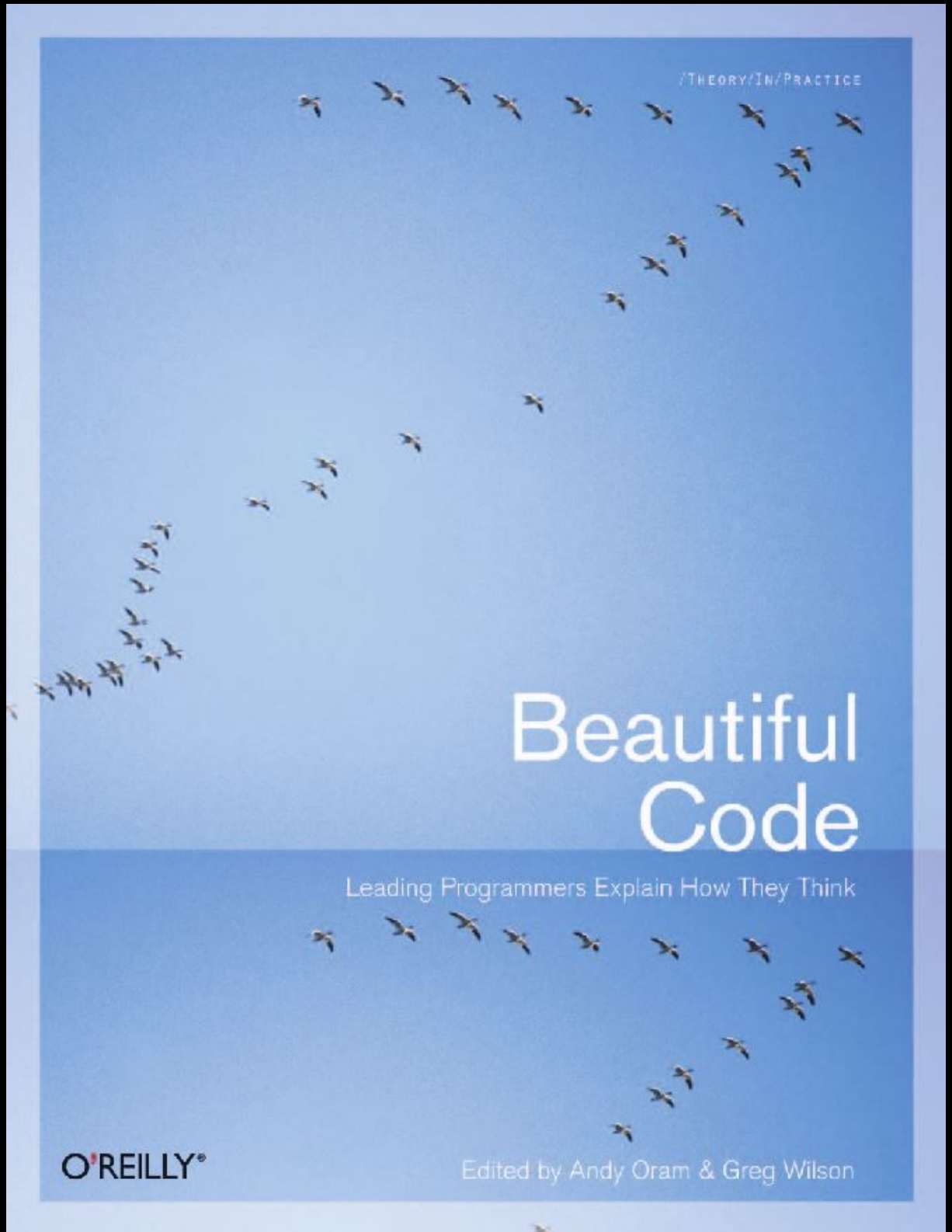
Objects in Linux

Trust us, we know what we are doing...

