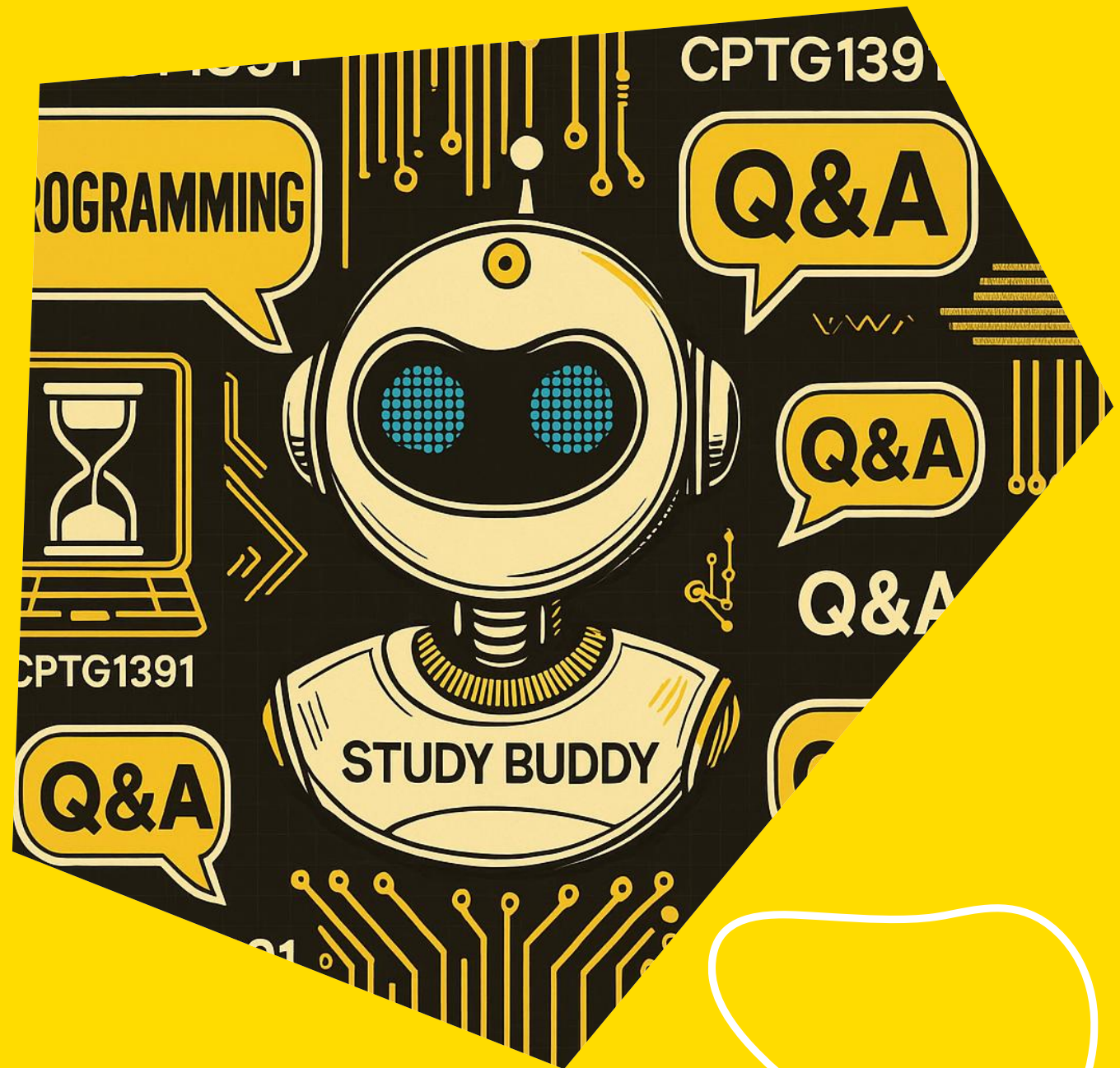


DPST1091 / CPTG1391
Introduction to Programming
Week 8 – Lecture 1

Lecturer and Course Convener:

