DPST1091 / CPTG1391

Introduction to Programming

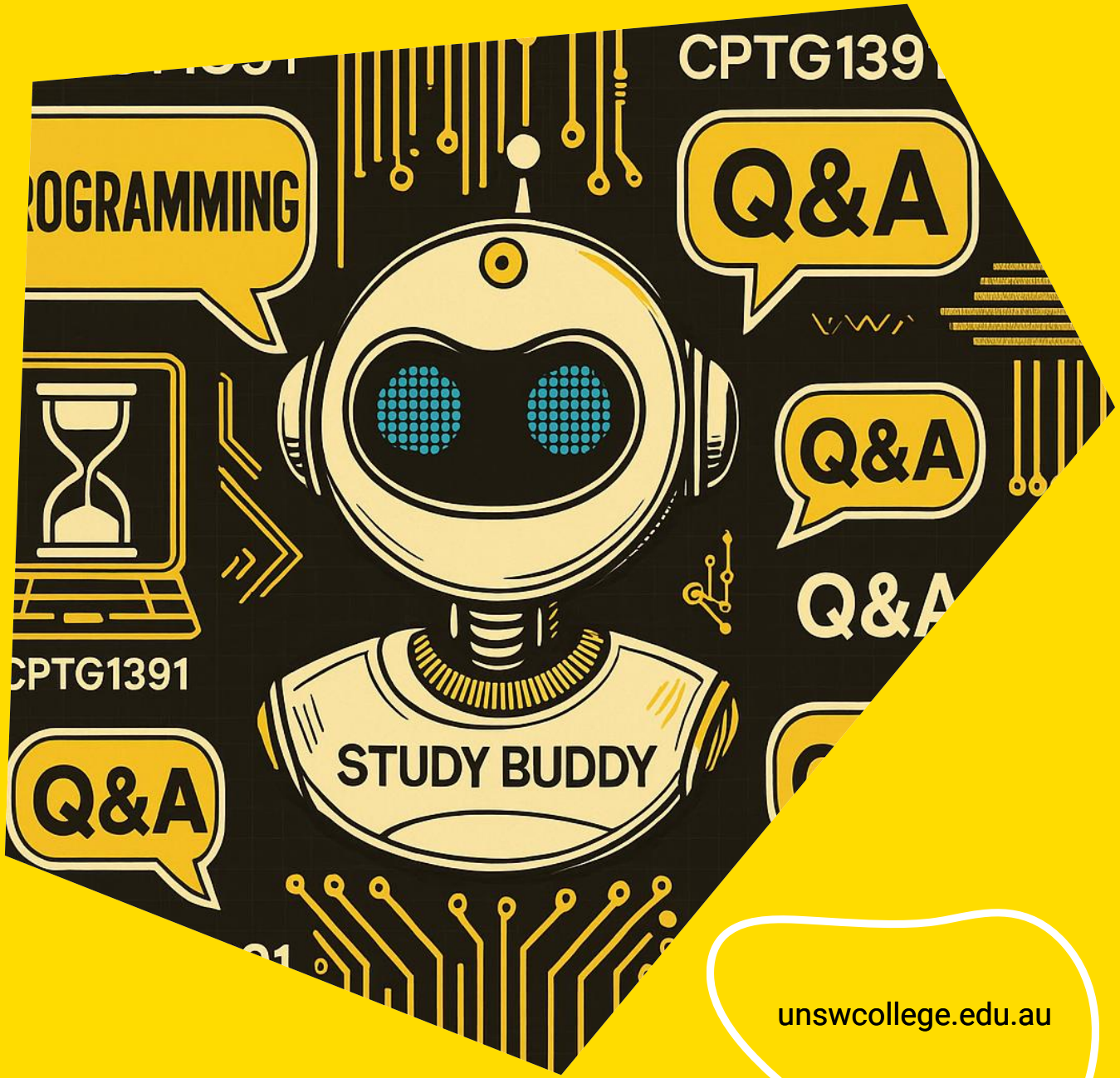
Week 6 – Lecture 2

Lecturer and Course Convener:

