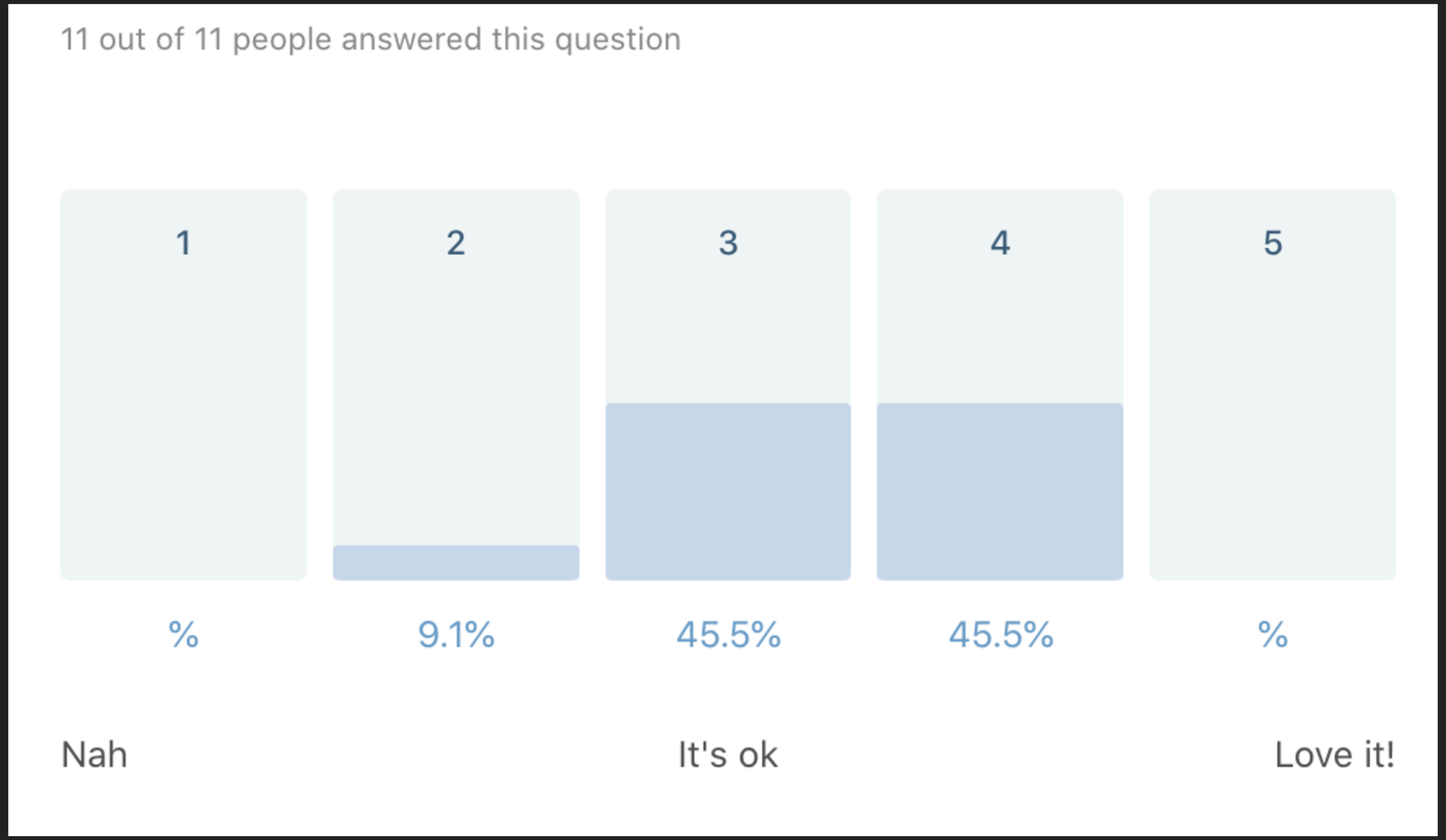
OCTOBER 28ST 2020

ELEMENTARY PROGRAMMING

SOME COVID BEST PRACTICES BEFORE WE START

- If you fill ill, go home
- Keep your distance to others
- Wash or sanitise your hands
- Disinfect table and chair
- Respect guidelines and restrictions

