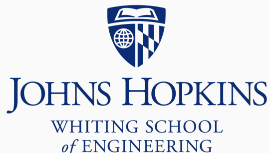
Dynamic memory allocation

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Dynamic memory allocation

`malloc(size)` allocates `size` bytes and returns a pointer to the allocated memory

The memory is *uninitialized*

`malloc` returns `NULL` upon error (e.g. not enough memory)

Dynamic memory allocation

```c
int n = get_length_of_array();
int *array = malloc( ? );
```

Dynamic memory allocation

```
int n = get_length_of_array();
int *array = malloc(n * sizeof(int));
```

`sizeof(int)` is better than simply putting 4; not every compiler/computer will agree that an `int` is 4 bytes

Dynamic memory allocation

Note the types in these prototypes (from `man malloc`)

```
void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
```

- `void *`: "wildcard" or "generic" pointer to any type
- `size_t`: usually the same thing as a `long unsigned int`

Dynamic memory allocation

Memory allocated with `malloc`, `calloc` or `realloc` is called *dynamically allocated* memory

Each returns a pointer to the newly-allocated memory

Allocated memory must eventually be deallocted, otherwise it "leaks" and isn't deallocated until the program exits

If your program accumulates enough "leaked" memory, it will exhaust memory and crash

- Possibly taking other programs with it

Dynamic memory allocation

- void * is the “wildcard” pointer; can point to any type

```
int *array = malloc(40 * sizeof(int));
// behind the scenes, C "automatically" changes
// the void * returned by malloc into an int *

char *array2 = malloc(40 * sizeof(char));
// same thing here, but for char *

float *array3 = malloc(40 * sizeof(float));
// same thing here, but for float *
```

Dynamic memory allocation

`malloc` allocates memory that is uninitialized at first

`calloc` allocates memory with all positions initialized to 0

`realloc` resizes a previously-allocated chunk of memory

- Useful for dynamically resizing a data structure that grows as more data comes in

`free` deallocates memory returned by any of the above

Dynamic memory allocation

```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.h> // for malloc, free & friends

double* scale(double arr[5], double factor) {
    double *scaled_array = malloc(sizeof(double) * 5);
    assert(scaled_array != NULL); // bad practice; use if instead
    for(int i = 0; i < 5; i++) {
        scaled_array[i] = arr[i] * factor;
    }
    return scaled_array;
}

int main() {
    double array[] = {1.0, 4.5, 8.4, 2.5, 8.3};
    double* scaled_array = scale(array, 2.0);
    printf("%0.2f %0.2f\n", scaled_array[0], scaled_array[4]);
    free(scaled_array);
    return 0;
}
```

Dynamic memory allocation

```
$ gcc -c scale_dynamic.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o scale_dynamic scale_dynamic.o
$ ./scale_dynamic
2.00 16.60
```

Dynamic memory allocation

This program has a *memory leak*:

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h> // for malloc, free & friends

double* scale(double arr[5], double factor) {
    double *scaled_array = malloc(sizeof(double) * 5);
    assert(scaled_array != NULL); // bad practice; use if instead
    for(int i = 0; i < 5; i++) {
        scaled_array[i] = arr[i] * factor;
    }
    return scaled_array;
}

int main() {
    double array[] = {1.0, 4.5, 8.4, 2.5, 8.3};
    double* scaled_array = scale(array, 2.0);
    printf("%0.2f %0.2f\n", scaled_array[0], scaled_array[4]);
    // free(scaled_array);  // *** LEAK ***
    return 0;
}
```

Dynamic memory allocation

This program has an especially bad leak (repeated in a loop):

```
int super_leaky(int n) {
    int done = 0;
    // assume we iterate many times before setting done=1
    while(!done) {
        int *array = malloc(n * sizeof(int));
        assert(array != NULL);

        // do stuff with array but don't deallocate it

        // we leak more memory *in each iteration*
    }
    return 0;
}
```

Dynamic memory allocation

Is this OK?

