COMP1511 PROGRAMMING FUNDAMENTALS

# LECTURE 10

fine, this time it is for real - STRINGS
The start of a beautiful friendship- Linked Lists

# LAST TIME...

- Revisited pointers to make a point
- Characters and some of their fun functions

# TODAY...

- Strings (or maybe I will just continue to string you along)
- The one, the only, the truly magical, magnificent Linked Lists

## WHERE IS THE CODE?

**Live lecture code can be found here:**

HTTPS://CGI.CSE.UNSW.EDU.AU/~CS1511/22T1/LIVE/WEEK05/

# STRINGS

## WHAT ARE THEY?

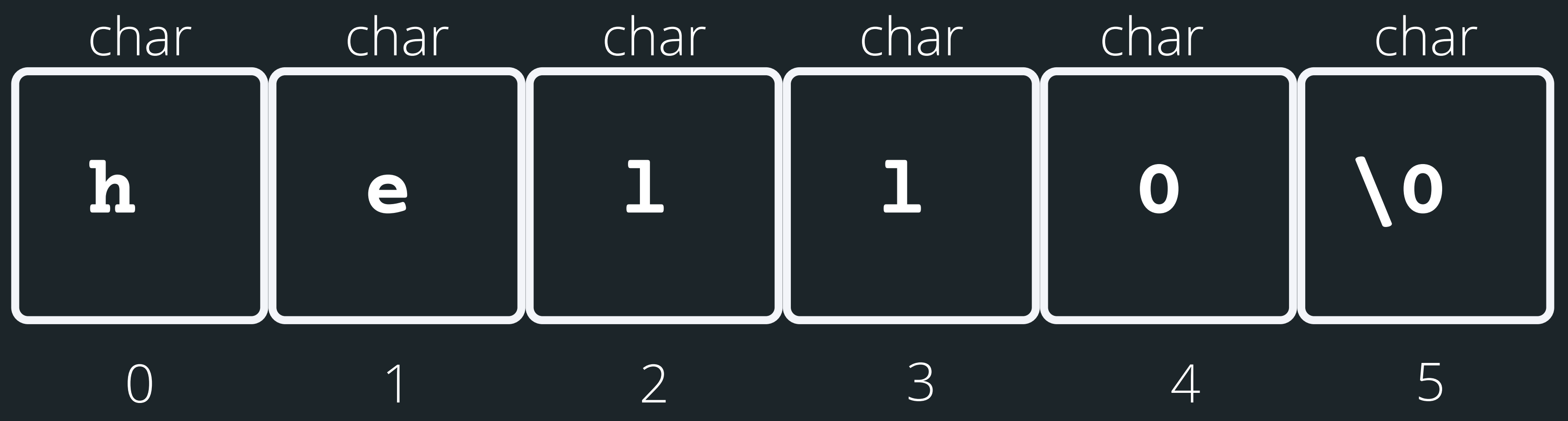
- Strings are a collection of characters that are joined together
    - an array of characters!
- There is one very special thing about strings in C - it is an array of characters that finishes with a
    - This symbol is called a null terminating character
- It is always located at the end of an array, therefore an array has to always be able to accomodate this character
- It is not displayed as part of the string
- It is a placeholder to indicate that this array of characters is a string
- It is very useful to know when our string has come to an end, when we loop through the array of characters

# HOW DO WE DECLARE A STRING?

## WHAT DOES IT LOOK LIKE VISUALLY?

- Because strings are an array of characters, the array type is char.
- To declare and initialise a string, you can use two methods:

```c
//the more convenient way
char word[] = "hello";
//this is the same as'\0':
char word[] = {'h','e','l','l','o','\0'};
```



| char | char | char | char | char | char |
|---|---|---|---|---|---|
| h | e | l | l | o | \0 |
| 0 | 1 | 2 | 3 | 4 | 5 |

# HELPFUL LIBRARY FUNCTIONS FOR STRINGS

## FGETS()

There is a useful function for reading strings:

**`fgets(array[], length, stream)`**

The function needs three inputs:

