COMP1511 - Programming Fundamentals

Term 2, 2019 - Lecture 13

What did we cover last week?

Pointers and Memory

- Pointers are variables that store memory addresses
- They allow us to access variables from anywhere in our code

Structs

- Custom variables made up of collections of variables
- Able to store different types of variables

What are we covering today?

Memory

- How functions work in memory
- Direct use of memory in C

Multi-File Projects

- Using more than one file for a program
- Using files to hide some information and provide a useful interface

Recap - Pointers

Pointers

- A pointer is a variable that stores a memory address (`&` to get an address)
- We can assign a memory location to a pointer from a variable
- We can access the memory the pointer is "aiming at" using `*`

```c
int i = 100;
// create a pointer called ip that points at
// the location of i
int *ip = &i;
printf("The value of the variable at %p is %d", ip, *ip);
```

Recap - Structs

Structs

- A struct is a collection of variables that can be accessed under one name
- They're used to collect custom information together

```
struct ninja {
    char name[50];
    char phrase[50];
    int power;
    int health;
};
```

Recap - Pointers and Structs

We often use pointers and structs together

- We use `->` to access fields when we have a pointer to a struct
- We often pass pointers to structs into functions

```
void display_person(struct person *hero) {
    printf("Name: %s\n", hero->name);
    printf("Powers:\n");
    int i = 0;
    while (i < hero->num_powers) {
        fputs(hero->powers[i], stdout);
        putchar('\n');
        i++;
    }
}
```

Functions and Memory

What actually gets passed to a function?

- Everything gets passed **"by value"**
- Variables are copied by the function
- The function will then work with their own versions of the variables

What happens to variables passed to functions?

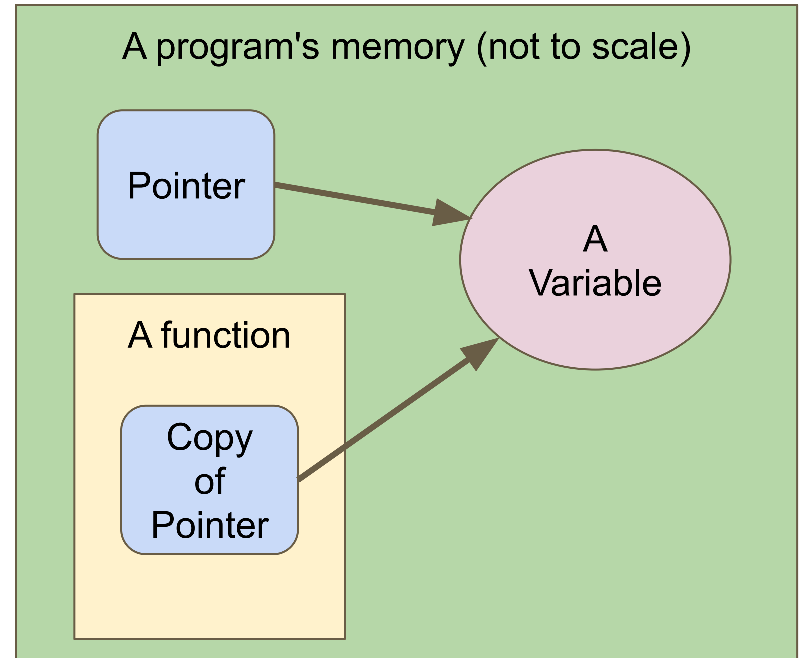
```
int main (void) {
    int x = 5;
    doubler(x);
    printf("x is %d.\n", x,);
    // "x is 5"
    // this is because the doubler function takes the value 5 from x
    // and copies it into the variable "number" which is a new variable
    // that only lasts as long as the doubler function runs
}

int doubler(int number) {
    number = number * 2;
    return number;
}
```

Functions and Pointers

What happens to pointers that are passed to functions?

- Everything gets passed "by value"
- But the value of a pointer is a memory address!
- The memory address will be copied into the function
- This means **both** pointers are accessing the same variable!



