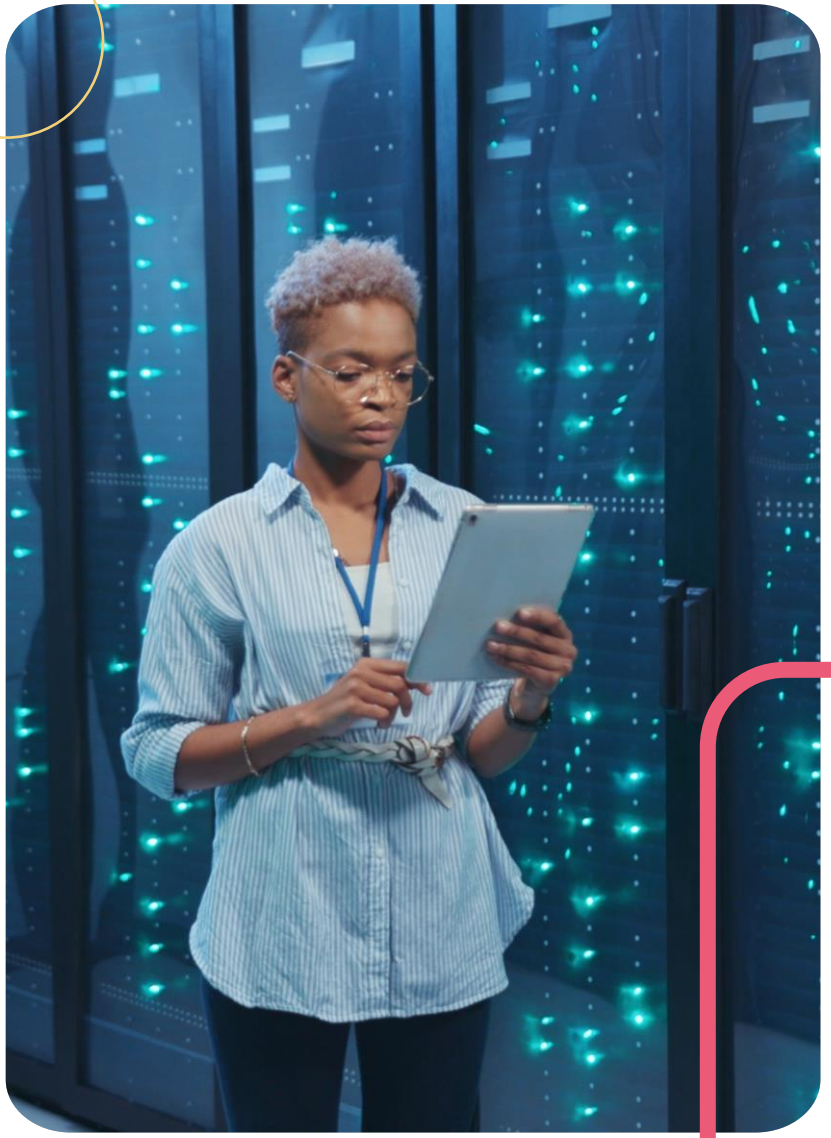
Diploma in

Computer Science

Pointers



Objectives

- Understand the concept of pointers as used in Computer Science
- Apply the concept of arithmetic operators to pointers
- Recognise the role of pointers in making software more efficient
- Appreciate the idea of linked lists as a data structure that uses pointers

The science of pointers

Understanding pointers

They are closely related to how microprocessors work.