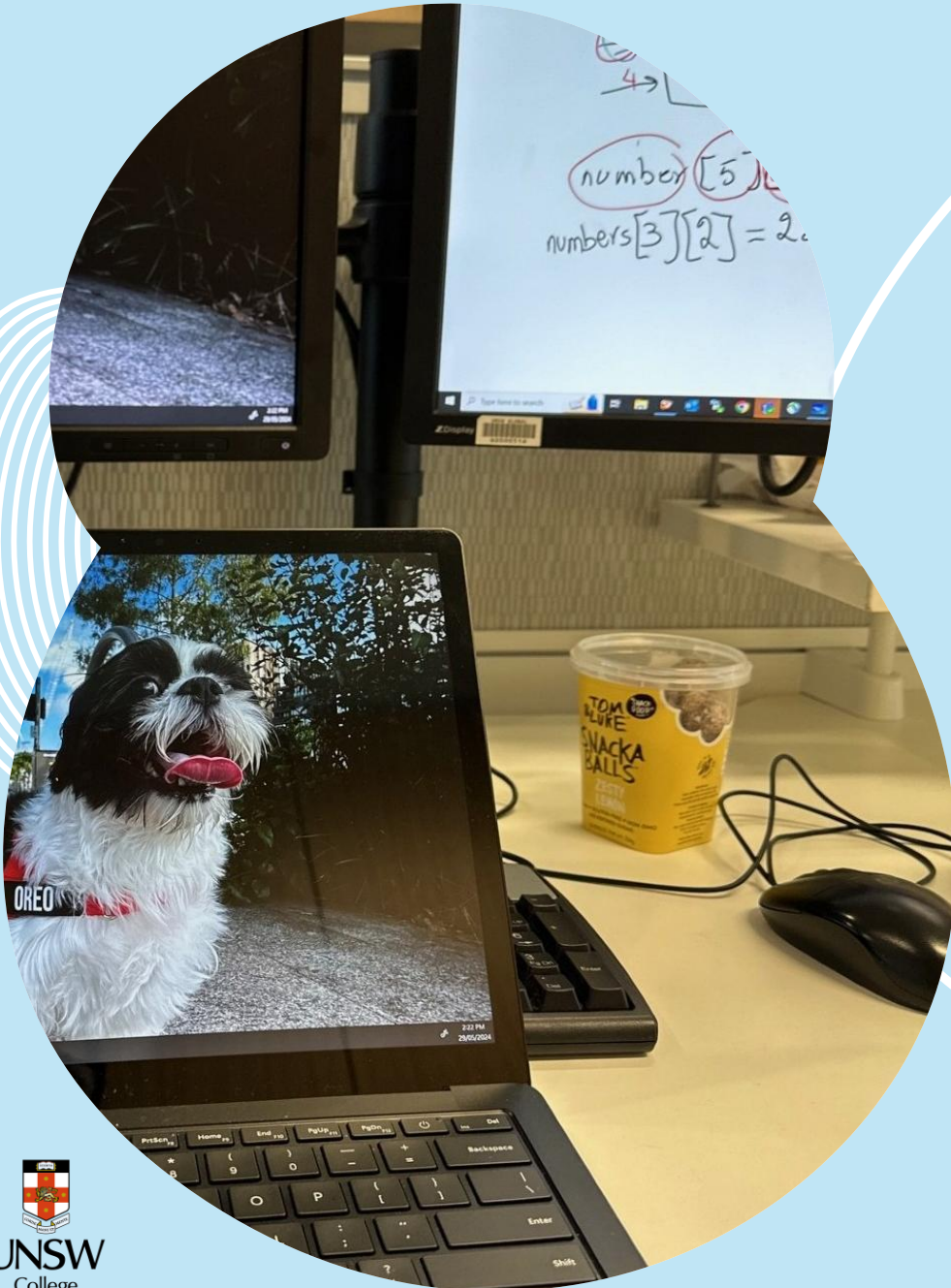
Dr Pantea Aria

Dynamic Memory, malloc and the heap



How was your flex week?

Assignment is due this Friday 9am



Agenda

- **Today**
 - **Dynamic Memory, malloc and the heap**

Link to the Live Lecture Code

<https://cgi.cse.unsw.edu.au/~dp1091/26T2/code/08>

Live lecture code is written for teaching, not perfection.
It may include extra comments and may not always follow
ideal coding style.



Pointers

Recap

Pointers are **variables** that **store the addresses** of other variables in memory.

When **declaring a pointer**, you **specify the data type it points to** and use an asterisk (*) to indicate that it is a pointer.

```
type_pointing_to *name_of_variable;
```

```
int *number_ptr;  
double *real_ptr;  
char *my_ptr;  
struct person *student_ptr;
```

Initialising a Pointer

```
int number = 2;

// number_ptr is declared
// and initialised and
// contains the address
// of int variable number
int *number_ptr = &number;
```

```
double value = 3.14;

// real_ptr declared
double *real_ptr;

// real_ptr is initialised
// and contains the
// address of double
// variable value
real_ptr = &value;
```

Visualise it in the memory

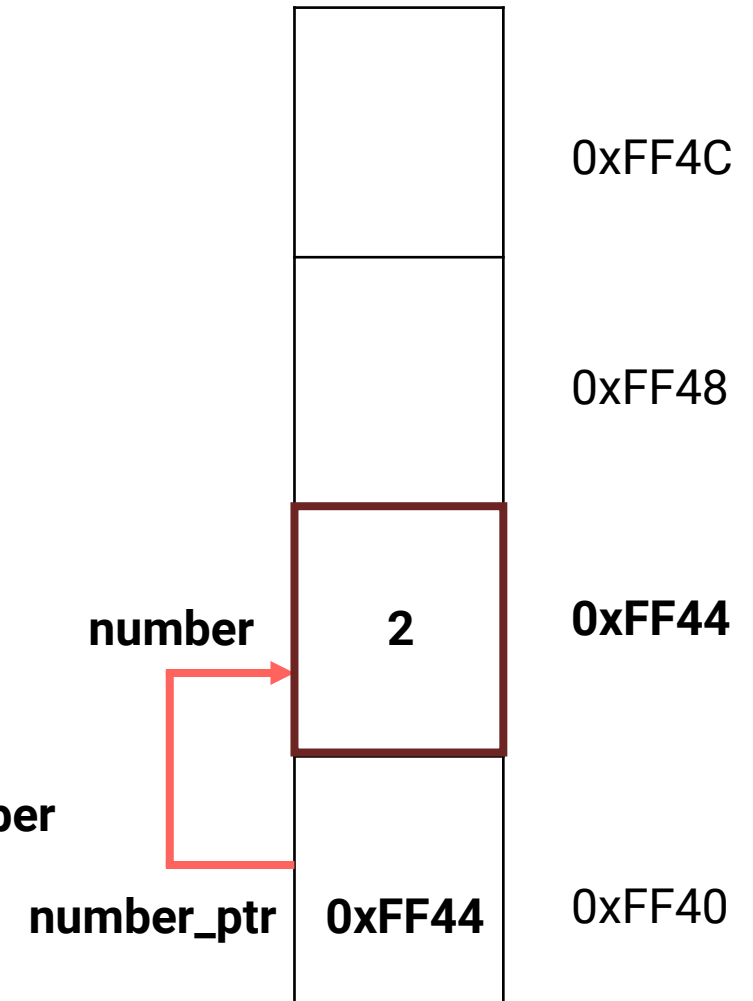
Referencing

```
int number = 2;  
  
// number_ptr is declared  
// and initialised and  
// contains the address  
// of int variable number  
int *number_ptr = &number;
```