Dr Pantea Aria

UNSW College

Pointers and Functions



unswcollege.edu.au

Next Week Flexibility Week

No Lecture/Tutorial/labs

We will run **consultations** for everyone

Assignment is due on **Friday Week 8**

UNSW College

Agenda

- **Last lecture**
  - **Pointers**
  - **Memory, stack**

- **Today**
  - **Pointers and Functions (Call by Reference)**

Memory and Addresses Recap

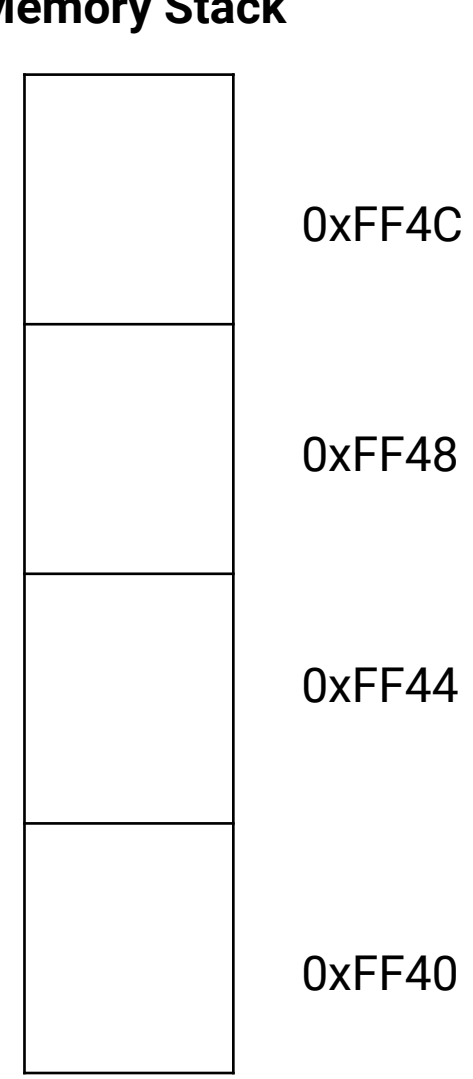
Memory can be thought of as one very **large array made up of bytes**.

Each **memory address** acts like an **index** that refers to a **specific byte** in that array.

Memory addresses are typically written in **hexadecimal**.

The **prefix 0x** indicates that the value is in hexadecimal.

On modern systems, memory addresses are usually **8 bytes** long and may look like:
`0x7ffcaa98655c`

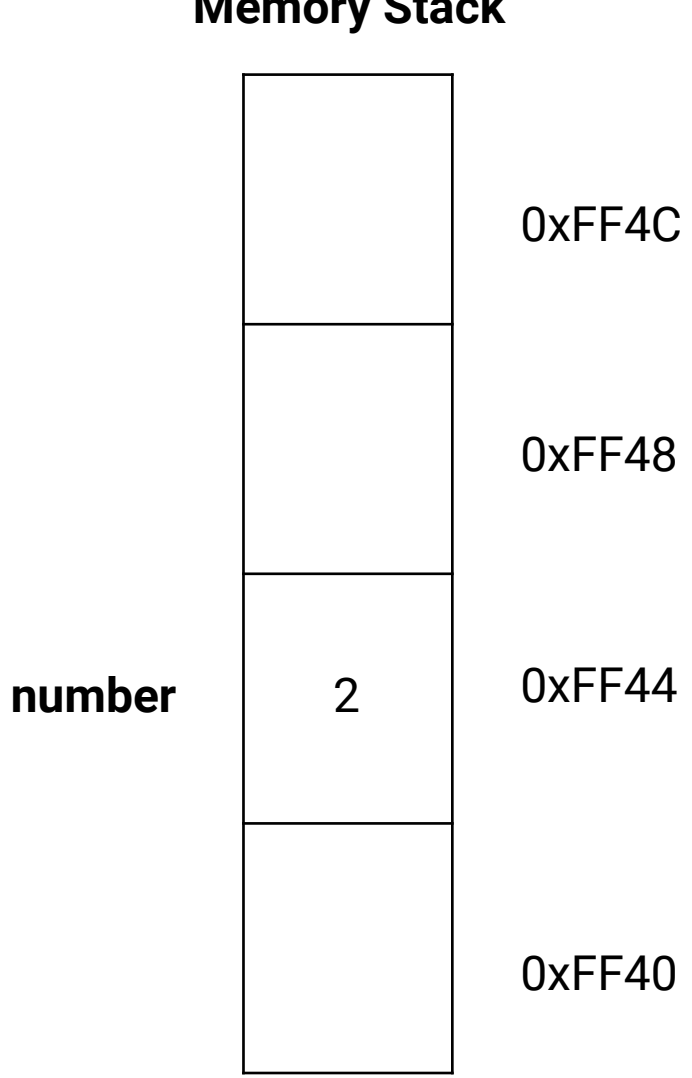


Memory and Variables

```c
// Declare a variable of type int
// Assign the value 2 to it
int number = 2;
```

During program execution, variables are stored in memory, with each variable located at a specific **memory address**.