Functions and Pointers

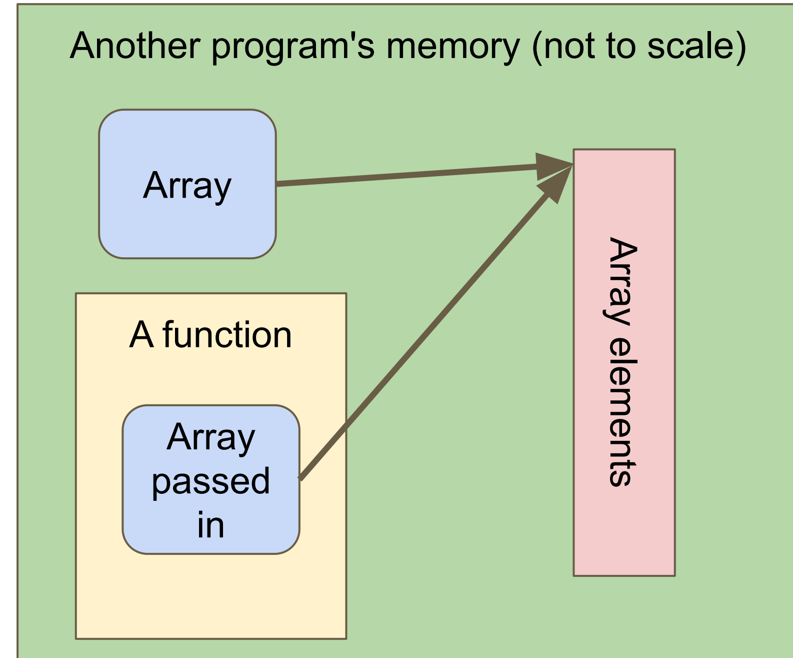
```
int main (void) {
    int x = 5;
    int *pointerX = &x;
    doublePointer(pointerX);
    printf("x is %d.\n", x);
    // "x is 10"
    // This is because doublePointer gets given access to x via its
    // copied pointer . . . since it changes what's at the other end of
    // that pointer, it affects x
}

// Double the value of the variable the pointer is aiming at
void doublePointer(int *numPointer) {
    *numPointer = *numPointer * 2;
}
```

Arrays are represented as pointers

Arrays and pointers are very similar

- An array is a variable
- It's not actually a variable containing all the elements
- When we use the array variable (no []), it's actually the memory address of the start of the elements
- Arrays and pointers act the same!



Functions and Arrays

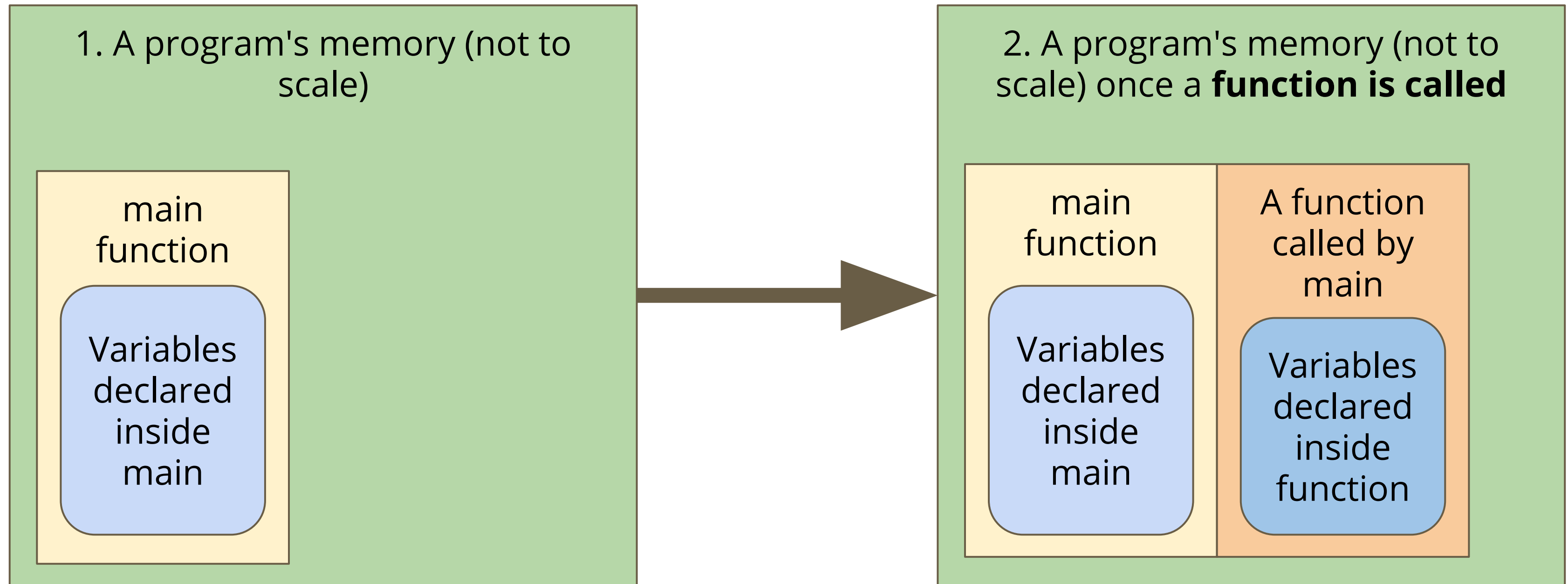
```
int main (void) {
    int myNums[3] = {1,2,3};
    doubleAll(3, myNums);
    printf("Array is: ");
    int i = 0;
    while(i < 3) {
        printf("%d ", myNums[i]);
        i++;
    }
    printf("\n");
    // "Array is 2 4 6"
    // Since passing an array to a function will pass the address
    // of the array, any changes made in the function will be made
    // to the original array
}
```

