Note 5: Memory Management

Kevin Moy

1 Structs in C

Out of the primitive data types that we have, i.e. `int, char`, etc. we can make arbitrary data types, or **structs**. Unlike Java, these `structs` are a little more restricted: they only just exist in characteristics, i.e. no struct methods are allowed.

Here's an example of a struct that represents an 3D coordinate in real space:

```
struct coordinate {
    int x;
    int y;
    int z;
};
```

Once a struct is declared, it is treated as a valid data type just like all primitives in C, so it is allowed as a function parameter.

We can also have **pointers to structs**- in fact, this is the preferred method because a pointer is almost always more efficient to pass into a function than the entire struct. However, dereferencing a struct pointer would just give us the original struct object. We'd still have to use **dot notation** after the dereference to access **fields** of the struct:

```
printf("x coordinate: %d\n", (*p).x);
```

This is kind of annoying, so C made an **arrow operator** `->` that dereferences a struct field all in one. So to access the x coordinate from coordinate pointer

`p`, we'd write `p->x`, which is exactly equivalent to `(*p).x`.

2 Struct Size, Word Alignment

How big are structs? Do we just add all the sizes of the constituent fields of the struct? Formally, what does `sizeof(struct)` return?

The answer is that it depends. The compiler might implement **word alignment**, in which the sum of sizes of fields is rounded UP to the nearest word (multiple of 4 bytes in our case). Consider struct `foo` below:

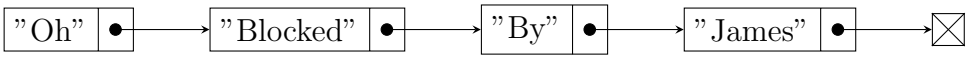
```
struct foo {
    char x;
    char y;
    char z;
    int xyz;
}
```

Summing up the total sizes gives $1 + 1 + 1 + 4 = 7$. However, if compilers do word alignment, `sizeof(foo)` would return 8, since that's the nearest multiple of 4 we can round 7 up to.

3 Our first struct

Given our knowledge of pointers and memory (de)allocation, now it's time to implement a practical `struct` for ourselves. We'll implement a linked list of strings in this section.

All linked lists consist of **nodes** that contain a value and a pointer to the next node. A node that points to a null value signifies the end node of a linked list.



In C code:

```
struct Node {
    char *value;
```

```
    struct Node *next;
}
typedef struct Node *List;

List ListNew(void){
    return NULL;
}
```

Note `typedef` is used when we want to give some other name to an already existing type: because a linked list is actually fully represented by the head node, we simply call the pointer to this node `List`.

We also made a "starter" function `ListNew` that returns a NULL list, i.e. a `Node` pointer that points to `NULL`.

Let's now implement adding a node to our linked list- in this case, adding a node with some string as its value. This node will be added to the FRONT of the linked list. Let's analyze the method given in lecture. Note that this is a STATIC method, since object-oriented methods don't exist for structs in C.

```
List list_add(List list, char *string) {
    struct Node *node = (struct Node*) malloc(sizeof(struct Node));
    node->value = (char*) malloc(strlen(string) + 1);
    strcpy(node->value, string);
    node->next = list;
    return node;
}
```

First notice our function signature: we are adding a node to a string linked list, so it makes sense that we need to first pass in our `List` (again, just a node pointer), and the `string` (`char` array) we want to place in our node as our value.

The very first thing we do is *dynamically* allocate space for our `Node` via `malloc`. Since `malloc` returns a `void` pointer to that allocated space, we cast it to a `Node` pointer then assign it to `node` for future use.

Next, we want to assign our node the string we passed into the function. Because we want to set a pointer *field* to something, we must *again* allocate

the necessary space via `malloc`: specifically, exactly as much space as `string` + 1 for the null terminator character.

Now that we've allocated this space and in essence allocated an array at `node->value`, we can simply copy the char array `string` to that array, effectively setting the node string.