So now: **number** is 2
AND
number_ptr is 0xFF44

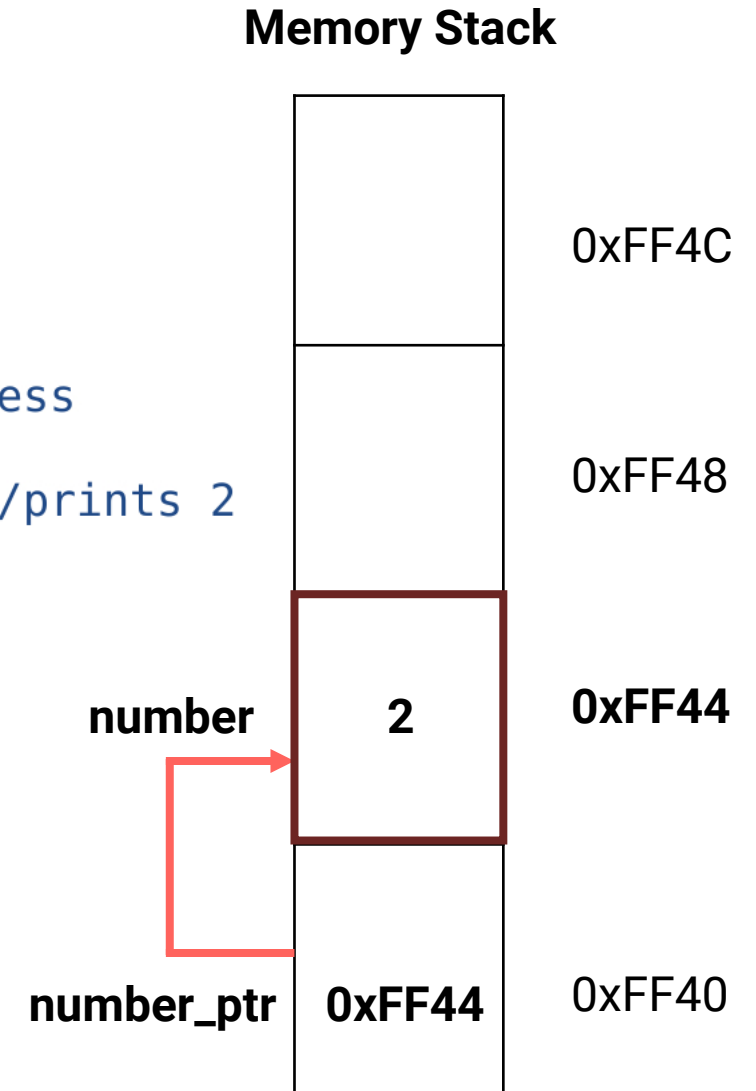
We can say **number_ptr** **references** **number**
or
number_ptr **points** to **number**

Memory Stack



Dereferencing

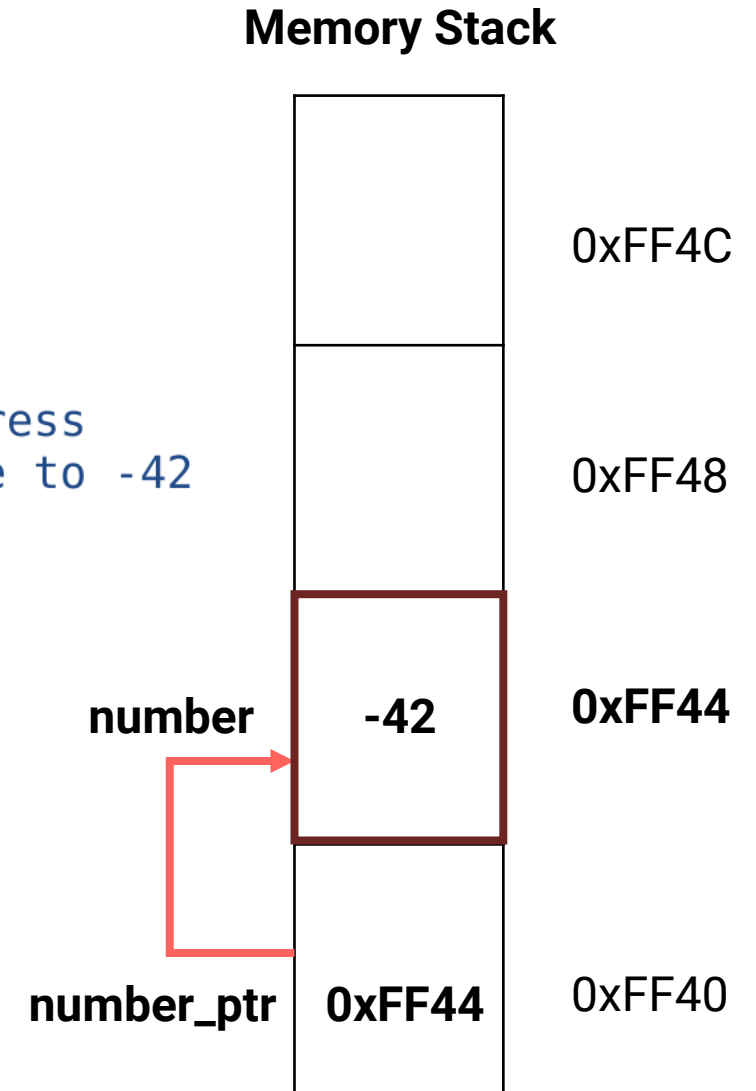
```
int number = 2;  
int *number_ptr = &number;  
  
// *number_ptr will go to address  
// 0xFF44 and get the value 2  
printf("%d\n", *number_ptr); //prints 2
```



