Supplement: Your Own Memory Allocator

Leonard König, April 12, 2021

Creating new Structures

In a previous exercise you had to iterate over given linked lists to schedule the correct process. The next step is for you to create and manage your own linked list as a means to manage (free) memory blocks. However, before we look into that, let's recap how we can actually *define* a new linked list type, or more precisely, a node type for use in a linked list. An example is given in **Listing 1**.

```
struct node {
    struct node *next;
    struct node *prev;

    int value;
};

struct node dummy = {
    .next = &dummy,
    .prev = &dummy,

    /* we don't care about the value */
};

struct node *head = &dummy;
```

Listing 1 Defining a new linked list

The code consists of three parts, the declaration of a new struct-type, the definition of a dummy value of the previously declared type, and the definition of a pointer to said dummy.

Structures are the way to define new composite data types in C, i.e. a new type is made up of its 'members'. In this case, a node should consist of a pointer to the next and previous node. A linked list as a data structure would be of not much use, however, if we couldn't save additional data in each node. In this case we chose to store an integer number in each node. Each variable with this new

type will be big enough to store all of the members, i.e. the following holds:[1]

```
sizeof (struct node) >= sizeof (struct node *) +
                        sizeof (struct node *) +
                        sizeof (int)
```

[1] You can pass types to `sizeof` as well as variables, the latter will behave as if you've passed the type of the variable.

In memory, one(!) possible layout of a single node could look like given in **Figure 2**.

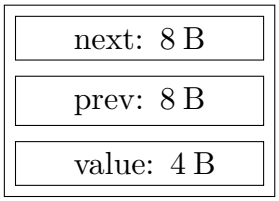

Figure 2 A node in memory

It is thus critical that the members of the structure are not of the type `struct node` but *pointers*, as otherwise a node would need be able to hold another node, which then would need to be able to hold another node in turn.

Memory Allocation

In order to create an instance of our newly defined structure we chose to create a global variable as this is the least obstructive way. We chose to name this specific instance 'dummy' as it will not hold any relevant data, but is merely there to denote the start/end of the list. It is initialized to be its own previous and succeeding value, thus creating a loop as shown in **Figure 3**.

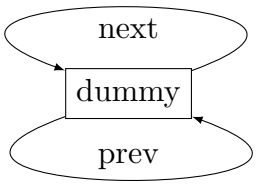

Figure 3 An empty list

If we hadn't such a dummy value we could not model such an empty list, but only a non-existant list and a list with $n > 0$ nodes.

When we later want to give this list to another function, we don't want to give it a copy of the dummy value, so we create a pointer to our dummy value instead and will call it 'head' as it *points to the head of the list*.

In order to re-create the situation in Figure ?? we need to create two new nodes and wire them up in such a way that they can be accessed by anyone who holds the head pointer. We defined the dummy value globally as this also implied that the dummy value will 'exist' for the whole runtime of the program.[2] This was fine with us at the time, but at least our new nodes are supposed to 'come and go' as we run through the program. There are two ways to handle this kind of allocation (setting aside enough memory to hold the data), automatic allocation and dynamic allocation. Till now we implicitly used automatic allocation as it, well, happens 'automatically'. Consider the code given in **Listing 4**.

[2] We call this 'static allocation'.

```
void foo(void)
{
    // automatic allocation: a is 'created'
    int a;

    while (/* some condition */) {
        // automatic allocation: b
        int b;

        /* ... */
    }
    // automatic de-allocation: b is 'destroyed'

}
// automatic de-allocation: a
```

Listing 4 Automatic allocation

We create two variables `a` and `b` which are allocated as soon as they are declared. This means the compiler handles creating the space 'somewhere' for us, s.t. we are later able to store an integer in it. As soon as they exit the innermost 'scope' delimited by curly braces

they are declared in, they are deallocated as well.[3] We frequently need an object (in this case a node) to persist longer than just the function we created it in. Otherwise it would be impossible to create function that takes an integer, creates a node to contain it and enqueues it in the list given:

```
void enqueue(struct node *head, int value);
```

The newly created node would be deallocated as `enqueue` finishes and the caller of the function would be left with a list which contain pointers to already freed memory (so-called dangling pointers). The **Figure 5** shows this scenario if the second node was enqueued in that way.