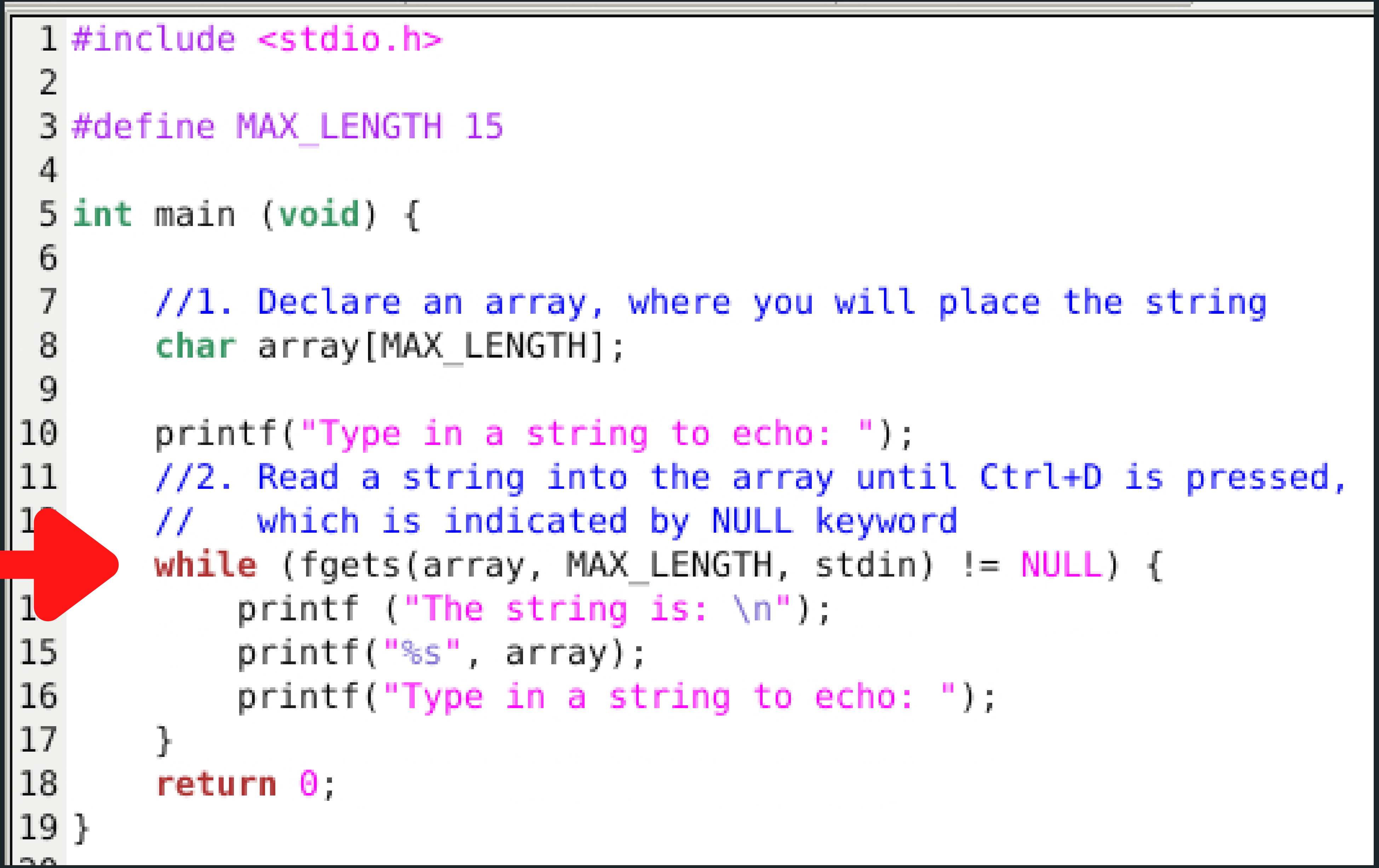
- array[] - the array that the string will be stored into
- length - the number of characters that will be read in
- stream - this is where this string is coming from - you don't have to worry about this one, in your case, it will always be stdin (the input will always be from terminal)

```
// Declare an array where you will place the
string that you read from somewhere
char array[MAX_LENGTH];
// Read in the string into array of length
MAX_LENGTH from terminal input
fgets(array, MAX_LENGTH, sdin)
```

# HOW DO I KEEP READING STUFF IN OVER AND OVER AGAIN?

Using the **NULL** keyword, you can continuously get string input from terminal until Ctrl+D is pressed

- fgets() stops reading when either length-1 characters are read, newline character is read or an end of file is reached, whichever comes first



```c
#include <stdio.h>

#define MAX_LENGTH 15

int main (void) {

    //1. Declare an array, where you will place the string
    char array[MAX_LENGTH];

    printf("Type in a string to echo: ");
    //2. Read a string into the array until Ctrl+D is pressed,
    //   which is indicated by NULL keyword
    while (fgets(array, MAX_LENGTH, stdin) != NULL) {
        printf ("The string is: \n");
        printf("%s", array);
        printf("Type in a string to echo: ");
    }
    return 0;
}
```

# HELPFUL LIBRARY FUNCTIONS FOR STRINGS

## FPUTS()

Another useful function to output strings:

```
fputs(array[], stream)
```

The function needs two inputs:

- array[] - the array that the string is be stored in
- stream - this is where this string will be output to, you don't have to worry about this one, in your case, it will always be stdout (the output will always be in terminal)

```
// Declare an array where you will place the
string that you read from somewhere
char array[MAX_LENGTH];
// Read in the string into array of length
MAX_LENGTH from terminal input
fgets(array, MAX_LENGTH, sdin)
//Output the array now
fputs(array, stdout)
```