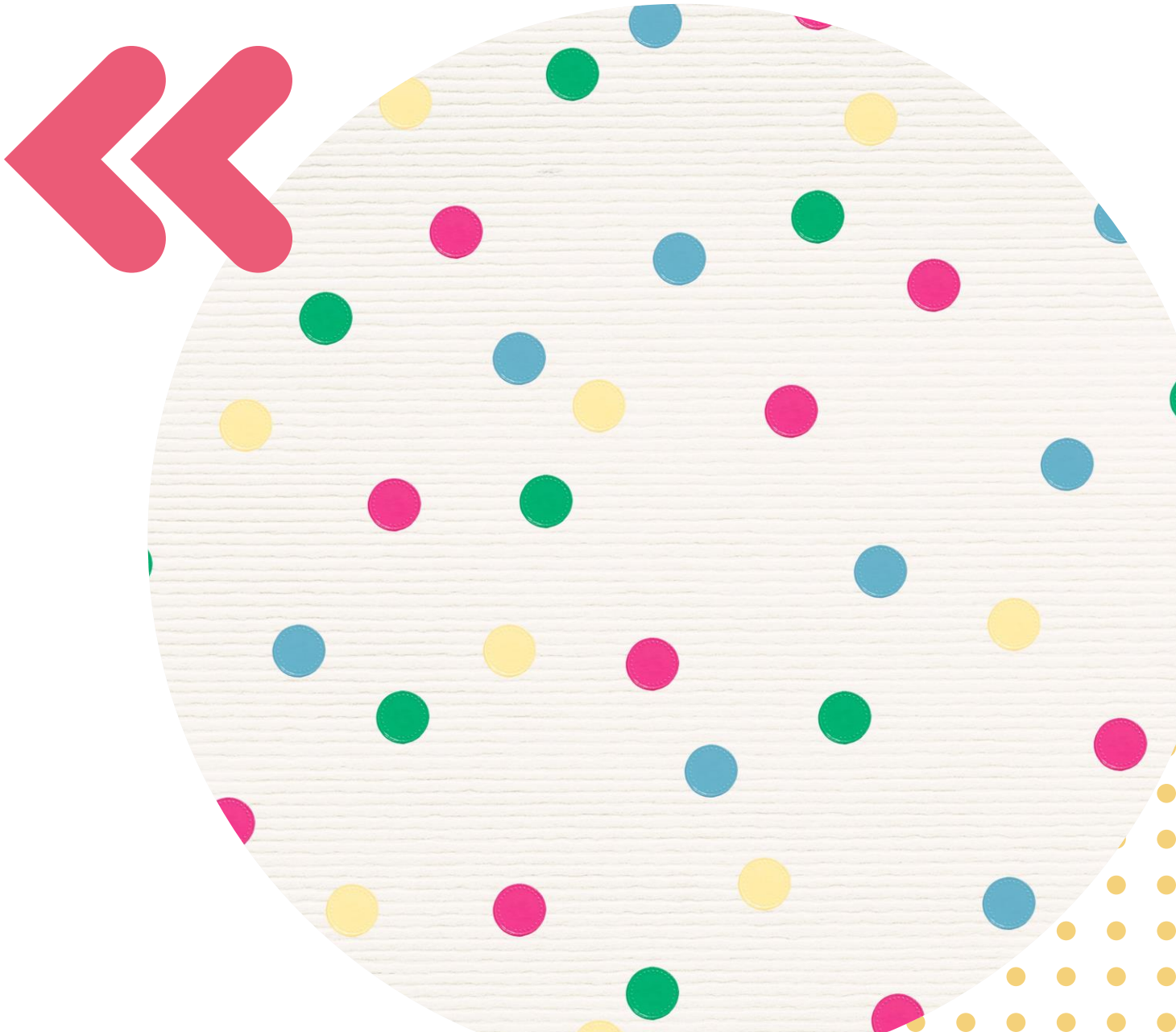
They make operating system kernels, embedded systems and device drivers efficient.

A pointer is a like a road sign.

A bit of theory...

A pointer is a reference to a data primitive.

- A **data primitive** is a piece of data that can be written or read from computer memory using an address.
- A **data aggregate** is a group of data primitives in a logical manner that can be viewed collectively.
- An **array** is a group of several data primitives of the same type.



A variable is a storage location

- Variable type: dictates the size and kind of data
- Variable name: to call and identify it
- Variable address: its physical or logical location
- We use the base address to calculate the span of the variable.

Declaring variables

```
long long int age=9;
```

Example

```
long long int age=9;
printf("%d", &age);
```

Example

Why do we need pointers?

- Useful with more complex data structures
- Used to store and manage arrays and data objects
- Reduce the system footprint of a program
- Allow modifications to a variable
- Help to save memory
- Allow you to return more than one value from a function

Pointer operations

Syntax

```
data type *pointer_name=&variable
```

Example

Declared with * before name AND value needs to be an address

```
#include<stdio.h>
int main(){
int age=9;
int *pointerToAge;
pointerToAge=&age;
printf("%d",pointerToAge);
printf("%d",pointerToAge);
}
```

Example

From then on, don't use *

```
Int *ptr_name;/* this is a pointer to an integer */
int *ptr1,name;/* ptr1 is a pointer to type integer and name is an integer
variable */
double *ptr2; /* pointer to a double */
float *ptr3; /* pointer to a float */
char *ch1; /* pointer to a character */
float *ptr, variable;/*ptr is a pointer to type float and variable is an ordinary
float variable */
```

Example

Pointers can be categorised in many ways based on what they can do.