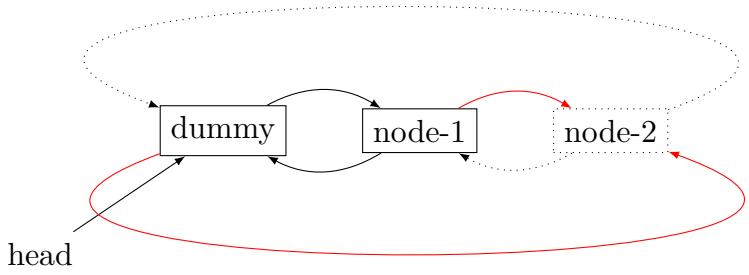

Figure 5 Dangling pointers

In order to be able to create objects of memory that persist longer but still can be created at demand, we need to look into dynamic memory allocation. Unfortunately, this needs help by the operating system, which in our case comes in form of the standard C library. The header `stdlib.h` provides a function for dynamic memory allocation which takes the size of the object to be created and returns a pointer to it, if the allocation succeeded (it may not, if we don't have enough memory!):

```
#include <stdlib.h>

void *malloc(size_t size);
```

Our example in **Listing 4** could be adapted to use more the more flexible but more tedious dynamic allocation for the allocation

[3] This is *usually* facilated by creating space on the 'stack' as the space is needed and removing it from the stack afterwards. Specifically, if a function ends, it will remove destroy all objects allocated in this way.

of the integers instead. As the `malloc` function has no way to actually 'return memory' it will only return a pointer to us, which we need to store in an automatically allocated variable as can be seen in **Listing 6**.

```
void foo(void)
{
    // automatic allocation of a pointer(!)
    int *pa;
    /* dynamic allocation of an int in memory,
     * address is assigned to the pointer
     */
    pa = malloc(sizeof (int));

    while (/* some condition */) {
        /* two in one:
         * automatic allocation of a pointer,
         * dynamic allocation of the int
         */
        int *pb = malloc(sizeof (*pb));

        /* ... */

        /* manual de-allocation of the object
         * pointed to by pb
         */
        free(pb);
    }

    // manual de-allocation
    free(pa);
}
```

Listing 6 Dynamic allocation

We also now need to manually de-allocate the memory after we are done using it, in order to not keep on asking the operating system for memory without giving anything back. So while the variable `pa` (or `pb`) might come and go when it comes out of scope, the actual memory object 'behind' it, persists, even if we'd have no pointer

pointing to it. This situation, 'memory leak', is what happened in **Figure 7**.

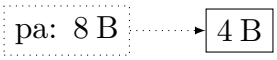

Figure 7 Memory leak

This danger is also the *strength* of dynamic allocation, as we aren't forced to deallocate the objects at the end of their pointers' scope—we merely did this to show equivalent code.

With this equipped, we can now make our first useful dynamic allocation, namely implementing the `enqueue` function from before which is given in **Listing 8**.

```
void enqueue(struct node *head, int value)
{
    struct node *new = malloc(sizeof (*new));
    new->value = value;

    new->next = head;
    new->prev = head->prev;

    new->prev->next = new;
    new->next->prev = new;
}
/* Note: the pointer new is deallocated,
 * but not the object it points to!
 */
```

Listing 8 Enqueue implementation

After we've allocated the new node and created a pointer to it, we can start copying the value over. While we could use the dot to access structures, we use the arrow to access structures behind a pointer, i.e. the following to are equivalent:

```
(*new).value = value; // dereference, then dot
new->value = value;   // "syntactic sugar"
```