number is stored at address 0xFF44

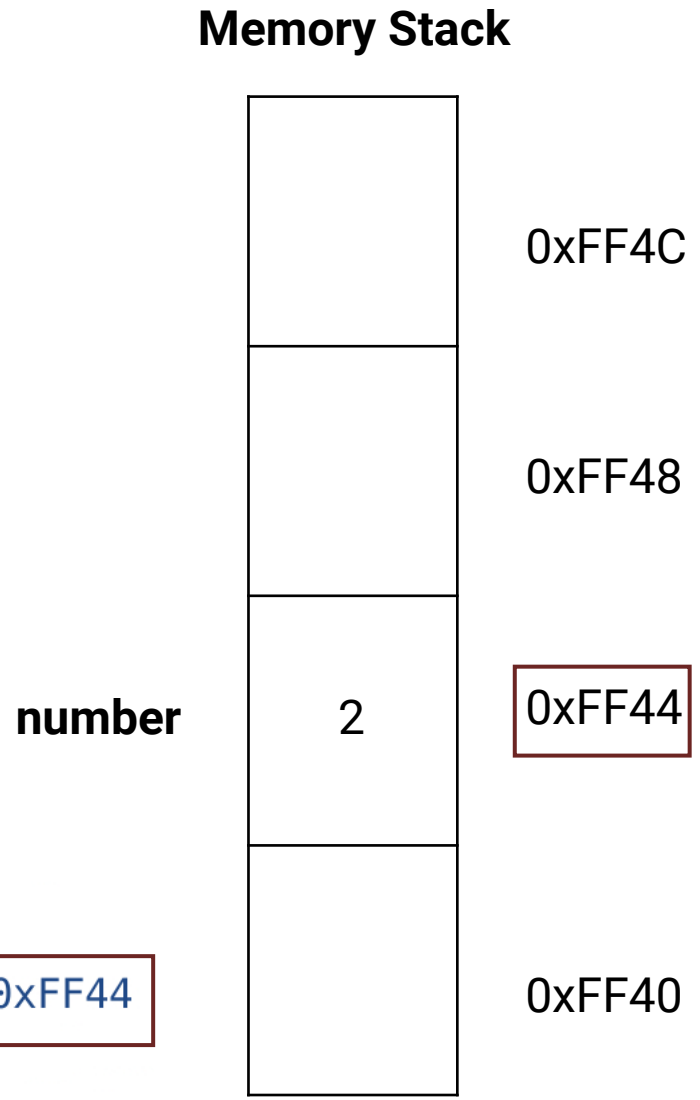


The Address of Operator

&

We can get the address of a variable using the **address of** operator **&**

```c
int number = 2;
// Print the address of x
// In this scenario it would print 0xFF44
printf("%p", &number);
```



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

```c
int number = 2;

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

