

## Writing an OS in Rust

Philipp Oppermann's blog

[« All Posts](#)

# Heap Allocation

Jun 26, 2019

This post adds support for heap allocation to our kernel. First, it gives an introduction to dynamic memory and shows how the borrow checker prevents common allocation errors. It then implements the basic allocation interface of Rust, creates a heap memory region, and sets up an allocator crate. At the end of this post all the allocation and collection types of the built-in `alloc` crate will be available to our kernel.

This blog is openly developed on [GitHub](#). If you have any problems or questions, please open an issue there. You can also leave comments [at the bottom](#). The complete source code for this post can be found in the [post-10](#) branch.

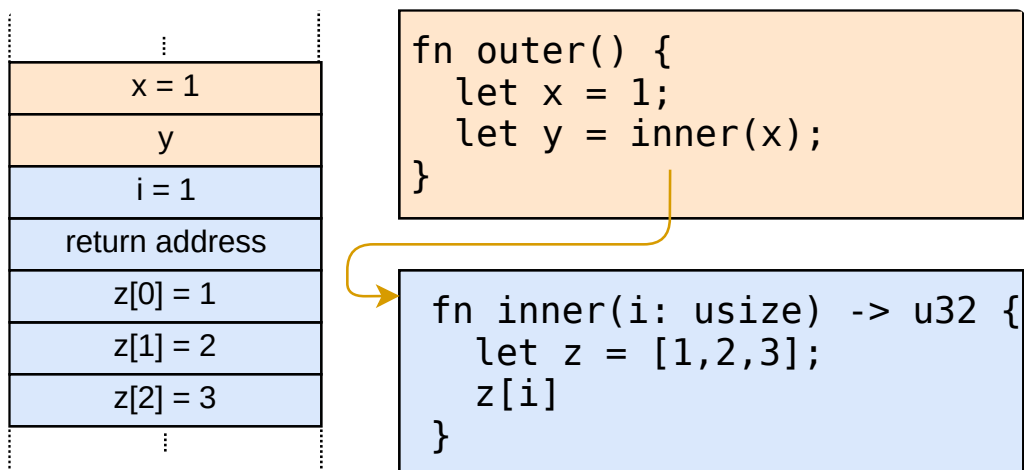
### ► Table of Contents

## Local and Static Variables

We currently use two types of variables in our kernel: local variables and `static` variables. Local variables are stored on the [call stack](#) and are only valid until the surrounding function returns. Static variables are stored at a fixed memory location and always live for the complete lifetime of the program.

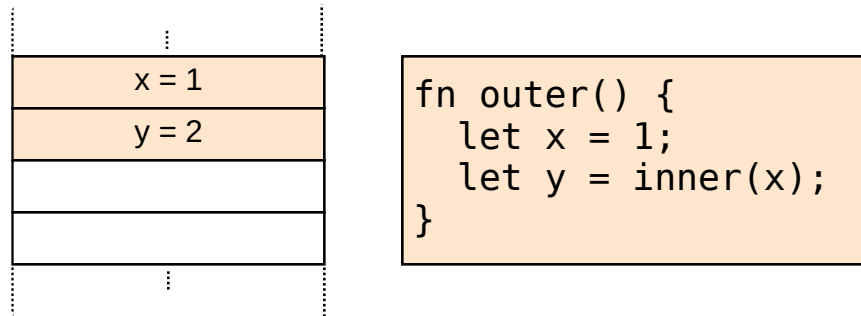
### Local Variables

Local variables are stored on the [call stack](#), which is a [stack data structure](#) that supports `push` and `pop` operations. On each function entry, the parameters, the return address, and the local variables of the called function are pushed by the compiler:



The above example shows the call stack after an `outer` function called an `inner` function. We see that the call stack contains the local variables of `outer` first. On the `inner` call, the parameter `1` and the return address for the function were pushed. Then control was transferred to `inner`, which pushed its local variables.

After the `inner` function returns, its part of the call stack is popped again and only the local variables of `outer` remain:



We see that the local variables of `inner` only live until the function returns. The Rust compiler enforces these lifetimes and throws an error when we use a value too long, for example when we try to return a reference to a local variable:

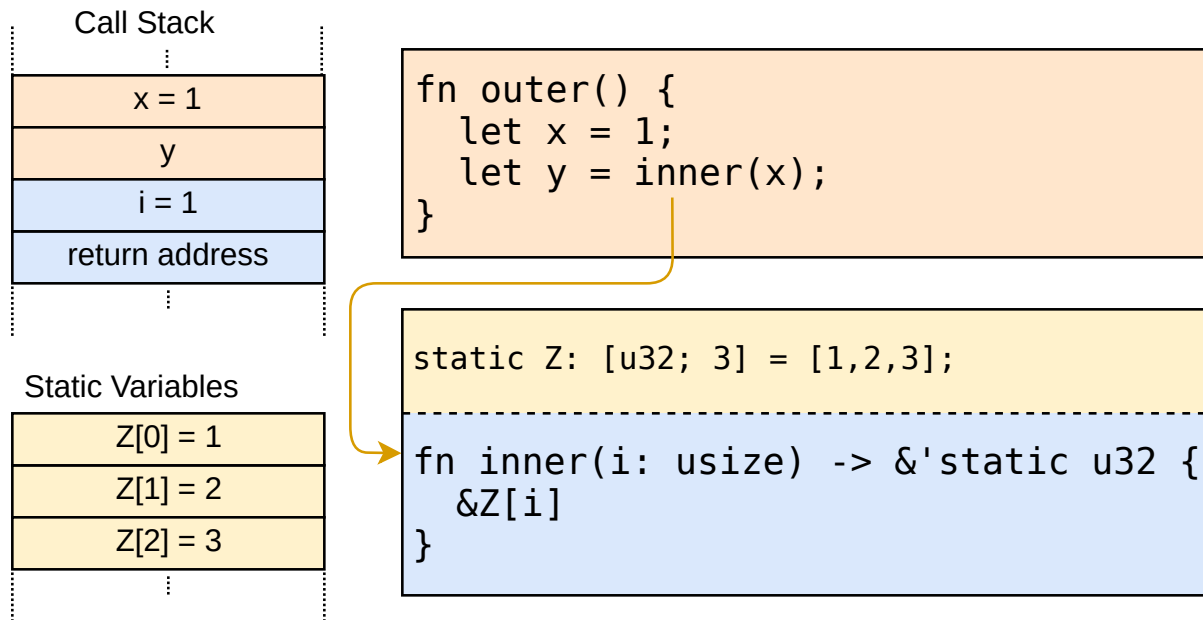
```
fn inner(i: usize) -> &'static u32 {
    let z = [1, 2, 3];
    &z[i]
}
```

(run the example on the playground)

While returning a reference makes no sense in this example, there are cases where we want a variable to live longer than the function. We already saw such a case in our kernel when we tried to [load an interrupt descriptor table](#) and had to use a `static` variable to extend the lifetime.

