

DPST1091 / CPTG1391

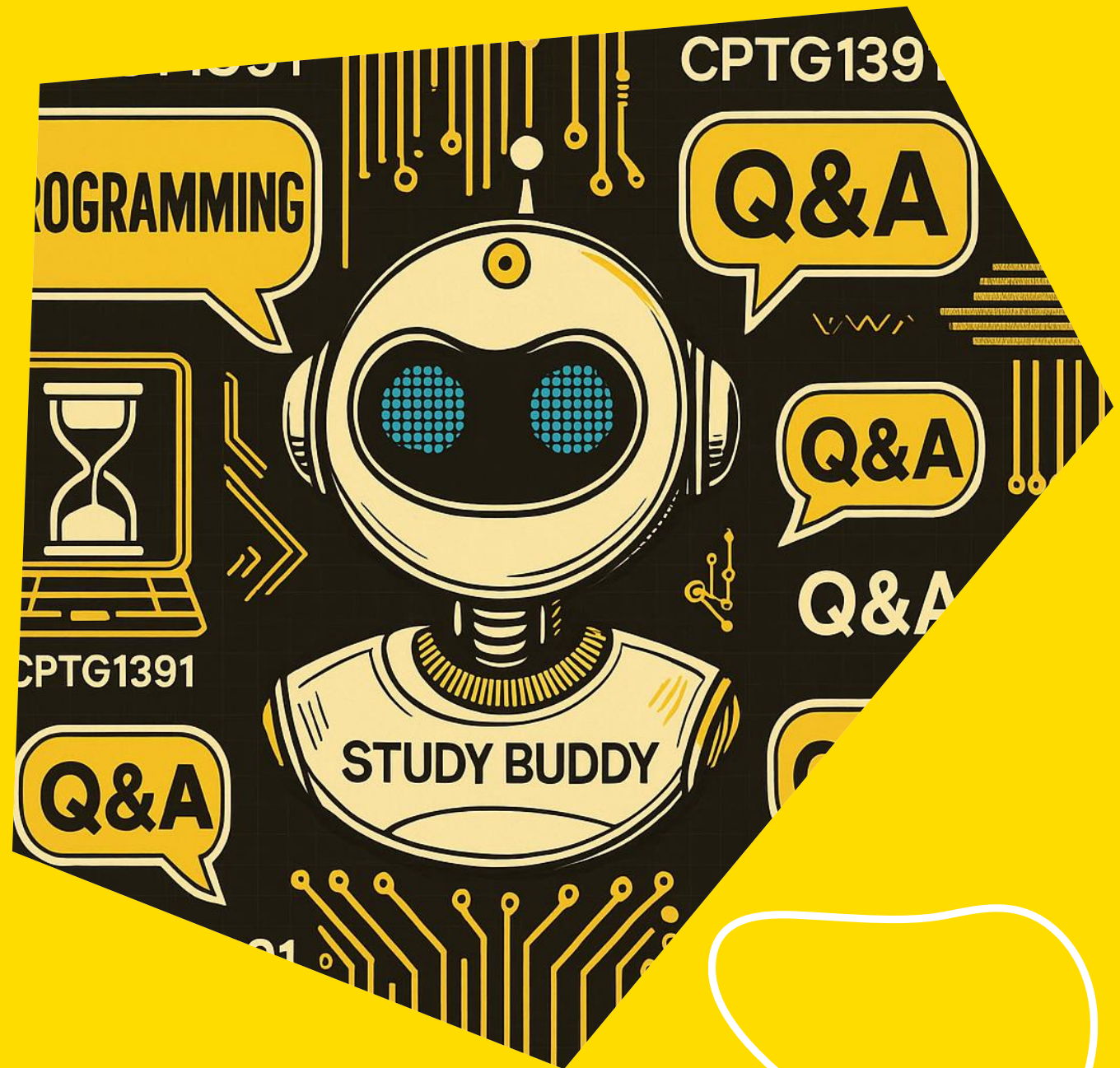
Introduction to Programming

Week 8 – Lecture 2

Lecturer and Course Convener:

Dr Pantea Aria

More malloc and Multi-File Projects



Agenda

- **Today**
More malloc
Multi-File Projects

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.