```
int possibly_leaky(int n) {
    int array[10];
    // do stuff with array
    return 0;
}
```

Dynamic memory allocation

Yes, this is perfectly OK

`int array[10]` is on the stack so allocation and deallocation are handled for us, behind the scenes

We're not attempting to `return array`, which would be bad

```
int possibly_leaky(int n) { // not leaky
    int array[10];
    // do stuff with array
    return 0;
}
```

Dynamic memory allocation

Be a good memory citizen

- Always explicitly deallocate memory you've allocated
- When you add an allocation to your program, think about where you should deallocate

Dynamic memory allocation

What should we have done differently?

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h> // for malloc, free & friends

int *sequence(int n) {
    int *x = malloc(n * sizeof(int));
    assert(x != NULL);
    for(int i = 0; i < n; i++) {
        x[i] = i;
    }
    return x;
}

int main() {
    int *seq = sequence(10);
    for(int i = 0; i < 10; i++) {
        printf("%d ", seq[i]);
    }
    return 0;
}
```

Dynamic memory allocation

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h> // for malloc, free & friends

// Returns a newly-allocated array
int *sequence(int n) {
    int *x = malloc(n * sizeof(int));
    assert(x != NULL);
    for(int i = 0; i < n; i++) {
        x[i] = i;
    }
    return x;
}

int main() {
    int *seq = sequence(10);
    // seq is a newly allocated array
    // you remind yourself "I have to deallocate this!"
    for(int i = 0; i < 10; i++) {
        printf("%d ", seq[i]);
    }
    free(seq); // that's better!!!
    return 0;
}
```

Dynamic memory allocation

Only `free` memory you allocated with `malloc` & friends

Don't `free` the same thing twice

`free` the same *exact* pointer you allocated; this won't work:

```
int *array = malloc(20 * sizeof(int));
assert(array != NULL);
array++; // skip over first element
free(array); // no longer equals pointer returned by malloc
```

Dynamic memory allocation

`calloc` is like `malloc` except:

The allocated memory is first initialized to all 0s

1st parameter to `calloc` is number of elements

2nd parameter is element size

```
int *array = malloc(40 * sizeof(int));
for(int i = 0; i < 40; i++) {
    array[i] = 0;
}

// ...is the same as...

int *array = calloc(40, sizeof(int));
```

Dynamic memory allocation

`realloc` “resizes” an existing memory allocation

Dynamic memory allocation

```c
#include <stdio.h>
#include <stdlib.h> // for malloc, free & friends

int main() {
    int *array = malloc(10 * sizeof(int));
    for(int i = 0; i < 10; i++) {
        array[i] = i * 2;
    }
    array = realloc(array, 20 * sizeof(int));

    // elements 0 through 9 are still set as above!
    // only need to set 10 through 19 here
    for(int i = 10; i < 20; i++) {
        array[i] = i * 2;
    }

    for(int i = 0; i < 20; i++) {
        printf("%d ", array[i]);
    }
    putchar('\n'); // print single newline to stdout
    return 0;
}
```

```
$ gcc -c realloc_eg1.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o realloc_eg1 realloc_eg1.o
$ ./realloc_eg1
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38
```

Dynamic memory allocation

`realloc`'s first parameter must have been allocated previously with `malloc` (or `calloc` or `realloc`)

```
int array[] = {0, 2, 4, 6};
realloc(array, 8 * sizeof(int)); // won't work!
```

Dynamic memory allocation

`new = realloc(old, new_size)`

- If new size is greater than old size, the new space is uninitialized

Behind the scenes, `realloc` will:

- `new = malloc(new_size)`
- Copy contents of `old` into beginning of `new`
- `free(old)`

Dynamic memory allocation

Be careful: after calling `realloc`, don't use any pointers corresponding to older versions of that allocation. The old memory was deallocated and no longer belongs to you.

Dynamic memory allocation

A quick reminder as to how big everything is

```
#include <stdio.h>

int main() {
    printf("char: %d\n",         (int)sizeof(char));
    printf("int: %d\n",          (int)sizeof(int));
    printf("unsigned int: %d\n", (int)sizeof(unsigned int));
    printf("float: %d\n",        (int)sizeof(float));
    printf("double: %d\n",       (int)sizeof(double));
    printf("void *: %d\n",       (int)sizeof(void *));
    printf("int *: %d\n",        (int)sizeof(int *));
    printf("size_t: %d\n",       (int)sizeof(size_t));
}
```

Dynamic memory allocation

```
$ gcc -c sizeof_all.c -std=c99 -pedantic -Wall -Wextra
$ gcc -o sizeof_all sizeof_all.o
$ ./sizeof_all
char: 1
int: 4
unsigned int: 4
float: 4
double: 8
void *: 8
int *: 8
size_t: 8
```

Dynamic memory allocation

New powers:

- Can (finally) allocate arbitrary-length arrays
- Arrays/variables allocated in this way live across scopes and function calls, until we deallocate

Dynamic memory allocation

New responsibilities:

- For every allocation, we must deallocate (no leaks)
- Check return value from `malloc` & friends; could be `NULL`
- Don't dereference an address (pointer) that doesn't point to your own properly-allocated memory
- Any new memory allocated by `malloc` or `realloc` is uninitialized; assign to it before using it
- Don't forget that that `free` and `realloc` deallocate their arguments