We now can start wiring up the node as an element of our list as visualized in **Figure 9** (try following the code tracing the paths

in the graphic!) We start by creating links from our new node to the start of the list (pointed to by `head` itself) and its end (the dummys previous element) respectively. It's now possible for us to access the rest of the list from our new node, but not vice versa, yet. As the previous element to our new one is known to be the last element of the list (we just defined it so in step 2), we can now say that its next element in turn shall link back to our new node. Similarly, we create the backlink from the first / dummy node to our newly created one.[4] As we are finished, the pointer `new` goes out of scope and thus is destroyed, i.e. the memory that holds the address of our freshly created node is invalidated. However, there are other pointers to this node, so all is fine.

[4] Note that rearranging the steps might lead to wrong behavior!

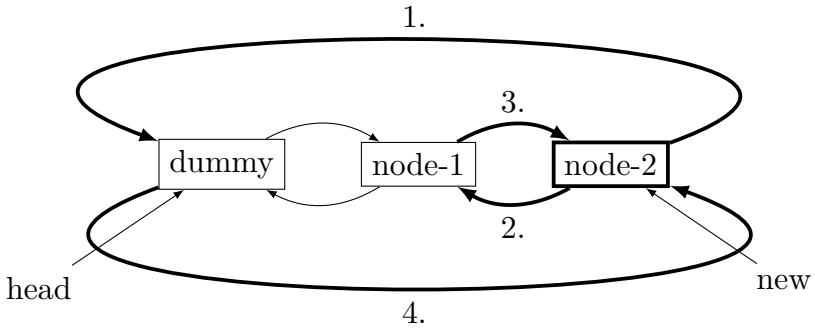

Figure 9 Enqueuing a new node

Your Own Memory Allocator

When we were given the memory by `malloc` the operating system gave us some space we could use for as long as we want. But it cannot just hand out memory as it likes, it needs to keep track of which memory areas it already 'gave away' and which it didn't. The most naïve way to do just that is... to use a linked list. Obviously this is some kind of a chicken-and-egg problem: To use a linked list and create nodes we need some kind of dynamic allocation in the first place, but we are just setting out to write our own!

We will solve this problem using some tricks to put our structures into memory that's already there—namely the one that we are trying to manpage. That means we are using up some of the memory just by managing it, called overhead.

We start by thinking of our complete RAM as one, big, continuous block. As we are actually not managing our actual RAM, we

cheat a little bit and create this block ourselves as a statically allocated array:

```
char memory[MEM_SIZE];
```

As a `char` in C is just another name for 'byte', we declare memory to be an array of as much bytes as the macro `MEM_SIZE` is defined to.

The simplest way to think of memory in a linked list is to think of our memory as consecutive free/allocated blocks, with each block being managed by a node *preceeding* this block of memory (a 'header'). Before we have any allocation request, all memory is free, which means we have one free block containing 'all' memory (well, all that is left after we subtract our overhead), and thus one element *in addition to the dummy* to manage said free space. However, we will make a small adjustment to our mental model and put the dummy at *the end of the memory* to delimit it (contrary to our list, our memory is not actually circular) as seen in **Figure 10**.

Figure 10 Before memory allocation

We will for now gloss over how we actually place our nodes 'inside' our array and consider a more complex snapshot of our memory.

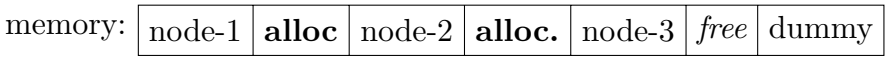

Figure 11 After some allocations

In **Figure 11** we have three nodes, managing the blocks directly following each, in total two allocated and one free area. As these are part of a (circular, doubly-) linked list, we can create pointers to each of the nodes.

Now we have to leave the abstract 'machine' that C defines and make the (rather common) assumption, that these pointers actually hold the *addresses of the nodes in memory*.[5] So, if we know that A is the address of node-1, and B the address of node-2, and each structure creates an overhead O, then the allocated memory inbetween is of size S given by:

$$S = B - A - O$$

[5] There's a pitfall here regarding pointer arithmetics that we will cover later. Pointers are, in fact, not addresses but just very closely related.