FEEDBACK CHECK

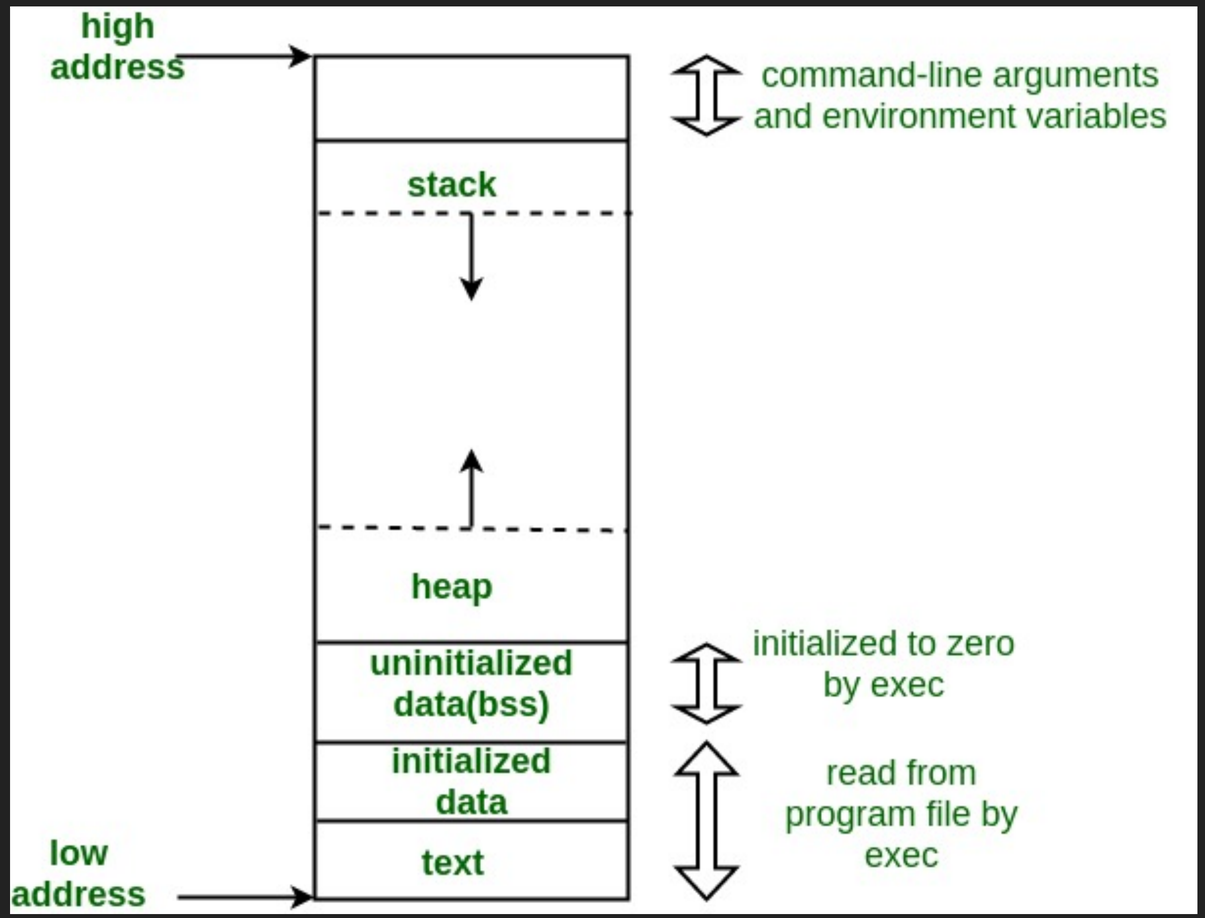


NEW FEEDBACK

- I would really like for you to take a survey at the end of the session
- Feedback is important, please take the time to do it
- Pretty please <3
- Type this in your browser http://bit.ly/elemprog8

MEMORY OF A PROGRAM (NOT ONLY WITH C)

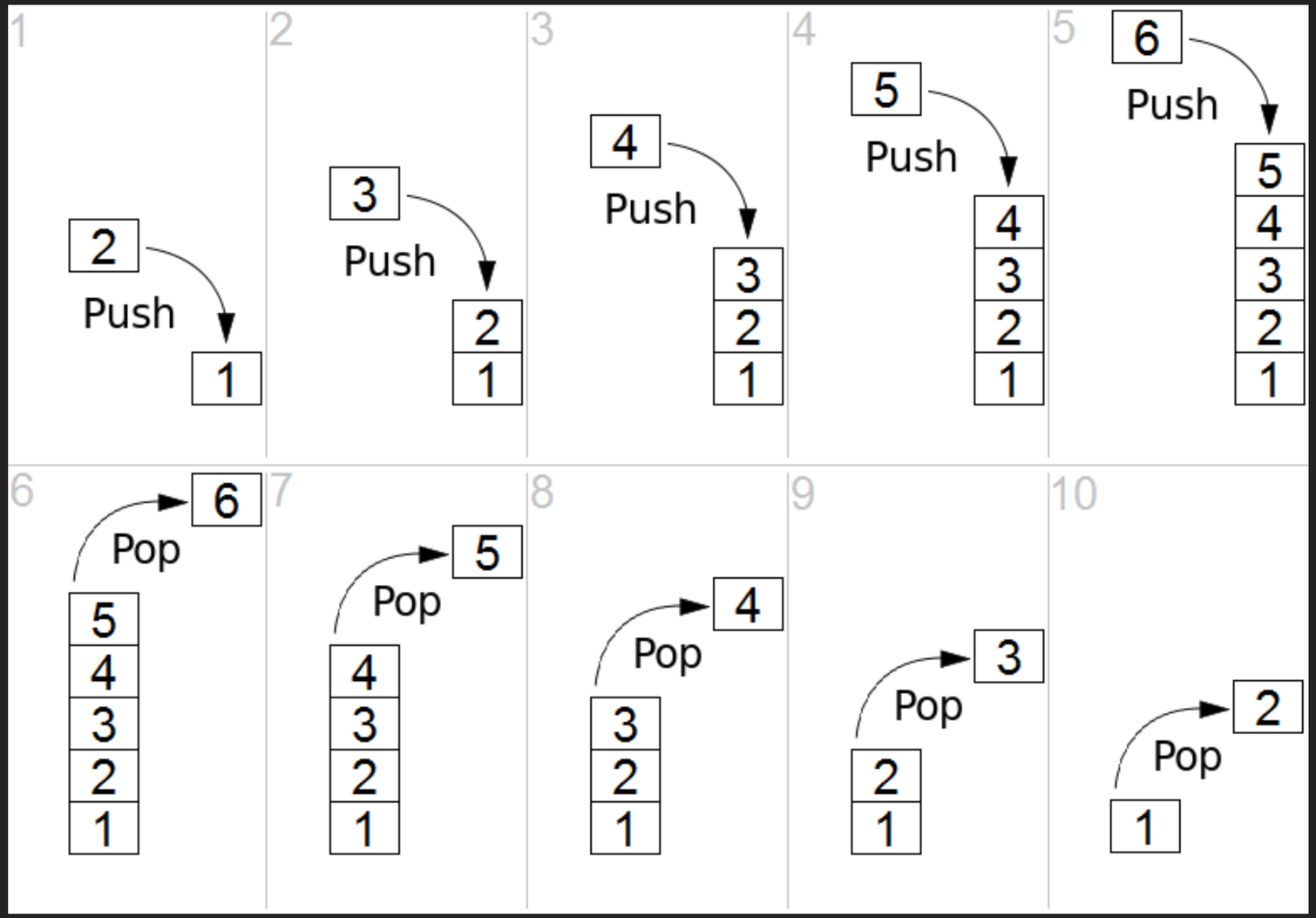
- The **code segment** (also called a **text segment**), where the compiled program sits in memory. The code segment is typically read-only.
- The **bss segment** (also called the **uninitialised data segment**), where zero-initialised global and static variables are stored.
- The **data segment** (also called the **initialised data segment**), where initialised global and static variables are stored.
- The **heap**, where dynamically allocated variables are allocated from.
- The **call stack**, where function parameters, local variables, and other function-related information are stored.