Greg Kroah-Hartman
gregkh@linuxfoundation.org


/Theory/In/Practice
Beautiful Code
Leading Programmers Explain How They Think
O'REILLY®
Edited by Andy Oram & Greg Wilson

Early 2000's
“Unify all Linux devices”

Pat Mochel – OSDL
Greg K-H - IBM

```
struct device {
        struct list_head node;
        struct list_head children;
        struct device *parent;

        char name[DEVICE_NAME_SIZE];
        bus_id[BUS_ID_SIZE];

        spinlock_t lock;

        atomic_t refcount;

        struct driver_dir_entry *dir;
        struct device_driver *driver;

        void *driver_data;
        void *platform_data;
        u32 current_state;
        unsigned char *saved_state;
  };
```

“sysfs looks like a web woven by a spider on drugs”

– lwn.net

```
int device_register(struct device *dev);
void device_unregister(struct device *dev);



struct bus_type {
        …
        int (*probe)(struct device *dev);
        int (*remove)(struct device *dev);
        …
};
```

```
struct usb_interface {
        struct usb_interface_descriptor *altsetting;
        int act_altsetting;
        int num_altsetting;
        int max_altsetting;
        struct usb_driver *driver;
        struct device dev;
};
```

```
static int usb_probe(struct device *d)
{
        struct usb_interface *intf;

        intf = container_of(d, struct usb_interface, dev);
        …

}
```

```
#define container_of(ptr, type, member) ({                  \
    const typeof( ((type *)0)->member ) *_ _mptr = (ptr); \
    (type *)( (char *)_ _mptr - offsetof(type,member) );})
```

```
intf = container_of(d, struct usb_interface, dev);
```

```
intf = ({
    const struct device *__mptr = d;
    (struct usb_interface *)( (char *)__mptr -
      offsetof(struct usb_interface, dev));
});
```

```
intf = ({
    const struct device *__mptr = d;
    (struct usb_interface *)( (char *)__mptr - 20));
});
```

```
intf = *d - 20;
```

No type safety at all, you just “know” what type of object you have.

```
struct device {
        struct list_head node;
        struct list_head children;
        struct device *parent;

        char name[DEVICE_NAME_SIZE];
        bus_id[BUS_ID_SIZE];

        spinlock_t lock;

        atomic_t refcount;

        struct driver_dir_entry *dir;
        struct device_driver *driver;

        void *driver_data;
        void *platform_data;
        u32 current_state;
        unsigned char *saved_state;
};
```

```
static inline void get_device(struct device *dev)
{
        BUG_ON(!atomic_read(&dev->refcount));
        atomic_inc(&dev->refcount);
}
```

```
void put_device(struct device *dev)
{
    if (!atomic_dec_and_lock(&dev->refcount,&device_lock))
        return;

...

    /* Tell the driver to clean up after itself.
     * Note that we likely didn't allocate the device,
     * so this is the driver's chance to free that up...
     */
    if (dev->driver && dev->driver->remove)
        dev->driver->remove(dev, REMOVE_FREE_RESOURCES);
}
```

```
struct kobject {
        char name[KBOJ_NAME_LEN];
        atomic_t refcount;
        struct list_head entry;
        struct kobject *parent
        struct subsystem *subsys;
        struct dentry *dentry;
};
```

```
struct device {
        struct list_head g_list;
        struct list_head node;
        struct list_head bus_list;
        struct list_head driver_list;
        struct list_head children;
        struct list_head intf_list;
        struct device *parent;

        struct kobject kobj;

        char bus_id[BUS_ID_SIZE];

...

};

#define to_dev(obj) \
    container_of(obj, struct device, kobj)
```

```
struct kref {
        atomic_t refcount;
};

static inline void kref_init(struct kref *kref)
{
        atomic_set(&kref->refcount, 1);
}

static inline void kref_get(struct kref *kref)
{
        /* If refcount was 0 before incrementing then
         * we have a race condition when this kref is
         * freeing by some other thread right now.
         */
        WARN_ON_ONCE(atomic_inc_return(&kref->refcount)
                        < 2);
}
```

```
int kref_put(struct kref *kref,
             void (*release)(struct kref *kref))
{
        WARN_ON(release == NULL);
        WARN_ON(release ==
                (void (*)(struct kref *))kfree);

        if (atomic_dec_and_test(&kref->refcount)) {
                release(kref);
                return 1;
        }
        return 0;
}
```

```
struct kobject {
        char name[KOBJ_NAME_LEN];
        struct kref kref;
        ...
};
```

```
void put_device(struct device *dev)
{
    if (!atomic_dec_and_lock(&dev->refcount,&device_lock))
        return;

...

    /* Tell the driver to clean up after itself.
     * Note that we likely didn't allocate the device,
     * so this is the driver's chance to free that up...
     */
    if (dev->driver && dev->driver->remove)
        dev->driver->remove(dev, REMOVE_FREE_RESOURCES);
}
```

```
void put_device(struct device *dev)
{
        /* might_sleep(); */
        if (dev)
                kobject_put(&dev->kobj);
}

void kobject_put(struct kobject *kobj)
{
        if (kobj) {
                if (!kobj->state_initialized)
                        WARN(1, …);
                kref_put(&kobj->kref, kobject_release);
        }
}
```

```
static inline int kref_put_mutex(struct kref *kref,
                                 void (*release)(struct kref *kref),
                                 struct mutex *lock)
{
        WARN_ON(release == NULL);

        if (!atomic_add_unless(&kref->refcount, -1, 1)) {
                mutex_lock(lock);
                if (!atomic_dec_and_test(&kref->refcount)) {
                        mutex_unlock(lock);
                        return 0;
                }
                release(kref);
                return 1;
        }
        return 0;
}
```