## Static Variables

Static variables are stored at a fixed memory location separate from the stack. This memory location is assigned at compile time by the linker and encoded in the executable. Statics live for the complete runtime of the program, so they have the `'static` lifetime and can always be referenced from local variables:



When the `inner` function returns in the above example, its part of the call stack is destroyed. The static variables live in a separate memory range that is never destroyed, so the `&Z[1]` reference is still valid after the return.

Apart from the `'static` lifetime, static variables also have the useful property that their location is known at compile time, so that no reference is needed for accessing it. We utilized that property for our `println` macro: By using a `static Writer` internally there is no `&mut Writer` reference needed to invoke the macro, which is very useful in `exception handlers` where we don't have access to any additional variables.

However, this property of static variables brings a crucial drawback: They are read-only by default. Rust enforces this because a `data race` would occur if e.g. two threads modify a static variable at the same time. The only way to modify a static variable is to encapsulate it in a `Mutex` type, which ensures that only a single `&mut` reference exists at any point in time. We already used a `Mutex` for our `static VGA buffer Writer`.

## Dynamic Memory

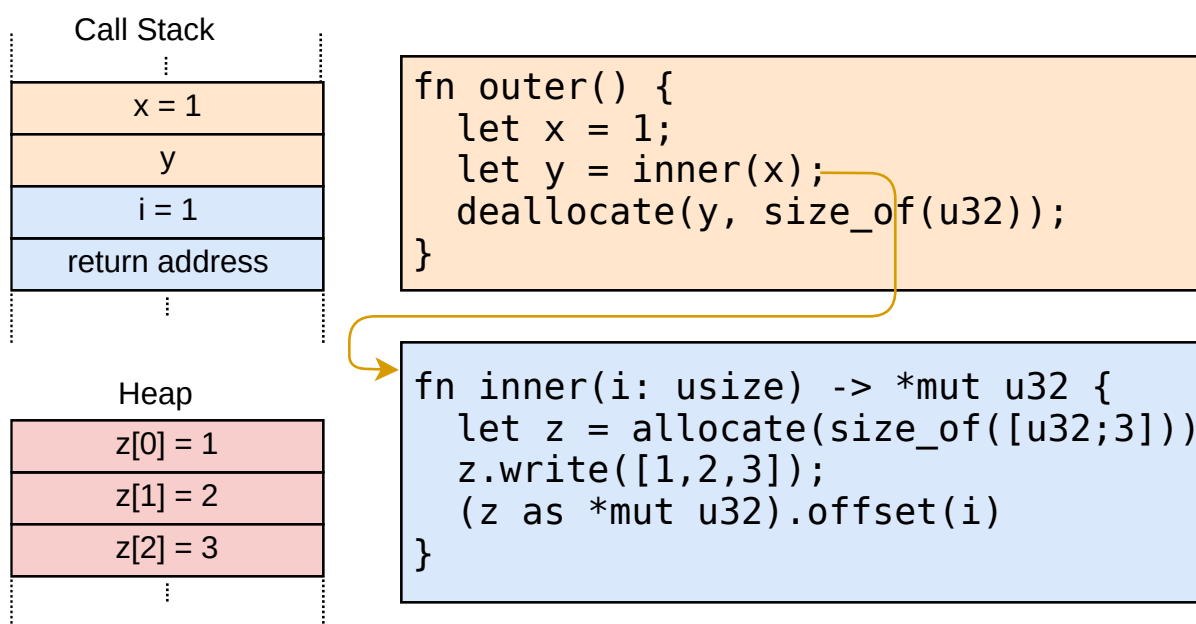
Local and static variables are already very powerful together and enable most use cases. However, we saw that they both have their limitations:

- Local variables only live until the end of the surrounding function or block. This is because they live on the call stack and are destroyed after the surrounding function returns.
- Static variables always live for the complete runtime of the program, so there is no way to reclaim and reuse their memory when they're no longer needed. Also, they have unclear ownership semantics and are accessible from all functions, so they need to be protected by a `Mutex` when we want to modify them.

Another limitation of local and static variables is that they have a fixed size. So they can't store a collection that dynamically grows when more elements are added. (There are proposals for [unsized rvalues](#) in Rust that would allow dynamically sized local variables, but they only work in some specific cases.)

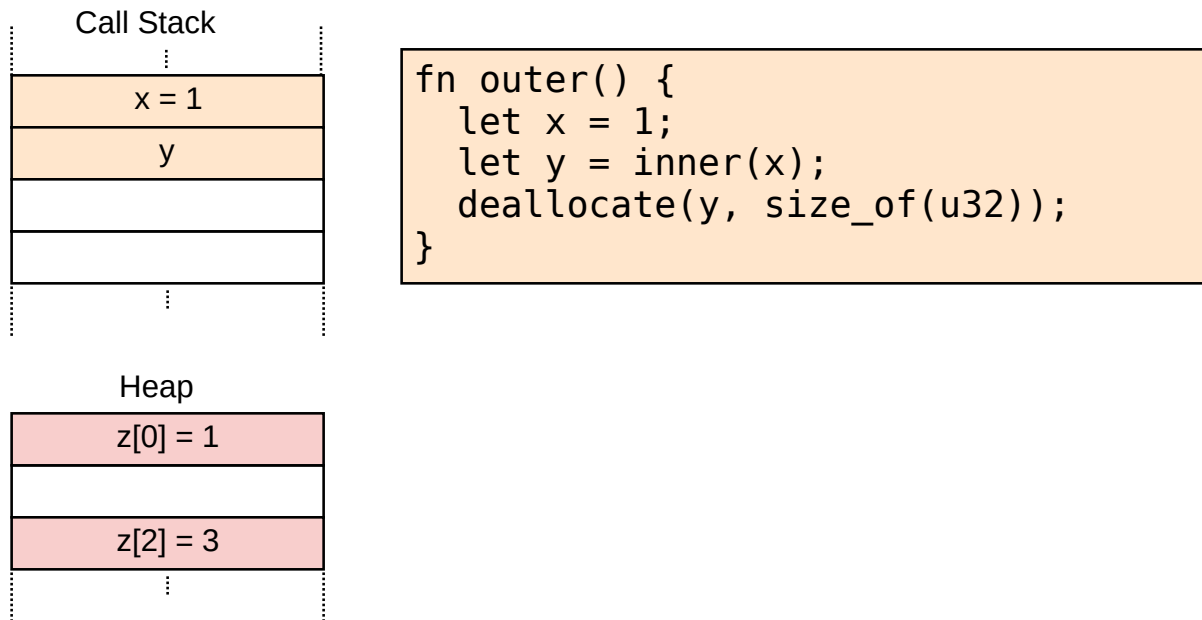
To circumvent these drawbacks, programming languages often support a third memory region for storing variables called the **heap**. The heap supports *dynamic memory allocation* at runtime through two functions called `allocate` and `deallocate`. It works in the following way: The `allocate` function returns a free chunk of memory of the specified size that can be used to store a variable. This variable then lives until it is freed by calling the `deallocate` function with a reference to the variable.

