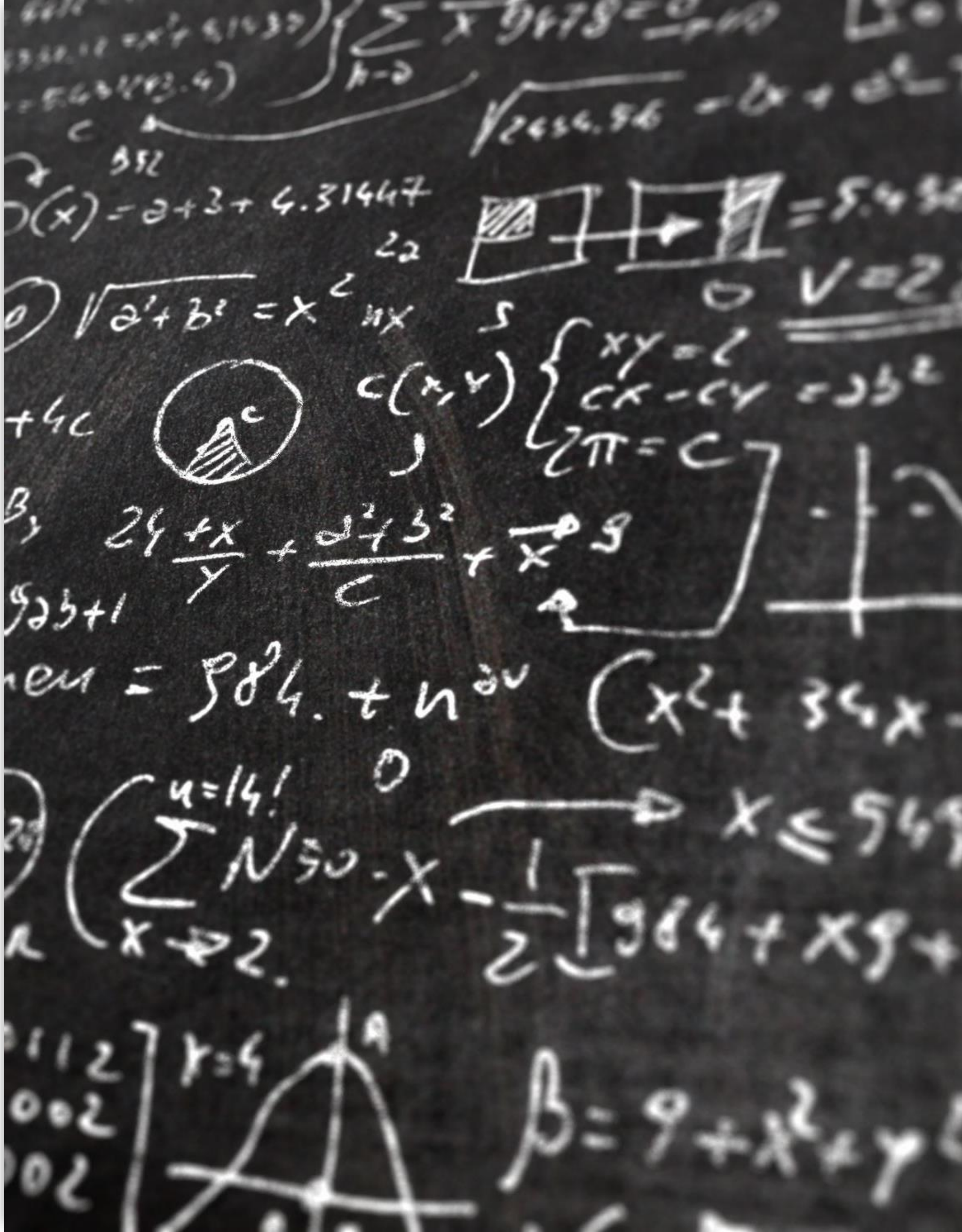
School of Computing & Communications | Lancaster University

SCC.111 Software Development – Lecture 12: Dynamic memory & Compound Types

Adrian Friday, Nigel Davies, Hansi Hettiarachchi, Saad Ezzini

This lecture

- A few final words on **strings** (consolidation)
- **Compound** variable types
- Limitless-ish possibilities! (**dynamic memory** + **compound types** + **pointers**!)