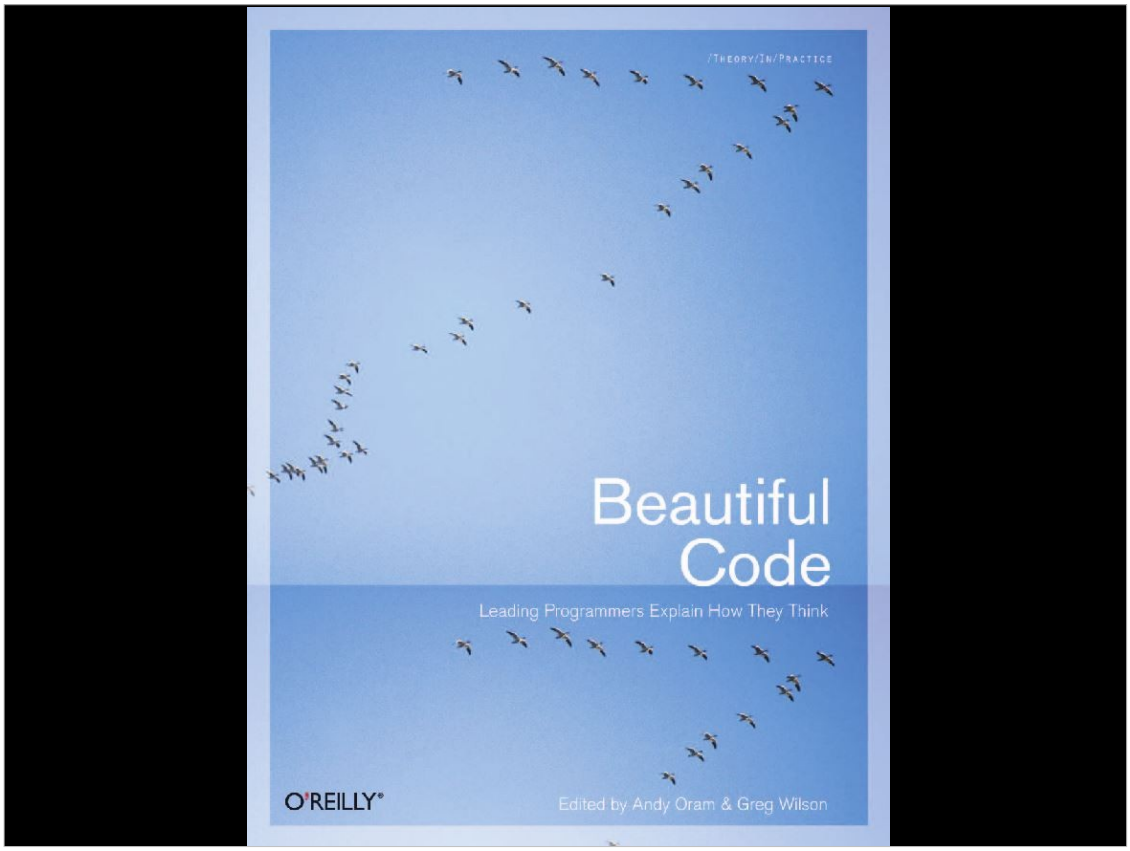
github.com/gregkh/presentation-kref

```
#define container_of(ptr, type, member) ({                  \
    const typeof( ((type *)0)->member ) *_ _mptr = (ptr); \
    (type *)( (char *)_ _mptr - offsetof(type,member) );})


intf = container_of(d, struct usb_interface, dev);

intf = ({
    const struct device *__mptr = d;
    (struct usb_interface *)( (char *)__mptr -
      offsetof(struct usb_interface, dev));
});

intf = ({
    const struct device *__mptr = d;
    (struct usb_interface *)( (char *)__mptr - 16));
});

intf = *d - 16;
```