Let's go through an example:



Here the `inner` function uses heap memory instead of static variables for storing `z`. It first allocates a memory block of the required size, which returns a `*mut u32 raw pointer`. It then uses the `ptr::write` method to write the array `[1,2,3]` to it. In the last step, it uses the `offset` function to calculate a pointer to the `i`th element and then returns it. (Note that we omitted some required casts and unsafe blocks in this example function for brevity.)

The allocated memory lives until it is explicitly freed through a call to `deallocate`. Thus, the returned pointer is still valid even after `inner` returned and its part of the call stack was destroyed. The advantage of using heap memory compared to static memory is that the memory can be reused after it is freed, which we do through the `deallocate` call in `outer`. After that call, the situation looks like this:



We see that the `z[1]` slot is free again and can be reused for the next `allocate` call. However, we also see that `z[0]` and `z[2]` are never freed because we never deallocate them. Such a bug is called a *memory leak* and often the cause of excessive memory consumption of programs (just imagine what happens when we call `inner` repeatedly in a loop). This might seem bad, but there are much more dangerous types of bugs that can happen with dynamic allocation.

## Common Errors

Apart from memory leaks, which are unfortunate but don't make the program vulnerable to attackers, there are two common types of bugs with more severe consequences:

- When we accidentally continue to use a variable after calling `deallocate` on it, we have a so-called **use-after-free** vulnerability. Such a bug causes undefined behavior and can often be exploited by attackers to execute arbitrary code.
- When we accidentally free a variable twice, we have a **double-free** vulnerability. This is problematic because it might free a different allocation that was allocated in the same spot after the first `deallocate` call. Thus, it can lead to an use-after-free vulnerability again.

These types of vulnerabilities are commonly known, so one might expect that people learned how to avoid them by now. But no, such vulnerabilities are still regularly found, for example this recent [use-after-free vulnerability in Linux](#) that allowed arbitrary code execution. This shows that even the best programmers are not always able to correctly handle dynamic memory in complex projects.

To avoid these issues, many languages such as Java or Python manage dynamic memory automatically using a technique called *garbage collection*. The idea is that the programmer never invokes `deallocate` manually. Instead, the program is regularly paused and scanned for

unused heap variables, which are then automatically deallocated. Thus, the above vulnerabilities can never occur. The drawbacks are the performance overhead of the regular scan and the probably long pause times.

Rust takes a different approach to the problem: It uses a concept called *ownership* that is able to check the correctness of dynamic memory operations at compile time. Thus no garbage collection is needed to avoid the mentioned vulnerabilities, which means that there is no performance overhead. Another advantage of this approach is that the programmer still has fine-grained control over the use of dynamic memory, just like with C or C++.

## Allocations in Rust

Instead of letting the programmer manually call `allocate` and `deallocate`, the Rust standard library provides abstraction types that call these functions implicitly. The most important type is `Box`, which is an abstraction for a heap-allocated value. It provides a `Box::new` constructor function that takes a value, calls `allocate` with the size of the value, and then moves the value to the newly allocated slot on the heap. To free the heap memory again, the `Box` type implements the `Drop` trait to call `deallocate` when it goes out of scope:

```
{
    let z = Box::new([1,2,3]);
    [...]
} // z goes out of scope and `deallocate` is called
```

This pattern has the strange name *resource acquisition is initialization* (or *RAII* for short). It originated in C++, where it is used to implement a similar abstraction type called `std::unique_ptr`.

Such a type alone does not suffice to prevent all use-after-free bugs since programmers can still hold on to references after the `Box` goes out of scope and the corresponding heap memory slot is deallocated:

```
let x = {
    let z = Box::new([1,2,3]);
    &z[1]
}; // z goes out of scope and `deallocate` is called
println!("{}", x);
```

This is where Rust's ownership comes in. It assigns an abstract *lifetime* to each reference, which is the scope in which the reference is valid. In the above example, the `x` reference is taken from the `z` array, so it becomes invalid after `z` goes out of scope. When you [run the above example on the playground](#) you see that the Rust compiler indeed throws an error:

```

error[E0597]: `z[_]` does not live long enough
--> src/main.rs:4:9
  |
2 |     let x = {
  |         - borrow later stored here
3 |         let z = Box::new([1,2,3]);
4 |         &z[1]
  |         ^^^^^ borrowed value does not live long enough
5 |     }; // z goes out of scope and `deallocate` is called
  |     - `z[_]` dropped here while still borrowed

```

The terminology can be a bit confusing at first. Taking a reference to a value is called *borrowing* the value since it's similar to a borrow in real life: You have temporary access to an object but need to return it sometime and you must not destroy it. By checking that all borrows end before an object is destroyed, the Rust compiler can guarantee that no use-after-free situation can occur.

Rust's ownership system goes even further and does not only prevent use-after-free bugs, but provides complete *memory safety* like garbage collected languages like Java or Python do. Additionally, it guarantees *thread safety* and is thus even safer than those languages in multi-threaded code. And most importantly, all these checks happen at compile time, so there is no runtime overhead compared to hand written memory management in C.

## Use Cases

We now know the basics of dynamic memory allocation in Rust, but when should we use it? We've come really far with our kernel without dynamic memory allocation, so why do we need it now?

First, dynamic memory allocation always comes with a bit of performance overhead, since we need to find a free slot on the heap for every allocation. For this reason local variables are generally preferable, especially in performance sensitive kernel code. However, there are cases where dynamic memory allocation is the best choice.