```c
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;
```

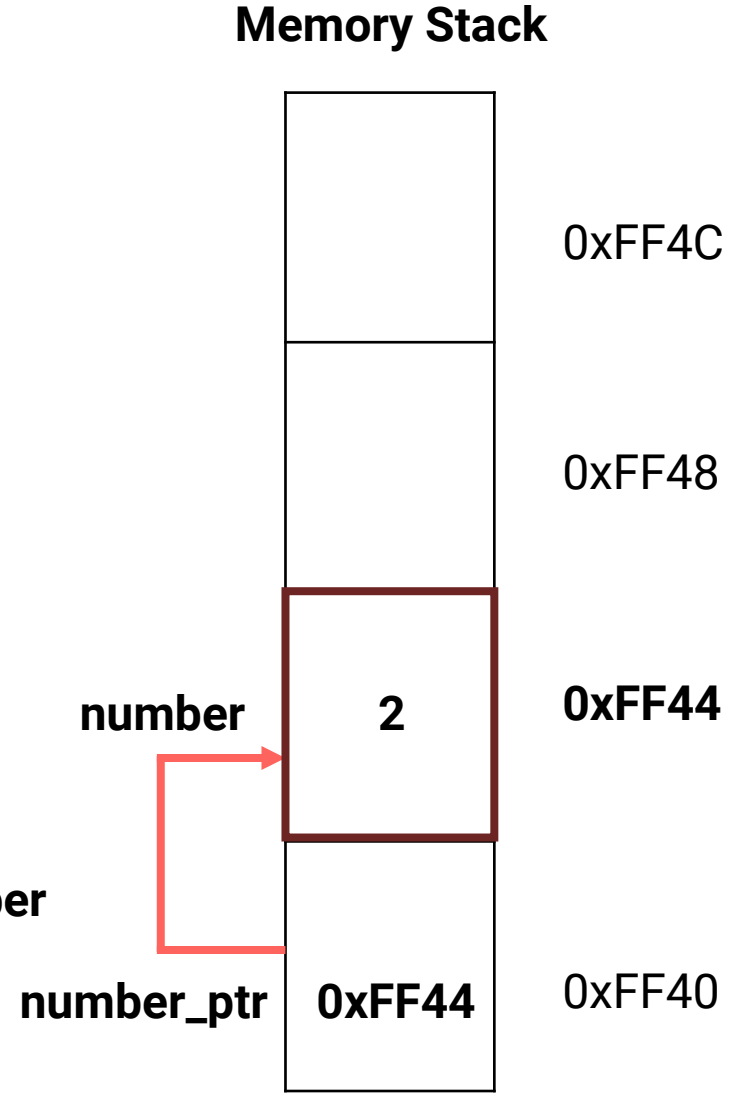
Visualise it in the memory

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



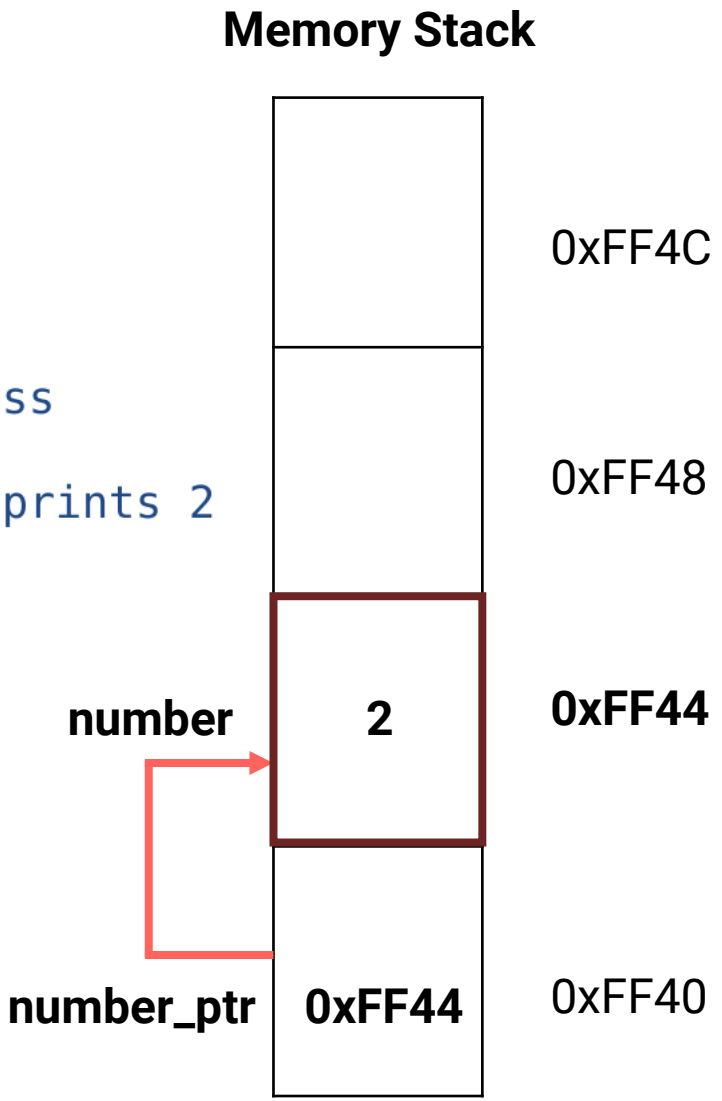
Dereferencing