# SOME OTHER INTERESTING STRING FUNCTIONS

## <STRING.H> STANDARD LIBRARY

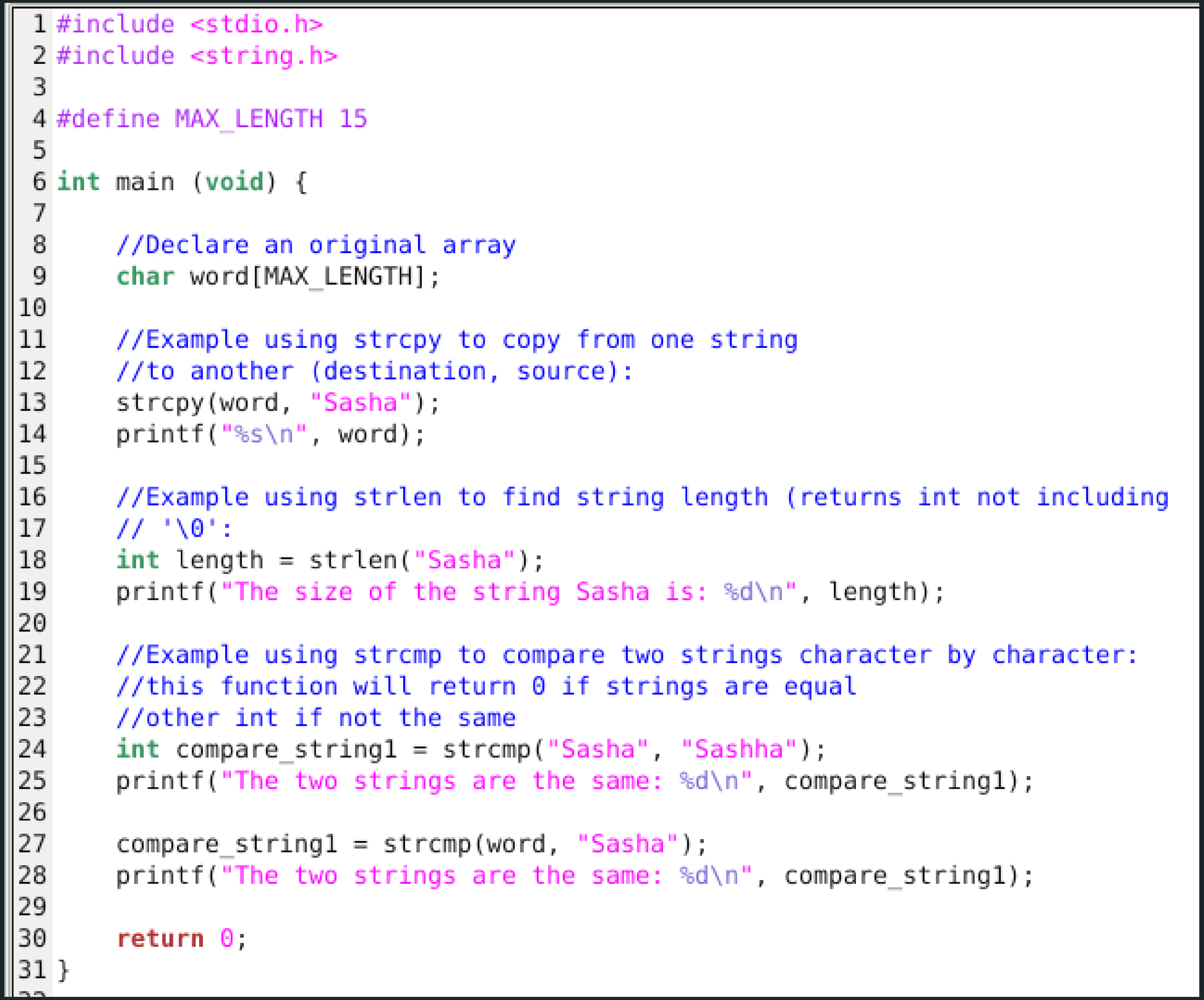
**CHECK OUT THE REST OF THE FUNCTIONS: HTTPS://WWW.TUTORIALSPOINT.COM/C_STANDARD_LIBRARY/STRING_H.HTM**

Some other useful functions for strings:

- **`strlen()`**gives us the length of the string (excluding the ’\0’
- **`strcpy()`**copy the contents of one string to another
- **`strcat()`**attach one string to the end of another (concatenate)
- **`strcmp()`**compare two strings
- **`strchr()`** find the first or last occurance of a character

# USING SOME OF THESE FUNCTIONS

# STRINGS



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

#define MAX_LENGTH 15

int main (void) {

    //Declare an original array
    char word[MAX_LENGTH];

    //Example using strcpy to copy from one string
    //to another (destination, source):
    strcpy(word, "Sasha");
    printf("%s\n", word);

    //Example using strlen to find string length (returns int not including
    // '\0':
    int length = strlen("Sasha");
    printf("The size of the string Sasha is: %d\n", length);

    //Example using strcmp to compare two strings character by character:
    //this function will return 0 if strings are equal
    //other int if not the same
    int compare_string1 = strcmp("Sasha", "Sashha");
    printf("The two strings are the same: %d\n", compare_string1);

    compare_string1 = strcmp(word, "Sasha");
    printf("The two strings are the same: %d\n", compare_string1);

    return 0;
}
```

# QUICK REHASH

## MEMORY

So far we have talked a bit about how variables are stored in memory, and live in their world {} in the stack memory

- This means that if we create data inside a function, it will die when that function finishes running
- This is memory that is allocated by the compiler at compile time…

