

# Lecture Notes on Memory Management

15-122: Principles of Imperative Computation  
Frank Pfenning

Lecture 21  
April 5, 2011

## 1 Introduction

Unlike C0 and other modern languages like Java, C#, or ML, C requires programs to explicitly manage their memory. Allocation is relatively straightforward, like in C0, requiring only that we correctly calculate the size of allocated memory. Deallocating (“freeing”) memory, however, is difficult and error-prone, even for experienced C programmers. Mistakes can either lead to attempts to access memory that has already been deallocated, in which case the result is undefined and may be catastrophic, or it can lead the running program to hold on to memory no longer in use, which may slow it down and eventually crash it when it runs out of memory. The second category is a so-called *memory leak*.

Your goal as a programmer should therefore be

- You should never free memory that is still in use.
- You should always free memory that is no longer in use.

Freeing memory counts as a final use, so the goals imply that you should not free memory twice. And, indeed, in C the behavior of freeing memory that has already been freed is undefined and may be exploited by an adversary.

The *golden rule of memory management* in C is

*You allocate it, you free it!*

By inference, if you *didn't* allocate it, you are *not* allowed to free it!

In the remainder of the lecture we explore how this rule plays out in common situations.

## 2 Simple Libraries

As a first example, consider again the stack data structure discussed in the last lecture. As a client, we are not supposed to know or exploit the implementation of stacks and we therefore cannot free the elements of the structure directly. Moreover, we (as the client) did not perform the actual allocation, so it is up to them (the library) to free the allocated space. In order to allow this, the data structure implementation must provide an interface function to free allocated memory.

```
void stack_free(stack S); /* S must be empty */
```

Because in this first scenario the stack must be empty, the implementation is simple:

```
void stack_free(stack S) {
    REQUIRES(is_stack(S) && stack_empty(S));
    ASSERT(S->top == S->bottom);
    free(S->top);
    free(S);
}
```

As a client, we call this usually in the same scope in which we create a stack

```
stack S = stack_new();
...
stack_free(S);
```

In this simple scenario we could successively pop elements of the stack and free them until the stack is empty, and then free  $S$  in the manner shown above.

## 3 Freeing Internal Structure

We would like to call `stack_free(S)` even if  $S$  is non-empty. In general, for richer data structures, the simple idea of successively deleting elements may not be possible and not even be supported in the interface. This means the `stack_free` function would have to look something like the following, passing the buck to another function to free list segments.

```
void stack_free(stack S) {
    REQUIRES(is_stack(S));
```

```
    segment_free(S->top, S->bottom);
    free(S->bottom);
    free(S);
}
```

Clearly, the `segment_free` function should not be visible to the client, just like the whole list type is not visible. This function would go over the list and free each node, but it would be incorrect to write the following:

```
void segment_free(list start, list end) {
    REQUIRES(is_segment(start, end));
    list p = start;
    while (p != end) {
        ASSERT(p != NULL);
        free(p);
        p = p->next; /* bug here! */
    }
    return;
}
```

In the marked line, we would access the next field of the struct that `p` is pointing to, but this has been deallocated in the preceding line. We must introduce a temporary variable `tmp` to hold the next pointer, which is a characteristic pattern for deallocation.

```
void segment_free(list start, list end) {
    REQUIRES(is_segment(start, end));
    while (p != NULL) {
        ASSERT(p != NULL);
        list tmp = p->next;
        free(p);
        p = tmp;
    }
    return;
}
```

What happens to the *data* stored at each node? Clearly, the library code cannot free this data, because it did not allocate it. Doing so would violate the *Golden Rule*! On the other hand, the client may not have any way to do so. For example, the stack might be the only place the data are stored, in which case they would become unreachable after the list has been deallocated. With a garbage collector, this is a common occurrence and the

correct behavior, because the garbage collector will now deallocate the unreachable data. With explicit memory management we need a different solution.

## 4 Function Pointers

A general solution is to pass a *function* as an argument to `stack_free` (and in turn to `list_free`) whose responsibility it is to free the embedded data elements. The client knows what this function should be, and is therefore in a position to pass it to the library. The library then applies this function to each data element in the list just before freeing the node where it is stored.