```c
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
```

Dereferencing is simply accessing **the value** at the address of a pointer

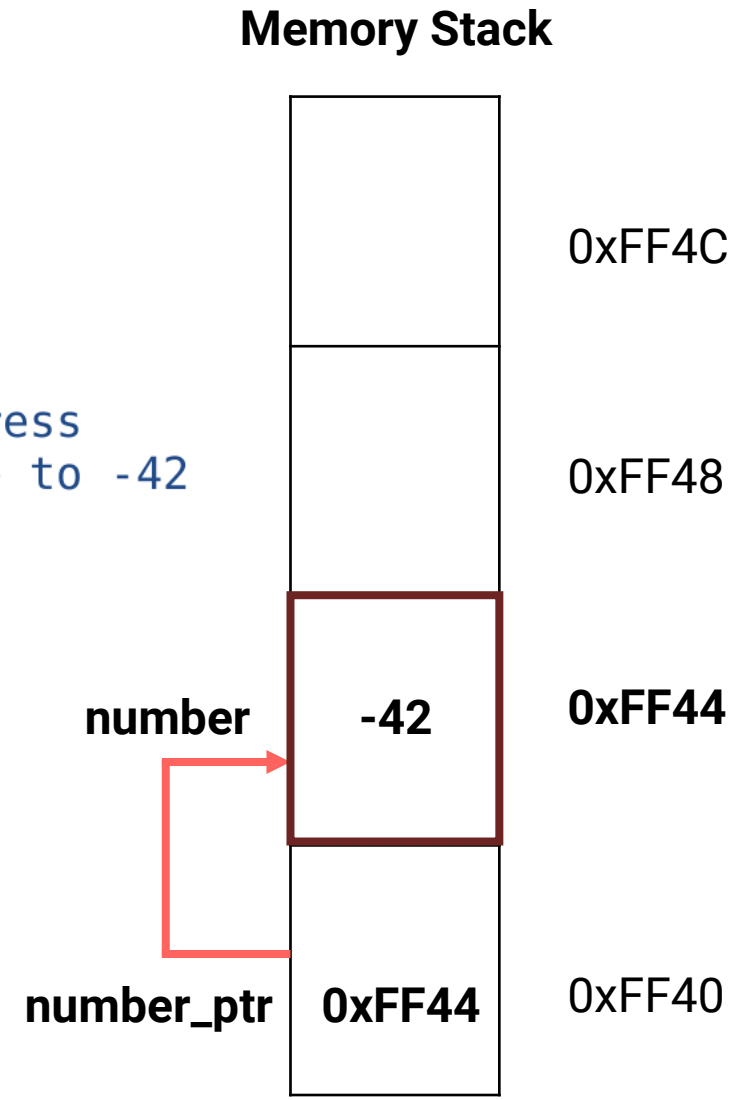
It uses the * symbol again (which causes confusion)

*number_ptr will get the integer at the address 0xFF44



Indirectly modify a variable using pointers

```c
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 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

```c
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?

```c
#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);
}
```

And what is the output of this program?