The Heap recap



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;
```

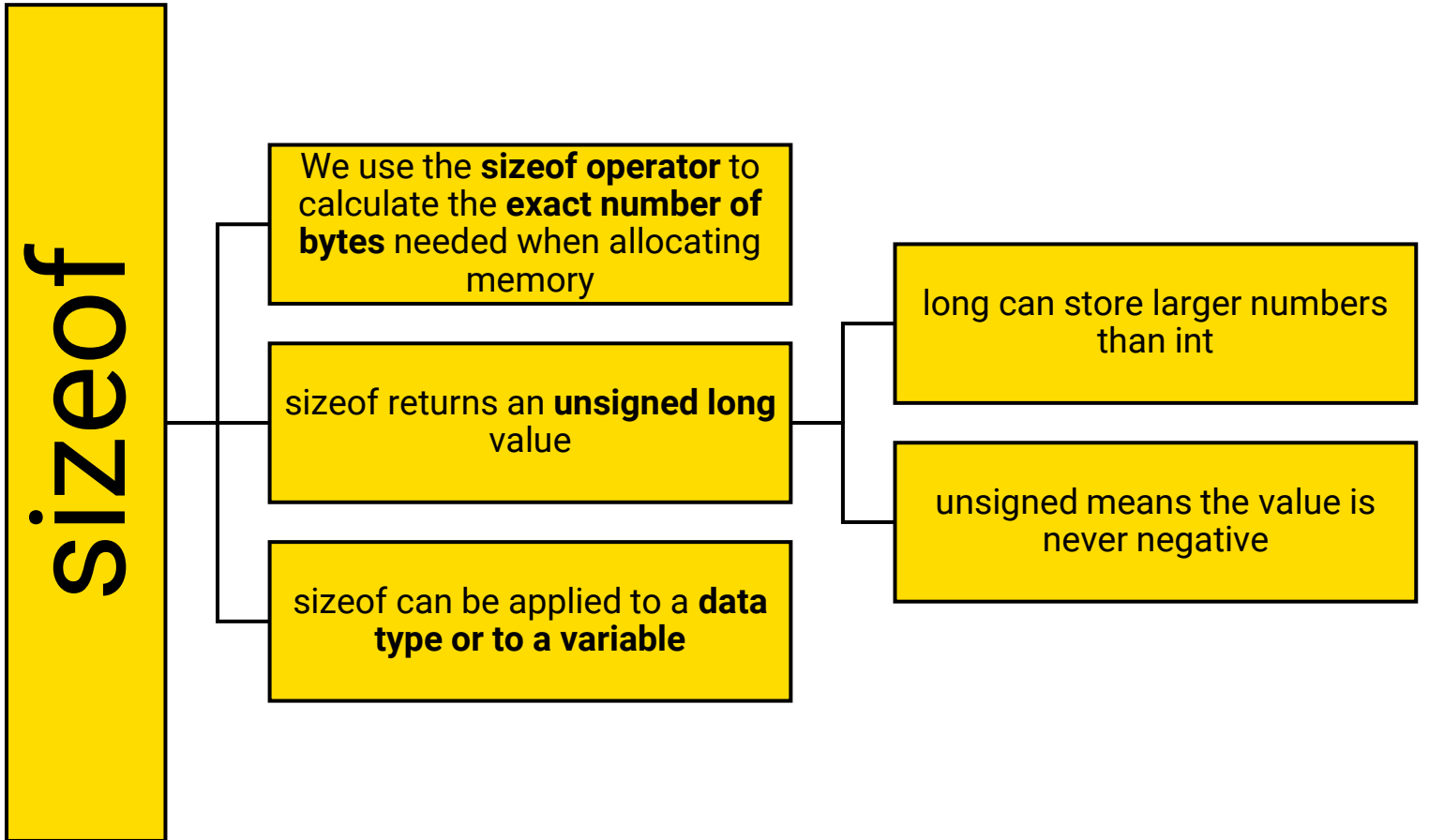
```
// Dereferencing a NULL pointer  
// causes a runtime error  
printf("%d\n", *my_ptr);
```

```
// 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



Example

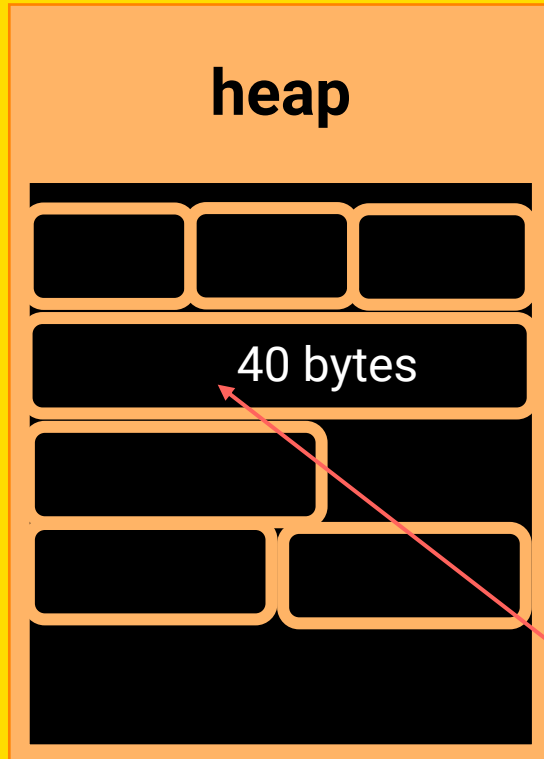
```
#include <stdio.h>

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;
}
```

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

Creating an array with malloc

```
int main(void) {  
    int size;  
    scanf("%d", &size);  
  
    char *buffer = malloc(size * sizeof(char));  
  
    if (buffer == NULL) {  
        printf("Out of memory\n");  
        return 1;  
    }  
}
```

You can treat arrays created with malloc just like normal arrays.