STACK

- It's a data structure
- It's a collection of elements
- It follows a Last In First Out policy (LIFO)
- It has to function associated:
  - Push => add an element in the stack
  - Pop => remove an element from the top of the stack

STACK


1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
6
Pop
5
4
3
2
1
7
5
Pop
4
3
2
1
8
4
Pop
3
2
1
9
3
Pop
2
1
10
2
Pop
1

STACK OF A PROGRAM

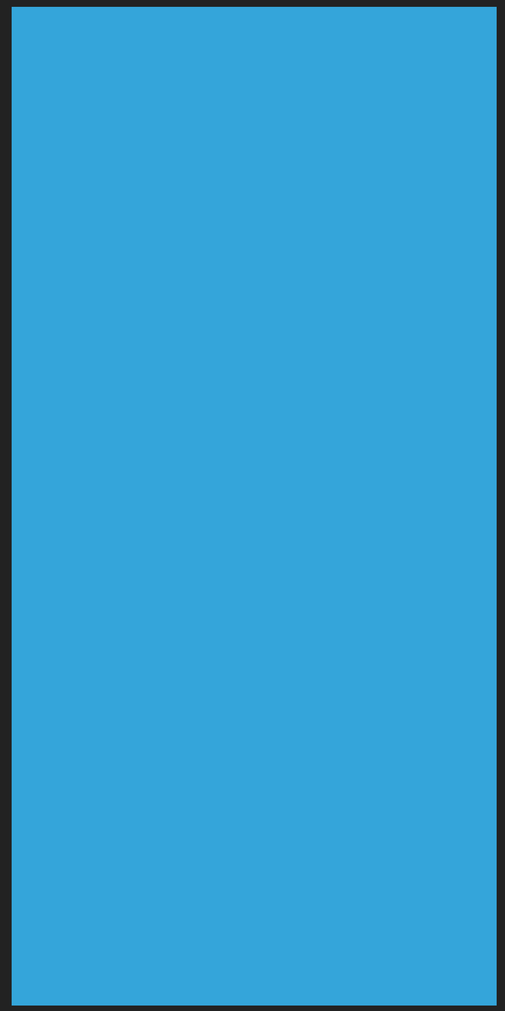
- The **stack area** contains the program stack, a LIFO structure, typically located in the higher parts of memory
- A **stack pointer** register tracks the top of the stack
- A **stack pointer** it is adjusted each time a value is “pushed” onto the stack
- The set of values pushed for one function call is termed a **stack frame**
- Memory is **managed** by the CPU

A STACK FRAME CONSISTS OF

- The address of the instruction beyond the function call (called the **return address**). This is how the CPU remembers where to return to after the called function exits.
- All **function arguments.**
- Memory for any **local variables**.
- **Saved copies of any registers modified by the function** that need to be restored when the function returns