```c
#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);
}
```

Stack Memory

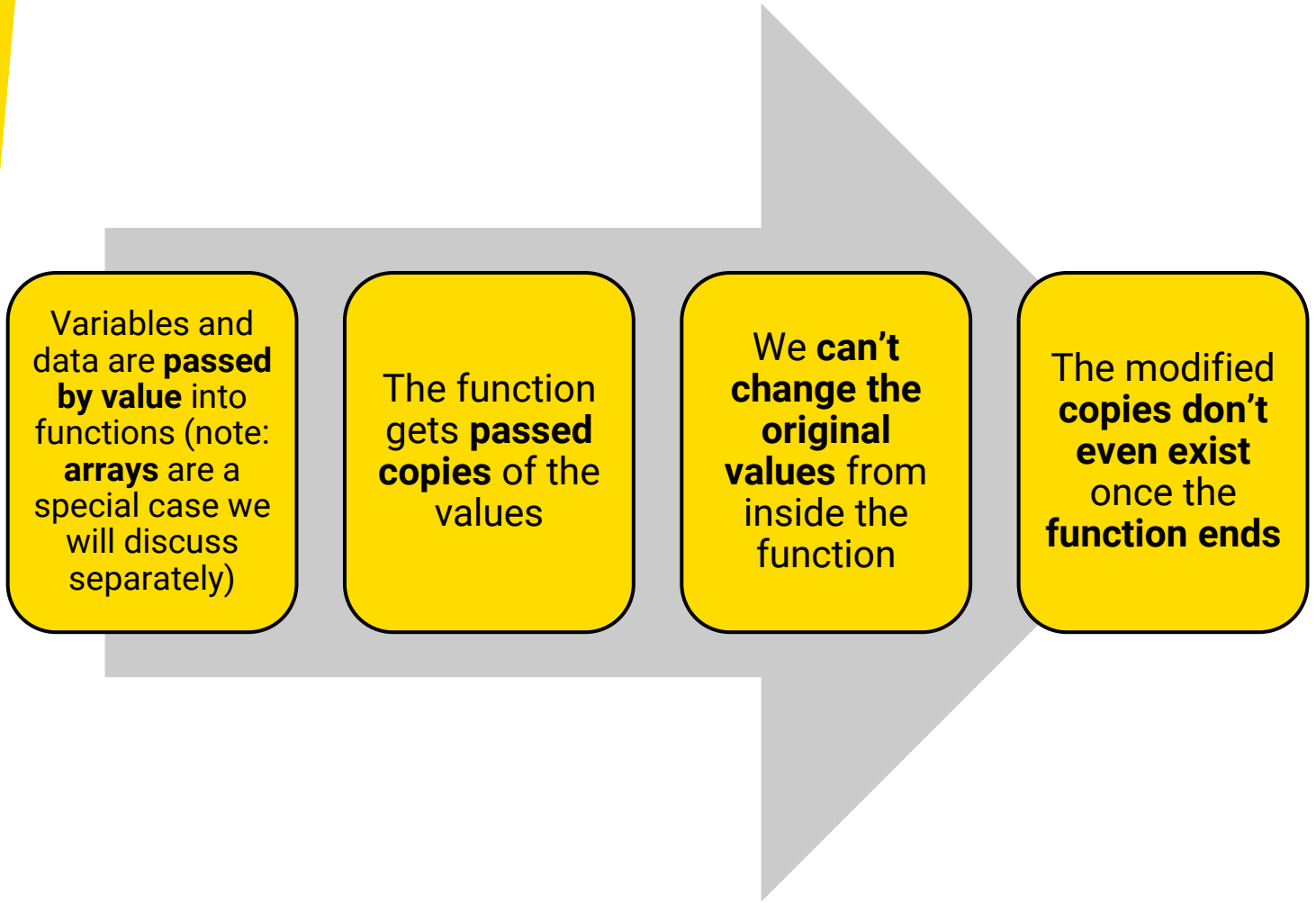
Stack memory stores information needed for your program to run, particularly details about **function calls**.

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

Functions And call by value




IT'S BREAK TIME!
#include <stdio.h>
#define ON_BREAK 1
int main(){
// Time for a 10 minute break! Switch to PARTY_MODE
#define PARTY_MODE ON_BREAK
if (PARTY_MODE == ON_BREAK) {
print("Program will resume in 10 minutes...");
sleep(600);
// Take a break
exit(0);
}
10 MINUTES BREAK!
BRB
CHIPS
Relax... We'll be back soon!

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**.

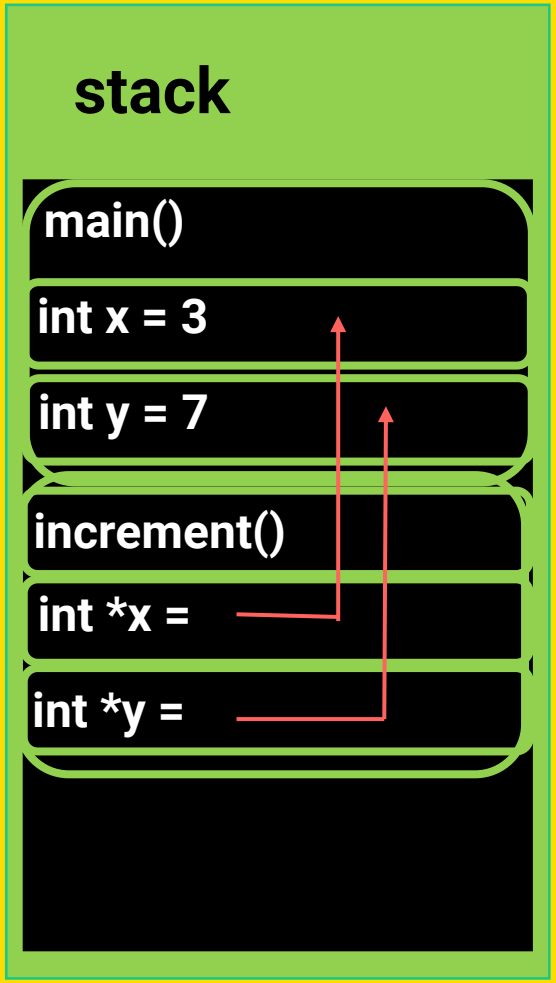
Example

Making function increment work using pointers