Pointers as arguments

- Used as function parameters – known as pass-by-reference
- No copy of actual value is made but a reference to the memory location of the value is made
- Reserve for functions that need to modify the value
- Can also return pointers to the calling function

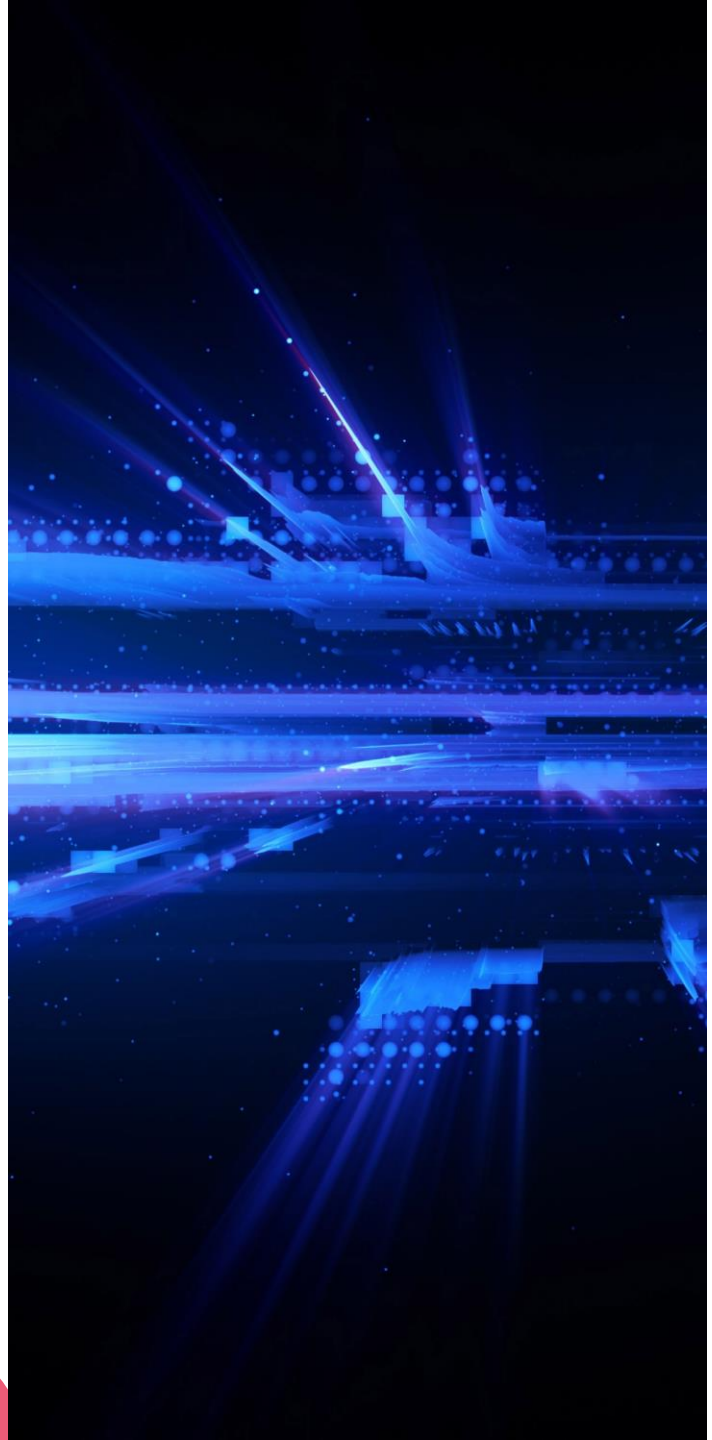
Function pointers

- Points to functions by storing the address of a function and calling it

```
void (*funtion_pointer)(data_type) =
&function_name;
```

Example

- Address of the function and returned data type is referenced in the pointer
- Can be used in place of a switch statement
- Can be passed as an argument and returned from a function to avoid code redundancy