```
// Make an array
int *create_array(void) {
    int numbers[10] = {0};
    // Return pointer to the array
    return numbers;
}
//However, when we close the curly brakes,our
//array is killed, so we are returning a
//pointer to memory that we no longer have...
```

# BUT WHAT HAPPENS IF I WANT TO SAVE SOME MEMORY?

## MALLOC()

- We do have the wonderful opportunity to allocate some memory by calling the function `malloc()` and letting this function know how many bytes of memory we want
    - this function returns a pointer to the piece of memory we created based on the number of bytes we specified as the input to this function
    - this also allows us to dynamically create memory as we need it - neat!
    - This means that we are now in control of this memory (cue the evil laugh!)

# WHAT IF I RUN WILD AND JUST KEEP ASKING FOR MEMORY?

## FREE()

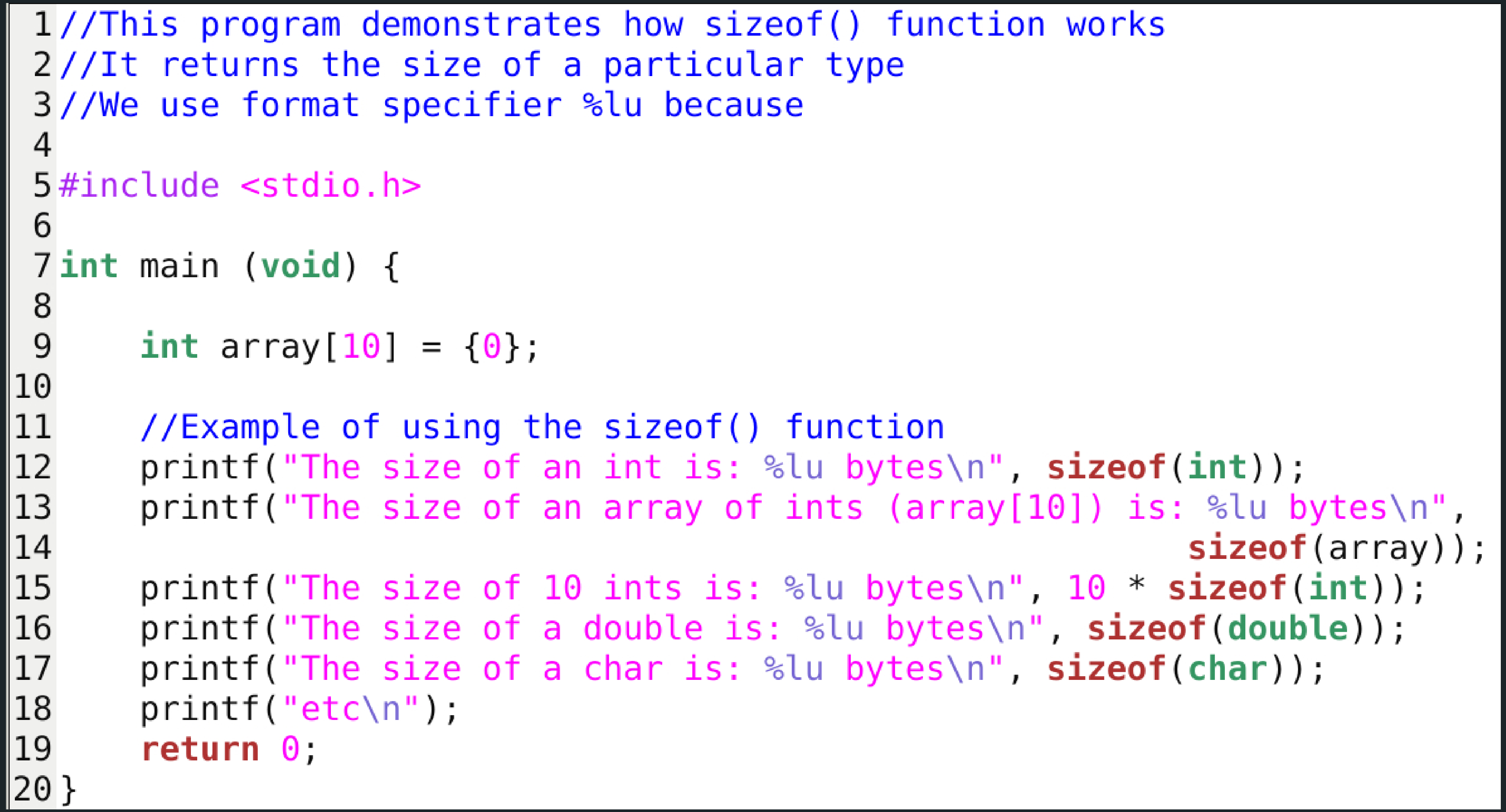
It would be very impolite to keep requesting memory to be made (and hog all that memory!), without giving some back...

- This piece of memory is ours to control and it is important to remember to kill it or you will eat up all the memory you computer has... often called a memory leak...
- A memory leak occurs when you have dynamically allocated memory (with **`malloc()`**) that you do not free - as a result, memory is lost and can never be free causing a memory leak
- You can free memory that you have created by using the function **`free()`**

# HOW DO I KNOW HOW MUCH MEMORY TO ASK FOR WHEN I USE MALLOC()

## SIZEOF()

- We can use the function **`sizeof()`** to give us the exact number of bytes we need to malloc (memory allocate)

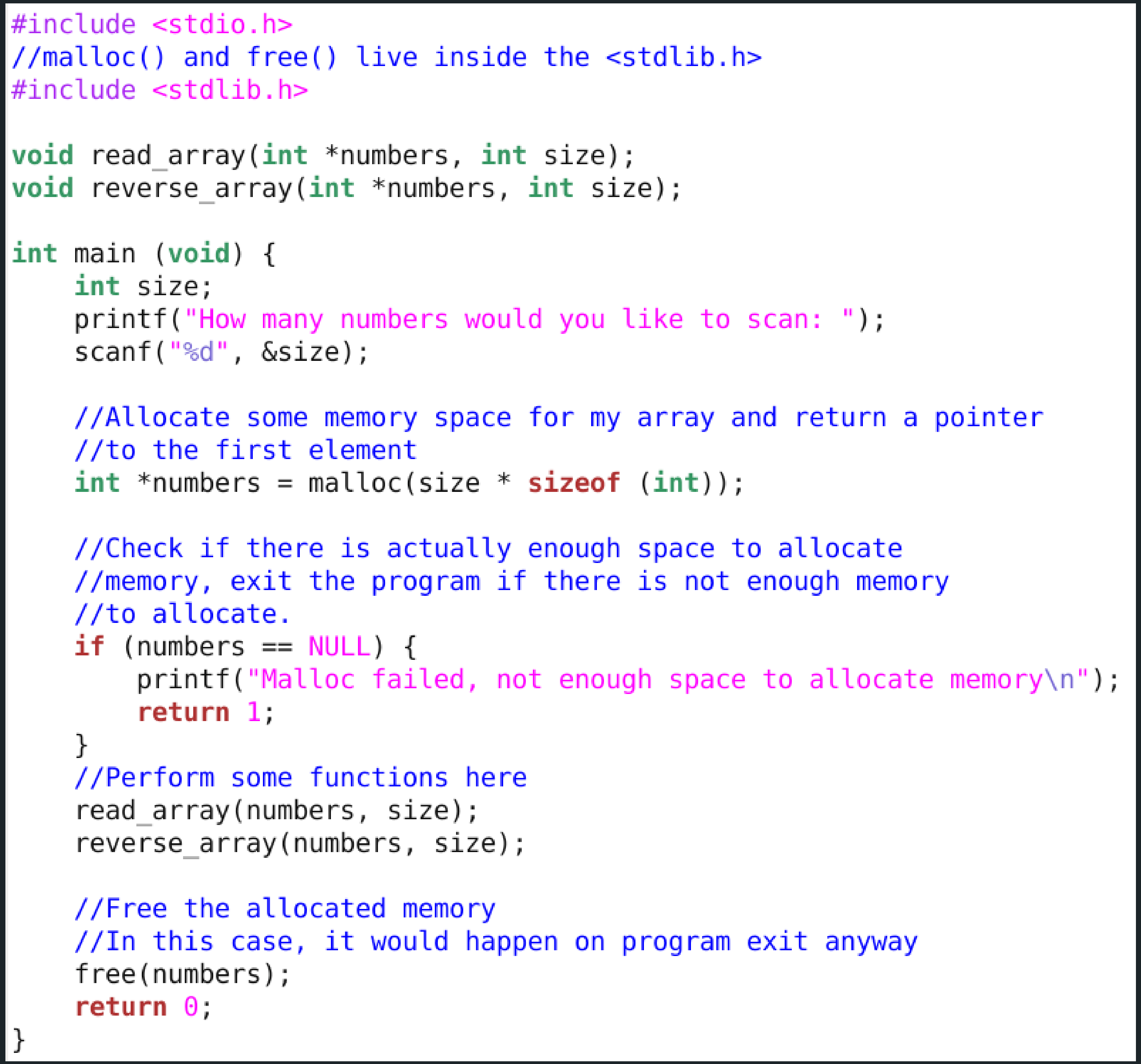