```c
#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);
}
```

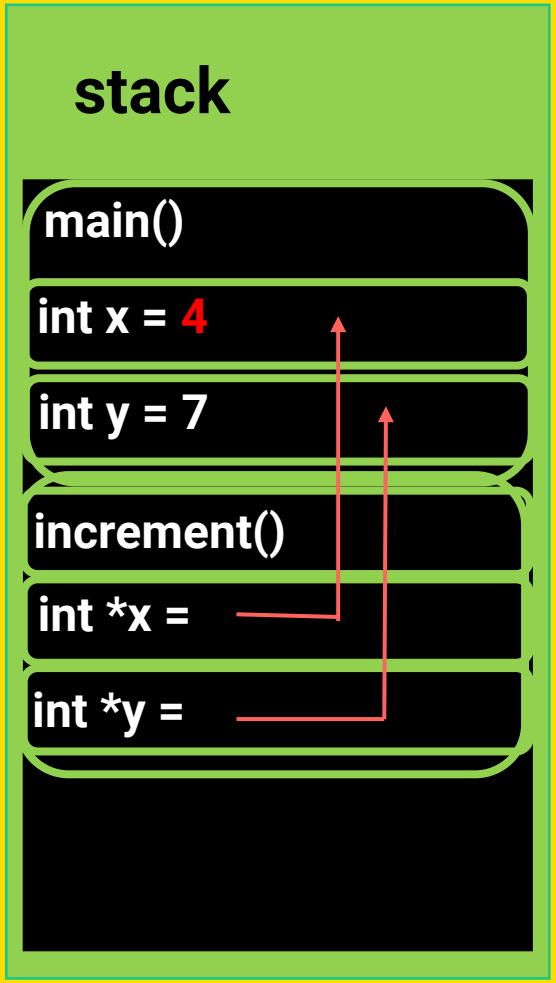
The **`main`** function must pass the **addresses** of `x` and `y` to the function.

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



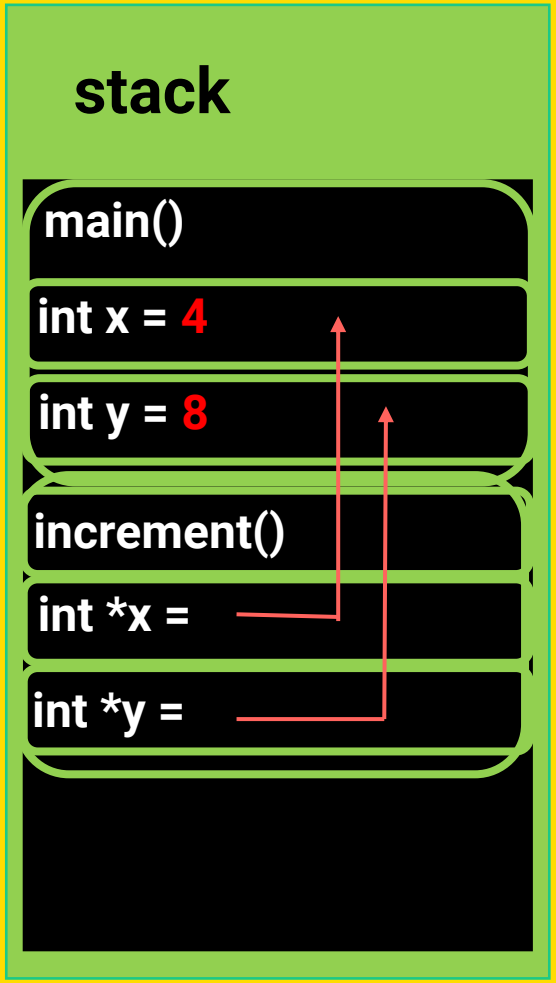

```c
#include <stdio.h>

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



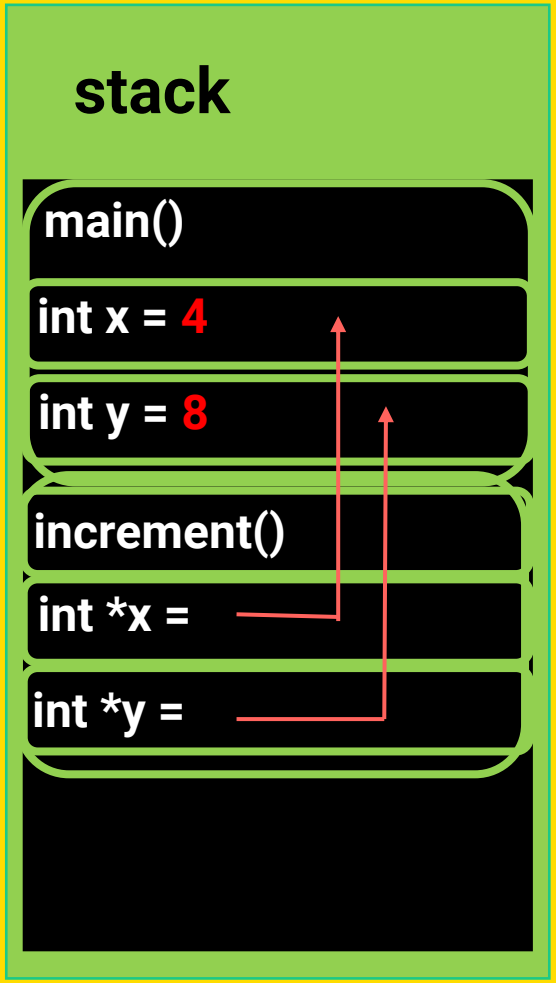

```c
#include <stdio.h>

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



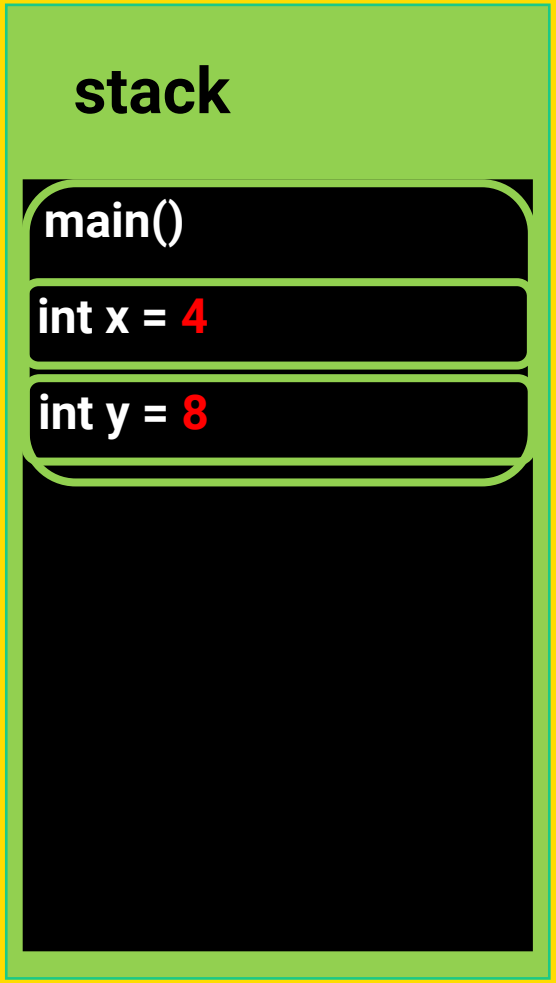