STACK EXAMPLE

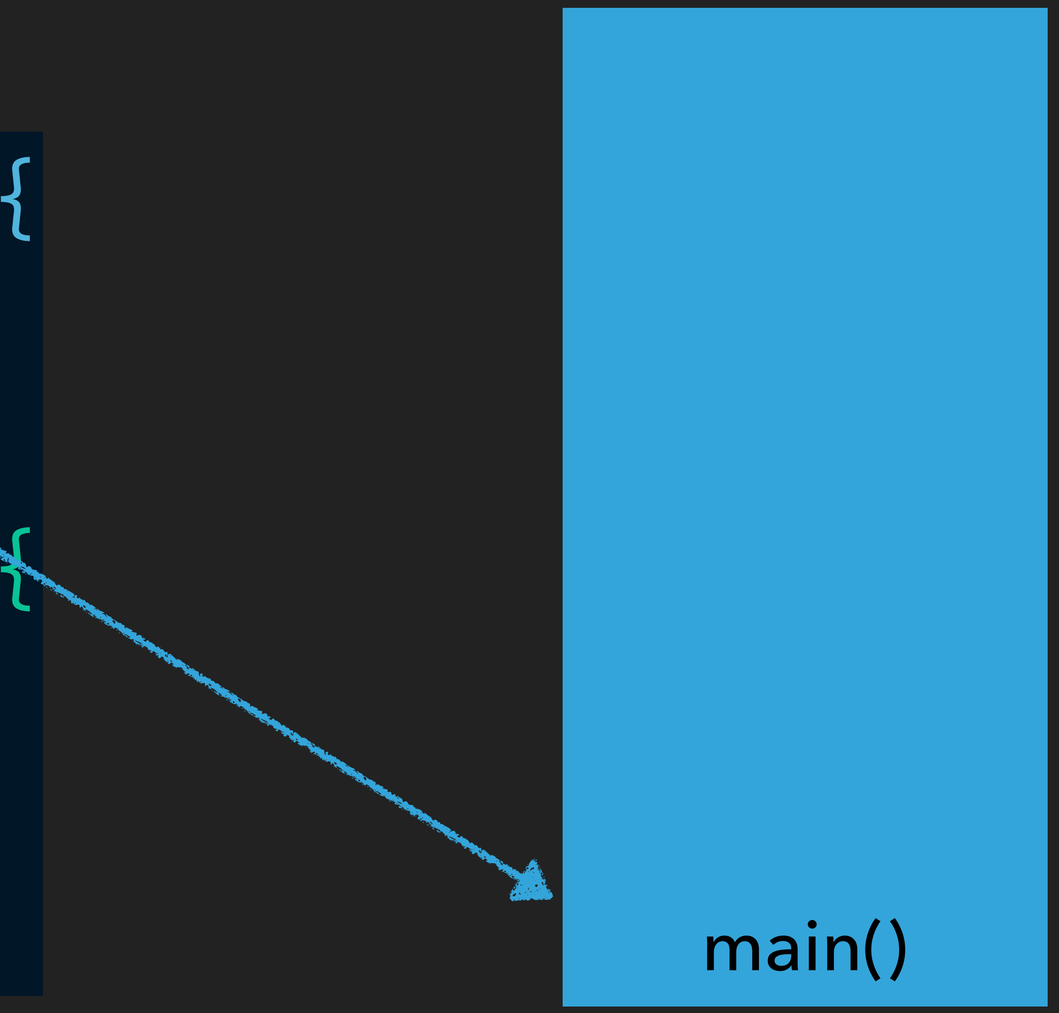
```
int foo(int x){
    return x;
}
int main(void){
    foo(5);
    return 0;
}
```



Stack

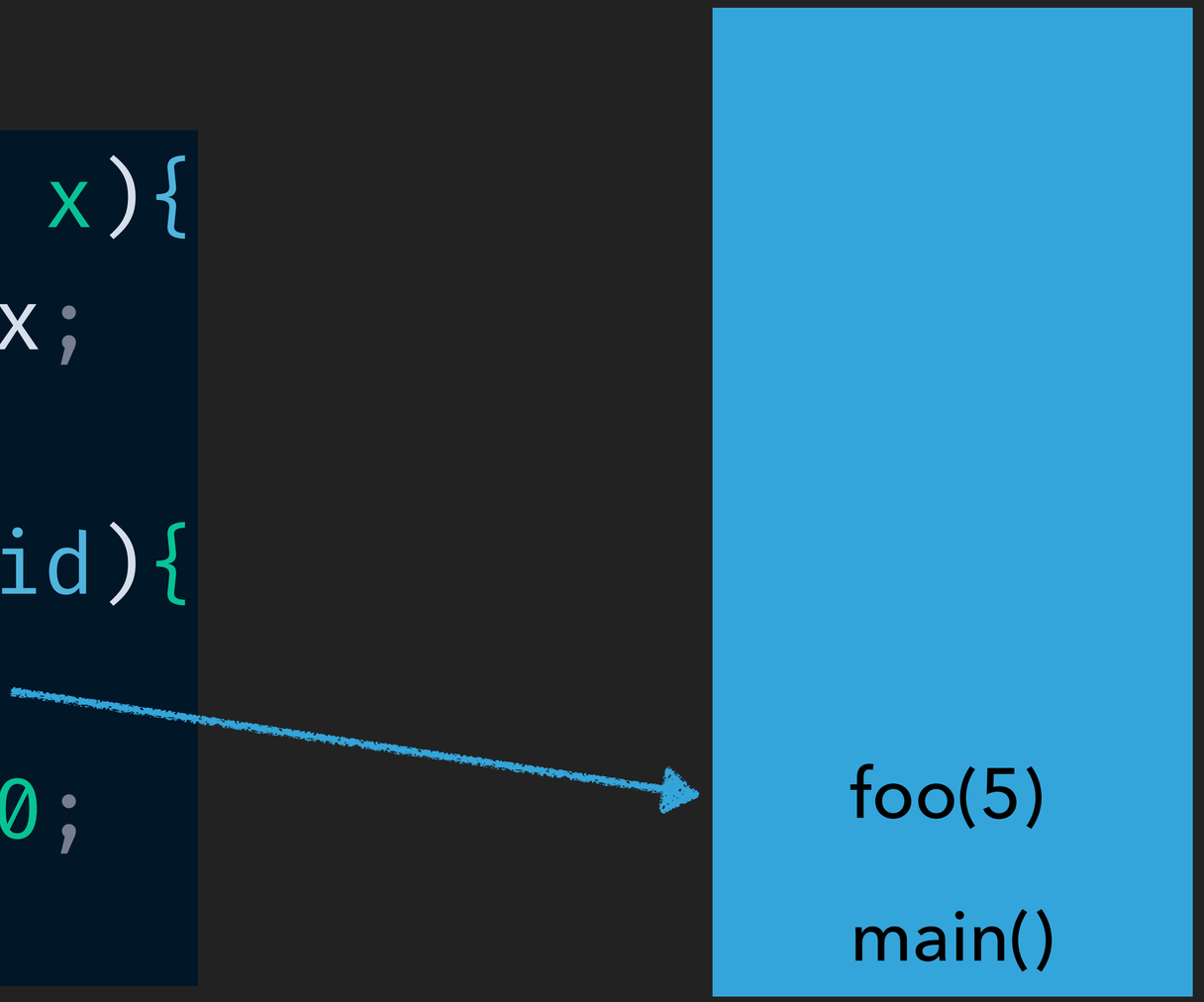
STACK EXAMPLE

```c
int foo(int x){
    return x;
}
int main(void){
    foo(5);
    return 0;
}
```



