Dynamic Memory Allocation

Review

- How do we pass arrays across functions?

Review

- What is the scope for variables declared inside a function?

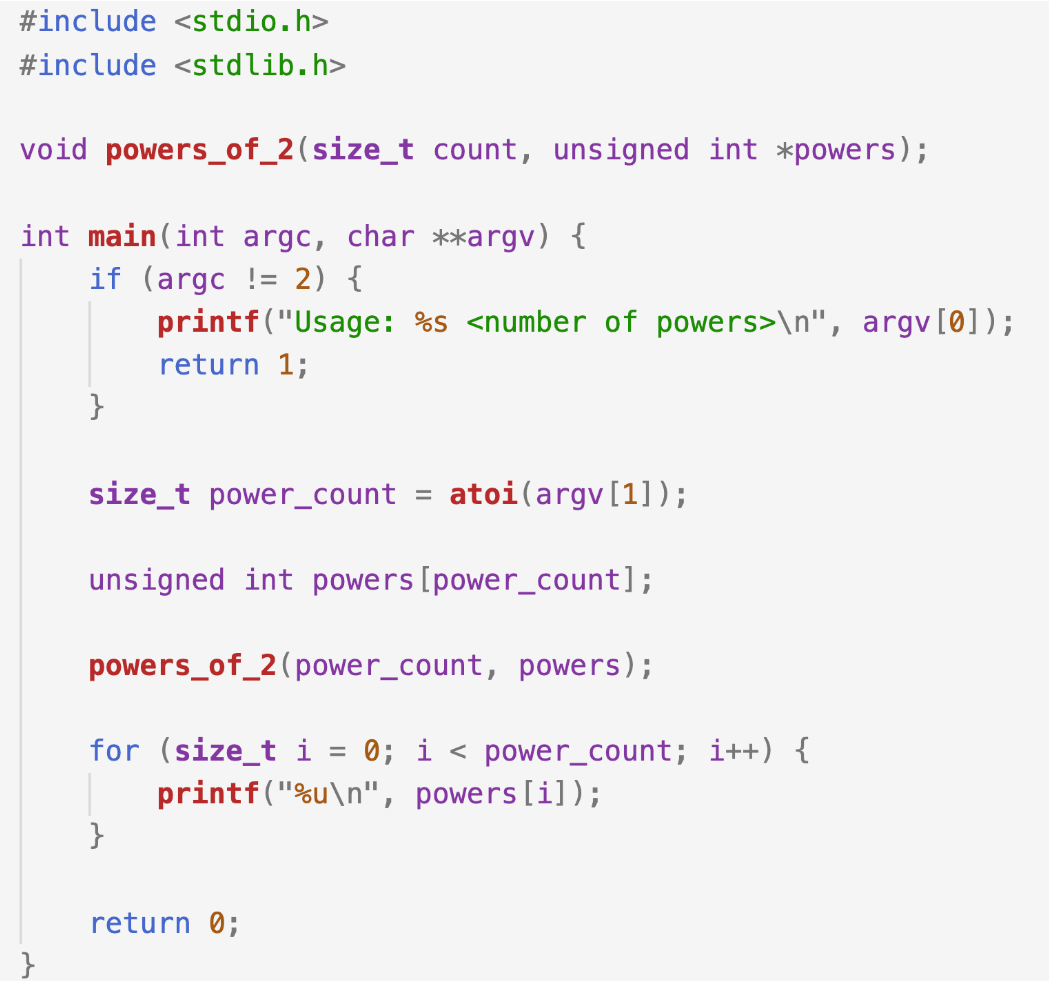
Review

- What happens to the automatic variables associated with a function after the stack frame for that function gets popped off the call stack?

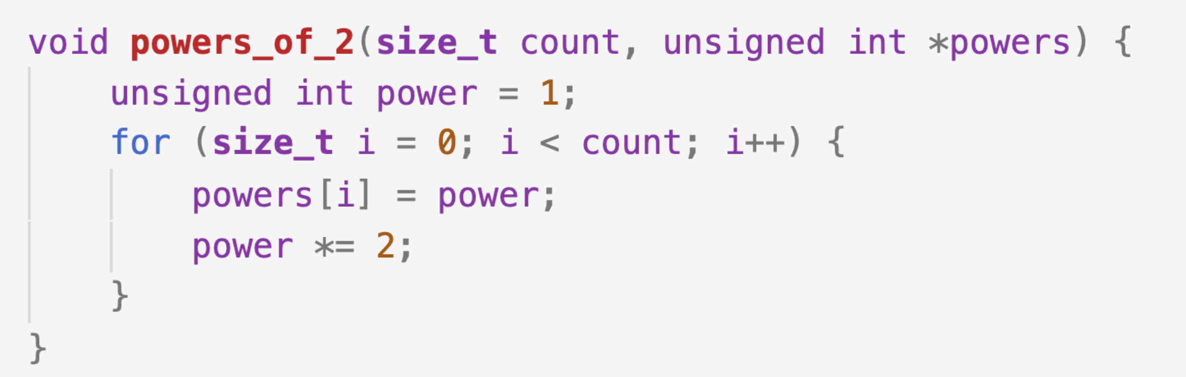
Review

- From examples discussed in last class, which was the first function to be pushed on to the call stack and the last to be popped off it?