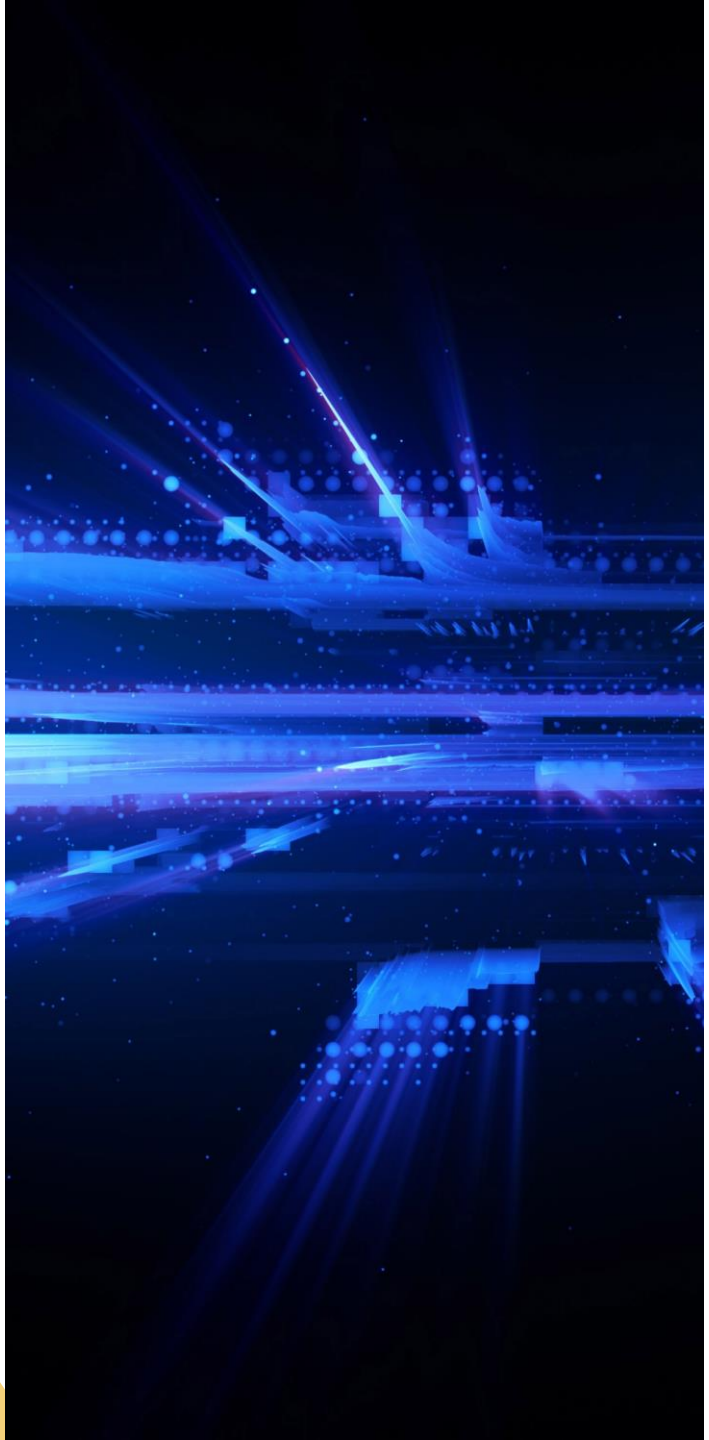


Null pointers

- The pointer will carry a zero value until a value is assigned to it.

```
int *pointerToAge=NULL;
```

Example



Void pointers

- Known as a generic pointer that does not have any data type
- Can store the address of any variable
- Cannot be dereferenced
- Cannot be used to perform pointer arithmetic (except GNU C)

Pointers, being variables, can also be used in arithmetic operations.

Pointer arithmetic

- Pointers store addresses.
- Pointers are like house numbers on a street.
- Pointers can only perform subtraction.
- The difference between two pointers gives you the number of elements of the type that can be stored between two pointers but spending them doesn't give you any meaningful functionality.

Pointer arithmetic

- Increment and decrement
- Addition or subtraction of an integer
- Subtracting one pointer from another
- Comparison of two pointers
- Assignment

Increment operator (++)

Increases the value of a pointer by the size of the data object to which the pointer points

```
pointer++;
```

Example

Decrement operator (--)

Decreases the value of the pointer

```
pointer--;
```

Example

Example illustrating that it is first multiplied by the size of the data type:

```
p++;
P--;
```