Actually, in C, functions are not first class, so we pass a *pointer* to a function instead. The syntax for function pointers is somewhat arcane. Here is what the interface declaration of `stack_free` looks like.

```
void stack_free(stack S, void (*data_free)(void* e));
```

The first argument name *S* is a stack. The second argument has name `data_free`. We read the declaration starting at the inside and moving outwards, considering what kind of operation can be applied to the argument.

1. `data_free` names an argument.
2. `*data_free` shows it can be dereferenced and therefore must be a pointer.
3. `(*data_free)(void* e)` means that the result of dereferencing `data_free` must be a function that can be applied to an argument of type `void*`.
4. `void (*data_free)(void* e)` finally shows that this function does not return a value.

In summary, `data_free` names a pointer to a function expecting a `void*` pointer as argument and returning no value. Its effect is intended to free the data element it was given. The stack is a generic data structure, so for reasons discussed in the last lecture, the data element is viewed as having type `void*`.

The implementation of `stack_free` is actually quite straightforward. Since it doesn't hold any data element, it just passes the function pointer to the `list_free` function.

```
void stack_free(stack S, void (*data_free)(void* e)) {
    REQUIRES(is_stack(S));
    segment_free(S->top, S->bottom, data_free);
    free(S->bottom);
    free(S);
    return;
}
```

Freeing the list elements has some pitfalls. Consider the following simple attempt.

```
void segment_free(list start, list end, void (*data_free)(void* e)) {
    REQUIRES(is_segment(start, end));
    while (p != end) {
        ASSERT(p != NULL);
        list tmp = p->next;
        (*data_free)(p->data);
        free(p);
        p = tmp;
    }
    return;
}
```

This actually has two somewhat subtle bugs. See if you can spot them before reading on.

The first problem is that `data_free` is a function pointer and therefore could be null. Attempting to dereference it would yield undefined behavior. The convention is that if we pass `NULL` we mean for the elements in the list *not* to be deallocated, perhaps because they are still needed elsewhere.

The second problem is that `p->data` may be null. We cannot free `NULL` because it has actually not been allocated and doesn't point to memory.

We therefore need to test these two conditions before we can apply the `*data_free` function.

```
void segment_free(list start, list end, void (*data_free)(void* e))
{
    REQUIRES(is_segment(start, end));
    list p = start;
    while (p != end) {
        ASSERT(p != NULL);
        list tmp = p;
        p = p->next;
        if (data_free != NULL && tmp->data != NULL)
            (*data_free)(tmp->data);
        free(tmp);
    }
    return;
}
```

When writing your own function along these lines, keep in mind that the order of the operations here is crucial: first we have to save the next pointer, then free `p->data`, then free `p` and then continue with the next element.

Finally, we examine the call site to see how we actually obtain a pointer to a function. First, we define an appropriate function. In this simple example, it just frees the memory holding an integer.

```
void int_free(void* p) {
    free((int*)p);          /* this coercion is optional */
}
```

We refer to this function below using the address-of operator `&`, after pushing two pointers onto the stack. The use of this operator before function names is optional, but preferred because it makes it clear that a pointer is passed, not the function itself (which cannot be done in C).

```
int main () {
    stack S = stack_new();
    int* x1 = xmalloc(sizeof(int));
    *x1 = 1;
    int* x2 = xmalloc(sizeof(int));
    *x2 = 2;
    push(x1, S);
    push(x2, S);
    stack_free(S, &int_free);
    ...
}
```

## 5 Double Free

In the above example, we have variables  $x_1$  and  $x_2$  in the main function, so we can deallocate the stack without deallocating its elements. This is done by passing NULL to `stack_free`, after which the main function itself can free  $x_1$  and  $x_2$ .

```
stack_free(S, NULL);
free(x1);
free(x2);
```

However, we have to be careful not to attempt freeing allocated memory more than once. The following has undefined behavior and is therefore a bug that may be security-critical.

```
stack_free(S, &int_free);
free(x1); /* bug; x1 already freed */
free(x2); /* bug; x2 already freed */
stack_free(S, NULL); /* bug, S already freed */
```

## 6 Stack Allocation

In C, we can also allocate data on the *system stack* (which is different from the explicit stack data structure used in the running example). As discussed in the lecture on memory layout, each function allocates memory in its so-called *stack frame* for local variables. We can obtain a pointer to this memory using the address-of operator. For example:

```
int main () {
    stack S = stack_new();
    int a1 = 1;
    int a2 = 2;
    push(&a1, S);
    push(&a2, S);
    ...
}
```

Note that there is no call to `malloc` or `calloc` which allocates spaces on the system heap (again, this is different from the heap data structure we used for priority queues).