Objects in Linux

Trust us, we know what we are doing...

Greg Kroah-Hartman
gregkh@linuxfoundation.org



Talk based on a chapter I wrote for “Beautiful Code – Leading Programmers Explain How They Think”

Edited by Andy Oram and Greg Wilson
O'Reilly
2007

Early 2000's
“Unify all Linux devices”

Pat Mochel – OSDL
Greg K-H - IBM

Pat wanted this for power management and suspend/resume

I wanted this for persistant device naming.

All devices and subsystems were islands

Both tasks needed a way to see all devices in the system, suspend/resume wanted to know which device to suspend in which order.

Naming needed a way to assign a character or block device to a specific hardware device

```
struct device {
        struct list_head node;
        struct list_head children;
        struct device *parent;

        char name[DEVICE_NAME_SIZE];
        bus_id[BUS_ID_SIZE];

        spinlock_t lock;

        atomic_t refcount;

        struct driver_dir_entry *dir;
        struct device_driver *driver;

        void *driver_data;
        void *platform_data;
        u32 current_state;
        unsigned char *saved_state;
  };
```

We came up with 'struct device'

All busses in the kernel were changed to create a structure based on this one. It was passed to the new driver core, and the driver core created a hierarchy of everything in the kernel.

This can be seen in sysfs (which used to be called driverfs)

> “sysfs looks like a web woven by a spider on drugs”
>
> – lwn.net

Pat wanted this for power management and suspend/resume

I wanted this for persistant device naming.

All devices and subsystems were islands

Both tasks needed a way to see all devices in the system, suspend/resume wanted to know which device to suspend in which order.

Naming needed a way to assign a character or block device to a specific hardware device

```
int device_register(struct device *dev);
void device_unregister(struct device *dev);



struct bus_type {
        …
        int (*probe)(struct device *dev);
        int (*remove)(struct device *dev);
        …
};
```

Pat wanted this for power management and suspend/resume

I wanted this for persistant device naming.

All devices and subsystems were islands

Both tasks needed a way to see all devices in the system, suspend/resume wanted to know which device to suspend in which order.

Naming needed a way to assign a character or block device to a specific hardware device

```
struct usb_interface {
        struct usb_interface_descriptor *altsetting;
        int act_altsetting;
        int num_altsetting;
        int max_altsetting;
        struct usb_driver *driver;
        struct device dev;
};
```

Pat wanted this for power management and suspend/resume

I wanted this for persistant device naming.