Example

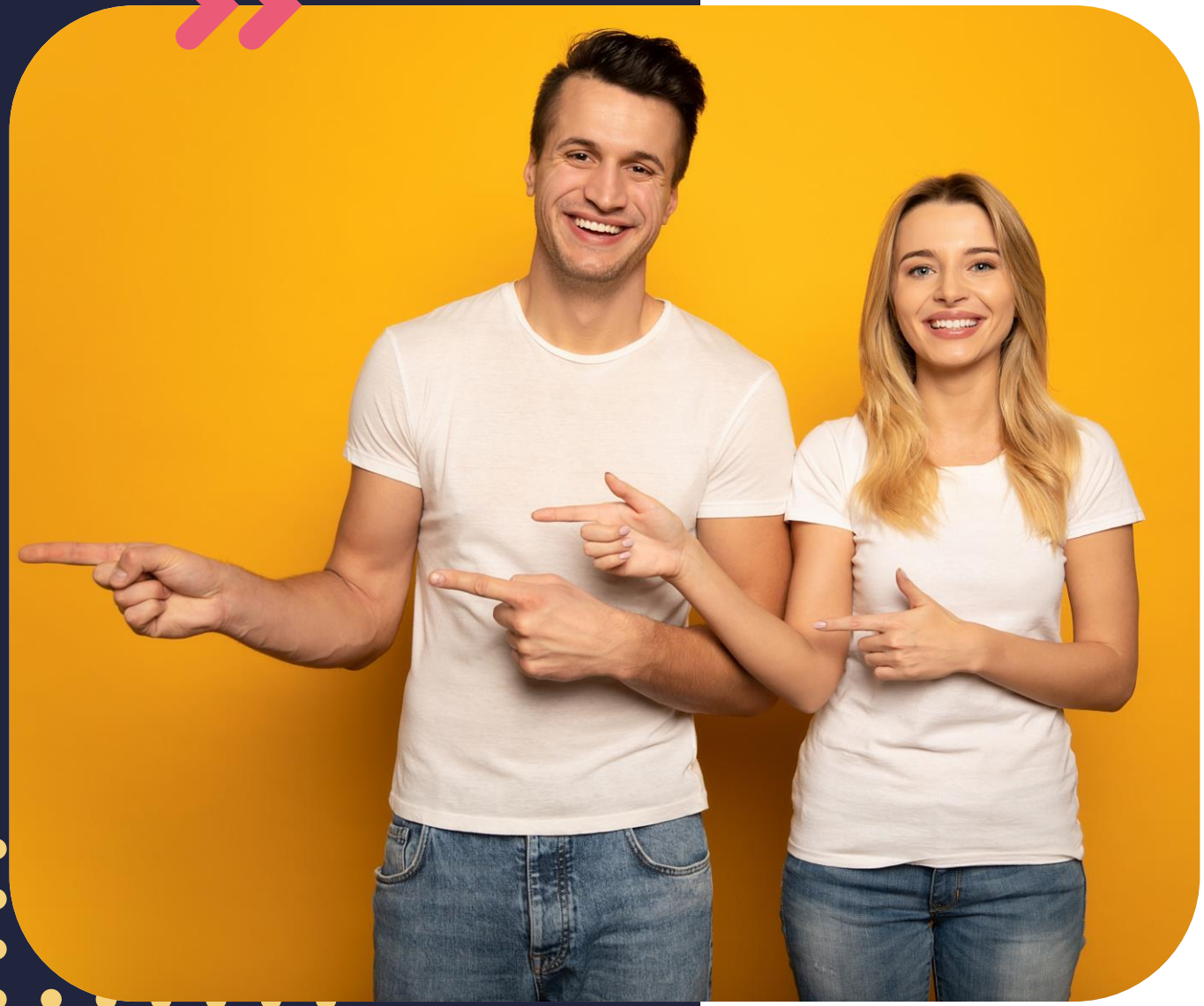
Addition and subtraction operators

```
p+4;
P-2;
```