Functions and Arrays continued

```
// Double all the elements of a given array
void doubleAll(int length, int numbers[]) {
    int i = 0;
    while(i < length) {
        numbers[i] = numbers[i] * 2;
        i++;
    }
}
```

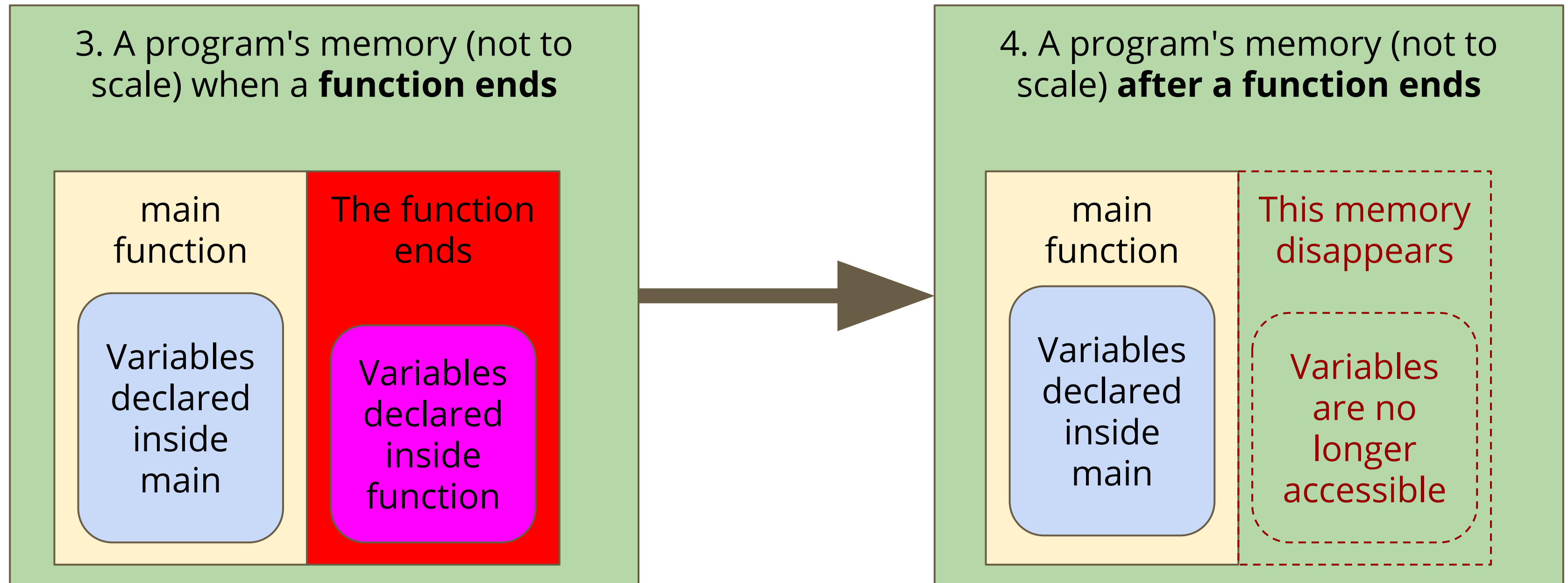
Memory in Functions

What happens to variables we create inside functions?