Indirectly modify a variable using pointers

```
int number = 2;  
int *number_ptr = &number;
```

```
// *number_ptr will go to address  
// 0xFF44 and change the value to -42  
// number is now -42  
*number_ptr = -42;
```



What was a pointer again??

Step	Description	Example
Declare a pointer	Use * to declare a pointer and specify the type of value it will point to	int *number_ptr;
Initialise the pointer	Assign the address of a variable to the pointer using the & operator	number_ptr = &x;
Declare and initialise	Declare the pointer and assign it an address in a single statement	int *number_ptr = &x;
Dereference the pointer	Use * to access the value stored at the address the pointer points to	*number_ptr

```
int score = 42;
```

```
// Declare a pointer  
int *score_ptr;
```

```
// Initialise pointer  
score_ptr = &score;
```

```
// Dereference pointer to get  
// 42, then add 1 so result is 43  
int next_score = *score_ptr + 1;
```

So, what is the point of pointers???

What is the output of this program?

```
#include <stdio.h>

void increment(int x, int y);

int main(void) {
    int x = 3;
    int y = 7;

    printf("Before increment: %d %d\n", x, y);

    increment(x, y);

    printf("After increment:  %d %d\n", x, y);

    return 0;
}

void increment(int x, int y) {
    x = x + 1;
    y = y + 1;

    printf("Inside increment: %d %d\n", x, y);
}
```

Stack Memory



Stack memory stores information needed for your program to run, particularly details about **function calls**.

Each time a function is called, a new block of data is **pushed onto the stack**, including: the **function's local variables** and information about **where the program should return** once the function finishes

When the function (or code block { }) finishes executing, its data, including variables, is **automatically removed from the stack**.

Stack memory is allocated **at run time**, and is managed **automatically by the program**, not directly by the developer.

Functions And call by value

Variables and data are **passed by value** into functions (note: **arrays** are a special case we will discuss separately)

The function gets **passed copies** of the values

We **can't change the original values** from inside the function

The modified **copies don't even exist** once the **function ends**

Call by Reference

We can pass the **addresses of variables** into functions, just like we do with `scanf`, allowing functions to **modify their values**.

When a function **receives an address**, it can access the corresponding memory location and **directly update the original variable**.

Although the function receives a **copy of the address**, that address still refers to the **same memory location**.

This gives us a way for functions to **modify local variables from the calling function**, even when those variables are **not arrays**.

Example

Making function increment work using pointers

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

```
void increment(int *x, int *y);
```

```
int main(void) {  
    int x = 3;  
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(&x, &y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
}
```

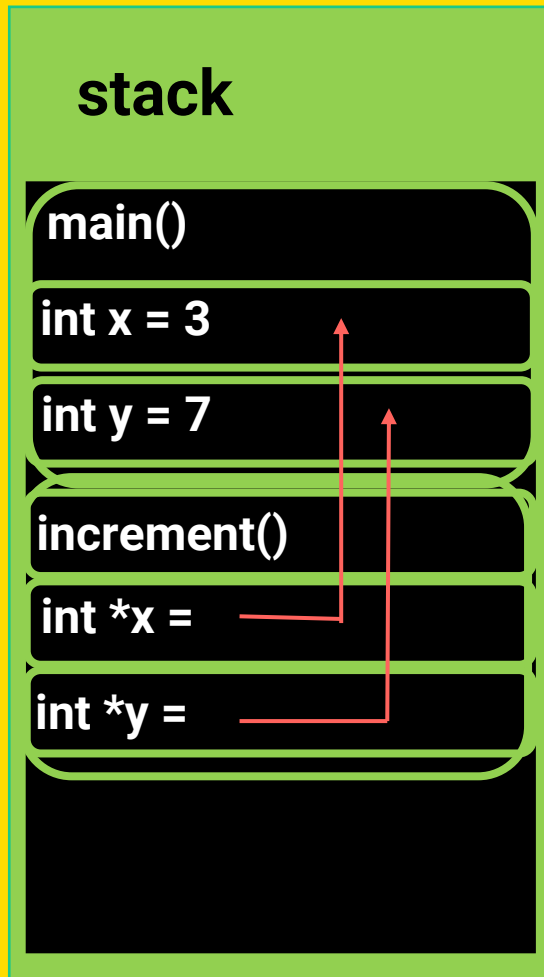
```
void increment(int *x, int *y) {  
    *x = *x + 1;  
    *y = *y + 1;
```

```
    printf("Inside increment: %d %d\n", *x, *y);
```

```
}
```

The **main** function must pass the **addresses** of `x` and `y` to the function.

The function must be defined with **pointer parameters**, because pointers are used to store memory addresses.