As a basic rule, dynamic memory is required for variables that have a dynamic lifetime or a variable size. The most important type with a dynamic lifetime is **Rc**, which counts the references to its wrapped value and deallocates it after all references went out of scope. Examples for types with a variable size are **Vec**, **String**, and other *collection types* that dynamically grow when more elements are added. These types work by allocating a larger amount of memory when they become full, copying all elements over, and then deallocating the old allocation.

For our kernel we will mostly need the collection types, for example for storing a list of active tasks when implementing multitasking in future posts.

# The Allocator Interface

The first step in implementing a heap allocator is to add a dependency on the built-in `alloc` crate. Like the `core` crate, it is a subset of the standard library that additionally contains the allocation and collection types. To add the dependency on `alloc`, we add the following to our `lib.rs`:

```
// in src/lib.rs

extern crate alloc;
```

Contrary to normal dependencies, we don't need to modify the `Cargo.toml`. The reason is that the `alloc` crate ships with the Rust compiler as part of the standard library, so the compiler already knows about the crate. By adding this `extern crate` statement, we specify that the compiler should try to include it. (Historically, all dependencies needed an `extern crate` statement, which is now optional).

Since we are compiling for a custom target, we can't use the precompiled version of `alloc` that is shipped with the Rust installation. Instead, we have to tell cargo to recompile the crate from source. We can do that, by adding it to the `unstable.build-std` array in our `.cargo/config.toml` file:

```
# in .cargo/config.toml

[unstable]
build-std = ["core", "compiler_builtins", "alloc"]
```

Now the compiler will recompile and include the `alloc` crate in our kernel.

The reason that the `alloc` crate is disabled by default in `#[no_std]` crates is that it has additional requirements. We can see these requirements as errors when we try to compile our project now:

```
error: no global memory allocator found but one is required; link to std or add
      #[global_allocator] to a static item that implements the GlobalAlloc trait.

error: `#[alloc_error_handler]` function required, but not found
```

The first error occurs because the `alloc` crate requires an heap allocator, which is an object that provides the `allocate` and `deallocate` functions. In Rust, heap allocators are described by the `GlobalAlloc` trait, which is mentioned in the error message. To set the heap allocator for the crate, the `#[global_allocator]` attribute must be applied to a `static` variable that implements the `GlobalAlloc` trait.



The second error occurs because calls to `allocate` can fail, most commonly when there is no more memory available. Our program must be able to react to this case, which is what the `#[alloc_error_handler]` function is for.

We will describe these traits and attributes in detail in the following sections.

## The GlobalAlloc Trait

The `GlobalAlloc` trait defines the functions that a heap allocator must provide. The trait is special because it is almost never used directly by the programmer. Instead, the compiler will automatically insert the appropriate calls to the trait methods when using the allocation and collection types of `alloc`.

Since we will need to implement the trait for all our allocator types, it is worth taking a closer look at its declaration:

```
pub unsafe trait GlobalAlloc {
    unsafe fn alloc(&self, layout: Layout) -> *mut u8;
    unsafe fn dealloc(&self, ptr: *mut u8, layout: Layout);

    unsafe fn alloc_zeroed(&self, layout: Layout) -> *mut u8 { ... }
    unsafe fn realloc(
        &self,
        ptr: *mut u8,
        layout: Layout,
        new_size: usize
    ) -> *mut u8 { ... }
}
```

It defines the two required methods `alloc` and `dealloc`, which correspond to the `allocate` and `deallocate` functions we used in our examples:

- The `alloc` method takes a `Layout` instance as argument, which describes the desired size and alignment that the allocated memory should have. It returns a [raw pointer](#) to the first byte of the allocated memory block. Instead of an explicit error value, the `alloc` method returns a null pointer to signal an allocation error. This is a bit non-idiomatic, but it has the advantage that wrapping existing system allocators is easy, since they use the same convention.
- The `dealloc` method is the counterpart and responsible for freeing a memory block again. It receives two arguments, the pointer returned by `alloc` and the `Layout` that was used for the allocation.

The trait additionally defines the two methods `alloc_zeroed` and `realloc` with default implementations:

- The `alloc_zeroed` method is equivalent to calling `alloc` and then setting the allocated memory block to zero, which is exactly what the provided default implementation does. An allocator implementation can override the default implementations with a more efficient custom implementation if possible.
- The `realloc` method allows to grow or shrink an allocation. The default implementation allocates a new memory block with the desired size and copies over all the content from the previous allocation. Again, an allocator implementation can probably provide a more efficient implementation of this method, for example by growing/shrinking the allocation in-place if possible.

## Unsafety

One thing to notice is that both the trait itself and all trait methods are declared as `unsafe` :

- The reason for declaring the trait as `unsafe` is that the programmer must guarantee that the trait implementation for an allocator type is correct. For example, the `alloc` method must never return a memory block that is already used somewhere else because this would cause undefined behavior.
- Similarly, the reason that the methods are `unsafe` is that the caller must ensure various invariants when calling the methods, for example that the `Layout` passed to `alloc` specifies a non-zero size. This is not really relevant in practice since the methods are normally called directly by the compiler, which ensures that the requirements are met.

## A DummyAllocator

Now that we know what an allocator type should provide, we can create a simple dummy allocator. For that we create a new `allocator` module:

```
// in src/lib.rs

pub mod allocator;
```

Our dummy allocator does the absolute minimum to implement the trait and always return an error when `alloc` is called. It looks like this:

```
// in src/allocator.rs

use alloc::alloc::{GlobalAlloc, Layout};
use core::ptr::null_mut;

pub struct Dummy;

unsafe impl GlobalAlloc for Dummy {
    unsafe fn alloc(&self, _layout: Layout) -> *mut u8 {
        null_mut()
    }
}
```

```

    }

    unsafe fn dealloc(&self, _ptr: *mut u8, _layout: Layout) {
        panic!("dealloc should be never called")
    }
}

```

The struct does not need any fields, so we create it as a [zero sized type](#). As mentioned above, we always return the null pointer from `alloc`, which corresponds to an allocation error. Since the allocator never returns any memory, a call to `dealloc` should never occur. For this reason we simply panic in the `dealloc` method. The `alloc_zeroed` and `realloc` methods have default implementations, so we don't need to provide implementations for them.

We now have a simple allocator, but we still have to tell the Rust compiler that it should use this allocator. This is where the `#[global_allocator]` attribute comes in.