Note that we can only free memory allocated with `malloc` or `calloc`, but not memory that is on the system stack. Such memory will automatically be freed when the function whose frame it belongs to returns. This has two important consequences. The first is that the following is a bug, because `stack_free` will try to free the memory holding  $a_1$  and  $a_2$  which are not on the heap.

```
int main () {
    stack S = stack_new();
    int a1 = 1;
    int a2 = 2;
    push(&a1, S);
    push(&a2, S);
    stack_free(S, &int_free); /* bug; a1 and a2 cannot be freed */
}
```

Instead, we must call `stack_free(S, NULL)`. The second consequence is pointers to data stored on the system stack do not survive the function's return. For example, the following is a bug:

```
void push1(stack S) {
    int a = 1;
    push(&a, S); /* bug: a is deallocated when push1 returns */
    return;
}
```

A correct implementation requires us to allocate on the system heap, using a call to `malloc` or `calloc` (or one of the library functions which calls them in turn).



```
void push1 (stack S) {
    int* x = xmalloc(sizeof(int));
    *x = 1;
    push(x, S); /* correct: x will persist when push1 returns */
    return;
}
```

## 7 Pointer Arithmetic in C

We have already discussed that C does not distinguish between pointers and arrays; essentially a pointer holds a memory address which may be the beginning of an array. In C we can actually calculate with memory addresses. Before we explain how, please heed our recommendation: recommendation

*Do not perform arithmetic on pointers!*

Code with explicit pointer arithmetic will generally be harder to read and is more error-prone than using the usual array access notation  $A[i]$ .

Now that you have been warned, here is how it works. We can add an integer to a pointer in order to obtain a new address. In our running example, we can allocate an array and then push pointers to the first, second, and third elements in the array onto a stack.

```
int* A = xcalloc(3, sizeof(int));
A[0] = 0; A[1] = 1; A[2] = 2;
push(A, S); /* push a pointer to A[0] onto stack */
push(A+1, S); /* push a pointer to A[1] onto stack */
push(A+2, S); /* push a pointer to A[2] onto stack */
```

The actual address denoted by  $A + 1$  depends on the size of the elements stored at  $*A$ , in this case, the size of an int. A much better way to achieve the same effect is

```
int* A = xcalloc(3, sizeof(int));
A[0] = 0; A[1] = 1; A[2] = 2;
push(&A[0], S); /* push a pointer to A[0] onto stack */
push(&A[1], S); /* push a pointer to A[1] onto stack */
push(&A[2], S); /* push a pointer to A[2] onto stack */
```

We cannot free array elements individually, even though they are located on the heap. The rule is that we can apply free only to pointers returned from malloc or calloc. So in the example code we can only free  $A$ .

```
int* A = xmalloc(3, sizeof(int));
A[0] = 0; A[1] = 1; A[2] = 2;
push(&A[0], S); /* push a pointer to A[0] onto stack */
push(&A[1], S); /* push a pointer to A[1] onto stack */
push(&A[2], S); /* push a pointer to A[2] onto stack */
stack_free(S, &int_free); /* bug: cannot free A[1] or A[2] separately */
```

The correct way to free this is as follows.

```
int* A = xmalloc(3, sizeof(int));
A[0] = 0; A[1] = 1; A[2] = 2;
push(&A[0], S); /* push a pointer to A[0] onto stack */
push(&A[1], S); /* push a pointer to A[1] onto stack */
push(&A[2], S); /* push a pointer to A[2] onto stack */
stack_free(S, NULL);
free(A);
```

We will see examples of code implementing generic data structures that depend on code provided by the client in the next lecture.