Example

Add 4 units to p: address will be 12361

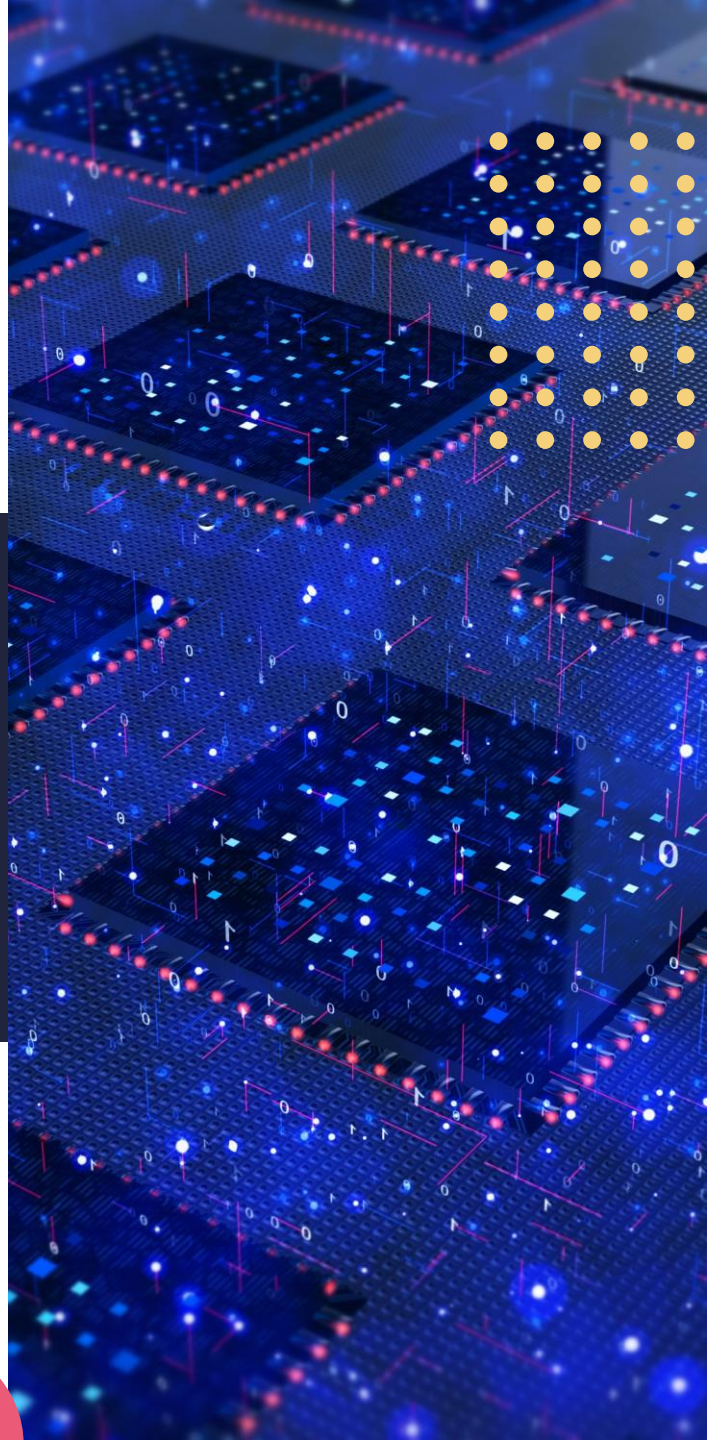
Subtract 2 units: address will be 12353



Comparison of two pointers

- If two pointers are pointing to the same array they can be compared.
- Achieved using all relational operators.
- Pointers can only be compared if they are contents of the same object.
- Data that has been grouped and is part of the same data aggregate.

Assignment refers to transferring an address into a pointer.



Pointers vs arrays

Arrays work hand in hand with pointers to achieve faster execution speeds and a smaller system footprint

```
int *p;
p = part_number;
// is the same as
p = &part_number[0]
```

Example

When you call an array without specifying an index, it will always refer to the base index

Hurts the readability of your code

Can create undefined behaviour

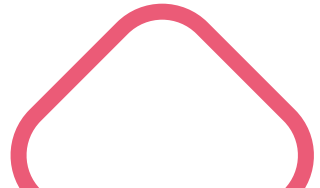


Diagram of an array

		0X856435
P=&part_number□	int part_number[0]	0X856436
	…	…
p+=14□	int part_number[14]	0X856450
	int part_number[15]	0X856451
	…	…
	int part_number[19]	0X856455
		0X856456
		0X856457
		0X856458
		0X856459

Rules for pre-increment operators

Syntax	Operation	Example

p++
*p++
*(p++)
	Post-increment pointer	new = *(p++);
		is the same as
		new = *p;
		p = p + 1;