Powers of 2: Declaring the array in main()

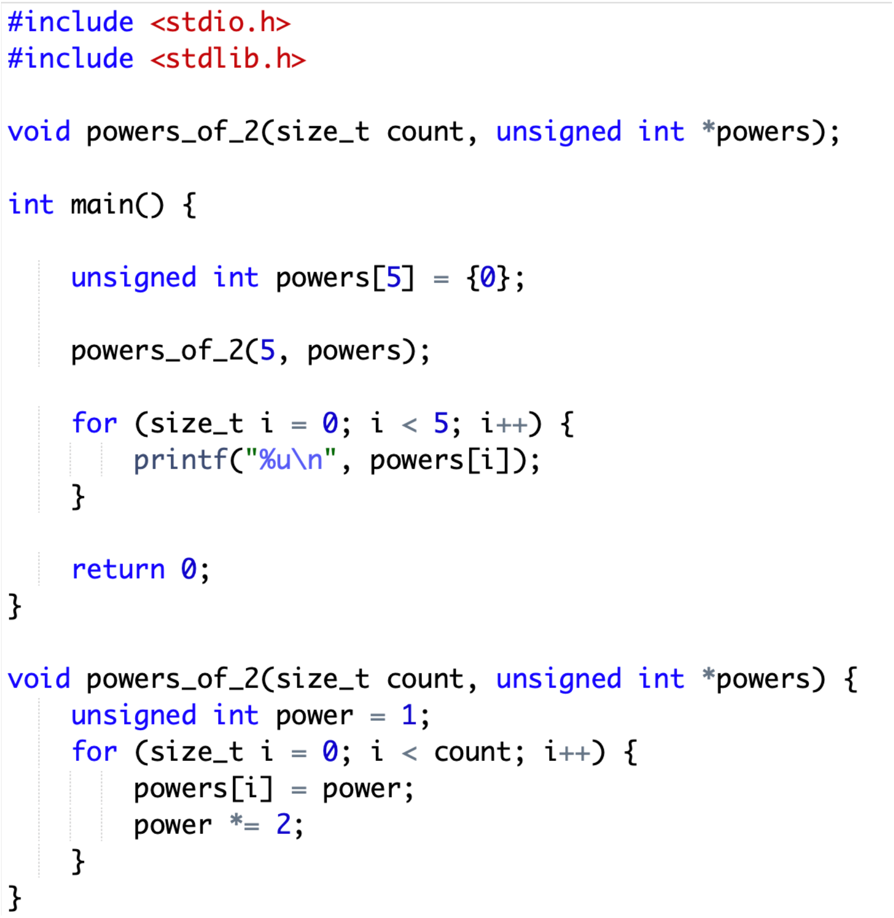


```c
#include <stdio.h>
#include <stdlib.h>

void powers_of_2(size_t count, unsigned int *powers);

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

    size_t power_count = atoi(argv[1]);

    unsigned int powers[power_count];

    powers_of_2(power_count, powers);

    for (size_t i = 0; i < power_count; i++) {
        printf("%u\n", powers[i]);
    }

    return 0;
}
```

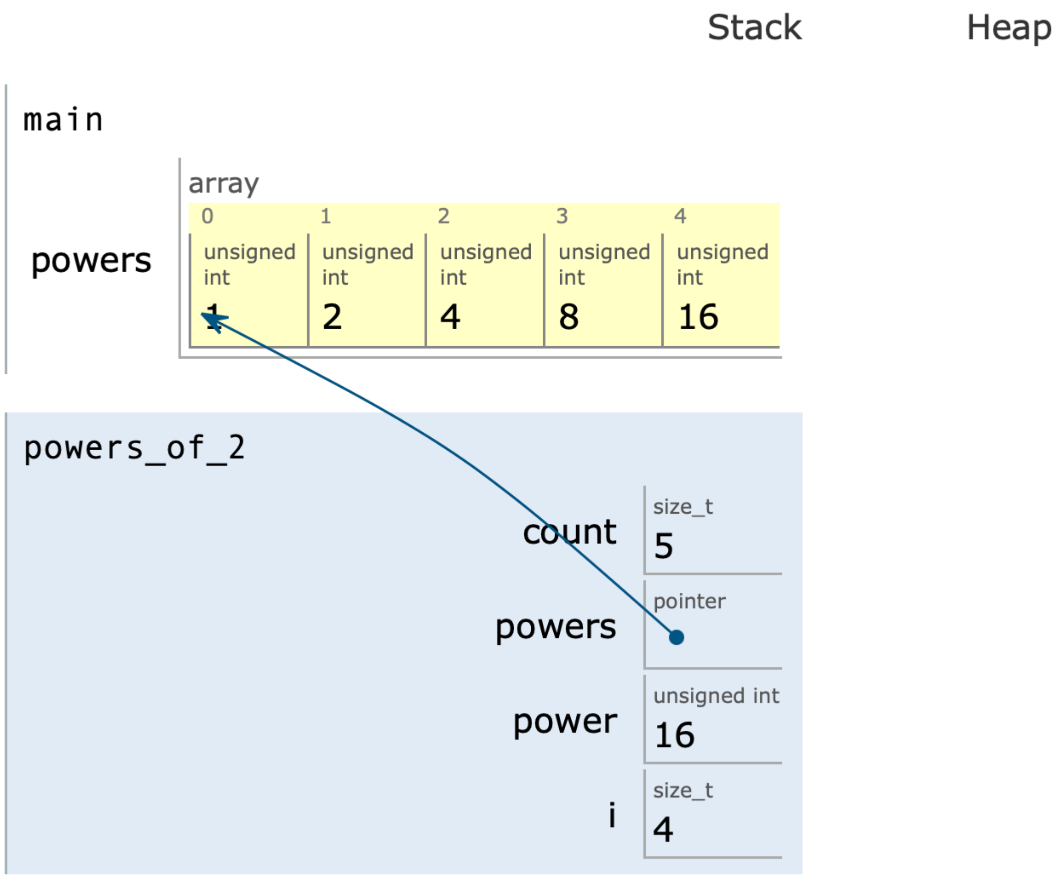


```c
void powers_of_2(size_t count, unsigned int *powers) {
    unsigned int power = 1;
    for (size_t i = 0; i < count; i++) {
        powers[i] = power;
        power *= 2;
    }
}
```