## The `#[global_allocator]` Attribute

The `#[global_allocator]` attribute tells the Rust compiler which allocator instance it should use as the global heap allocator. The attribute is only applicable to a `static` that implements the `GlobalAlloc` trait. Let's register an instance of our `Dummy` allocator as the global allocator:

```

// in src/allocator.rs

#[global_allocator]
static ALLOCATOR: Dummy = Dummy;

```

Since the `Dummy` allocator is a [zero sized type](#), we don't need to specify any fields in the initialization expression.

When we now try to compile it, the first error should be gone. Let's fix the remaining second error:

```

error: `#[alloc_error_handler]` function required, but not found

```

## The `#[alloc_error_handler]` Attribute

As we learned when discussing the `GlobalAlloc` trait, the `alloc` function can signal an allocation error by returning a null pointer. The question is: how should the Rust runtime react to such an allocation failure? This is where the `#[alloc_error_handler]` attribute comes in. It specifies a function that is called when an allocation error occurs, similar to how our panic handler is called when a panic occurs.

Let's add such a function to fix the compilation error:

```
// in src/lib.rs

#![feature(alloc_error_handler)] // at the top of the file

#[alloc_error_handler]
fn alloc_error_handler(layout: alloc::alloc::Layout) -> ! {
    panic!("allocation error: {:?}", layout)
}
```

The `alloc_error_handler` function is still unstable, so we need a feature gate to enable it. The function receives a single argument: the `Layout` instance that was passed to `alloc` when the allocation failure occurred. There's nothing we can do to resolve the failure, so we just panic with a message that contains the `Layout` instance.

With this function, the compilation errors should be fixed. Now we can use the allocation and collection types of `alloc`, for example we can use a `Box` to allocate a value on the heap:

```
// in src/main.rs

extern crate alloc;

use alloc::boxed::Box;

fn kernel_main(boot_info: &'static BootInfo) -> ! {
    // [...] print "Hello World!", call `init`, create `mapper` and `frame_allocator`

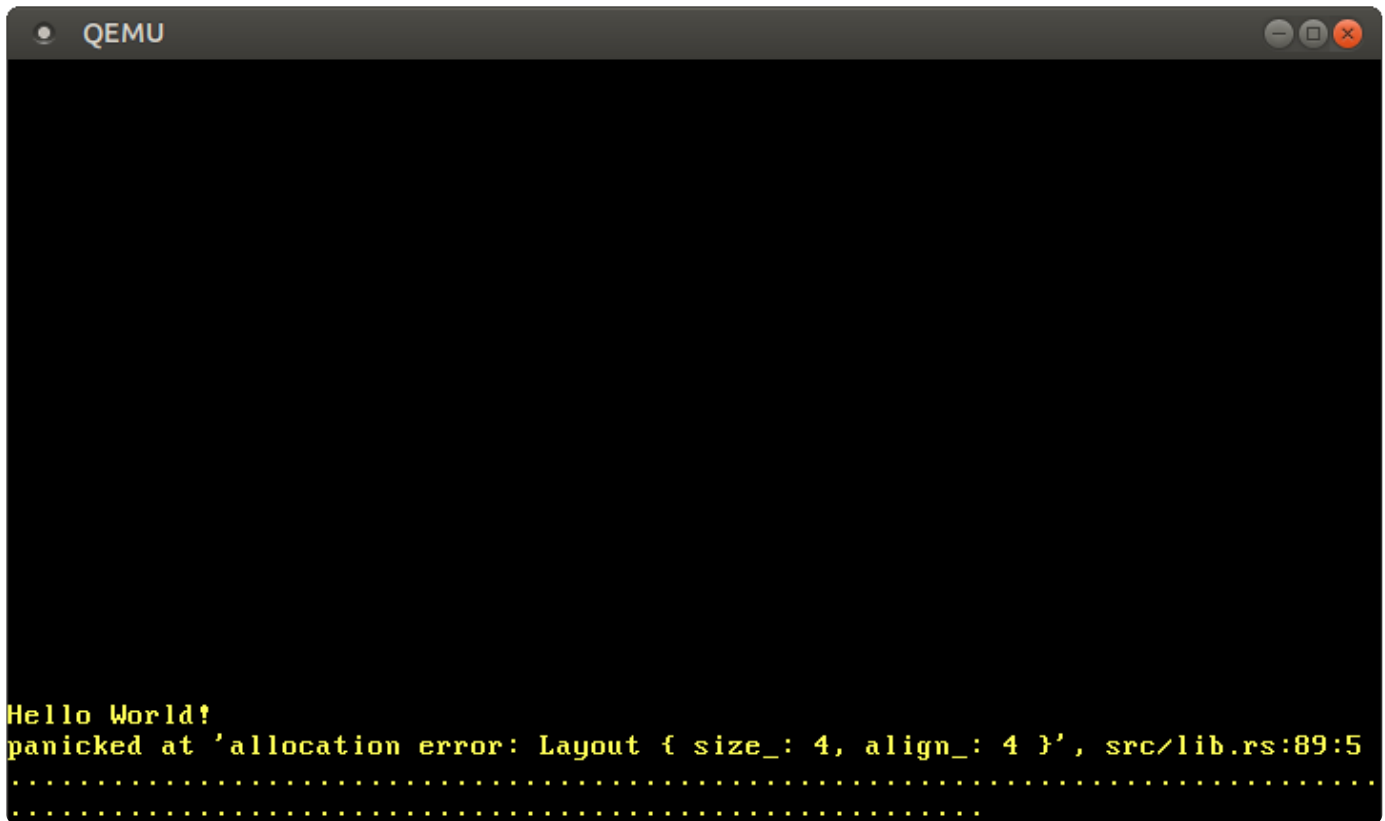
    let x = Box::new(41);

    // [...] call `test_main` in test mode

    println!("It did not crash!");
    blog_os::hlt_loop();
}
```

Note that we need to specify the `extern crate alloc` statement in our `main.rs` too. This is required because the `lib.rs` and `main.rs` part are treated as separate crates. However, we don't need to create another `#[global_allocator]` static because the global allocator applies to all crates in the project. In fact, specifying an additional allocator in another crate would be an error.

When we run the above code, we see that our `alloc_error_handler` function is called:

A screenshot of a QEMU terminal window. The window has a dark background and a title bar that says "QEMU". The terminal output shows "Hello World!" followed by a panic message: "panicked at 'allocation error: Layout { size\_: 4, align\_: 4 }', src/lib.rs:89:5". The message is followed by several lines of dots, indicating a stack trace or additional details that are not fully visible.

```
QEMU
Hello World!
panicked at 'allocation error: Layout { size_: 4, align_: 4 }', src/lib.rs:89:5
.....
.....
```

The error handler is called because the `Box::new` function implicitly calls the `alloc` function of the global allocator. Our dummy allocator always returns a null pointer, so every allocation fails. To fix this we need to create an allocator that actually returns usable memory.