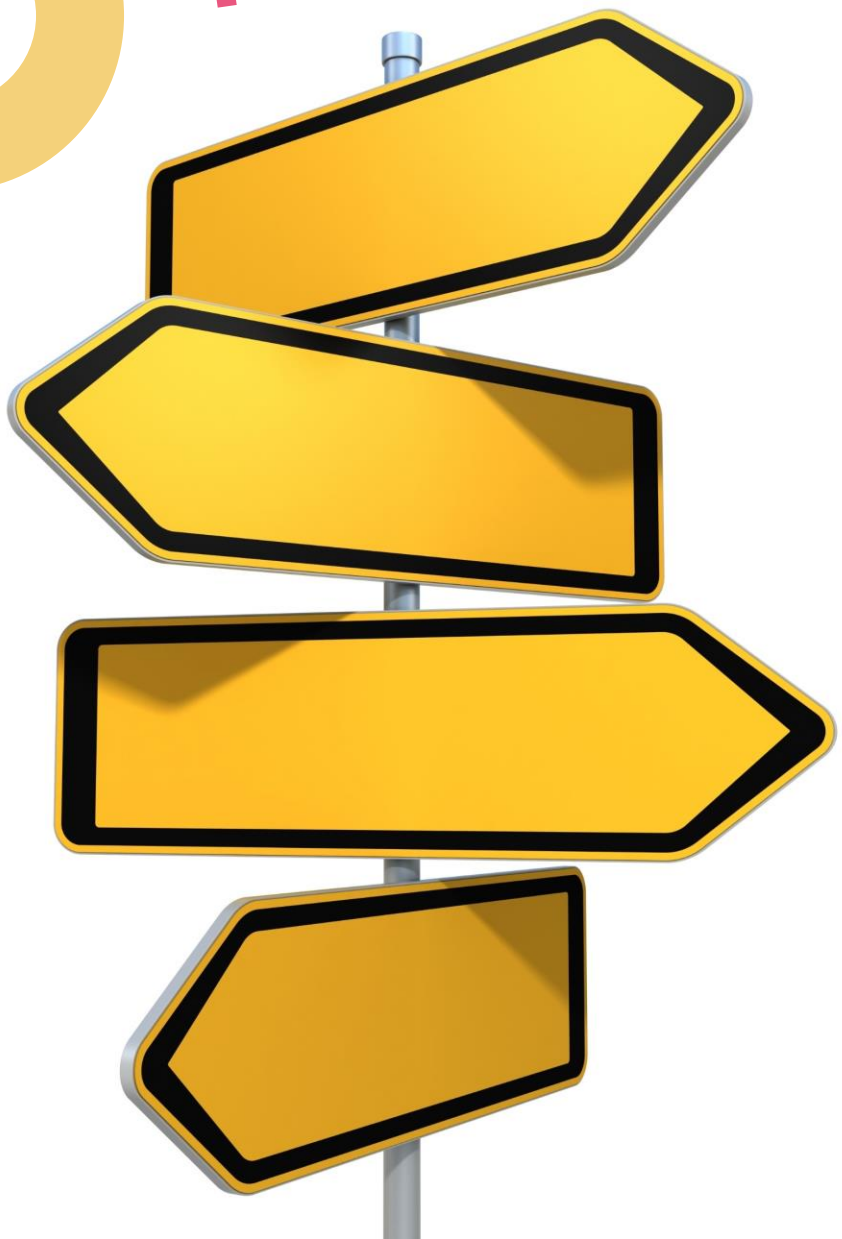
(*p)++	Post-increment data pointed to by pointer	new = (*p)++;
		is the same as
		new = *p;
		*p = *p + 1;

Rules for pre-increment operators

Syntax	Operation	Example

++p
++*p
*(++p)
	Pre-increment pointer	new = *(++p);
		is the same as
		p = p + 1;
		new = *p;

++(*p)	Pre-increment
	data pointed to
	by pointer
		new = ++(*p);
		is the same as
		*p = *p + 1;
		new = *p;



To modify the pointer…

```
pre-inc: *(++p) or *++p or ++p
post-inc: *(p++) or *p++ or p++
```

To modify the value the pointer points to…

```
++(*p) and (*p)++
```

Example

Pointer to pointer (double pointers)

The first pointer is used to store the address of a variable.

The second pointer is used to store the address of the first pointer.

```
int a = 5;
int *ptr = &a; //ptr references a
int **dptr = &ptr; //dptr references ptr
```

Example

The double pointer has a double asterisk, which immediately tells it apart from a usual pointer.