Anonymous Q&A

Join at menti.com | use code **7504 3610**

Part 1 – reprise of string functions

A string is just a sequence of characters… (an array, **notice** \0)

"hello"

‘h’	‘e’	‘l’	‘l’	‘o’	‘\0’

We can print strings using %s

```
printf("You typed %s\n", answer);
```

Assuming ‘answer’ is either a char array or a pointer to a char

```
char *answer = "hello";
char answer[] = "hello";
```

Vs. an integer – *why is this different?*

```
int answer;
scanf("%d", &answer);
```

We can get strings using %s

```
char answer[20]; // max length including \0 is 20!

scanf("%s", answer);
```

Join at menti.com | use code 4894 7111

Why do we need '&' for scanf("%d", &answer); but not scanf("%

0	0	0	0
C functions are pass by value	Arrays are effectively pointers	We need a layer of indirection	It's a mistake on the slide

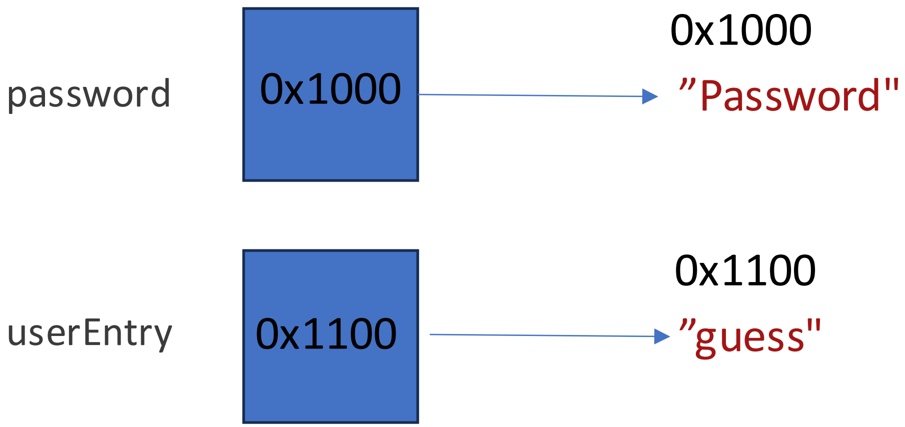
Responses are hidden

Press H to show responses

Care when comparing strings, *they're not basic types*!

- Given what we know about strings, what happens when we compare strings?

```
int main()
{
 char *password = "Password",
      *userEntry = "guess";

 if (userEntry == password)
  printf("Correct login\n");
}
```



Function signature in #include <string.h>

```
int strcmp(char*, char*)
```

strcmp()

compares two strings and returns -1, 0 or 1 (less, equal, greater)

Comparing strings *correctly*(!) with strcmp()

```
char *password = "pass123";

......

if (strcmp(password, "pass123") == 0 ||
    strcmp(password, "secret") == 0)
  printf("Yes\n");
```

In <string.h>

- int strlen(char *) // find the length of string s
- char *strchr(char *, int) // find a character within a string (pointer or NULL)
- strcat(char *d, char *s) // append string s to d
- strcpy(char *d, char *s) // copy string s to d

- *Take particular care your string is large enough, when copying and concatenating strings especially!*

Part 2 – beyond basic variables

Variable “Issues”

- Variables so far can only represent “simple” scalar values (int, char, etc.) or as fixed length arrays of values
- They are declared with a fixed size
- Variables go ‘out of scope’ at the end of the block they are declared in
- They incur overhead when copied (e.g. when we pass them into a function)