This means we can infer the size of each memory block, just by looking closely at the list. What we cannot deduce, however, is whether a block is free or allocated. Our structure for a node must thus hold a variable which is either true or false, specifying whether the following block is yet available.

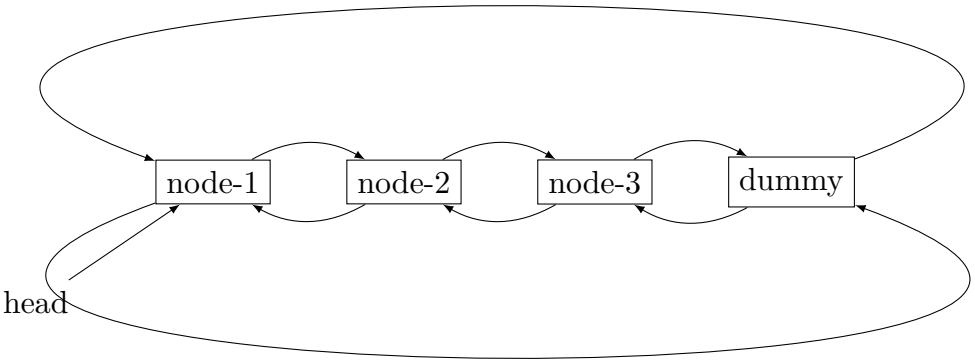

Figure 12 Linked list for memory management

From Theory to Practice

With these abstract observations in mind, we can actually look more closely on the code. If we want to recreate the situation in **Figure 10**, we need to place two structures within a given object of a different type. To do that, we first create a pointer to the start of our array `memory` (i.e. pointer to its first element) and force C to think of it as a pointer to our structure instead, which we can then use to fill it with our nodes data. This process is called 'casting' and makes us *reinterpret* the memory previously known as 'the first few bytes of the array `memory`'.

```
char memory[MEM_SIZE];
struct node *head = (struct node*) (&memory[0]);
```

Listing 13 Casting to a different pointer

Similarly, we place our dummy node at the very end of the array, but not at `memory[MEM_SIZE]`, as this would leave no room for the dummy node to go. Specifically, while we used the 0th element in

the memory array for placing the head, we want to use the index `MEM_SIZE - sizeof (struct node)` for the dummy. Afterwards we can just let `head->next` point to our `dummy` and so on.

We can now calculate the free memory we have left between those nodes, shown in **Listing 14** using our previously found formula:

```
size_t free = (char*)(dummy) - (char*)(head) -
    sizeof (struct node);
```

Listing 14 Calculating free memory

We cannot simply subtract two pointers and expect the difference to be the difference in bytes, however! As far as C is concerned, pointers are 'abstract' and don't even need to be numbers. C does though define pointer-arithmetics, specifically, the difference between two pointer objects is the *number of elements of the type of object they point to, that would fit in-between.* If the pointer is not pointing to a type of the size of one single byte, this number of elements is definitely lower than the number of bytes inbetween. However, if we treat our pointers as pointers to bytes (`char`) with a type-cast, those two numbers are the same.

A final more detailed overview of the design is given on **page 11**.

In Summary

You now know how to setup your own linked list and place it into memory, iterate over the list and calculate the free space in each block (difference between the addresses of consecutive nodes), are able to use this knowledge to find a free block that is large enough to fit the request (if one exists, that is) and allocate it [6]. Remember to return the address of the block after the node, not address to the node itself. Finally, given the address memory block, you can subtract the size of your management structure to calculate the start of the respective node. This should help clear the technical obstacles to implementing your own memory allocated, good luck!

[6] Note, if you end up splitting a block because it is large, you create a new metadata block, actually reducing the free memory left!