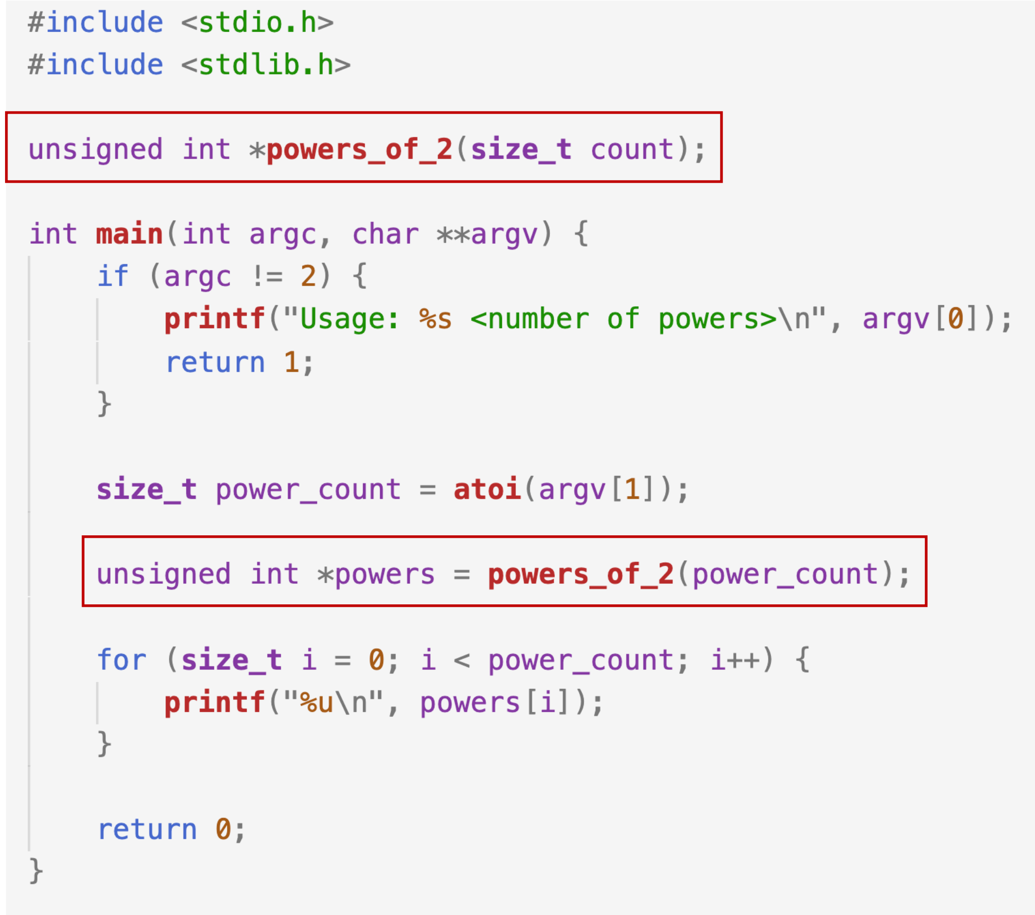
Puthontutor Visualization



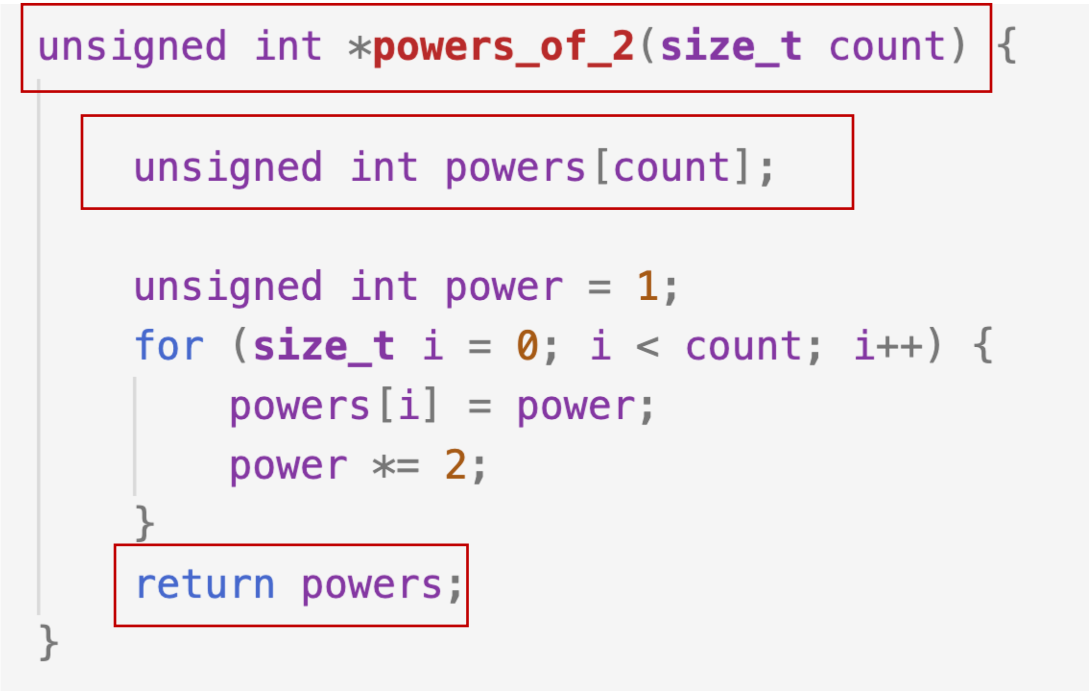
```c
#include <stdio.h>
#include <stdlib.h>

void powers_of_2(size_t count, unsigned int *powers);

int main() {

    unsigned int powers[5] = {0};

    powers_of_2(5, powers);

    for (size_t i = 0; i < 5; i++) {
        printf("%u\n", powers[i]);
    }

    return 0;
}

void powers_of_2(size_t count, unsigned int *powers) {
    unsigned int power = 1;
    for (size_t i = 0; i < count; i++) {
        powers[i] = power;
        power *= 2;
    }
}
```