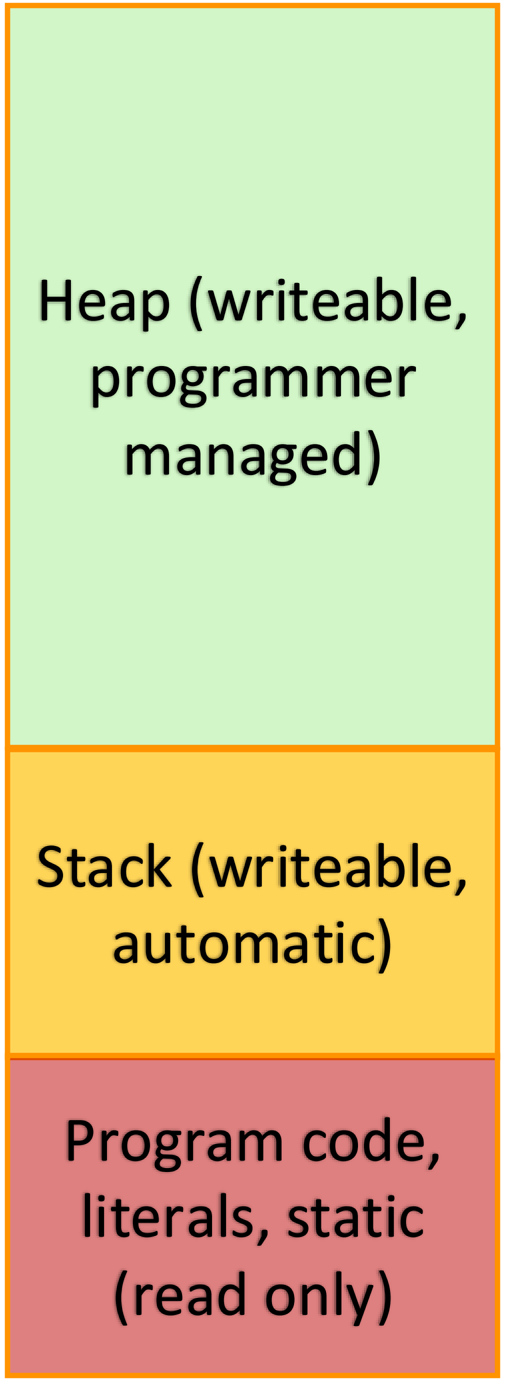
We can overcome these issues by using compound variables, ‘dynamic’ memory, and passing pointers