Finally, to connect that node to the beginning of the linked list, we simply point `node->next` to the original (head node) `list`. Note we're setting it to `list`, another node pointer: setting a pointer's value to another pointer is essentially just points both to the same place, since both pointers will contain the same memory address. So now both `node->next` and `list` will point to the **second** node of our linked list, as they should.

Finally, we return our new List `node`- it points to our new head node and thus is the new representative of our linked list.

4 Memory Allocation

Keep in mind that declaring `struct`s does NOT allocate any memory- only `malloc` and variable declaration does! So far we've covered a couple ways to allocate memory in C: either by beginning-of-function variable declarations like `int i; char* str;` OR dynamic allocation by calling `malloc` (we'll soon learn some other `(.*)alloc` functions).

Side note: Any variables declared outside of `main` are called **global** variables, since they have global scope (throughout file OR project directory)- memory is allocated for them in the exact same way as local variables.

5 C Memory

How does C manage its memory? There are three main sections of memory: **stack, heap, and static** memory.

Stack memory is reserved for local variables and parameters. It grows downwards as more things are added to the stack. It is also where functions' **return addresses** are placed- i.e. a pointer to the location of code where execution resumes after the function returns.

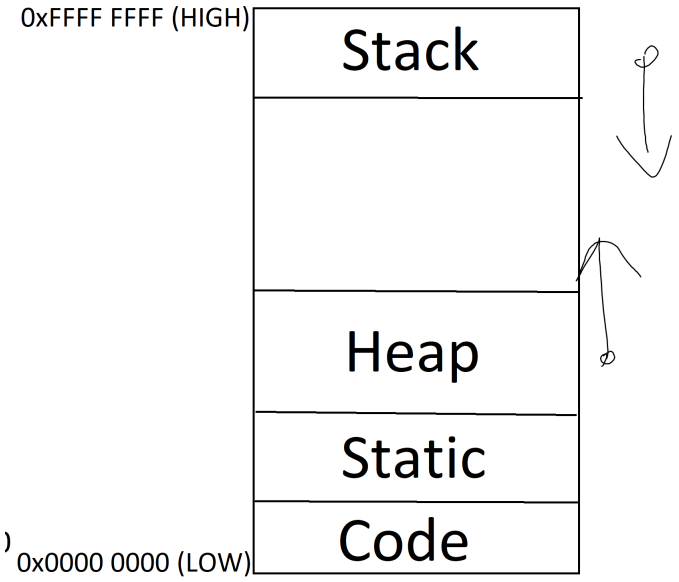
Heap memory is reserved entirely for dynamic memory allocation and storage. Allocation of heap memory is not contiguous- simple chunks of memory sufficient for a `malloc` request are just taken wherever. Heap memory is considered taken up until it is *specifically* deallocated via `free`, as you learned last note.

Finally, **static memory** is reserved for data that needs to exist for the duration of the entire program, i.e. `global` and `static` variables.

Note also that unlike Java, C requires knowledge of memory locations of objects in order for them to be used correctly.

5.1 Address Space Regions

4 regions of address space. From lowest to highest address, they are: code, static, heap, stack. Stack grows downwards, heap grows upwards and non-contiguously. Crude diagram below:



Unlike the heap and stack, the static and code sections always stay the same size. In particular, the code portion of memory is loaded immediately upon program start and never changes throughout execution.

6 Variable Allocation in Memory

Variables declared **outside** of any function are global and immediately placed into static storage. Variables declared **inside** functions, including inside `main`, are local variables and are allocated to the stack (in downwards fashion). When a function returns local variables are useless, so they are removed from the stack at that point. So if we have

```
int x;
int main() {
    int y;
    char z;
}
```

`y` and `z` will be allocated to stack memory, and `x` to static memory.

6.1 Stack Memory

For each function/subroutine called there is a **stack frame** in stack memory, which contains space for local variables, parameters for that function as well as that function's return address. There is also a **stack pointer** that points to the TOP stack frame- the earliest function called that hasn't returned yet.