Memory in Functions

What happens to variables we create inside functions?



Keeping memory available

What if we want to create something in a function?

- We often want to run functions that create data
- We can't always pass it back as an output

```
// Make an array and return its address
int *createArray() {
    int numbers[10] = {0};
    return numbers;
}
// This example will return a pointer to memory that we no longer have!
```

Memory Allocation

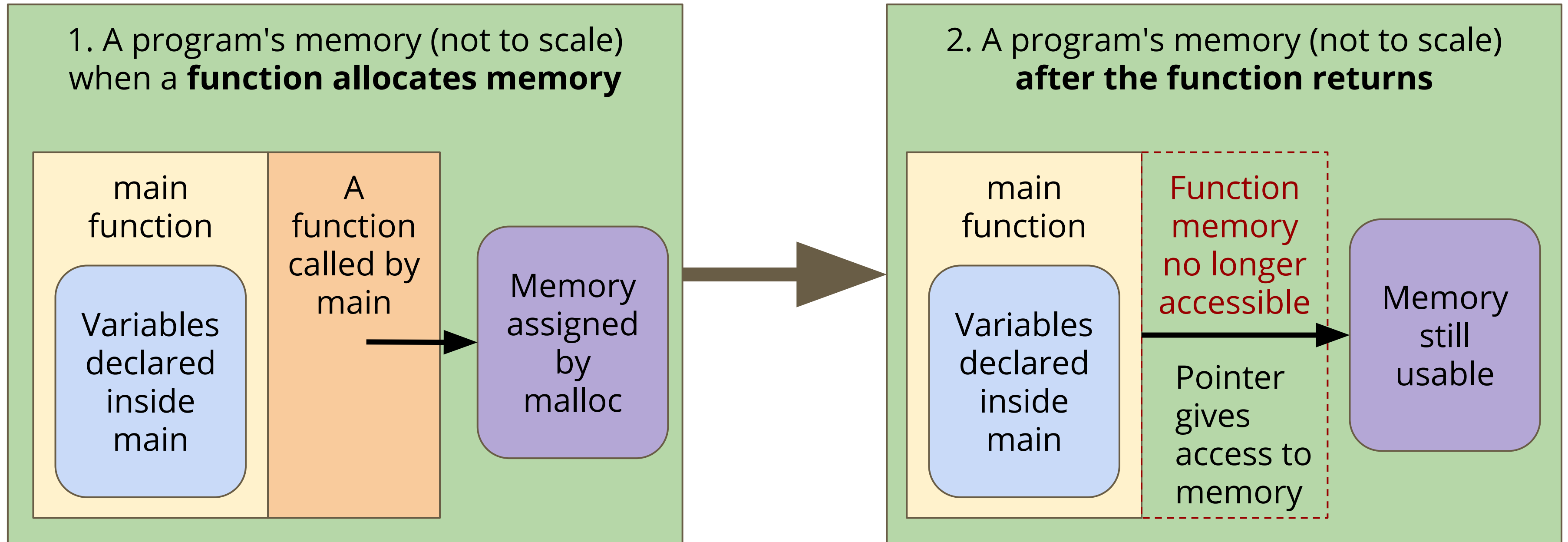
C has the ability to allocate memory

- A function called **`malloc(bytes)`** returns a pointer to memory
- Allows us to take control of a block of memory

- This won't automatically be cleaned up when a function ends
- To clean up the memory, we call **`free(pointer)`**
- **`free()`** will use the pointer to find our previous memory to clean it up

What malloc() does

Using malloc, we can assign some memory that is not tied to a function



Malloc() in code

We can assign a particular amount of memory for use

- The function **`sizeof()`** allows us to see how many bytes a variable needs
- We can use **`sizeof()`** to allocate the correct amount of memory

```
// Allocate memory for a number and return a pointer to them
int *mallocNumber() {
    int *intPointer = malloc(sizeof(int));
    *intPointer = 10;
    return intPointer;
}
// This example will return a pointer to memory we can use
```

Cleaning up after ourselves

Allocated memory is never cleaned up automatically

- We need to remember to use **`free()`**
- Every pointer that is aimed at allocated memory must be freed!