```c
#include <stdio.h>

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



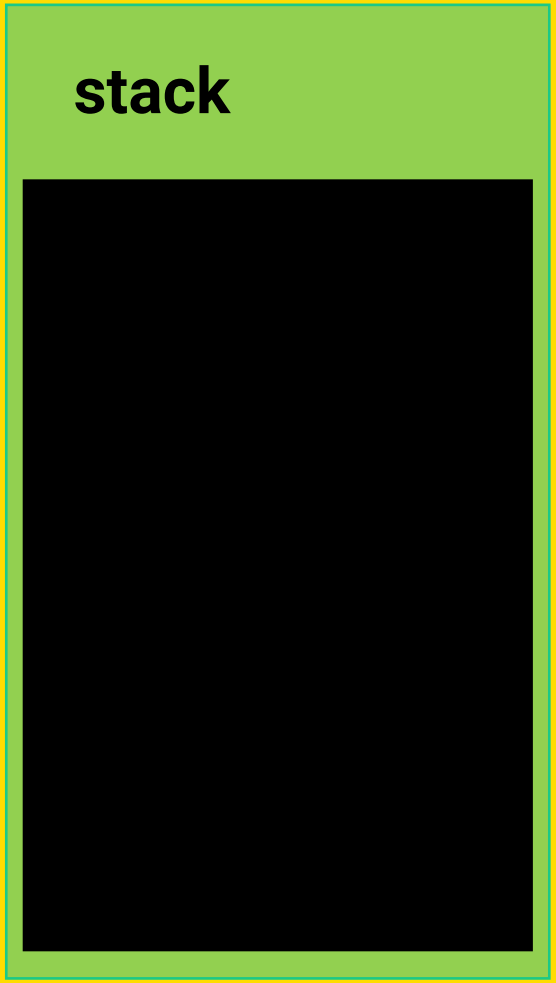

```c
#include <stdio.h>

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




```c
#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




```c
#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

```c
#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);
}
```

Demo

⟶pointer_add.c
⟶Pointer_add_product.c

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

UNSW College

Pointers to structs

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

```c
#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

```c
#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

```c
#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?

```c
#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

```c
#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.

```c
#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

- When an array is passed to a function, the function receives the **address of the first element** of the array.
- The entire array is **not copied** into the function.
- Instead, only a **copy of the array's starting address** is passed.
- Because both the function and the caller refer to the same memory, the function can **modify the original array values**.

Example:

What is the output of this program?

```c
#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++;
    }
}
```

Demo

→pointer_array_function.c

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

UNSW College

Pointers and strings (array of char)

```c
#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

→pointer_string.c
→Pointer_string_function.c

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

UNSW College

Voice of the Student

Anonymous ongoing feedback
Anything you wanted to share with me

26T1 Voice of the Student – Fill out form

See you soon …

UNSW College