Hypothetical situation: Returning an array from a function

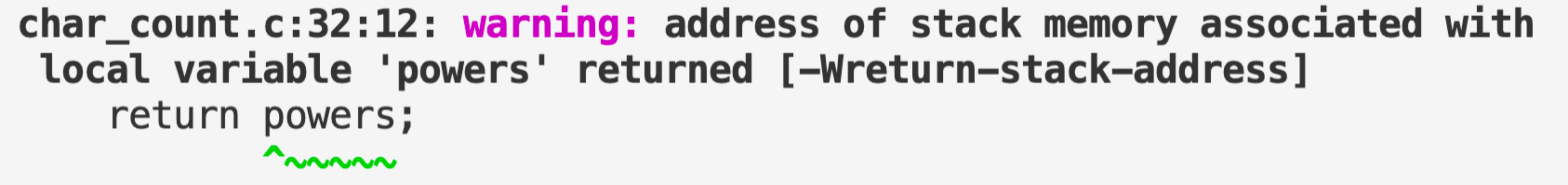


```c
#include <stdio.h>
#include <stdlib.h>

unsigned int *powers_of_2(size_t count);

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

    size_t power_count = atoi(argv[1]);

    unsigned int *powers = powers_of_2(power_count);

    for (size_t i = 0; i < power_count; i++) {
        printf("%u\n", powers[i]);
    }

    return 0;
}
```



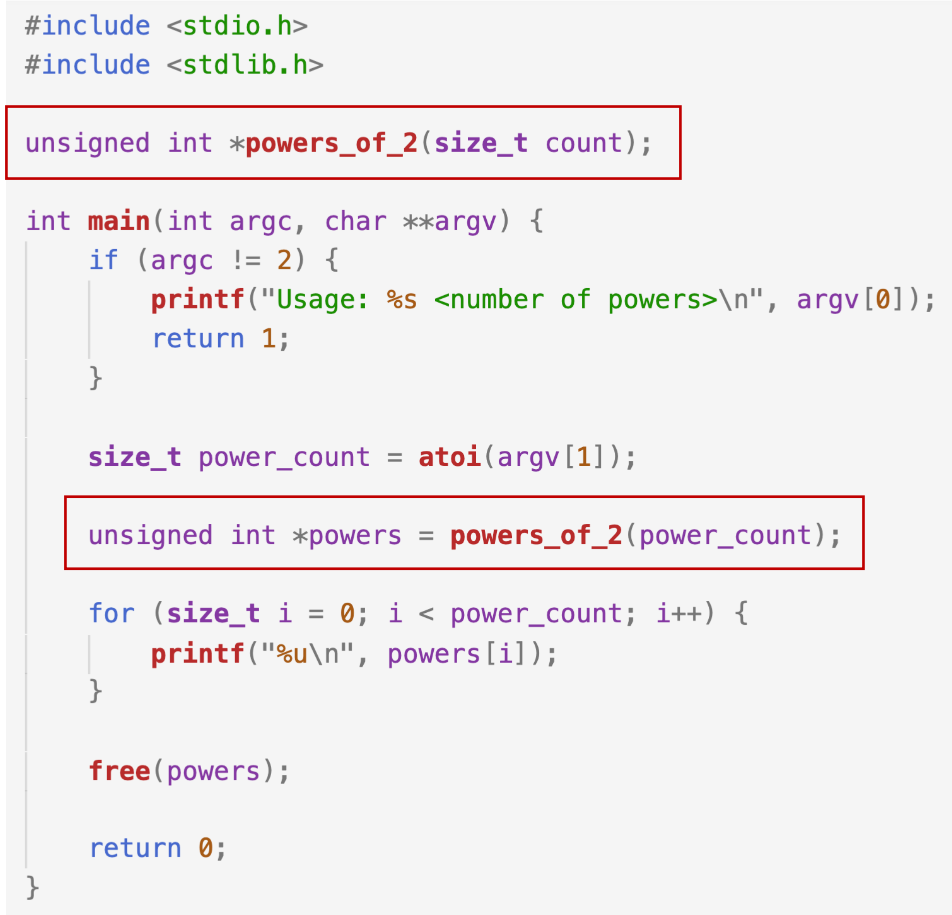
```c
unsigned int *powers_of_2(size_t count) {

    unsigned int powers[count];

    unsigned int power = 1;
    for (size_t i = 0; i < count; i++) {
        powers[i] = power;
        power *= 2;
    }
    return powers;
}
```

Warning!