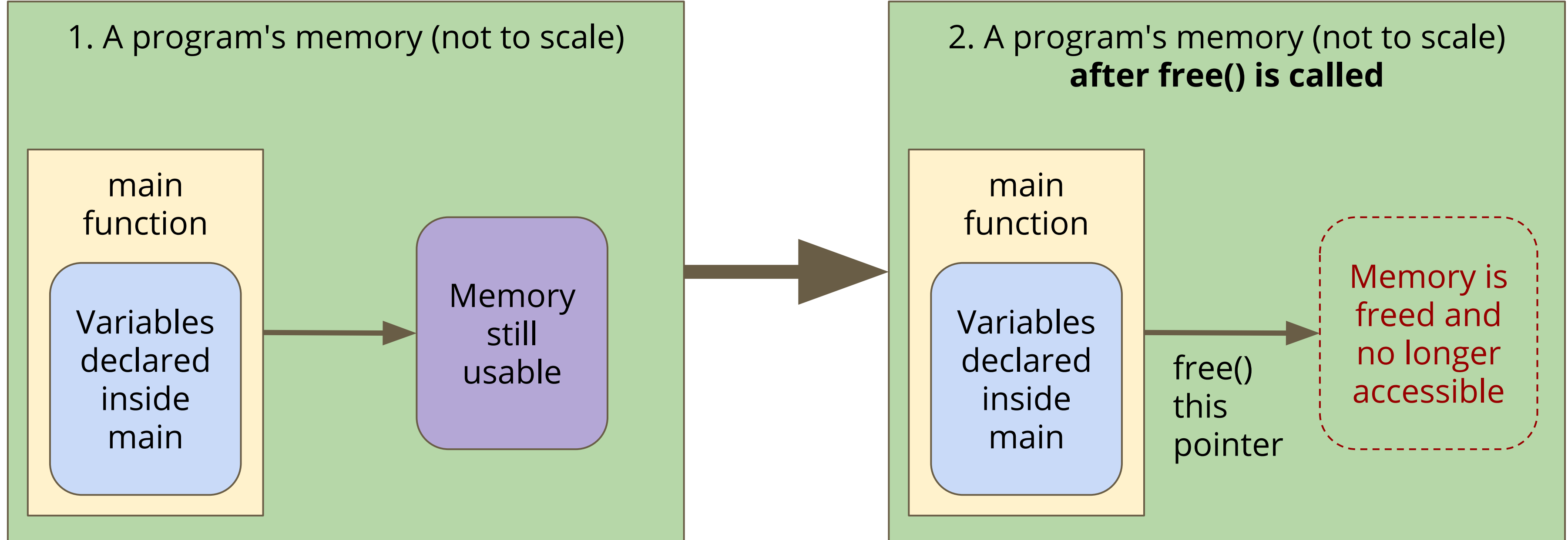
```
// Use an allocated variable via its pointer then free it
int main(void) {
    int *iPointer = mallocNumber();

    *iPointer += 25;

    free(iPointer);
    return 0;
}
```

Freeing up memory

Calling free will clean up the allocated memory that we're finished with



Using memory

Some things to think about with malloc() and free()

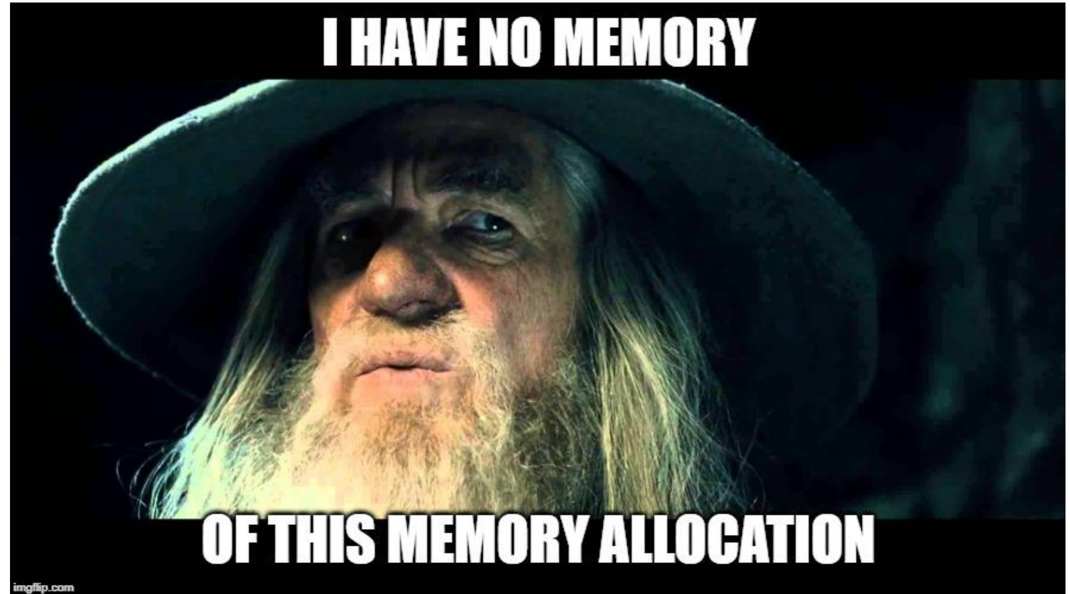
- You can use `sizeof()` to figure out how many bytes something needs
- We can malloc arrays and structs as well as variables
- In general, always use `sizeof()` with `malloc()`

