Lecture 11: Dynamic Memory Allocation

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/11-dynamic-memory-allocation.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

Here we cover dynamic memory allocation, i.e. allocating memory space at runtime.

Dynamic Memory Allocation: Motivation

Until now we have only worked with data that is allocated statically. This means that the amount of memory space reserved for all the variables we used is determined at compile time. But there are some scenarios in which the amount of space needed is only known at runtime. Here is an example with a program taking an integer as command line argument and allocating an array which size is based on that number:

```
int main(int argc, char **argv) {
    if(argc != 2) {
        printf("Usage: %s <array size>\n", argv[0]);
        return -1;
    }

    int size = atoi(argv[1]);
    int array[size]; // variable-sized array

    for(int i=0; i<size; i++) {
        array[i] = i;
        printf("array[%d] = %d\n", i, array[i]);
    }
    return 0;
}
```

`size` is only known at runtime and thus this is a variable-sized array. Variable-sized arrays are rarely used in C because they have many downsides. They are not supported in some versions of C. Most importantly, they are stored on a location in memory called the stack, that has very limited space. For example if when running the code above with a large number for `size`, a memory error will happen because the stack will overflow. We need another solution.

Program Memory Layout

Before explaining dynamic memory allocation, let's talk a bit about how the program memory is organised We take this example program manipulating 2 arrays:

```
int large_array[10000000];
```

```c
int process_array(int *array, int size) { /* do some stuff with array */ }

int main(int argc, char **argv) {
    int size;

    /* check argc ... */

    size = atoi(argv[1]);
    if(size < 500) {            /* Make sure small_array does not overflow the stack */
        int small_array[size];
        process_array(small_array, size);
    }

    process_array(large_array, 10000000);
    return 0;
}
```

We have a static array (static meaning that its size is known at compile-time), `large_array`, with a relatively large size, declared as a global variable. And we have a variable-sized array for which we make sure that the size is not too big.

In memory, there is an area named the **static memory** that contains many things and in particular the global variables. Its size can be very large. And the size of what is contained in there is determined at compile-time. For example here the size of 'large_array" is 10 million of ints and it won't change. It goes into static memory.

Another area is the **stack**, storing things like local variables and function parameters. Its size is very small, by default a few megabytes on Linux. Most allocation sizes on the stack are fixed at compile time, although this is where variable-sized arrays live too.

Finally, we have a third area called the **heap**. It has a large size and all allocations there are realised dynamically at runtime, i.e. the size of what we allocate there does not need to be known at compile-time. There is a function used to perform dynamic memory allocation on the heap. It is called `malloc`. Here is an example of its usage:

```c
#include <stdio.h>
#include <stdlib.h> // needed for malloc

int process_array(int *array, int size) { /* do something with array */ }
int main(int argc, char **argv) {
    int *heap_array;
    /* ... */

    int size = atoi(argv[1]);

    heap_array = malloc(size * sizeof(int)); /* Allocate size * sizeof(int) bytes on the heap */
    if(heap_array == NULL)
        return -1;

    process_array(heap_array, size);

    free(heap_array); /* free the area previously allocated */
    return 0;
}
```

We declare an `int` pointer that will point to the allocated space. Then we call `malloc`. `malloc` takes one parameter, the size in bytes of the space we want to allocate. Here it is the size of the array we wish to allocate. It is equal to the number of elements we wish the array to contain, `size`, multiplied by the size of one element, that is `sizeof(int)`. `malloc` allocates a contiguous memory space in memory on the heap, and returns a pointer to it. A very important thing with `malloc` is to check its return value. It can be `NULL` (an alias for `0`) if the system runs out of memory. Without checking the execution will continue with the pointer set to `NULL` which will lead to bugs that can be hard

to fix.

One last thing to know about dynamic memory allocation is that the programmer is responsible for releasing the allocated memory. This is done with the `free` function, taking as parameter a pointer that points to a space previously allocated with malloc. It is important to free up memory as soon as the program is done with it. Otherwise, the program is holding memory for nothing, it is called a memory leak.