```c
//This program demonstrates how sizeof() function works
//It returns the size of a particular type
//We use format specifier %lu because

#include <stdio.h>

int main (void) {

    int array[10] = {0};

    //Example of using the sizeof() function
    printf("The size of an int is: %lu bytes\n", sizeof(int));
    printf("The size of an array of ints (array[10]) is: %lu bytes\n",
                                                        sizeof(array));
    printf("The size of 10 ints is: %lu bytes\n", 10 * sizeof(int));
    printf("The size of a double is: %lu bytes\n", sizeof(double));
    printf("The size of a char is: %lu bytes\n", sizeof(char));
    printf("etc\n");
    return 0;
}
```

# PUTTING IT ALL TOGETHER:

## MALLOC(SIZEOF())
## FREE()

- Using all of these together in a simple example:

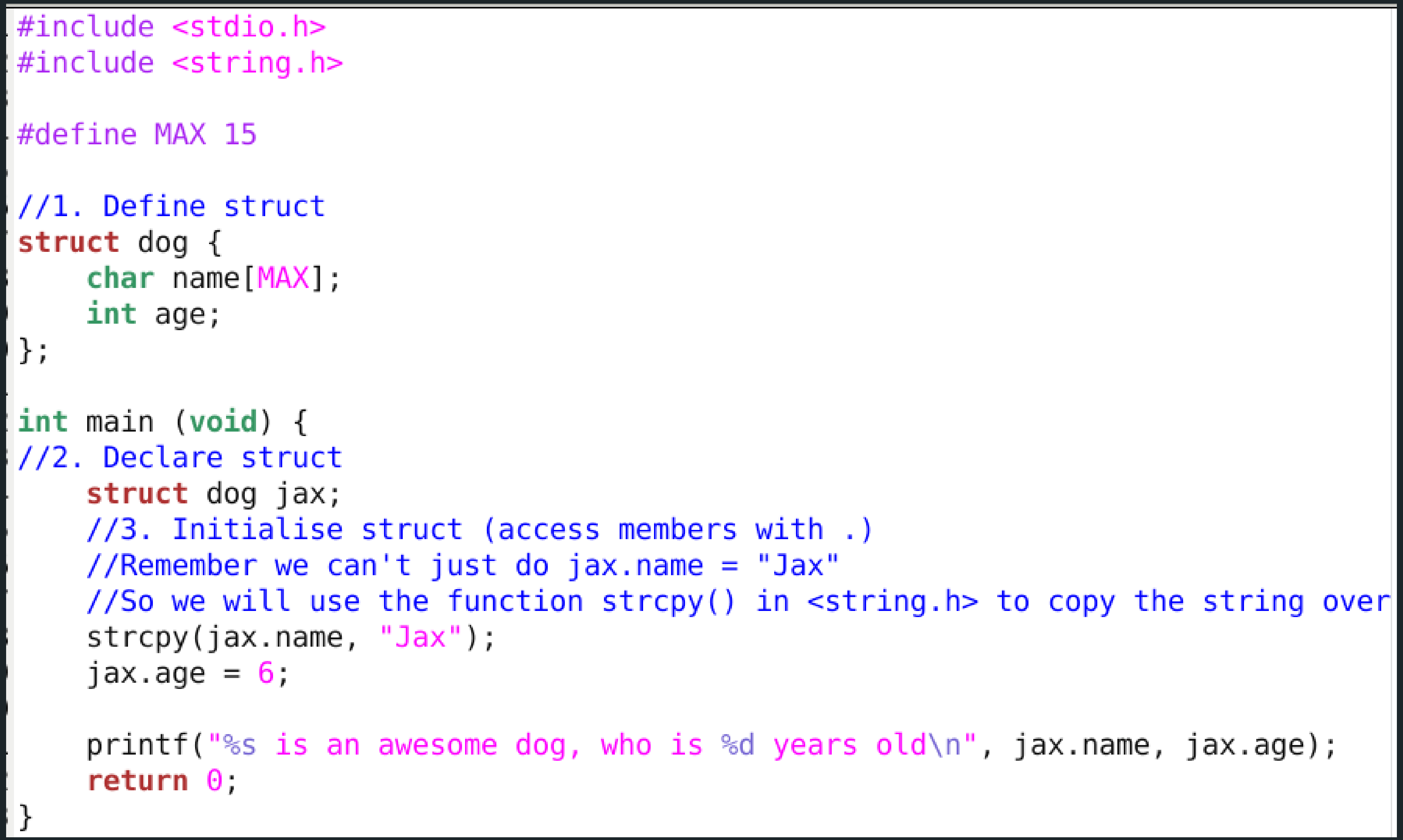


```c
#include <stdio.h>
//malloc() and free() live inside the <stdlib.h>
#include <stdlib.h>

void read_array(int *numbers, int size);
void reverse_array(int *numbers, int size);

int main (void) {
    int size;
    printf("How many numbers would you like to scan: ");
    scanf("%d", &size);

    //Allocate some memory space for my array and return a pointer
    //to the first element
    int *numbers = malloc(size * sizeof (int));

    //Check if there is actually enough space to allocate
    //memory, exit the program if there is not enough memory
    //to allocate.
    if (numbers == NULL) {
        printf("Malloc failed, not enough space to allocate memory\n");
        return 1;
    }
    //Perform some functions here
    read_array(numbers, size);
    reverse_array(numbers, size);

