc__))

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Type-Attributes.html#index-mode-type-attribute
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html#index-mode-variable-attribute
 */
#define __mode(x)                       __attribute__((__mode__(x)))

/*
 * Optional: only supported since gcc >= 7
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/x86-Function-Attributes.html#index-no_005fcaller_005fsaved_005fre
gisters-function-attribute_002c-x86
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#no-caller-saved-registers
 */
#if __has_attribute(__no_caller_saved_registers__)
# define __no_caller_saved_registers    __attribute__((__no_caller_saved_registers__))
#else
# define __no_caller_saved_registers
#endif

/*
 * Optional: not supported by clang
 *
 *  gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-noclone-function-attribute
 */
#if __has_attribute(__noclone__)
# define __noclone                      __attribute__((__noclone__))
#else
# define __noclone
#endif

/*
 * Add the pseudo keyword 'fallthrough' so case statement blocks
 * must end with any of these keywords:
 *   break;
 *   fallthrough;
 *   continue;
 *   goto <label>;
 *   return [expression];
 *
 *  gcc: https:// gcc.gnu.org/onlinedocs/gcc/Statement-Attributes.html#Statement-Attributes
 */
#if __has_attribute(__fallthrough__)
# define fallthrough                    __attribute__((__fallthrough__))
#else
# define fallthrough                    do {} while (0)  /* fallthrough */
#endif

/*
 * gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#Common-Function-Attributes
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#flatten
 */
# define __flatten                      __attribute__((flatten))

/*
 * Note the missing underscores.
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-noinline-function-attribute
 * clang: mentioned
 */
#define   noinline                      __attribute__((__noinline__))

/*
 * Optional: only supported since gcc >= 8
 * Optional: not supported by clang
```

```
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html#index-nonstring-variable-attribut
e
 */
#if __has_attribute(__nonstring__)
# define __nonstring                    __attribute__((__nonstring__))
#else
# define __nonstring
#endif

/*
 * Optional: only supported since GCC >= 7.1, clang >= 13.0.
 *
 *      gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-no_005fprofile_005finstr
ument_005ffunction-function-attribute
 *    clang: https:// clang.llvm.org/docs/AttributeReference.html#no-profile-instrument-function
 */
#if __has_attribute(__no_profile_instrument_function__)
# define __no_profile                   __attribute__((__no_profile_instrument_function__))
#else
# define __no_profile
#endif

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-noreturn-function-attribute
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#noreturn
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#id1
 */
#define __noreturn                      __attribute__((__noreturn__))

/*
 * Optional: only supported since GCC >= 11.1, clang >= 7.0.
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-no_005fstack_005fprotector-
function-attribute
 *   clang: https:// clang.llvm.org/docs/AttributeReference.html#no-stack-protector-safebuffers
 */
#if __has_attribute(__no_stack_protector__)
# define __no_stack_protector           __attribute__((__no_stack_protector__))
#else
# define __no_stack_protector
#endif

/*
 * Optional: not supported by gcc.
 *
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#overloadable
 */
#if __has_attribute(__overloadable__)
# define __overloadable                 __attribute__((__overloadable__))
#else
# define __overloadable
#endif

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Type-Attributes.html#index-packed-type-attribute
 * clang: https:// gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html#index-packed-variable-attribute
 */
#define __packed                        __attribute__((__packed__))

/*
 * Note: the "type" argument should match any __builtin_object_size(p, type) usage.
 *
 * Optional: not supported by gcc.
 *
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#pass-object-size-pass-dynamic-object-size
 */
#if __has_attribute(__pass_dynamic_object_size__)
# define __pass_dynamic_object_size(type)       __attribute__((__pass_dynamic_object_size__(type)))
#else
# define __pass_dynamic_object_size(type)
#endif
#if __has_attribute(__pass_object_size__)
# define __pass_object_size(type)       __attribute__((__pass_object_size__(type)))
#else
# define __pass_object_size(type)
#endif

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-pure-function-attribute
 */
#define __pure                          __attribute__((__pure__))

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-section-function-attribute
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html#index-section-variable-attribute
```

```
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#section-declspec-allocate
 */
#define __section(section)              __attribute__((__section__(section)))

/*
 * Optional: only supported since gcc >= 12
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html#index-uninitialized-variable-attr
ibute
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#uninitialized
 */
#if __has_attribute(__uninitialized__)
# define __uninitialized                __attribute__((__uninitialized__))
#else
# define __uninitialized
#endif

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-unused-function-attribute
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Type-Attributes.html#index-unused-type-attribute
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html#index-unused-variable-attribute
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Label-Attributes.html#index-unused-label-attribute
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#maybe-unused-unused
 */
#define __always_unused                 __attribute__((__unused__))
#define __maybe_unused                  __attribute__((__unused__))

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-used-function-attribute
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html#index-used-variable-attribute
 */
#define __used                          __attribute__((__used__))

/*
 * The __used attribute guarantees that the attributed variable will be
 * always emitted by a compiler. It doesn't prevent the compiler from
 * throwing 'unused' warnings when it can't detect how the variable is
 * actually used. It's a compiler implementation details either emit
 * the warning in that case or not.
 *
 * The combination of both 'used' and 'unused' attributes ensures that
 * the variable would be emitted, and will not trigger 'unused' warnings.
 * The attribute is applicable for functions, static and global variables.
 */
#define __always_used                   __used __maybe_unused