- Anything allocated with `malloc()` must be `free()` after you've finished with it
- Otherwise we get what's known as memory leaks!
- `dcc --leak-check` can be used to tell you if you have any memory leaks

Break Time

Memory allocation is tricky

- It's easy to forget what you've allocated
- Then you might forget to free it!



C Projects with Multiple Files

For readability and also to separate code by subject

- We've already seen #include
- We can also #include our own files!
- This allows us to join projects together

Reusable sub-projects

- We'll often make some code that we can use again
- If we make it in its own file, with its own interface, we can #include it in our projects

Header Files and C (Implementation) Files

Two different files for different purposes

- Header and C files usually go together in pairs

Header *.h file

- Shows the capabilities of a code file
- Enough to use it without needing to understand what's in it

C Implementation *.c file

- Contains the underlying implementation of the H file

File.h

Header Files show you what the code's functions are

- This file shows a programmer all they need to know to use our code
- `typedef` (Type Define) is a way of allowing us to create our own C Type out of another Type
- This protects our struct from access and keeps our data safe!
- Function Declarations with no definitions
- Comments that describe how the functions can be used
- No running code!

File.c

Implementation Files show you how the code runs in detail

- We can hide the complicated running code in this file
- Has includes, especially `#include "File.h"` (joins the two files together)
- Implements the struct mentioned in the typedef from the header
- Implements all the functions declared in the header

Main.c and other Files

Our Entry Point into our code

- The main function is always what runs first
- For any code file (*.c) to use the functionality provided by another file, it must #include that file
- In our example, main.c needs to include person.h to be able to access the functionality provided by the person code

Compiling a Project with Multiple Files

How do we compile multi-file project?

- We need to compile all *.c files that we will use
- The *.c files will **`#include`** the necessary *.h files
- Amongst the *.c files there should be exactly one **`main()`** function
- The compiled program will run from the start of the **`main()`** function

Let's look at a multi-file project

I'm Batman!

- A set of files that allow us to define a "person"
- Each person has a name and some super powers
- But also, they have a pointer to their secret identity!
- person.h shows how we can use a person
- person.c has the underlying details
- main.c shows how we can include and use this code

person.h

What's in the Header file?

- A Typedef saying we can use `Person` to mean a pointer to a `struct person`
- No mention of what `struct person` is! We don't have direct access
- Functions to let us create and free a person
- A function to let us give powers to a person
- A function to display a person (by printing to the terminal)

person.c

Our implementation file

- The actual and hidden implementation of `struct person`
- This means that the code in the C file can use `struct person` but the main.c can only use `Person`
- Implementations of all the functions listed in person.h

main.c

The main file

- Contains the main function. There is always exactly one main function in any project. It will be where the program starts running
- #includes the person.h file (always include headers, but not C files)
- Uses things like Person and the functions provided in the header

Using the multi-file project

Compiling

- We'll compile all the C files (but no H files) into a single program
- We rely on #includes to get the information we need from H files
- In this case: `dcc main.c person.c -o person_demo`

Using Multi-file projects in COMP1511

- We will be keeping these reasonably simple in COMP1511
- Assignment 2 will have a multi-file project, but you will not need to create a multi-file project to pass this course

What did we learn today?

Functions and Memory

- How functions have their own piece of memory
- How we lose access to anything in a function once it returns
- How we can specifically allocate memory

Multi-File Projects

- How C separates functionality in a Header and C (Implementation) file
- How we can include our own files
- How headers make it easier to read what a set of files can do