    //Free the allocated memory
    //In this case, it would happen on program exit anyway
    free(numbers);
    return 0;
}
```

# STRUCTS AND POINTERS

## -> VERSUS .

- Remember that when we access members of a struct we use a .

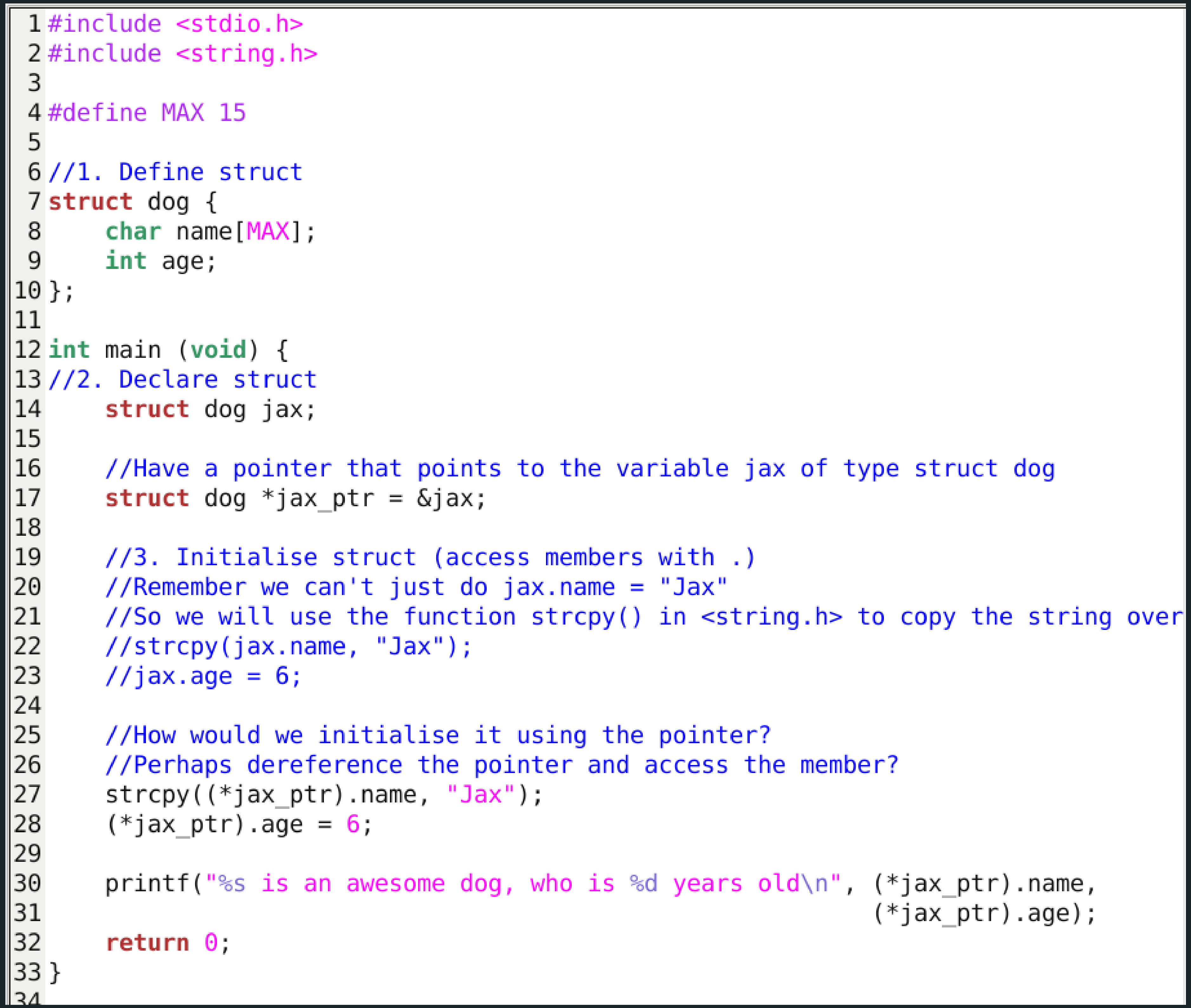


```c
#include <stdio.h>
#include <string.h>

#define MAX 15

//1. Define struct
struct dog {
    char name[MAX];
    int age;
};

int main (void) {
//2. Declare struct
    struct dog jax;
    //3. Initialise struct (access members with .)
    //Remember we can't just do jax.name = "Jax"
    //So we will use the function strcpy() in <string.h> to copy the string over
    strcpy(jax.name, "Jax");
    jax.age = 6;

    printf("%s is an awesome dog, who is %d years old\n", jax.name, jax.age);
    return 0;
}
```

# STRUCTS AND POINTERS

## -> VERSUS .

- What happens if we make a pointer of type struct? How do we access it then?

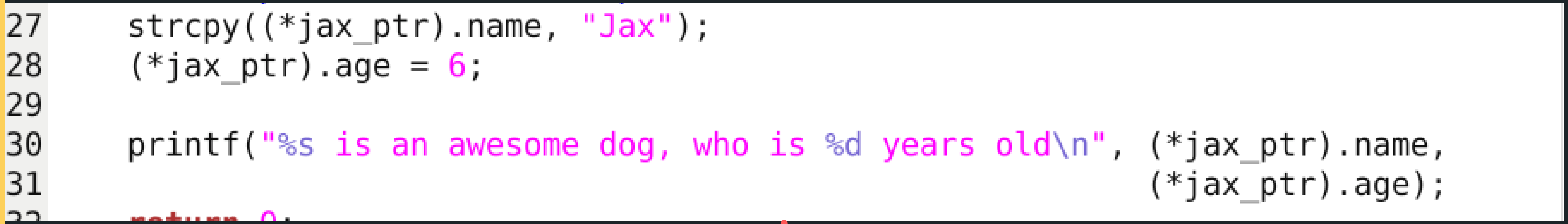