Pointer to pointer uses

To create two-dimensional arrays

As function arguments to modify the pointer

To interface with APIs form other programs

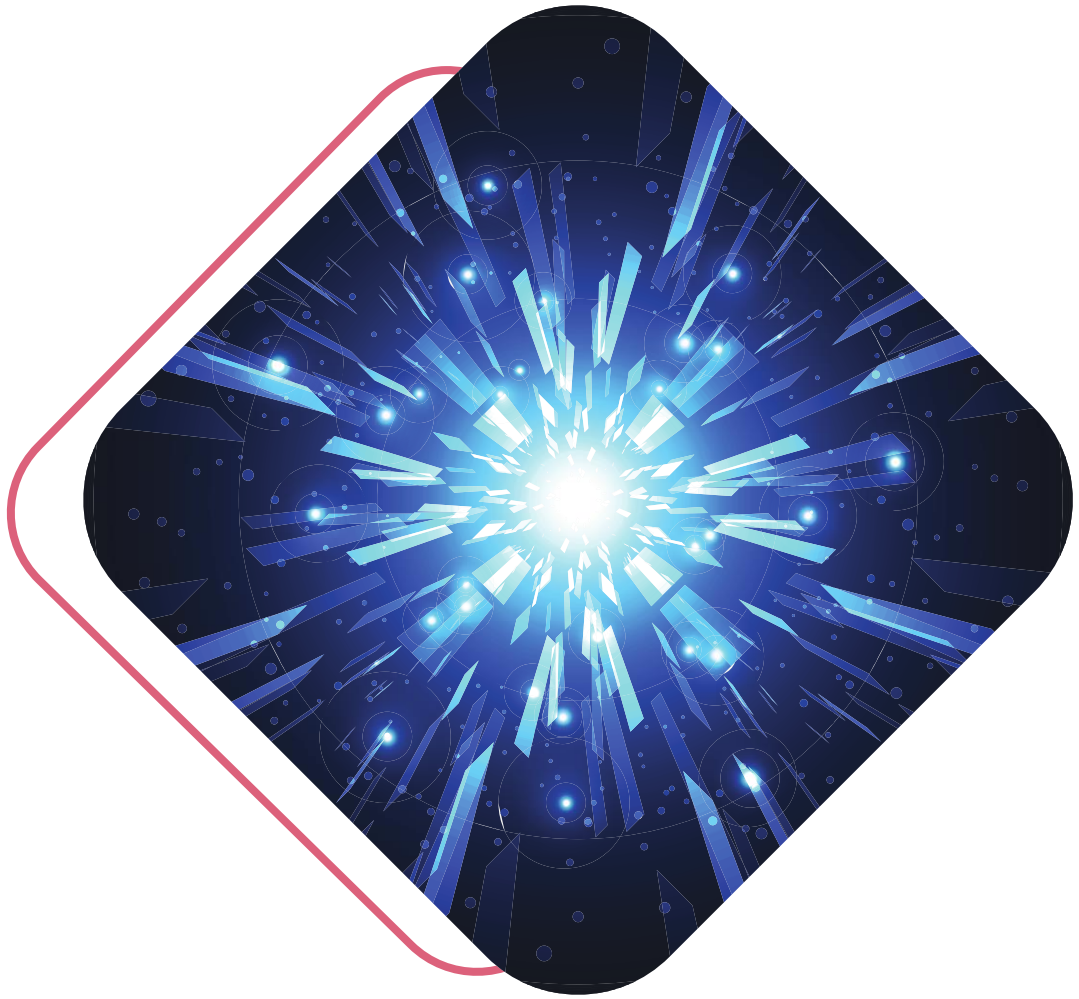
Linked lists are the simplest dynamic data structure that use pointers.

Linked lists

- Similar to as both linear data structures
- Not necessarily contiguous in memory
- Data elements connected by pointers

Linked lists

- Have dynamic size
- Do not allow random element access
- Not suitable for caching
- Extra memory space for the pointer is required for each element in the list



Linked lists

- Represented by a pointer that points to the first node in the list
- Node is made up of two parts: the data element then a pointer

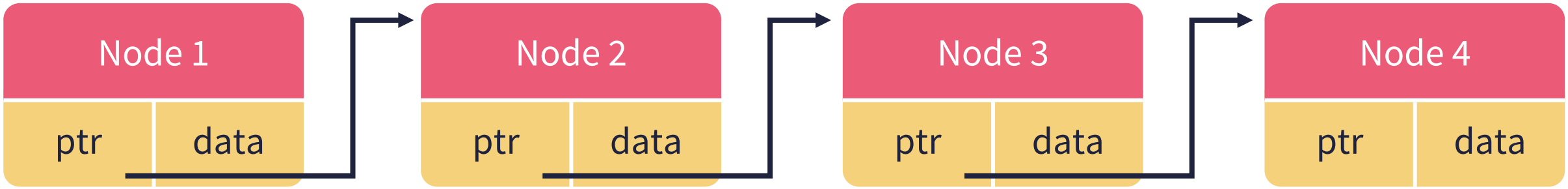
Linked lists

- Define a linked list in C with a structure
- Structures allow you to represent several data items of different types
- Basic syntax:

```
struct [structure name {
    member;
    member;
    ...
    member;
} [structure variables];
```

Example

A linked list in memory...



Things to note when using pointers...

A pointer can become a dangling point when …

It points at memory that was de-allocated

A variable goes out of scope

Challenge

Create a programme that traverses an array using a pointer.