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

```
void increment(int *x, int *y);
```

```
int main(void) {  
    int x = 3;  
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(&x, &y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
}
```

```
void increment(int *x, int *y) {
```

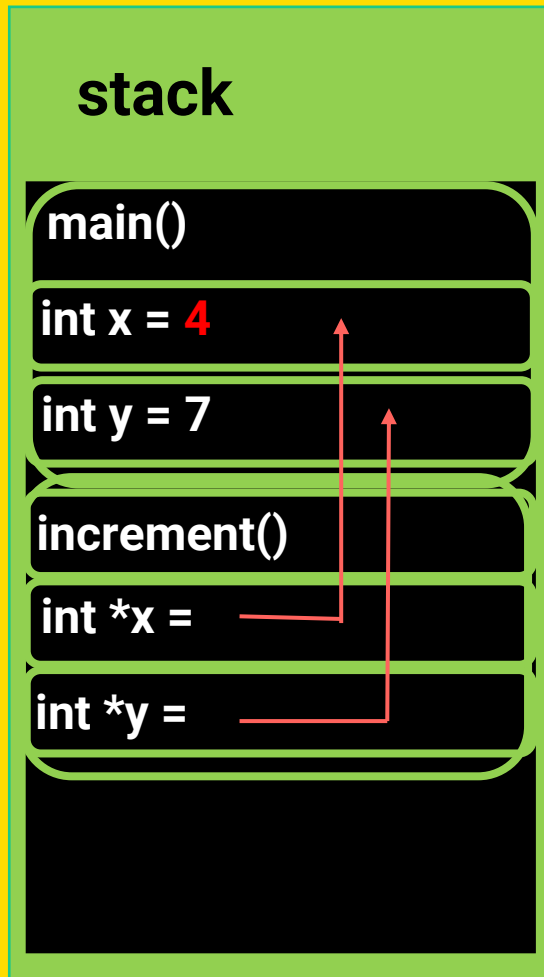
```
    *x = *x + 1;
```

```
    *y = *y + 1;
```

```
    printf("Inside increment: %d %d\n", *x, *y);
```

```
}
```

Output: Before increment: 3 7



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

```
void increment(int *x, int *y);
```

```
int main(void) {  
    int x = 3;  
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(&x, &y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
}
```

```
void increment(int *x, int *y) {
```

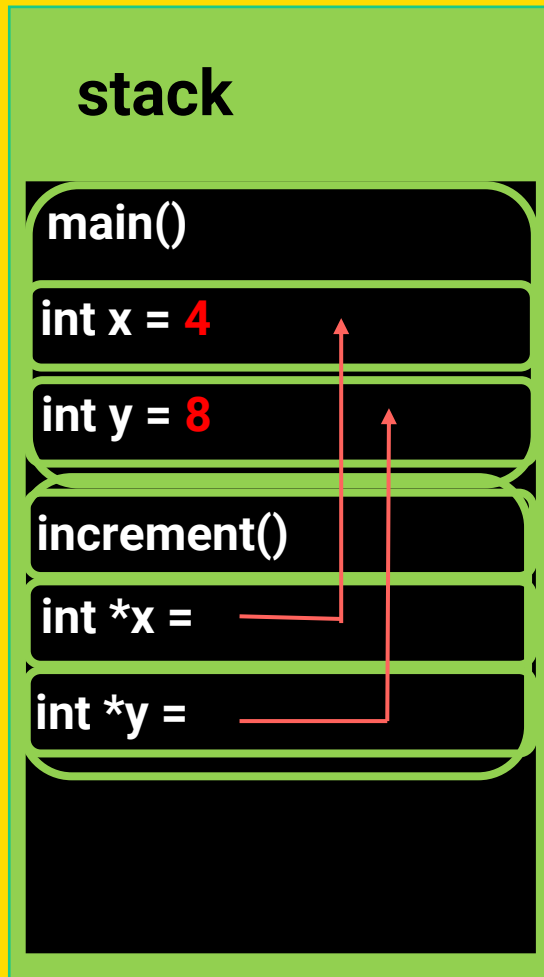
```
    *x = *x + 1;
```

```
    *y = *y + 1;
```

```
    printf("Inside increment: %d %d\n", *x, *y);
```

```
}
```

Output: Before increment: 3 7



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

```
void increment(int *x, int *y);
```

```
int main(void) {  
    int x = 3;  
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(&x, &y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

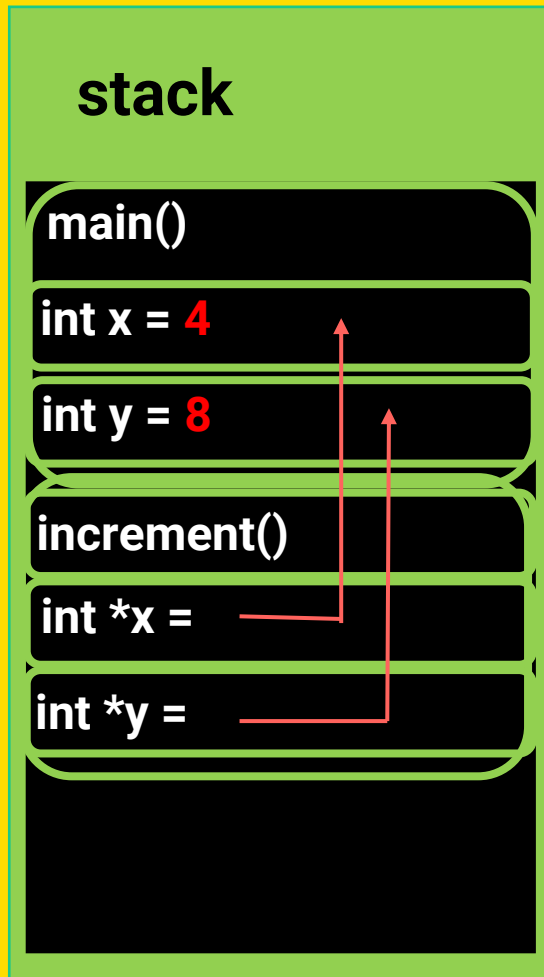
```
}
```