```c
#include <stdio.h>
#include <string.h>

#define MAX 15

//1. Define struct
struct dog {
    char name[MAX];
    int age;
};

int main (void) {
//2. Declare struct
    struct dog jax;

    //Have a pointer that points to the variable jax of type struct dog
    struct dog *jax_ptr = &jax;

    //3. Initialise struct (access members with .)
    //Remember we can't just do jax.name = "Jax"
    //So we will use the function strcpy() in <string.h> to copy the string over
    //strcpy(jax.name, "Jax");
    //jax.age = 6;

    //How would we initialise it using the pointer?
    //Perhaps dereference the pointer and access the member?
    strcpy((*jax_ptr).name, "Jax");
    (*jax_ptr).age = 6;

    printf("%s is an awesome dog, who is %d years old\n", (*jax_ptr).name,
                                                          (*jax_ptr).age);

    return 0;
}
```

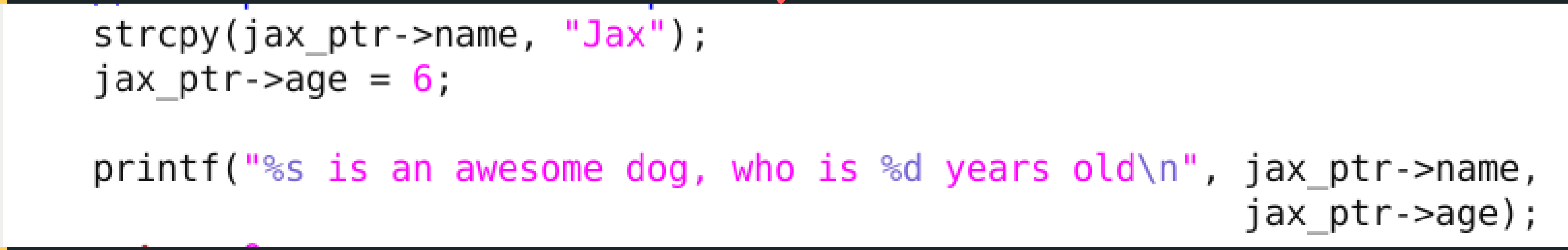
# STRUCTS AND POINTERS

## -> VERSUS .

- Those brackets can get quite confusing, so there is a shorthand way to do this with an ->
- There is no need to use *(jax_ptr) and instead can just straight jax_ptr ->



```
    strcpy((*jax_ptr).name, "Jax");
    (*jax_ptr).age = 6;

    printf("%s is an awesome dog, who is %d years old\n", (*jax_ptr).name,
                                                           (*jax_ptr).age);
    return 0;
```



```
    strcpy(jax_ptr->name, "Jax");
    jax_ptr->age = 6;

    printf("%s is an awesome dog, who is %d years old\n", jax_ptr->name,
                                                           jax_ptr->age);
```

# WHY ARE YOU HURTING US WITH ALL THIS STUFF?

## WE HAVE COME TO THE ULTIMATE REVEAL.

- Now that you have become comfortable with arrays, we are going to become acquainted with another important data structure (drum roll please ):

- The one and only LINKED LIST

# INTRODUCING A NEW DATA STRUCTURE

## LINKED LISTS

- Like an array, a linked list is used to store a collection of the same data type
- So what's the point?
    - Linked lists are dynamically sized, that means we can grow and shrink them as needed - efficient for memory!
    - Elements of a linked list (called nodes) do NOT need to be stored contiguously in memory, like an array.
    - Unlike arrays, linked lists are not random access data structures! You can only access items sequentially, starting from the beginning of the list.

# LET'S VISUALISE IT

## LINKED LISTS

# HAVE A RESTFUL FLEX WEEK!

- We hope that you all have a good rest and catch up over the Flex Week time.
  - There are no formal classes next week!
  - There is a social COM-PUN-TITION event
  - There are two bonus ethics talks for those interested in how ethics is dealt with in computing - it is a fascinating topic!
- Help Sessions are still running, please check the timetable
- Forum will be monitored closely to help you with any Assignment 1 queries

# SOCIAL AND BONUS STREAMS IN FLEX WEEK

Social events and Bonus Streams next week (none of the material is examinable and optional if you are interested!):

Com-pun-tition = Make me laugh until I cry
Wednesday 1pm  (Register via QR Code)

Ethics guest talks by Dr Sebastian Sequoiah-Grayson (videos will be available next week, and then log in for a chat about ethics and what role it plays in computing - fascinating!)

Tuesday 4pm: Normative Ethics for Computer Programmers
Friday 3pm: Meta ethics for Computer Programmers

# Feedback please!

I value your feedback and use to pace the lectures and improve your overall learning experience. If you have any feedback from today's lecture, please follow the link below. Please remember to keep your feedback constructive, so I can action it and improve the learning experience.

https://www.menti.com/ynkk1gomx7

# WHAT DID WE LEARN TODAY?

## STRINGS (FINALLY!)

strings.c

string_functions.c

## LINKED LISTS

size_of.c

memory.c

struct_pointer.c

# REACH OUT

## CONTENT RELATED QUESTIONS

Check out the forum

## ADMIN QUESTIONS

cs1511@cse.unsw.edu.au