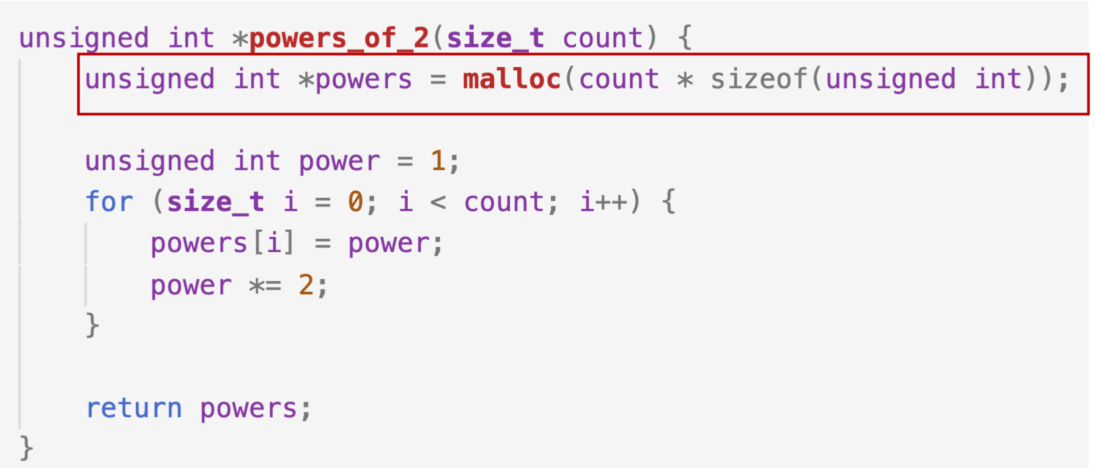


```
char_count.c:32:12: warning: address of stack memory associated with
 local variable 'powers' returned [-Wreturn-stack-address]
    return powers;
           ^~~~~~
```

Dynamic Memory Allocation

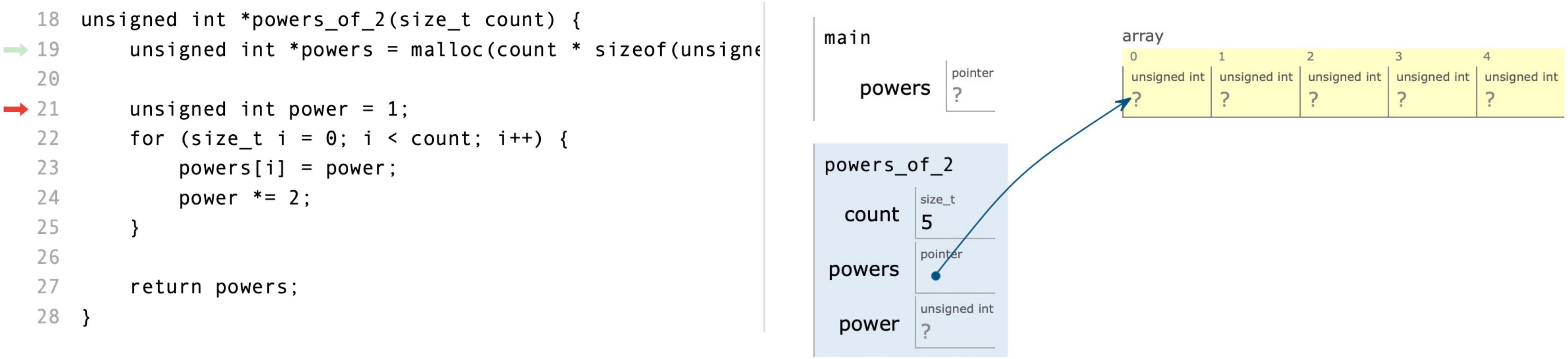


```c
#include <stdio.h>
#include <stdlib.h>

unsigned int *powers_of_2(size_t count);

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

    size_t power_count = atoi(argv[1]);

    unsigned int *powers = powers_of_2(power_count);

    for (size_t i = 0; i < power_count; i++) {
        printf("%u\n", powers[i]);
    }

    free(powers);

    return 0;
}
```



```c
unsigned int *powers_of_2(size_t count) {
    unsigned int *powers = malloc(count * sizeof(unsigned int));

    unsigned int power = 1;
    for (size_t i = 0; i < count; i++) {
        powers[i] = power;
        power *= 2;
    }

    return powers;
}
```