All devices and subsystems were islands

Both tasks needed a way to see all devices in the system, suspend/resume wanted to know which device to suspend in which order.

Naming needed a way to assign a character or block device to a specific hardware device

```
static int usb_probe(struct device *d)
{
        struct usb_interface *intf;

        intf = container_of(d, struct usb_interface, dev);
        …

}
```

Pat wanted this for power management and suspend/resume

I wanted this for persistant device naming.

All devices and subsystems were islands

Both tasks needed a way to see all devices in the system, suspend/resume wanted to know which device to suspend in which order.

Naming needed a way to assign a character or block device to a specific hardware device

```
#define container_of(ptr, type, member) ({                  \
    const typeof( ((type *)0)->member ) *_ _mptr = (ptr); \
    (type *)( (char *)_ _mptr - offsetof(type,member) );})
```

Pat wanted this for power management and suspend/resume

I wanted this for persistant device naming.

All devices and subsystems were islands

Both tasks needed a way to see all devices in the system, suspend/resume wanted to know which device to suspend in which order.

Naming needed a way to assign a character or block device to a specific hardware device

```
intf = container_of(d, struct usb_interface, dev);
```

```
intf = ({
    const struct device *__mptr = d;
    (struct usb_interface *)( (char *)__mptr -
      offsetof(struct usb_interface, dev));
});
```

```
intf = ({
    const struct device *__mptr = d;
    (struct usb_interface *)( (char *)__mptr - 20));
});
```

```
intf = *d - 20;
```

No type safety at all, you just “know” what type of object you have.

```
struct device {
        struct list_head node;
        struct list_head children;
        struct device *parent;

        char name[DEVICE_NAME_SIZE];
        bus_id[BUS_ID_SIZE];

        spinlock_t lock;

        atomic_t refcount;

        struct driver_dir_entry *dir;
        struct device_driver *driver;

        void *driver_data;
        void *platform_data;
        u32 current_state;
        unsigned char *saved_state;
  };
```

When the structure was initialized, this field was set to 1. Whenever any code wished to use the structure, it had to first increment the reference count by calling the function get_ device , which checked that the reference count was valid and incremented the reference count of the structure:

```
static inline void get_device(struct device *dev)
{
        BUG_ON(!atomic_read(&dev->refcount));
        atomic_inc(&dev->refcount);
}
```

When the structure was initialized, this field was set to 1. Whenever any code wished to use the structure, it had to first increment the reference count by calling the function get_ device , which checked that the reference count was valid and incremented the reference count of the structure:

```
void put_device(struct device *dev)
{
    if (!atomic_dec_and_lock(&dev->refcount,&device_lock))
        return;

...

    /* Tell the driver to clean up after itself.
     * Note that we likely didn't allocate the device,
     * so this is the driver's chance to free that up...
     */
    if (dev->driver && dev->driver->remove)
        dev->driver->remove(dev, REMOVE_FREE_RESOURCES);
}
```

This function decremented the reference count and then, if it was the last user of the
object, would tell the object to clean itself up and call a function that was previously set up
to free it from the system.

```
struct kobject {
        char name[KBOJ_NAME_LEN];
        atomic_t refcount;
        struct list_head entry;
        struct kobject *parent
        struct subsystem *subsys;
        struct dentry *dentry;
};
```

```
struct device {
        struct list_head g_list;
        struct list_head node;
        struct list_head bus_list;
        struct list_head driver_list;
        struct list_head children;
        struct list_head intf_list;
        struct device *parent;

        struct kobject kobj;

        char bus_id[BUS_ID_SIZE];

...

};

#define to_dev(obj) \
    container_of(obj, struct device, kobj)
```