```
void increment(int *x, int *y) {  
    *x = *x + 1;  
    *y = *y + 1;
```

```
    printf("Inside increment: %d %d\n", *x, *y);
```

```
}
```

Output: Before increment: 3 7



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

```
void increment(int *x, int *y);
```

```
int main(void) {  
    int x = 3;  
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(&x, &y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
}
```

```
void increment(int *x, int *y) {  
    *x = *x + 1;  
    *y = *y + 1;
```

```
    printf("Inside increment: %d %d\n", *x, *y);
```

```
}
```

Output: Before increment: 3 7
Output: Inside increment: 4 8

stack

main()

int x = 4

int y = 8

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

```
void increment(int *x, int *y);
```

```
int main(void) {
```

```
    int x = 3;
```

```
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(&x, &y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
}
```

```
void increment(int *x, int *y) {
```

```
    *x = *x + 1;
```

```
    *y = *y + 1;
```

```
    printf("Inside increment: %d %d\n", *x, *y);
```

```
}
```

Output: Before increment: 3 7

Output: Inside increment: 4 8

Output: After increment: 4 8

stack

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

```
void increment(int *x, int *y);
```

```
int main(void) {
```

```
    int x = 3;
```

```
    int y = 7;
```

```
    printf("Before increment: %d %d\n", x, y);
```

```
    increment(&x, &y);
```

```
    printf("After increment:  %d %d\n", x, y);
```

```
    return 0;
```

```
void increment(int *x, int *y) {
```

```
    *x = *x + 1;
```

```
    *y = *y + 1;
```

```
    printf("Inside increment: %d %d\n", *x, *y);
```

```
}
```

Output: Before increment: 3 7

Output: Inside increment: 4 8

Output: After increment: 4 8

Example: Let's modify function swap together

```
#include <stdio.h>

void swap(int x, int y);

int main(void) {
    int x = 3;
    int y = 7;

    printf("Before swap: %d %d\n", x, y);

    swap(x, y);

    printf("After swap:  %d %d\n", x, y);

    return 0;
}

void swap(int x, int y) {
    int temp = x;
    x = y;
    y = temp;

    printf("Inside swap: %d %d\n", x, y);
}
```

Pointers to structs

When working with a **structure variable**, we use the **dot operator** (.) to access its members.

```
#include <stdio.h>

struct point {
    int x;
    int y;
};

int main(void) {
    struct point location;

    location.x = 5;
    location.y = 12;

    printf("Point coordinates: (%d, %d)\n", location.x, location.y);

    return 0;
}
```

When using a pointer to a struct, accessing members with the dot operator requires extra parentheses

```
#include <stdio.h>

struct point {
    int x;
    int y;
};

int main(void) {
    struct point location;
    struct point *location_ptr = &location;

    (*location_ptr).x = 10;
    (*location_ptr).y = 9;

    printf("Location: (%d, %d)\n",
           (*location_ptr).x, (*location_ptr).y);

    return 0;
}
```

which can make the code harder to read

Use -> when accessing struct members through a pointer instead of dot operator

```
#include <stdio.h>

struct point {
    int x;
    int y;
};

int main(void) {
    struct point location;
    struct point *location_ptr = &location;

    location_ptr->x = 10;
    location_ptr->y = 9;

    printf("Location: (%d, %d)\n",
           location_ptr->x, location_ptr->y);

    return 0;
}
```

Example: What is the result of this program??

How can we fix it?

```
#include <stdio.h>

struct point {
    int x;
    int y;
};

void increment(struct point location);

int main(void) {
    struct point location;

    location.x = 10;
    location.y = 9;

    printf("Before increment: (%d, %d)\n", location.x, location.y);

    increment(location);

    // These will stay the same
    printf("After increment:  (%d, %d)\n", location.x, location.y);

    return 0;
}

void increment(struct point location) {
    location.x = location.x + 1;
    location.y = location.y + 1;

    printf("Inside increment: (%d, %d)\n", location.x, location.y);
}
```

Can I not use a pointer??

In this case Yes!

You can return the updated location

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

```
struct point {  
    int x;  
    int y;  
};
```

```
struct point increment(struct point location);
```

```
int main(void) {  
    struct point location;
```

```
    location.x = 10;  
    location.y = 9;
```

```
    printf("Before increment: (%d, %d)\n", location.x, location.y);
```

```
    location = increment(location);
```

```
    printf("After increment: (%d, %d)\n", location.x, location.y);
```

```
    return 0;
```

```
}  
struct point increment(struct point location) {  
    location.x = location.x + 1;  
    location.y = location.y + 1;
```

```
    return location;  
}
```

Or pass the address of location to the function

Passing the address of a struct allows a function to modify the original struct without returning it.