## Creating a Kernel Heap

Before we can create a proper allocator, we first need to create a heap memory region from which the allocator can allocate memory. To do this, we need to define a virtual memory range for the heap region and then map this region to physical frames. See the ["Introduction To Paging"](#) post for an overview of virtual memory and page tables.

The first step is to define a virtual memory region for the heap. We can choose any virtual address range that we like, as long as it is not already used for a different memory region. Let's define it as the memory starting at address `0x_4444_4444_0000` so that we can easily recognize a heap pointer later:

```
// in src/allocator.rs

pub const HEAP_START: usize = 0x_4444_4444_0000;
pub const HEAP_SIZE: usize = 100 * 1024; // 100 KiB
```

We set the heap size to 100 KiB for now. If we need more space in the future, we can simply increase it.

If we tried to use this heap region now, a page fault would occur since the virtual memory region is not mapped to physical memory yet. To resolve this, we create an `init_heap` function that maps the heap pages using the `Mapper` API that we introduced in the *"Paging Implementation"* post:

```
// in src/allocator.rs

use x86_64::{
    structures::paging::{
        mapper::MapToError, FrameAllocator, Mapper, Page, PageTableFlags, Size4KiB,
    },
    VirtAddr,
};

pub fn init_heap(
    mapper: &mut impl Mapper<Size4KiB>,
    frame_allocator: &mut impl FrameAllocator<Size4KiB>,
) -> Result<(), MapToError<Size4KiB>> {
    let page_range = {
        let heap_start = VirtAddr::new(HEAP_START as u64);
        let heap_end = heap_start + HEAP_SIZE - 1u64;
        let heap_start_page = Page::containing_address(heap_start);
        let heap_end_page = Page::containing_address(heap_end);
        Page::range_inclusive(heap_start_page, heap_end_page)
    };

    for page in page_range {
        let frame = frame_allocator
            .allocate_frame()
            .ok_or(MapToError::FrameAllocationFailed)?;
        let flags = PageTableFlags::PRESENT | PageTableFlags::WRITABLE;
        unsafe {
            mapper.map_to(page, frame, flags, frame_allocator)?.flush()
        };
    }

    Ok(())
}
```

The function takes mutable references to a `Mapper` and a `FrameAllocator` instance, both limited to 4KiB pages by using `Size4KiB` as generic parameter. The return value of the function is a `Result` with the unit type `()` as success variant and a `MapToError` as error variant, which is the error type returned by the `Mapper::map_to` method. Reusing the error type makes sense here because the `map_to` method is the main source of errors in this function.

The implementation can be broken down into two parts:

- **Creating the page range::** To create a range of the pages that we want to map, we convert the `HEAP_START` pointer to a `VirtAddr` type. Then we calculate the heap end address from it by adding the `HEAP_SIZE`. We want an inclusive bound (the address of the last byte of the heap), so we subtract 1. Next, we convert the addresses into `Page` types using the `containing_address` function. Finally, we create a page range from the start and end pages using the `Page::range_inclusive` function.
- **Mapping the pages:** The second step is to map all pages of the page range we just created. For that we iterate over the pages in that range using a `for` loop. For each page, we do the following:
  - We allocate a physical frame that the page should be mapped to using the `FrameAllocator::allocate_frame` method. This method returns `None` when there are no more frames left. We deal with that case by mapping it to a `MapToError::FrameAllocationFailed` error through the `Option::ok_or` method and then apply the `question mark operator` to return early in the case of an error.
  - We set the required `PRESENT` flag and the `WRITABLE` flag for the page. With these flags both read and write accesses are allowed, which makes sense for heap memory.
  - We use the `Mapper::map_to` method for creating the mapping in the active page table. The method can fail, therefore we use the `question mark operator` again to forward the error to the caller. On success, the method returns a `MapperFlush` instance that we can use to update the *translation lookaside buffer* using the `flush` method.

The final step is to call this function from our `kernel_main`:

```
// in src/main.rs

fn kernel_main(boot_info: &'static BootInfo) -> ! {
    use blog_os::allocator; // new import
    use blog_os::memory::{self, BootInfoFrameAllocator};

    println!("Hello World{}", "!");
    blog_os::init();

    let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
    let mut mapper = unsafe { memory::init(phys_mem_offset) };
    let mut frame_allocator = unsafe {
        BootInfoFrameAllocator::init(&boot_info.memory_map)
    };

    // new
    allocator::init_heap(&mut mapper, &mut frame_allocator)
```

```

        .expect("heap initialization failed");

    let x = Box::new(41);

    // [...] call `test_main` in test mode

    println!("It did not crash!");
    blog_os::hlt_loop();
}

```

We show the full function for context here. The only new lines are the `blog_os::allocator` import and the call to `allocator::init_heap` function. In case the `init_heap` function returns an error, we panic using the `Result::expect` method since there is currently no sensible way for us to handle this error.

We now have a mapped heap memory region that is ready to be used. The `Box::new` call still uses our old `Dummy` allocator, so you will still see the "out of memory" error when you run it. Let's fix this by using a proper allocator.

## Using an Allocator Crate

Since implementing an allocator is somewhat complex, we start by using an external allocator crate. We will learn how to implement our own allocator in the next post.

A simple allocator crate for `no_std` applications is the `linked_list_allocator` crate. It's name comes from the fact that it uses a linked list data structure to keep track of deallocated memory regions. See the next post for a more detailed explanation of this approach.

To use the crate, we first need to add a dependency on it in our `Cargo.toml` :

```

# in Cargo.toml

[dependencies]
linked_list_allocator = "0.9.0"

```

Then we can replace our dummy allocator with the allocator provided by the crate:

```

// in src/allocator.rs

use linked_list_allocator::LockedHeap;

#[global_allocator]
static ALLOCATOR: LockedHeap = LockedHeap::empty();

```

The struct is named `LockedHeap` because it uses the `spinning_top::Spinlock` type for synchronization. This is required because multiple threads could access the `ALLOCATOR` static at



the same time. As always when using a spinlock or a mutex, we need to be careful to not accidentally cause a deadlock. This means that we shouldn't perform any allocations in interrupt handlers, since they can run at an arbitrary time and might interrupt an in-progress allocation.