The type of the pointer returned by `malloc` is `void *`. It's a generic type so `malloc` can be used to allocate data that will be pointed by pointers of all types.

Multidimensional Arrays and `malloc`

Below is an example of multidimensional array of size 3 (rows) by 2 (columns) elements allocated with `malloc`:

```
int a = 3; int b = 2;
int **array;

array = malloc(a * sizeof(int *));
if(array == NULL) return -1;

for(int i=0; i<a; i++) {
    array[i] = malloc(b * sizeof(int));
    if(array[i] == NULL) return -1;
}

array[2][0] = 12;
/* ... */

for(int i=0; i<a; i++)
    free(array[i]);
free(array);
```

The key idea is to store each row in its own contiguous area of memory allocated with `malloc`. And to have another area storing columns, i.e. a set of pointers, each pointing to the area representing a row.

We declare `array`, that will be the base pointer of our bi-dimensional array. `array` will hold the pointers to the rows: it is an `int **` variable, i.e. a pointer of pointer of int. With `malloc` we allocate a first area of memory and have `array` point to it. Its size is equals to the first dimension of the array we wish to create, i.e. the number of rows, `3`, multiplied by the size of one element, a pointer of `int`: `sizeof(int *)`.

`array` now points to an area with enough space to store 3 pointers of `int`. Next we allocate in a loop the area for each row with `malloc`: 3 areas. Each has a size equal to the second dimension of the array, i.e. the number of columns, `2`, multiplied by the size of the final elements held in the array: `sizeof int`.

After each call to `malloc` we check its success and abort (`return -1`) if it fails. After the rows and columns have been successfully allocated, we can use the original pointer `array` and the square brackets to index the array in the exact same way we would do with a static array.

Before exiting the program, we deallocate the memory with `free` in reverse order: first the row pointers in a loop, then what is pointed by `array` itself.

Memory Leaks

When the programmer forgets to free memory it is called a memory leak. Such wasted memory is inefficient and can lead to crashes when the system runs out of memory, for example with long-running programs such as servers. Valgrind[1] is a very useful tool to detect leaks, amongst other features. Take this example program containing a leak:

```
/* leak.c */

void print_ten_integers() {
```

[1] https://valgrind.org/

```
    int *array = malloc(10 * sizeof(int));
    if(!array) {
        printf("cannot allocate memory ...\n");
        return;
    }

    for(int i=0; i<10; i++) {
        array[i] = rand()%100;
        printf("%d ", array[i]);
    }

    printf("\n");
    /* array is never freed, leaking 10*sizeof(int) of memory each iteration */
}

int main(int argc, char **argv) {
    int iterations = atoi(argv[1]);
    for(int i=0; i<iterations; i++)
        print_ten_integers();
    return 0;
}
```

For Valgrind to work correctly, the program needs to be compiled with additional debug metadata so use the -g flag when calling gcc:

```
$ gcc -g leak.c -o leak
```

Valgrind reports the amount of bytes leaked, as well as the line in the sources of the corresponding calls to malloc:

```
$ valgrind --leak-check=full ./leak 10
...
==11613==
==11613== HEAP SUMMARY:
==11613==     in use at exit: 400 bytes in 10 blocks
==11613==   total heap usage: 11 allocs, 1 frees, 1,424 bytes allocated
==11613==
==11613== 400 bytes in 10 blocks are definitely lost in loss record 1 of 1
==11613==    at 0x483577F: malloc (vg_replace_malloc.c:299)
==11613==    by 0x109196: print_ten_integers (leak.c:5)
==11613==    by 0x109294: main (leak.c:32)
==11613==
==11613== LEAK SUMMARY:
==11613==    definitely lost: 400 bytes in 10 blocks
==11613==    indirectly lost: 0 bytes in 0 blocks
==11613==      possibly lost: 0 bytes in 0 blocks
==11613==    still reachable: 0 bytes in 0 blocks
==11613==         suppressed: 0 bytes in 0 blocks
```