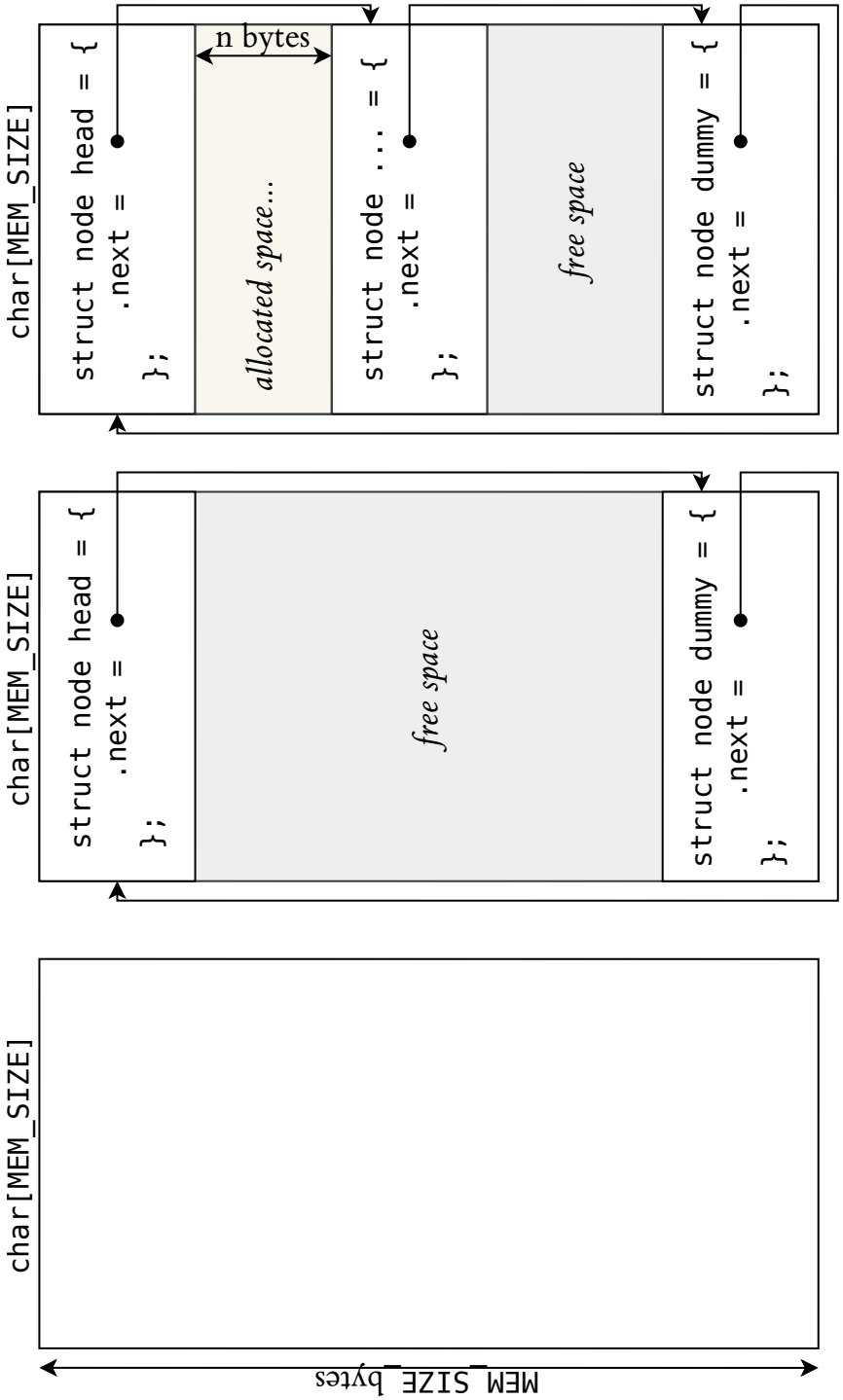

Figure 15 Overview: Memory Allocation