Setting the `LockedHeap` as global allocator is not enough. The reason is that we use the `empty` constructor function, which creates an allocator without any backing memory. Like our dummy allocator, it always returns an error on `alloc`. To fix this, we need to initialize the allocator after creating the heap:

```
// in src/allocator.rs

pub fn init_heap(
    mapper: &mut impl Mapper<Size4KiB>,
    frame_allocator: &mut impl FrameAllocator<Size4KiB>,
) -> Result<(), MapToError<Size4KiB>> {
    // [...] map all heap pages to physical frames

    // new
    unsafe {
        ALLOCATOR.lock().init(HEAP_START, HEAP_SIZE);
    }

    Ok(())
}
```

We use the `lock` method on the inner spinlock of the `LockedHeap` type to get an exclusive reference to the wrapped `Heap` instance, on which we then call the `init` method with the heap bounds as arguments. It is important that we initialize the heap *after* mapping the heap pages, since the `init` function already tries to write to the heap memory.

After initializing the heap, we can now use all allocation and collection types of the built-in `alloc` crate without error:

```
// in src/main.rs

use alloc::{boxed::Box, vec, vec::Vec, rc::Rc};

fn kernel_main(boot_info: &'static BootInfo) -> ! {
    // [...] initialize interrupts, mapper, frame_allocator, heap

    // allocate a number on the heap
    let heap_value = Box::new(41);
    println!("heap_value at {:p}", heap_value);

    // create a dynamically sized vector
    let mut vec = Vec::new();
    for i in 0..500 {
        vec.push(i);
    }
}
```

```

    .....
}
println!("vec at {:p}", vec.as_slice());

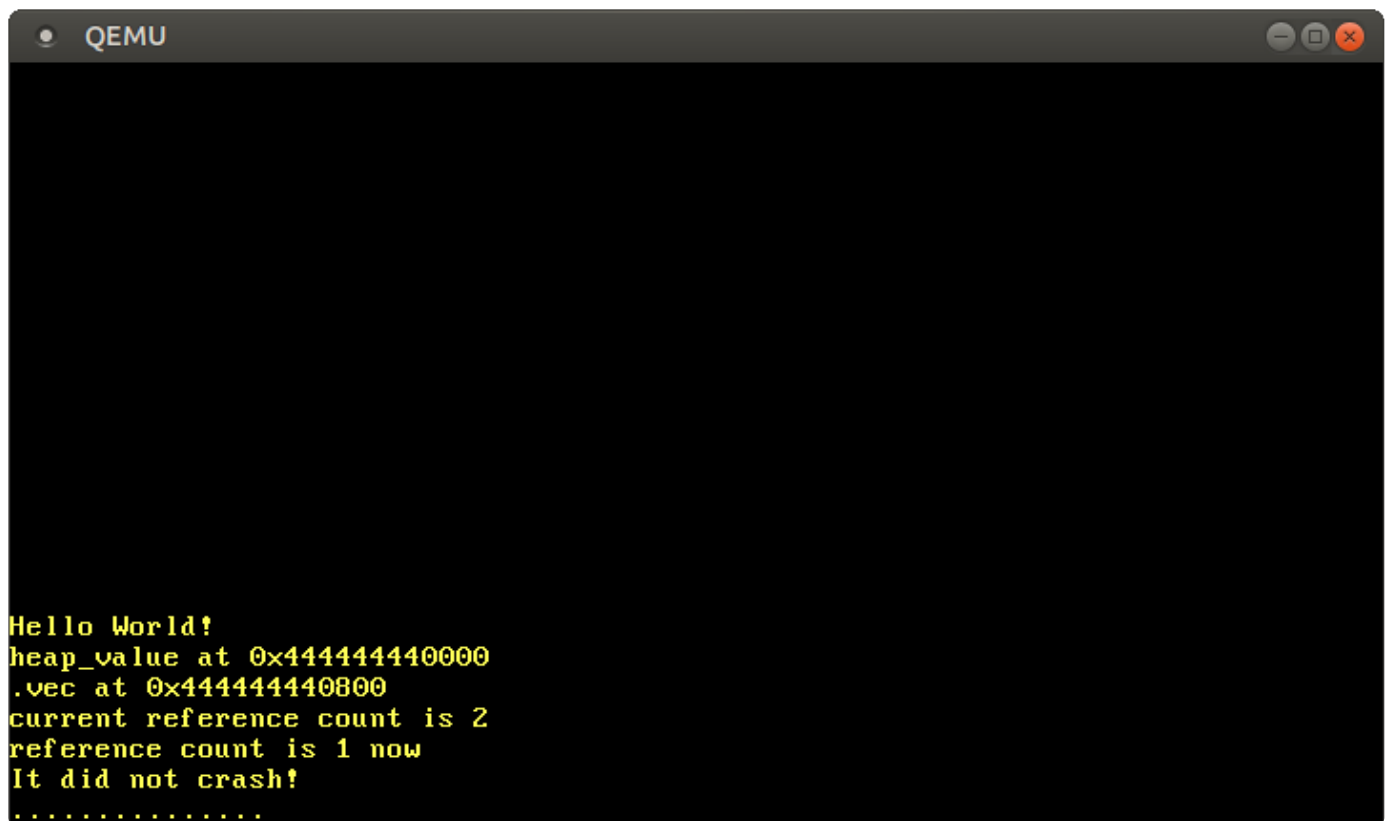
// create a reference counted vector -> will be freed when count reaches 0
let reference_counted = Rc::new(vec![1, 2, 3]);
let cloned_reference = reference_counted.clone();
println!("current reference count is {}", Rc::strong_count(&cloned_reference));
core::mem::drop(reference_counted);
println!("reference count is {} now", Rc::strong_count(&cloned_reference));

// [...] call `test_main` in test context
println!("It did not crash!");
blog_os::hlt_loop();
}

```

This code example shows some uses of the `Box`, `Vec`, and `Rc` types. For the `Box` and `Vec` types we print the underlying heap pointers using the `{:p}` formatting specifier. For showcasing `Rc`, we create a reference counted heap value and use the `Rc::strong_count` function to print the current reference count, before and after dropping an instance (using `core::mem::drop`).

When we run it, we see the following:



```

QEMU
Hello World!
heap_value at 0x444444440000
.vec at 0x444444440800
current reference count is 2
reference count is 1 now
It did not crash!
.....

```

As expected, we see that the `Box` and `Vec` values live on the heap, as indicated by the pointer starting with the `0x_4444_4444_*` prefix. The reference counted value also behaves as

expected, with the reference count being 2 after the `clone` call, and 1 again after one of the instances was dropped.