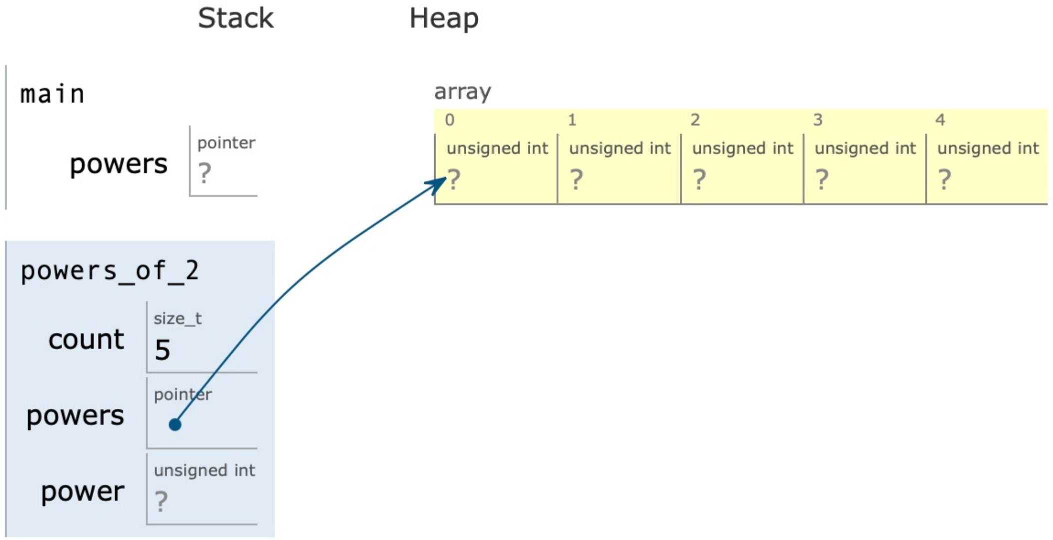
Dynamic Memory Allocation



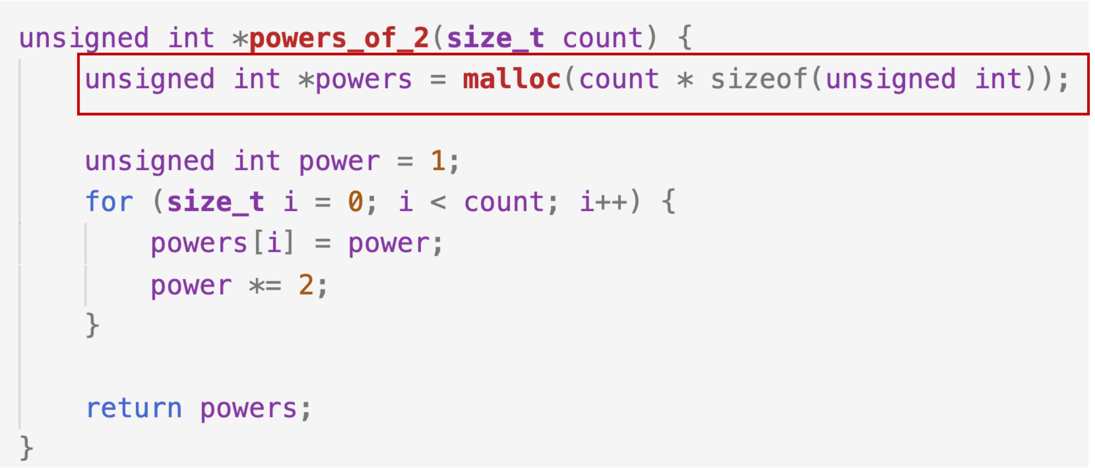
```
unsigned int *powers_of_2(size_t count) {
    unsigned int *powers = malloc(count * sizeof(unsigne

    unsigned int power = 1;
    for (size_t i = 0; i < count; i++) {
        powers[i] = power;
        power *= 2;
    }

    return powers;
}
```

malloc()

- malloc() dynamically allocates memory on the heap
- returns a pointer *to that memory.*



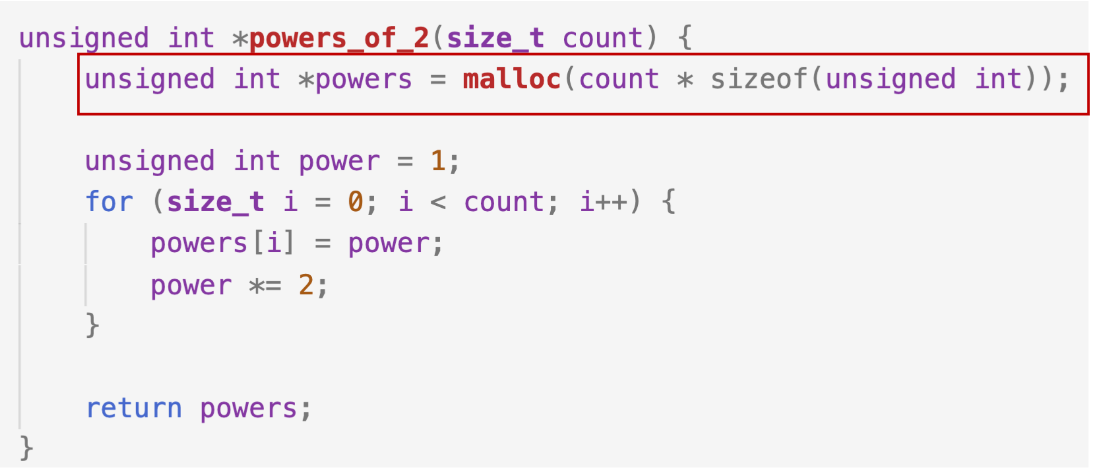
malloc()



```
unsigned int *powers_of_2(size_t count) {
    unsigned int *powers = malloc(count * sizeof(unsigned int));

    unsigned int power = 1;
    for (size_t i = 0; i < count; i++) {
        powers[i] = power;
        power *= 2;
    }

    return powers;
}
```

- The parameter for malloc() is the number of bytes we want to allocate
- To allocate an array we need to multiply the number of elements we want by the number of bytes each element needs.