So if I had the code below:

```
int main() {
    int x;
    foo();
}
void foo() {
    int y;
    bar();
}
void bar() {
    int z;
}
```

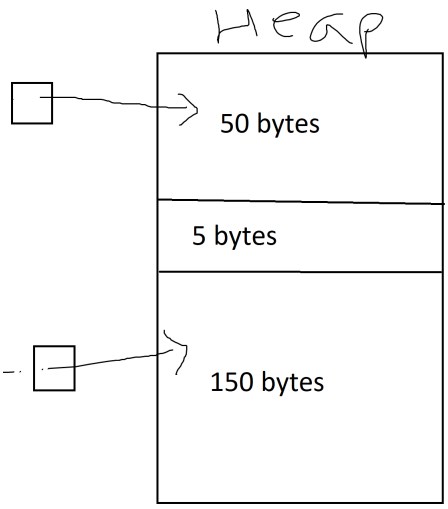
By the time `bar` is called we'd have our stack frames like this:

Notice the downward growth. When a function returns, its corresponding stack frame is removed from the stack- which means eventually after all subroutines return we'll just have a single `main` stack frame. These properties follow the stack's **Last-In, First Out (LIFO)** data structure property: the last stack frame inserted, in the above example `bar`, is also the first frame to be removed (when it returns).

6.2 Heap Memory

We know that heap memory is memory allocated dynamically (at runtime), and is not allocated contiguously unlike stack memory. It works pretty much like how Java allocates memory with its `new` keyword for objects. UNlike Java, however, remember that `malloc` makes us specify the exact amount of memory we need in bytes.

Managing heap memory is significantly more tricky than stack/static/code memory because memory can be (de)allocated at **any** time during execution and in a noncontiguous fashion. For example, this might lead to something called **fragmentation**, where we have a bunch of small and unusable chunks of free memory as a result of the noncontiguous allocation. For example:



Since `malloc` cannot "gather and combine" different chunks of memory (must allocate a *single* continuous block per call), that 5 byte chunk in the middle is useless for most practical purposes and thus wasted heap memory. If this happens a lot, you can easily see how these "in between" chunks accumulate to a significant amount of wasted memory!

7 Heap Memory Implementations

Given the trickiness of managing heap memory, many have tried to come up with system of management.

7.1 K&R Implementation

In the Kernigan and Ritchie (textbook) implementation, each memory block should contain a **header** that specifies its size in bytes as well as contains a pointer to the next memory block. The very end of this memory-block linked list will point back to the first free block, creating a **circular linked list**. If one of these blocks of memory is allocated, there's no real reason for this block's header to have a pointer, so it's set to null.

When `malloc` is called, the circular linked list is searched for a sufficient block- and either allocates and returns a pointer to it or fails. Seems pretty simple.

The interesting part occurs When `free` is called. The function first frees the specified memory block, then checks if **adjacent** blocks (to the freed block) are also free. If they are, then the operating system **converges** this sequence of free blocks into a single free block. In other words, a few nodes from the free block linked list will be collapsed into a single node. Otherwise, the OS simply adds the freed block back to the list.

7.2 Choosing Free Blocks

Let's say `malloc` requests for 100 bytes of memory. We see a 150, 100, 120, and 80-byte blocks in our free block list. Which do we choose for the request? There are also various implementations for this:

- **Best-Fit**: Smallest block out of all free blocks that is sufficient for the request. So we'd choose the 100-byte block in our above example.
- **First-Fit**: FIRST sufficiently-sized free block. So we'd choose the 150-byte block in this case.
- **Next-Fit**: Like first-fit, but after each search we **save our node location** and resume our search for the next `malloc` call from that save-point.

8 Conclusion

Memory management is important in C because memory is ultimately what contains all of our data. There are three main storage sections in C: stack memory, heap memory, and static memory. Stack memory is used for local variables and parameters. Heap memory is used for dynamically allocated and arbitrary amounts of memory. Static memory is used for global and static variables. There are various memory-management implementations, such as differences on how to represent free blocks in memory as well as how to choose such free blocks given a memory request.