The reason that the vector starts at offset `0x800` is not that the boxed value is `0x800` bytes large, but the [reallocations](#) that occur when the vector needs to increase its capacity. For example, when the vector's capacity is 32 and we try to add the next element, the vector allocates a new backing array with capacity 64 behind the scenes and copies all elements over. Then it frees the old allocation.

Of course there are many more allocation and collection types in the `alloc` crate that we can now all use in our kernel, including:

- the thread-safe reference counted pointer `Arc`
- the owned string type `String` and the `format!` macro
- `LinkedList`
- the growable ring buffer `VecDeque`
- the `BinaryHeap` priority queue
- `BTreeMap` and `BTreeSet`

These types will become very useful when we want to implement thread lists, scheduling queues, or support for `async/await`.

## Adding a Test

To ensure that we don't accidentally break our new allocation code, we should add an integration test for it. We start by creating a new `tests/heap_allocation.rs` file with the following content:

```
// in tests/heap_allocation.rs

#![no_std]
#![no_main]
#![feature(custom_test_frameworks)]
#![test_runner(blog_os::test_runner)]
#![reexport_test_harness_main = "test_main"]

extern crate alloc;

use bootloader::{entry_point, BootInfo};
use core::panic::PanicInfo;

entry_point!(main);

fn main(boot_info: &'static BootInfo) -> ! {
    unimplemented!();
}

#[panic_handler]
```

```
fn panic(info: &PanicInfo) -> ! {
    blog_os::test_panic_handler(info)
}
```

We reuse the `test_runner` and `test_panic_handler` functions from our `lib.rs`. Since we want to test allocations, we enable the `alloc` crate through the `extern crate alloc` statement. For more information about the test boilerplate check out the [Testing](#) post.

The implementation of the `main` function looks like this:

```
// in tests/heap_allocation.rs

fn main(boot_info: &'static BootInfo) -> ! {
    use blog_os::allocator;
    use blog_os::memory::{self, BootInfoFrameAllocator};
    use x86_64::VirtAddr;

    blog_os::init();
    let phys_mem_offset = VirtAddr::new(boot_info.physical_memory_offset);
    let mut mapper = unsafe { memory::init(phys_mem_offset) };
    let mut frame_allocator = unsafe {
        BootInfoFrameAllocator::init(&boot_info.memory_map)
    };
    allocator::init_heap(&mut mapper, &mut frame_allocator)
        .expect("heap initialization failed");

    test_main();
    loop {}
}
```

It is very similar to the `kernel_main` function in our `main.rs`, with the differences that we don't invoke `println`, don't include any example allocations, and call `test_main` unconditionally.

Now we're ready to add a few test cases. First, we add a test that performs some simple allocations using `Box` and checks the allocated values, to ensure that basic allocations work:

```
// in tests/heap_allocation.rs
use alloc::boxed::Box;

#[test_case]
fn simple_allocation() {
    let heap_value_1 = Box::new(41);
    let heap_value_2 = Box::new(13);
    assert_eq!(*heap_value_1, 41);
    assert_eq!(*heap_value_2, 13);
}
```

Most importantly, this test verifies that no allocation error occurs.

Next, we iteratively build a large vector, to test both large allocations and multiple allocations (due to reallocations):

```
// in tests/heap_allocation.rs

use alloc::vec::Vec;

#[test_case]
fn large_vec() {
    let n = 1000;
    let mut vec = Vec::new();
    for i in 0..n {
        vec.push(i);
    }
    assert_eq!(vec.iter().sum:::<u64>(), (n - 1) * n / 2);
}
```

We verify the sum by comparing it with the formula for the [n-th partial sum](#). This gives us some confidence that the allocated values are all correct.

As a third test, we create ten thousand allocations after each other:

```
// in tests/heap_allocation.rs

use blog_os::allocator::HEAP_SIZE;

#[test_case]
fn many_boxes() {
    for i in 0..HEAP_SIZE {
        let x = Box::new(i);
        assert_eq!(*x, i);
    }
}
```

This test ensures that the allocator reuses freed memory for subsequent allocations since it would run out of memory otherwise. This might seem like an obvious requirement for an allocator, but there are allocator designs that don't do this. An example is the bump allocator design that will be explained in the next post.

Let's run our new integration test:

```
> cargo test --test heap_allocation
[...]
```

Running 3 tests

```
simple_allocation... [ok]
large_vec... [ok]
many_boxes... [ok]
```

All three tests succeeded! You can also invoke `cargo test` (without `--test` argument) to run all unit and integration tests.

## Summary

This post gave an introduction to dynamic memory and explained why and where it is needed. We saw how Rust's borrow checker prevents common vulnerabilities and learned how Rust's allocation API works.

After creating a minimal implementation of Rust's allocator interface using a dummy allocator, we created a proper heap memory region for our kernel. For that we defined a virtual address range for the heap and then mapped all pages of that range to physical frames using the `Mapper` and `FrameAllocator` from the previous post.

Finally, we added a dependency on the `linked_list_allocator` crate to add a proper allocator to our kernel. With this allocator, we were able to use `Box`, `Vec`, and other allocation and collection types from the `alloc` crate.

## What's next?

While we already added heap allocation support in this post, we left most of the work to the `linked_list_allocator` crate. The next post will show in detail how an allocator can be implemented from scratch. It will present multiple possible allocator designs, shows how to implement simple versions of them, and explain their advantages and drawbacks.

## Support Me

Creating and [maintaining](#) this blog and the associated libraries is a lot of work, but I really enjoy doing it. By supporting me, you allow me to invest more time in new content, new features, and continuous maintenance.

The best way to support me is to [sponsor me on GitHub](#), since they don't charge any fees. If you prefer other platforms, I also have [Patreon](#) and [Donorbox](#) accounts. The latter is the most flexible as it supports multiple currencies and one-time contributions.

Thank you!

---

[« Paging Implementation](#)

[Allocator Designs »](#)

---

**0 reactions****0 comments**

---

Write

Preview

Sign in to comment

Styling with Markdown is supported

Sign in with GitHub

Instead of authenticating the [giscus](#) application, you can also comment directly on the on GitHub. Just click the "*X comments*" link at the [top](#) — or the date of any comment — to go to the GitHub discussion.

## About Me

I'm a Rust freelancer with a master's degree in computer science. I love systems programming, open source software, and new challenges.

If you want to work with me, reach out on [LinkedIn](#) or write me at [job@phil-opp.com](mailto:job@phil-opp.com).

---

© 2021. All rights reserved. [Contact](#)