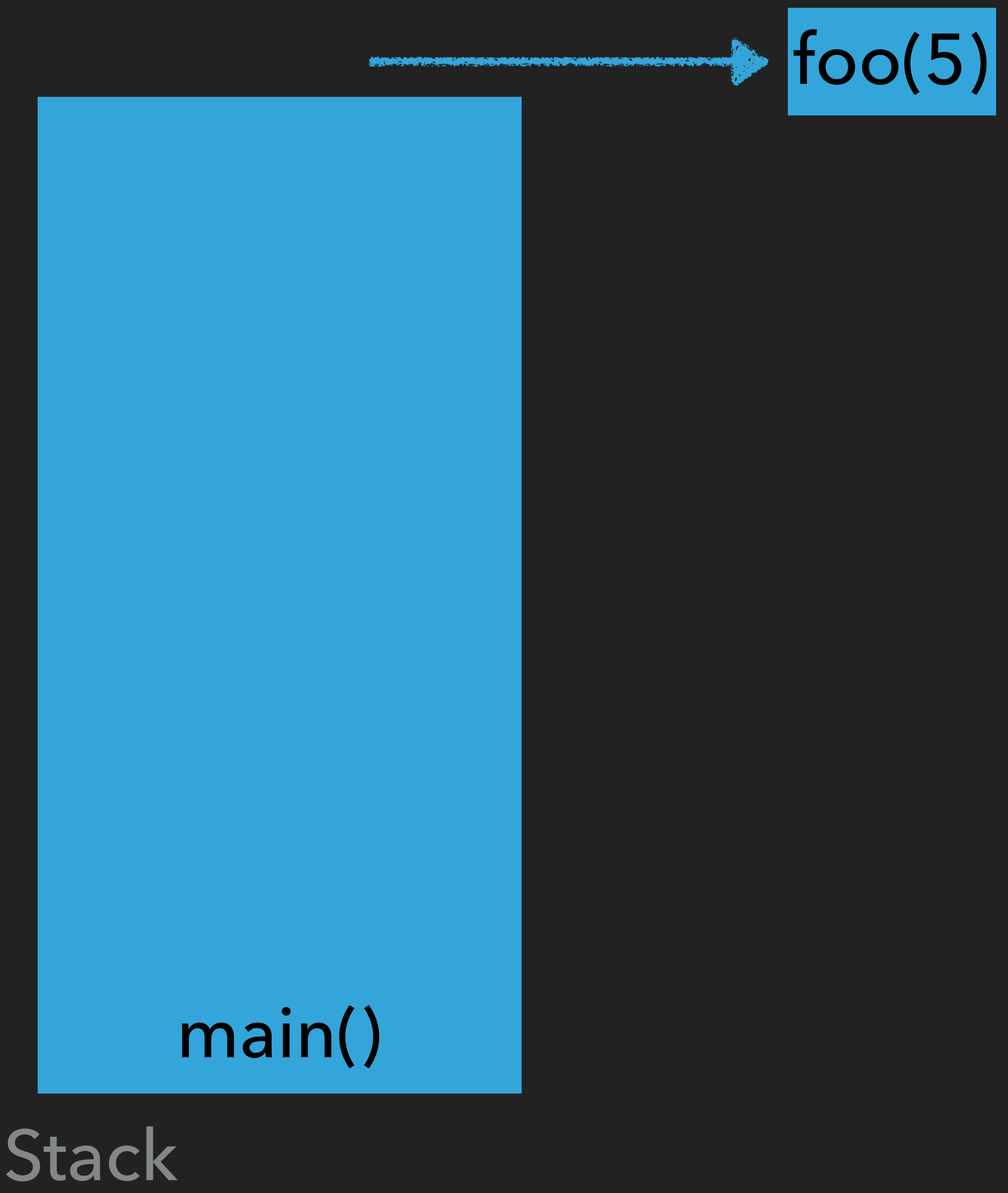
STACK EXAMPLE

```c
int foo(int x){
    return x;
}
int main(void){
    foo(5);
    return 0;
}
```