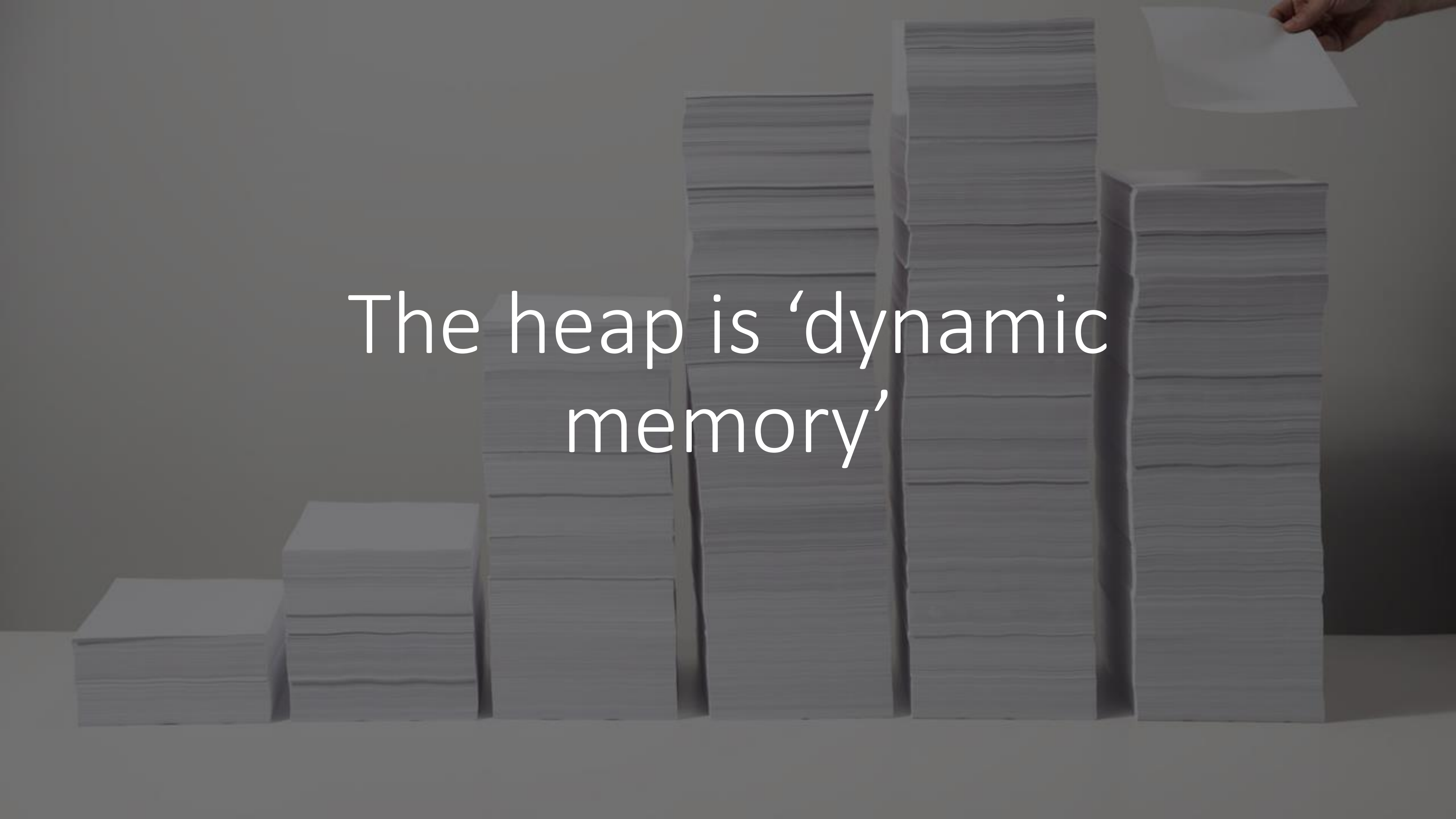
Challenge 1: addressing fixed size

Dynamic memory management

Different types of storage

- Literals, e.g. “hello, world” are compiled into our program
- Standard ‘variables’ are managed automatically by the compiler (on the stack) – they go in/out of scope
- Global variables – *yuk*!
- Dynamic variables allocated from ‘the heap’


Heap (writeable, programmer managed)
Stack (writeable, automatic)
Program code, literals, static (read only)



The heap is ‘dynamic memory’

Dynamic memory

- Allocated at runtime by your code / the programmer
- Needs *managing* (e.g. free and return to the heap when you're done)
- *Extreme care* required to avoid memory problems, memory leaks, crashing programs!

malloc() - get a pointer to some memory

```
// create pointer str to 100 bytes
char *str = (char *) malloc(sizeof(char) * 100);

// do something with it
str[0] = 'h';
strcpy(str, "hello");

// return str to the heap
free(str);
```

malloc will return ‘NULL’ if it fails (e.g. we ask for too much)

How to read what the malloc line does!

sizeof(char)	How big is a char?
* 100	Multiply by the size we want
malloc(100)	Allocate that heap space (return a pointer)
char *str	Declare pointer ’str’
= (char *)	‘cast’ the pointer to be of the type ‘points to char’

Challenge 2: more than just scalar values

Compound types combine simple types into ‘an entity’ (closer to our problem)

- The question & its answer… are always a pair…
- A map location (latitude + longitude)… do we want 2 separate variables, or an array of **latitudes** and an array of **longitudes** that we keep in sync?
- An image, its dimensions and size… *be great if we could bundle these together!*

The ‘struct’

- A struct is a user defined grouping of variables (possibly even other nested structs):

```
/* Declare a new type (not variable, struct person) */

struct person {
  char name[20]; // array of chars (string name)
  int age;
  char gender; // gender will be a single letter, e.g. 'f'
};

/* Declare the variable of type struct person */

struct person aPerson;
```

- The parts of the structs are accessed with special ‘.’ dot notation:

```
struct person aPerson;
```

```
strcpy(aPerson.name, "Nigel");
aPerson.age = 30;
aPerson.gender = 'm';
```