You can use indexes on them the same way as usual.

You can pass them into functions without any issues.

You can even return them from functions!

That's because malloc puts the array on the heap, so it still exists after the function finishes.

```
// creates and returns a malloced array
// filled with even numbers
int *make_even_array(void) {
    int size;
    scanf("%d", &size);

    int *numbers = malloc(size * sizeof(int));
    if (numbers == NULL) {
        return NULL;
    }

    for (int i = 0; i < size; i++) {
        numbers[i] = i * 2;
    }

    return numbers;
}
```

Demo

→ malloc_array.c
→ memory_leak.c

The realloc Function

If the array was created dynamically, we can resize it using **realloc**.

```
// Allocate space for 10 integers
int *numbers = malloc(10 * sizeof(int));

// Later, we realise we need more space
// realloc increases the size without
// losing the existing data
numbers = realloc(numbers, 20 * sizeof(int));
```

Demo

→realloc_intro.c

Creating struct with malloc

We use malloc for a struct when:

- ☐ We are building dynamic data structures (linked lists, trees, graphs)
- ☐ We need memory that is controlled manually (not automatically removed)
- ☐ We need the struct to live beyond the current function

Demo

→ malloc_struct.c

Multi-File Projects

Large programs are usually **split** across **multiple files** rather than written all in one place. This has a few big advantages:

The code is **easier to read** because each file is **shorter**. Different parts can be written, **tested**, and updated **independently**.

Related code can be **grouped** together, which makes programs more **modular**.

We've actually been **using** this idea **already** without realising it.

Every time we use **#include**, we're pulling in code from **another file**.

So far, we've only been including files from the **C standard library**.

Multi-File Projects

You can also **#include your own files** (this is where it gets fun!).

It lets us **break larger projects** into smaller pieces and join them together.

It also makes it possible for **multiple people to work** on the same project, like in **real-world software teams**.

We often **write code once** and then **reuse** it in other projects.

That's exactly what the C standard libraries are: collections of useful functions **reused** in many programs.

Coming up, **Assignment 2** will use **multiple files**.
• **Assignment 1 will not** – do **not** split Assignment 1 into multiple files.

In a multi-file project, you'll usually see a few different types of files:

One or more **header files** (.h), similar to the standard library headers you've already been using.

One or more **implementation files** (.c), which contain the actual code for what's declared in the headers.

A .c file with the **main** function, this is the **program's entry point**, and we usually keep it as small and simple as possible.

Header (.h) Files usually contains:

Function prototypes for functions that are implemented in the corresponding .c files.

Comments that explain how those functions should be used.

#define constants and **enum** definitions.

Header files are **important** because they:

Give programmers the information they need to use the code correctly, a bit like documentation.

Give the compiler the details it needs to check types and catch errors when the header is included in .c files.

Implementation (.c) Files usually contains:

There should be **exactly one .c file** that contains the **main** function.

Any **other .c files** usually contain: The **implementations of functions** declared in their matching header files.
.c files include the headers they depend on.

When including your own headers, **use double quotes** instead of angle brackets.

For example: `#include "array_functions.h"`

Example

Imagine a **project**
made up of three
files:

- A header file called: `arrays_functions.h`
- An implementation file called: `arrays_functions.c`, which includes:
`#include "arrays_functions.h"`
- A file containing the **main** function, `mfp.c`, which also includes:
`#include "arrays_functions.h"`

Another **project**
made up of three
files:

- A header file called: `arrays_functions.h`
- An implementation file called: `arrays_functions.c`, which includes:
`#include "arrays_functions.h"`
- Another file containing the **main** function, `another_mfp.c`, which also includes
`#include "arrays_functions.h"`

How to compile and run Multi-File Programs

You don't compile `.h` files directly.

- Header files are included inside the relevant `.c` files.

Instead, you compile all the `.c` files together to create a single executable.

- Exactly one of those `.c` files must contain the `main` function.

```
$ gcc -o mfp mfp.c array_functions.c
$ ./mfp

$ gcc -o another_mfp another_mfp.c array_functions.c
$ ./another_mfp
```

Demo

→mfp.c

→arrays_functions.c

→arrays_functions.h

→another_mfp.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 ...