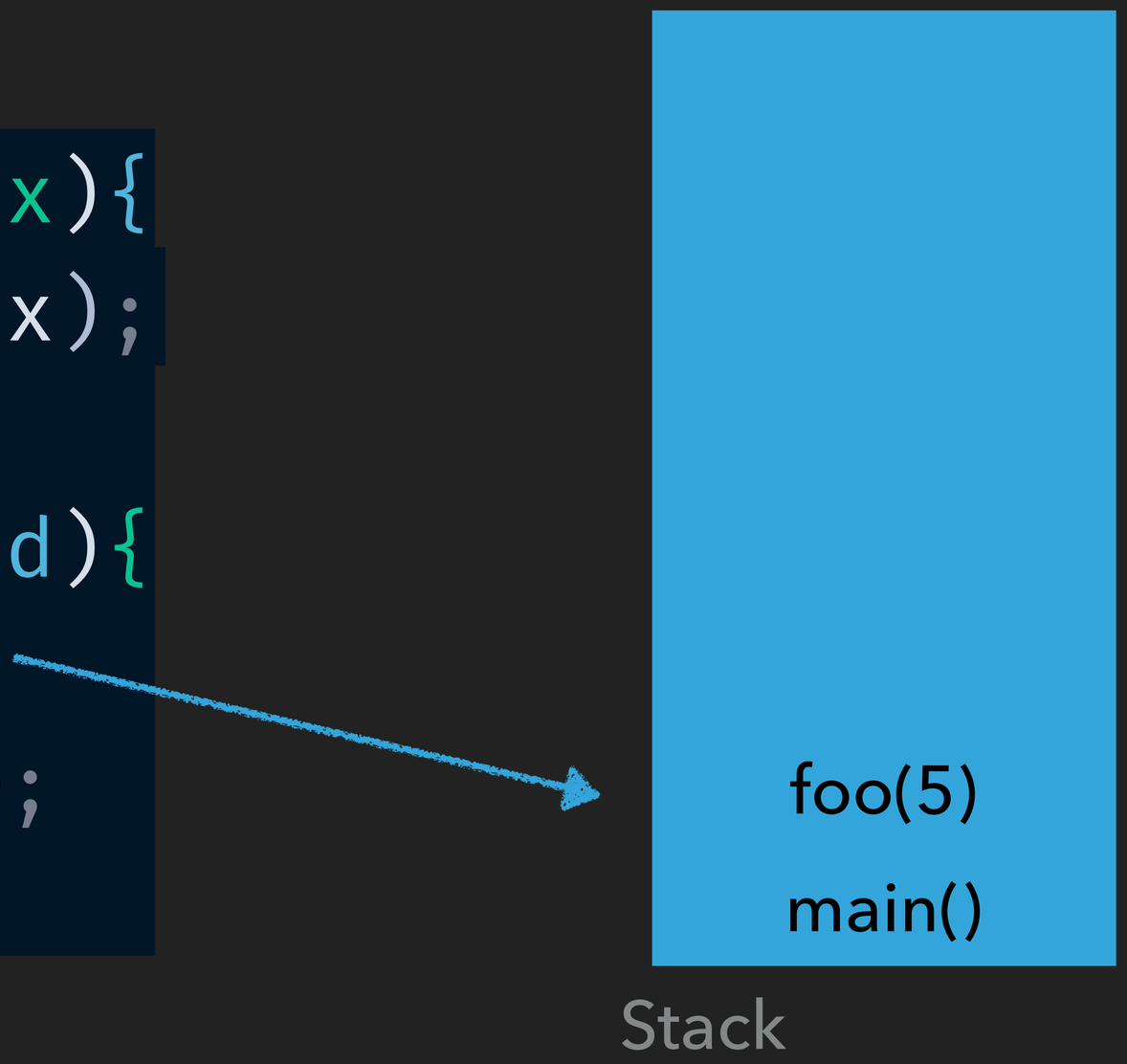
STACK EXAMPLE

```
int foo(int x){
    return x;
}
int main(void){
    foo(5);
    return 0;
}
```