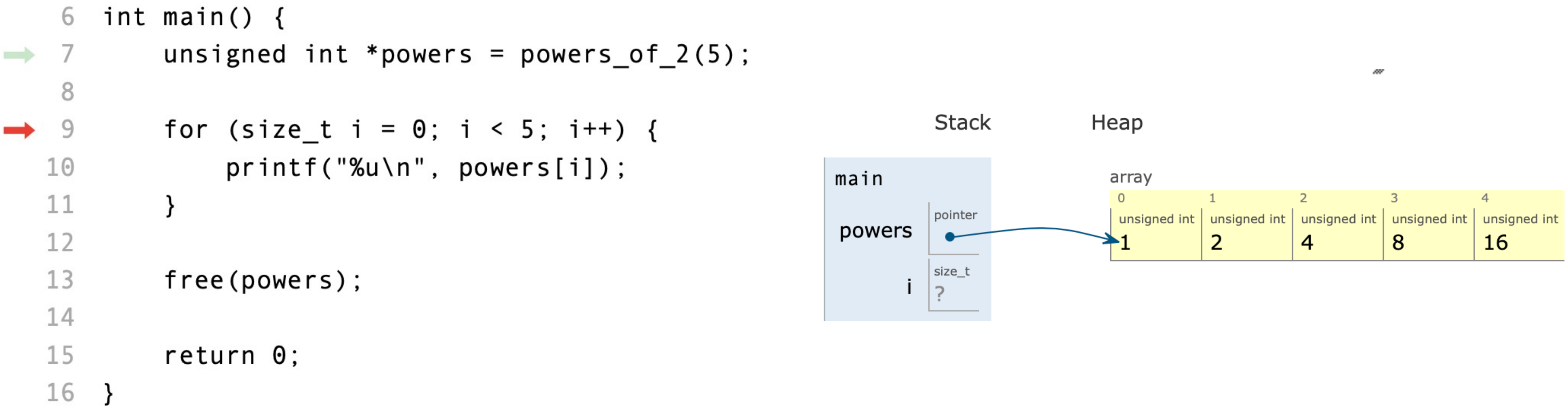
malloc()



```c
unsigned int *powers_of_2(size_t count) {
    unsigned int *powers = malloc(count * sizeof(unsigned int));

    unsigned int power = 1;
    for (size_t i = 0; i < count; i++) {
        powers[i] = power;
        power *= 2;
    }

    return powers;
}
```

- return type: void *
- Generic pointer that is automatically cast to a pointer of a specific type on assignment
- sizeof() finds the size of the type you are using
    - malloc(10 * sizeof(int)) - Allocates enough storage for 10 ints

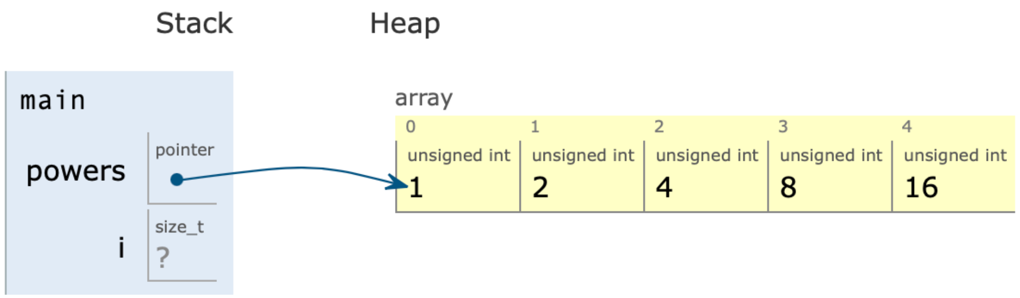
Visualizing malloc()



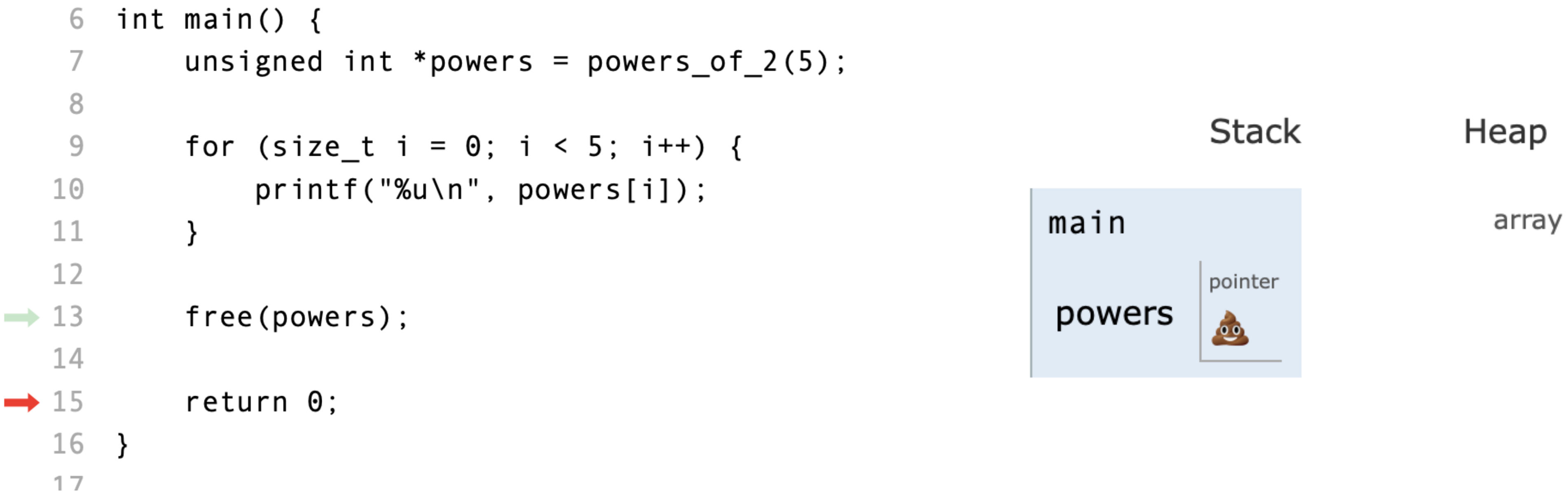
```c
int main() {
    unsigned int *powers = powers_of_2(5);

    for (size_t i = 0; i < 5; i++) {
        printf("%u\n", powers[i]);
    }

    free(powers);

    return 0;
}
```

Heap

- Area where dynamically allocated memory is stored
- Memory in the heap is not associated with names, like stack variables are
- Memory in the heap can only be accessed through pointers