```
#include <stdio.h>

struct point {
    int x;
    int y;
};

void increment(struct point *location);

int main(void) {
    struct point location;

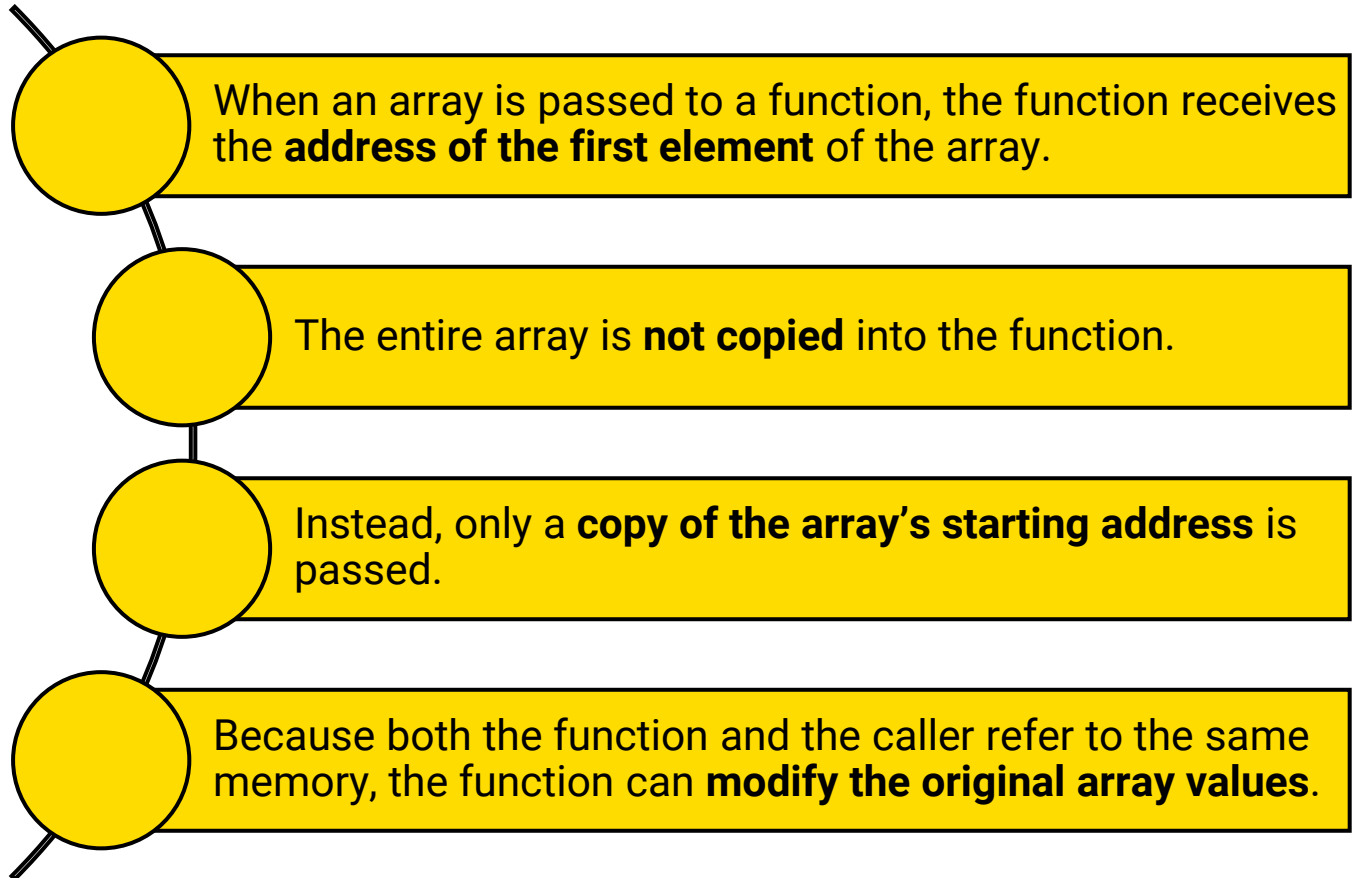
    location.x = 10;
    location.y = 9;

    printf("Before increment: (%d, %d)\n", location.x, location.y);
    increment(&location);
    printf("After increment:  (%d, %d)\n", location.x, location.y);

    return 0;
}

void increment(struct point *location) {
    location->x = location->x + 1;
    location->y = location->y + 1;
}
```

Functions and Arrays



Example:

What is the output of this program?

```
#include <stdio.h>

void double_values(int values[], int size);

int main(void) {
    int scores[] = {3, 5, 7, 9};
    int i = 0;

    double_values(scores, 4);

    while (i < 4) {
        printf("%d ", scores[i]);
        i++;
    }

    return 0;
}

void double_values(int values[], int size) {
    int i = 0;

    while (i < size) {
        values[i] = values[i] * 2;
        i++;
    }
}
```

Pointers and strings (array of char)

```
#include <stdio.h>

int main(void) {
    char word[] = "Computer";
    char *ptr = &word[3];

    printf("%c\n", *ptr);
    printf("%c\n", ptr[0]);
    printf("%c\n", ptr[1]);
    printf("%s\n", ptr);

    return 0;
}
```

What is happening?

- **word** is a **character array** storing the string "Computer"
- **ptr** is a **pointer to a character**
- **&word[3]** points to the **fourth character** of the string ('p')

Understanding the output

- ***ptr**
→ prints 'p'
- **ptr[0]**
→ same as ***ptr**, prints 'p'
- **ptr[1]**
→ moves one character forward, prints 'u'
- **printf("%s", ptr);**
→ prints the string starting from 'p':
puter

Demo

→ Week 6 `pointer_string_function.c`

We can also return a pointer from a function

```
#include <stdio.h>

int *increment(int *x);

int main(void) {
    int num = 5;

    int *ptr = increment(&num);

    printf("%d\n", *ptr); // prints 6
    printf("%d\n", num);  // prints 6

    return 0;
}

// function definition (after main)
int *increment(int *x) {
    *x = *x + 1;
    return x;
}
```

But you cannot return the address of a local variable, because it is destroyed when the function ends.