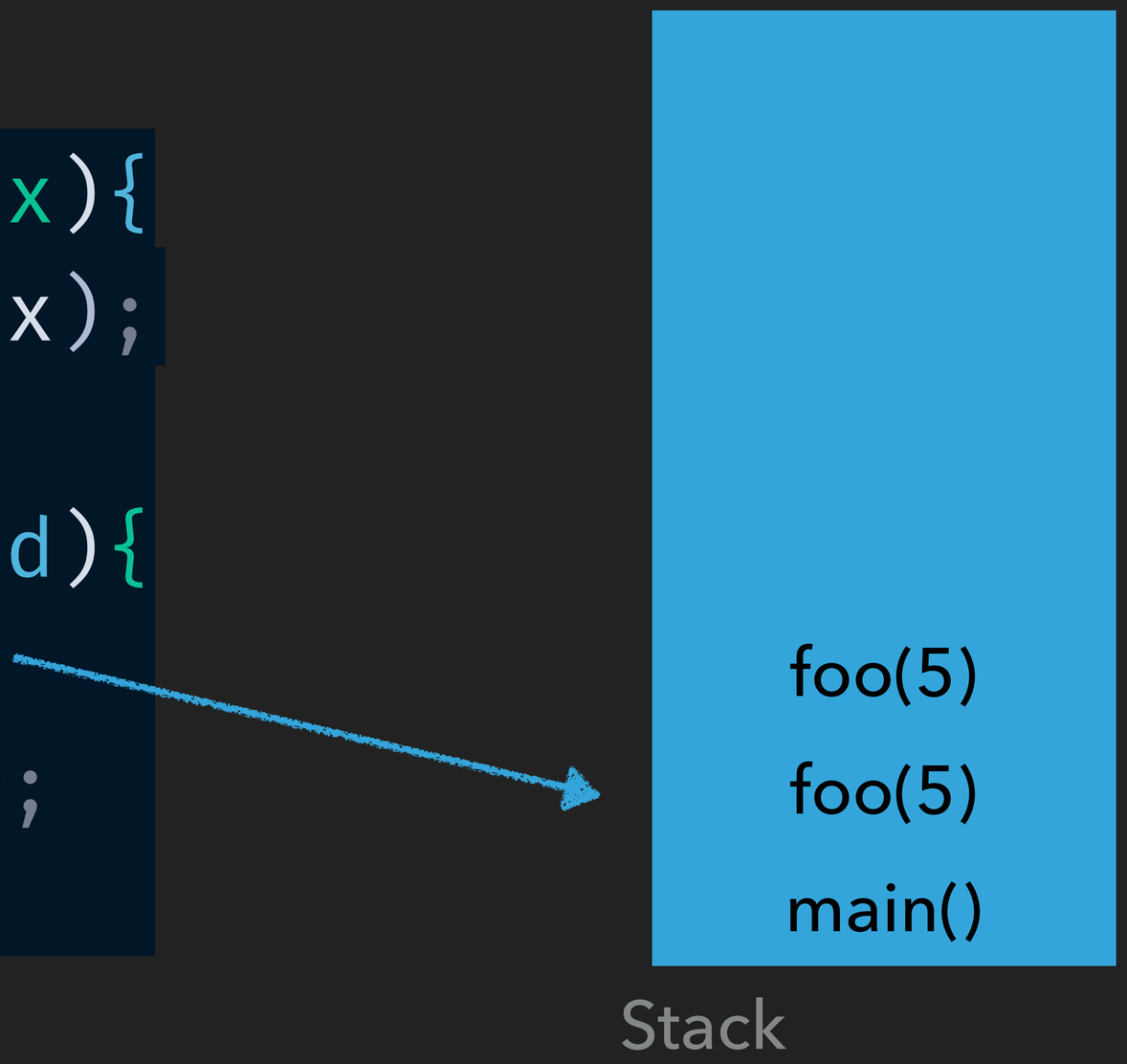
BAD STACK EXAMPLE

```
int foo(int x){
 return foo(x);
}
int main(void){
    foo(5);
    return 0;
}
```