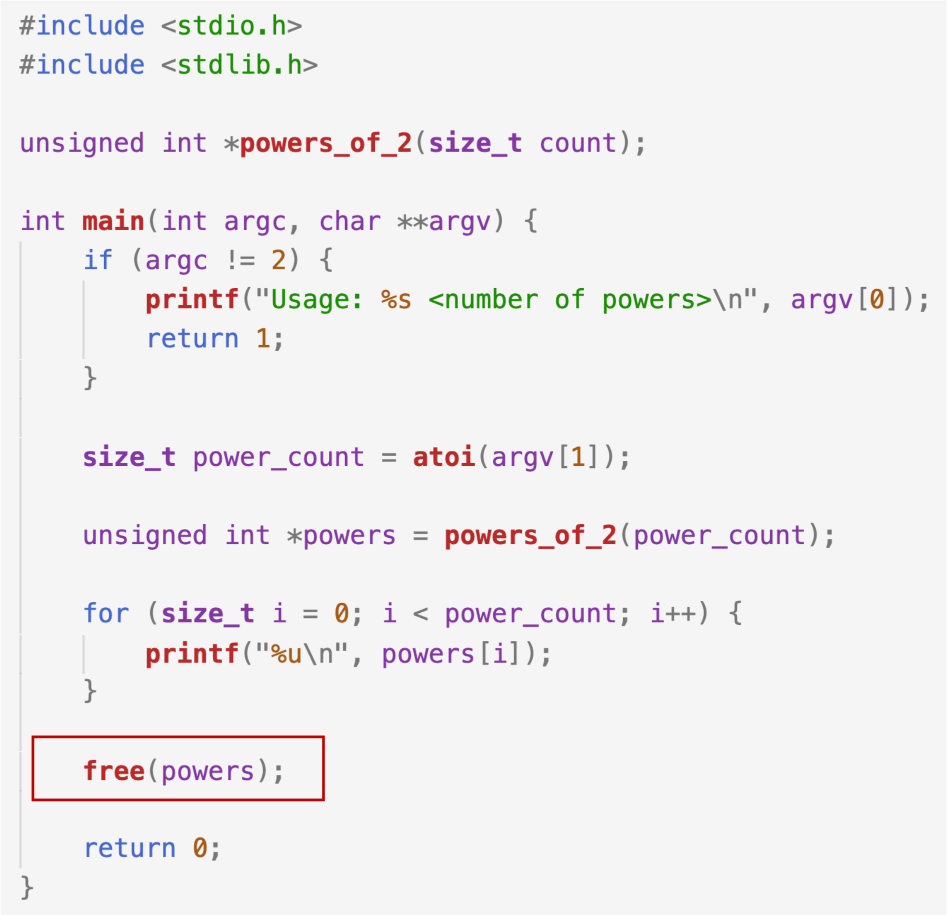


free()

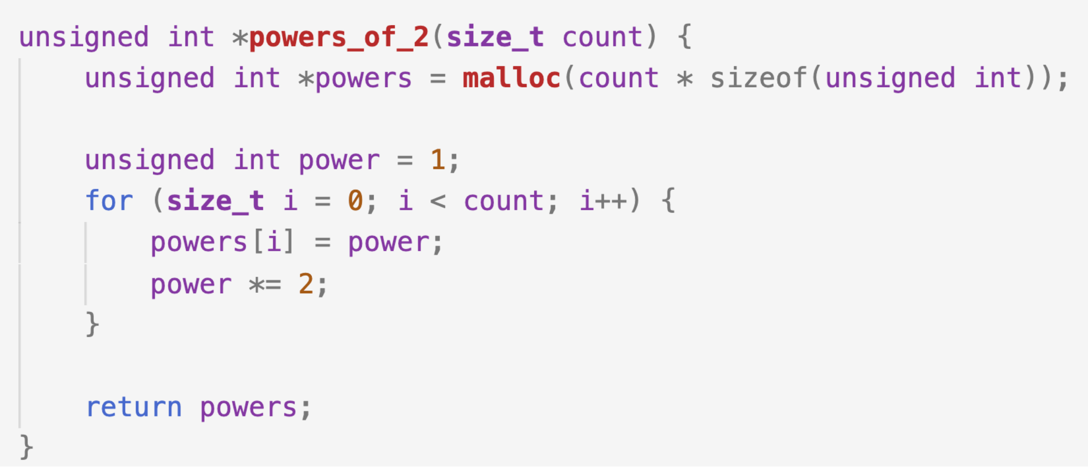


```
int main() {
    unsigned int *powers = powers_of_2(5);

    for (size_t i = 0; i < 5; i++) {
        printf("%u\n", powers[i]);
    }

    free(powers);

    return 0;
}
```

free()



```c
#include <stdio.h>
#include <stdlib.h>

unsigned int *powers_of_2(size_t count);

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

    size_t power_count = atoi(argv[1]);

    unsigned int *powers = powers_of_2(power_count);

    for (size_t i = 0; i < power_count; i++) {
        printf("%u\n", powers[i]);
    }

    free(powers);

    return 0;
}
```



```c
unsigned int *powers_of_2(size_t count) {
    unsigned int *powers = malloc(count * sizeof(unsigned int));

    unsigned int power = 1;
    for (size_t i = 0; i < count; i++) {
        powers[i] = power;
        power *= 2;
    }

    return powers;
}
```

Memory leak

- If ptr is a pointer to heap memory, reassigning ptr to point to something else will not free the memory automatically.
- If ptr is automatically deallocated when a function returns, the heap memory will also not be deallocated
- Heap memory that has no pointer pointing to it is inaccessible and cannot be freed until the program exits

calloc()

- calloc()
  - void *calloc(size_t count, size_t size)
  - Allocates enough memory to store count items of size bytes each
  - Memory is initialized to 0
- int *array = calloc(10, sizeof(int))
  - Create an array of 10 ints, initialized to 0