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-warn_005funused_005fresult-
function-attribute
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#nodiscard-warn-unused-result
 */
#define __must_check                    __attribute__((__warn_unused_result__))

/*
 * Optional: only supported since clang >= 14.0
 *
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-warning-function-attribute
 */
#if __has_attribute(__warning__)
# define __compiletime_warning(msg)     __attribute__((__warning__(msg)))
#else
# define __compiletime_warning(msg)
#endif

/*
 * Optional: only supported since clang >= 14.0
 *
 * clang: https:// clang.llvm.org/docs/AttributeReference.html#disable-sanitizer-instrumentation
 *
 * disable_sanitizer_instrumentation is not always similar to
 * no_sanitize((<sanitizer-name>)): the latter may still let specific sanitizers
 * insert code into functions to prevent false positives. Unlike that,
 * disable_sanitizer_instrumentation prevents all kinds of instrumentation to
 * functions with the attribute.
 */
#if __has_attribute(disable_sanitizer_instrumentation)
# define __disable_sanitizer_instrumentation \
         __attribute__((disable_sanitizer_instrumentation))
#else
# define __disable_sanitizer_instrumentation
#endif

/*
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-weak-function-attribute
 *   gcc: https:// gcc.gnu.org/onlinedocs/gcc/Common-Variable-Attributes.html#index-weak-variable-attribute
 */
```

```
#define __weak                          __attribute__((__weak__))

/*
 * Used by functions that use '__builtin_return_address'. These function
 * don't want to be splited or made inline, which can make
 * the '__builtin_return_address' get unexpected address.
 */
#define __fix_address noinline __noclone

#endif /* __LINUX_COMPILER_ATTRIBUTES_H */
```

---

https:// yuankun.me/posts/the-cleanup-variable-attribute-in-gcc/

The Cleanup Variable Attribute in GCC
2023-06-06

C++ has a powerful idiom called RAII (Resource Acquisition Is Initialization)[^1], although the name might leave something to be desired. The fundamental idea is to represent a resource by a local object, tying the resource's lifecycle to the lifecycle of the local object' lifecycle. In other words, we could say the local object is the owner of the resource (Hmm, I smell something rusty here). The local object is responsible for releasing the resource in its destructor. Once the local object goes out of its scope, the resource is released. This mechanism remains valid even in the event of an exception within the scope. This is one of my most wanted feature when I'm programming in C. Without it, some error handling situations often lead to numerous goto failure statements, where the failure block is merely used for releasing resources in a carefully crafted sequence. Appearently, the C standard committee is discussing to add a defer mechanism to C, taking inspiration from the defer construct in Golang.

Today I learned that GCC has already provided the semantic through an extension called the cleanup variable attribute. The cleanup attribute allows us to define an auto variable with a cleanup handler that will be invoked when the auto variable goes out of score. The cleanup handler must take one parameter, which is a pointer to a type compatible with the variable. The return value of the function (if any) is ignored. Below is its syntax.

```
void handler(void *p) {
    /* cleanup logic */
}

/* Must be applied only on auto variables */
attribute __((cleanup(handler))) var;
```

Usage

Here's tiny program that illustrates how to use the cleanup attribute. In the code, I've added a switch, USE_CLEANUP, which can be toggled to activate or deactivate the application of the cleanup variable attribute. In the foo function, I've initiated two pointers, x and y, to point to two int objects on the heap. Note that I didn't deallocate the two int objects in the foo function.

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

#ifdef USE_CLEANUP
#define CLEANUP __attribute__ ((cleanup(cleanup_func)))
#else
#define CLEANUP
#endif

static void cleanup_func(void *p) {
    printf("[cleanup_func] value=%d\n", **(int **)p);
    free(*(void **)p);
}

void foo() {
    CLEANUP int *x = malloc(sizeof(int));
    CLEANUP int *y = malloc(sizeof(int));
    *x = 42;
    *y = 39;
    printf("[foo] x=%d\n", *x);
    printf("[foo] y=%d\n", *y);
}

int main(void) {
    foo();
    return 0;
}
```

Now, compile the program with the -DUSE_CLEANUP flag and run it. We see that the cleanup handler is invoked twice, in the reverse order of how we defined the two integer pointers.

```
[foo] x=42
[foo] y=39
[cleanup_func] value=39
[cleanup_func] value=42
```

Implementation

In order to implement the cleanup attribute, GCC inserts instructions to call the handler before returning from the current function. We can confirm this by comparing the two variants of the above program - one that compiled with the -DUSE_CLEANUP flag and another that does not. I've turned off stack protector to avoid nay unrelated stack protection instructions.

```
$> cc -DUSE_CLEANUP -fno-stack-protector -S -o with_cleanup.s main.c
$> cc -fno-stack-protector -S -o no_cleanup.s main.c
$> diff -y no_cleanup.s with_cleanup.s
```

[https:// yuankun.me/img/cleanup-diff.png]/img/cleanup-diff.png

The diff result clearly shows that in the -DUSE_CLEANUP version, GCC generates instructions to call the cleanup handler twice at the termination of the foo function. That's the only distinction between the two versions.

From this implementation, we can draw a few insights:

* The cleanup handler won't be called if the local scope executes any of the exit functions or the abort function.
* The cleanup handler might not be called or might not complete in the event of a terminate signal, depending on the timing of the signal reception.
* The cleanup handler won't be called if the local scope performs a long jump instead of a normal return.

Moreover, as per the GCC documentation, it is undefined what happens if the cleanup handler does not return normally (e.g., if it performs a long jump).