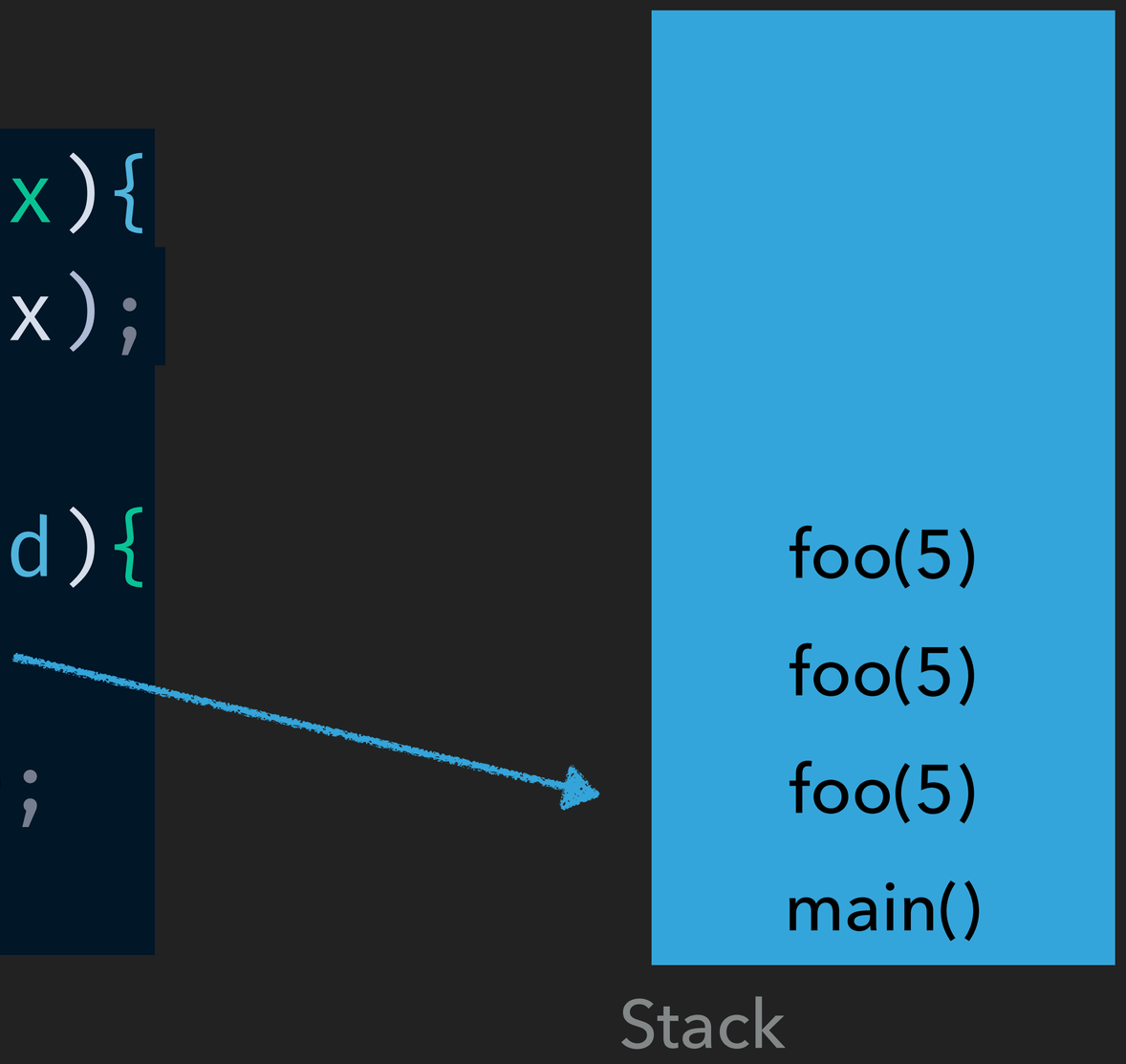
BAD STACK EXAMPLE

```
int foo(int x){
 return foo(x);
}
int main(void){
    foo(5);
    return 0;
}
```



BAD STACK EXAMPLE

```
int foo(int x){
 return foo(x);
}
int main(void){
    foo(5);
    return 0;
}
```



BAD STACK EXAMPLE

```
int foo(int x){
 return
}
int mai
    foo
    ret
}
```

BOOM!

Stack

THE HEAP

- The heap is the area of the memory where **dynamic allocation** happens
- You can access the heap with **pointers**
- The **programmer** needs to manage the memory and avoid memory leaks

HOW TO WORK ON THE HEAP

- C gives you three functions from `<stidlib.h>`:
  - `malloc` allocates a block of memory but doesn't initialise it
  - `calloc` allocates a block of memory and clears it
  - `realloc` resizes a previously allocated block of memory

MALLOC VS CALLOC

- `malloc` and `calloc` differs because after you used `malloc` you cannot start using the pointer right away, whereas with `calloc` you have the initialisation to 0

HOW TO USE MALLOC

```c
int *arr  = malloc(10000000 * sizeof(int));
 if (arr == NULL) {
     printf("memory could not be allocated\n");
     exit(EXIT_FAILURE);
 }
 for (int i = 0; i < 15; i++) {
     printf("%d\n", arr[i]);
 }
```

```c
int *newArrL = calloc(5, sizeof(int));
if (newArrL == NULL) {
    printf("memory could not be allocated\n");
    exit(EXIT_FAILURE);
}
```

HOW TO USE CALLOC

```
int *newArrL = calloc(5, sizeof(int));
if (newArrL == NULL) {
    printf("memory could not be allocated\n");
    exit(EXIT_FAILURE);
}
```

HOW TO USE REALLOC

```
    int *newArr = realloc(arr, 5 * sizeof(int));
    if (newArr == NULL) {
        printf("memory could not be allocated\n");
        exit(EXIT_FAILURE);
    }
```

SURPRISE SURPRISE: ALWAYS CLEAN

```
int *newArrL = calloc(5, sizeof(int));
    if (newArrL == NULL) {
        printf("memory could not be allocated\n”);
        exit(EXIT_FAILURE);
    }
    // do your things
    free(newArrL);
```

NULL POINTER

- A `NULL` pointer is a pointer that points to nothing:
  - when you initialise without assignment your pointer will point to `NULL`
  - when you assign a pointer to a function the returns a pointer (eg. `realloc`), if pointer is `NULL` means something bad happened

STRUCT

- A struct is composite data type that allows me to do pretty cool thing like model reality
- Being a composite data type means that I can use it to create new types

EXAMPLE OF A STRUCT

```
typedef struct {
    char * name;
    int code;
    int    maxStudents;
} Course;
```

HOW TO INITIALIZE A STRUCT

```
Course c;
c.name       = "Something";
c.code       = 1234;
c.maxStudents = 4;
```

NEW FEEDBACK

- I would really like for you to take a survey at the end of the session
- Feedback is important, please take the time to do it
- Pretty please <3
- Type this in your browser http://bit.ly/elemprog8

SOME COVID BEST PRACTICES BEFORE WE LEAVE

- Disinfect table and chair
- Maintain your distance to others
- Wash or sanitise your hands
- Respect guidelines and restrictions outside