When the structure was initialized, this field was set to 1. Whenever any code wished to use the structure, it had to first increment the reference count by calling the function get_ device , which checked that the reference count was valid and incremented the reference count of the structure:

```
struct kref {
        atomic_t refcount;
};

static inline void kref_init(struct kref *kref)
{
        atomic_set(&kref->refcount, 1);
}

static inline void kref_get(struct kref *kref)
{
        /* If refcount was 0 before incrementing then
         * we have a race condition when this kref is
         * freeing by some other thread right now.
         */
        WARN_ON_ONCE(atomic_inc_return(&kref->refcount)
                     < 2);
}
```

```
int kref_put(struct kref *kref,
             void (*release)(struct kref *kref))
{
        WARN_ON(release == NULL);
        WARN_ON(release ==
                (void (*)(struct kref *))kfree);

        if (atomic_dec_and_test(&kref->refcount)) {
                release(kref);
                return 1;
        }
        return 0;
}
```

After these two checks are made, the reference count is atomically decremented, and if this is the last reference, the release function is called and 1 is returned. If this is not the last reference on the object, 0 is returned. This return value is used just to determine whether the caller was the last holder of the object, not whether the object is still in mem-ory (it can’t guarantee that the object still exists because someone else might come in and release it after the call returns).

```
struct kobject {
        char name[KOBJ_NAME_LEN];
        struct kref kref;
        ...
};
```

With the creation of struct kref , the struct kobject structure was changed to use it:

With all of these different structures embedded within other structures, the result is that the original struct usb_interface described earlier now contains a struct device , which contains a struct kobject , which contains a struct kref.

And who said it was hard to do object-oriented programming in the C language....

```
void put_device(struct device *dev)
{
    if (!atomic_dec_and_lock(&dev->refcount,&device_lock))
        return;

...

    /* Tell the driver to clean up after itself.
     * Note that we likely didn't allocate the device,
     * so this is the driver's chance to free that up...
     */
    if (dev->driver && dev->driver->remove)
        dev->driver->remove(dev, REMOVE_FREE_RESOURCES);
}
```

The original put_device()

```
void put_device(struct device *dev)
{
        /* might_sleep(); */
        if (dev)
                kobject_put(&dev->kobj);
}

void kobject_put(struct kobject *kobj)
{
        if (kobj) {
                if (!kobj->state_initialized)
                        WARN(1, …);
                kref_put(&kobj->kref, kobject_release);
        }
}
```

put_device() and kobject_put() today

What's wrong with this picture…

No lock!!!!

Bug has endured for over a decade. We got lucky, the device lock went away, but busses added/removed devices all sequentially.

Things get messy when you use kref() on your own.

Things are buggy if you use a kref on your own.

```
static inline int kref_put_mutex(struct kref *kref,
                        void (*release)(struct kref *kref),
                        struct mutex *lock)
{
        WARN_ON(release == NULL);

        if (!atomic_add_unless(&kref->refcount, -1, 1)) {
                mutex_lock(lock);
                if (!atomic_dec_and_test(&kref->refcount)) {
                        mutex_unlock(lock);
                        return 0;
                }
                release(kref);
                return 1;
        }
        return 0;
}
```

Behaves identical to kref_put with one exception. If the reference count drops to zero, the lock will be taken atomically wrt dropping the reference count.

Adds -1 to the reference count only if refcount was not already 1
If it succeeded (i.e. refcount was not 1) then the function returns 0

If it failed, then refcount was 1, it's now 0, so try to grab the lock.

After we grab the lock, we then try to decrement the reference count again to see if someone else incremented it before we got the lock. If we return 0, then someone else got here before we did, so unlock and return. If we don't return 0, then we were the last reference, so release the memory, and have the release function unlock the lock.

Obligatory Penguin Picture

```
#define container_of(ptr, type, member) ({                  \
    const typeof( ((type *)0)->member ) *_ _mptr = (ptr); \
    (type *)( (char *)_ _mptr - offsetof(type,member) );})


intf = container_of(d, struct usb_interface, dev);

intf = ({
    const struct device *__mptr = d;
    (struct usb_interface *)( (char *)__mptr -
      offsetof(struct usb_interface, dev));
});

intf = ({
    const struct device *__mptr = d;
    (struct usb_interface *)( (char *)__mptr — 16));
});

intf = *d - 16;
```