aPerson

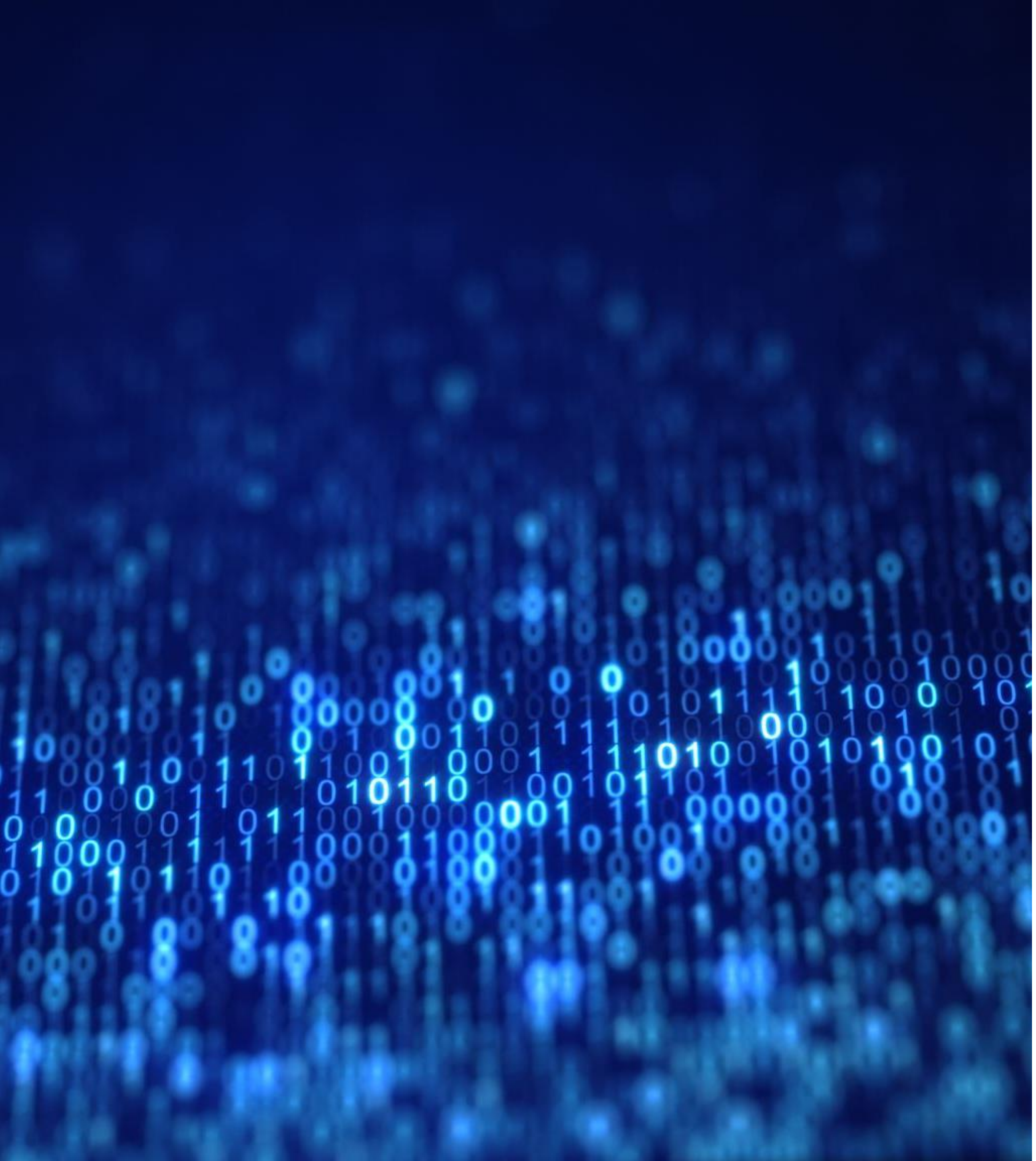
name	char array	Nigel
age	int	30
gender	char	‘m’

An array of structs

```
struct person *people; // A pointer to a ‘type struct person’

// Now try and allocate some memory (an array of struct persons)

if ((people = (struct person *)
              malloc(sizeof(struct person) * 100)) != NULL) {
  // it worked, do something with 100 people
  // free when done

  people[50].age = 18; // age field in 50th person in people array

  free(people);
}
else {
  // oh no, out of memory!
}
```

Access a struct field from a pointer…

```
struct person *p = &aPerson;

strcpy(p->name, "Nigel");
p->age = 30;
p->gender = 'm’;

printf("%s's age is %d\n", p->name, p->age);
```



Summary

- How to **compare** and **print** strings
- **Structs**/ compound variable types
- Dynamic memory & how to allocate on the heap (**malloc/free**)
- *Next lecture: even more powerful uses of pointers!*