```
int *f(void) {  
    int numbers[] = {1, 2, 3};  
    return numbers;  
}
```



Or this?

```
int *f(void) {  
    int x = 3;  
    return &x;  
}
```



The Heap



We recall that when a stack frame is created, enough memory to store everything in the frame is allocated to the frame

What if the **amount of memory** needed **isn't known in advance**?

The program must know **exactly how much space** a stack frame requires **before** the function starts running.

In this case, we **can't use stack memory**.

Because the **size is unknown at compile time**, the stack cannot be used to allocate this memory.

Why do we need the heap?

We want to **create arrays** inside functions and **return them**

We also want to create arrays whose **size is only known at runtime**

Unlike stack memory, heap memory is allocated **explicitly by the programmer**

Heap memory **remains allocated** until the programmer **frees** it

This gives you **full control over memory** allocation and lifetime

But remember with great power comes great responsibility

C provides us some functions to interact with the heap.

malloc

Allocates memory on the heap

Returns a **pointer** to the allocated memory

Allows the programmer to **choose the size** of the allocation

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

The NULL Pointer

What does malloc return?

A **pointer to the allocated block of memory**, or

NULL if
the allocation
fails (not
enough
memory
available)

Because
allocation can
fail, you should
**always check
that the returned
pointer is not
NULL** before
using it.

The NULL Pointer

- Sometimes we initialise a pointer with a special value to indicate that it is not pointing to any valid memory yet
- This special value is called NULL
- Dereferencing a NULL pointer will cause a runtime error

```
int *my_ptr = NULL;
```

```
// Dereferencing a NULL pointer  
// causes a runtime error  
printf("%d\n", *my_ptr);
```

```
// Safe usage  
// Always check that a pointer is not NULL before  
// dereferencing it
```

```
int *my_ptr = NULL;
```

```
if (my_ptr != NULL) {  
    printf("%d\n", *my_ptr);  
}
```

Sizeof Operator

sizeof

We use the **sizeof operator** to calculate the **exact number of bytes** needed when allocating memory

sizeof returns an **unsigned long** value

sizeof can be applied to a **data type** or to a **variable**

long can store larger numbers than int

unsigned means the value is never negative

Example

```
#include <stdio.h>

int main(void) {
    double values[5];

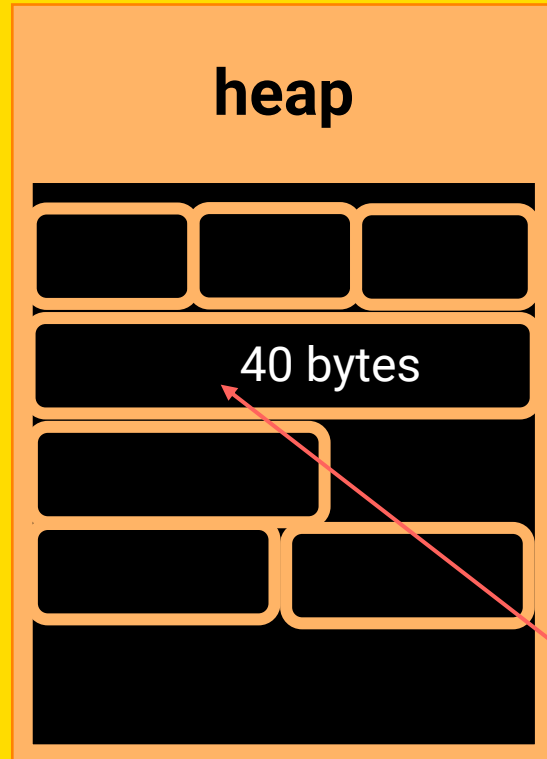
    printf("Size of a double: %lu bytes\n", sizeof(double));
    printf("Size of 5 doubles: %lu bytes\n", 5 * sizeof(double));

    return 0;
}
```

Demo

→size_of.c

Using malloc



To calculate how much memory to allocate, **multiply the number of elements by the size of each element's type** using sizeof.

This gives the total number of bytes that malloc should allocate.

malloc then returns a **pointer to the first byte** of the allocated block of memory.

```
int *numbers = malloc(10 * sizeof(int));
```

The Free Function

free tells the system that a **block of heap memory** is **no longer needed**

Every call to malloc should have a **matching call to free**

If memory is allocated but never freed, the program will **consume more and more memory** – this is known as a **memory leak**

Memory leaks are especially problematic in **long-running programs**

Although the operating system reclaims memory when a program exits, relying on this is **not good practice**

Using memory **after it has been freed**, or **freeing it more than once**, can lead to **serious and hard-to-debug errors**

Example:

```
// allocate space for values
double *values = malloc(count * sizeof(double));

// check malloc was successful
if (values == NULL) {
    return 1;
}

// use the array
// (e.g. values[0] = 3.14;)

// free the memory when done
free(values);
```

You can check for **memory leaks** using dcc with the flag
dcc --leak-check

Demo

→ malloc_intro.c

Voice of the Student

Anonymous ongoing feedback
Anything you wanted to share with me



Voice of the Student



[Voice of the Student – Fill out form](#)

See you soon ...