

Session 2 - The Cequel

Welcome to the Cequel!

See what I did there

I hope you enjoyed Session 1 and are now comfortable with using variables, for loops, and if statements to create very basic **C** programs (including commenting and being able to compile your code), and are familiar with arrays and the interesting part of **C**: undefined behaviour and how you can do many things in **C** which shouldn't be allowed (we'll see some more examples in this session!).

It's getting to the harder - but more interesting - stuff ...

Good luck and happy coding!

- CHANGE THE README

Contents

- Pointers
 - Memory Addresses
 - The Point (Intro to Pointers)
 - Dereferencing
 - The Null Pointer
 - Pointer Exercises
- Functions
 - Void
 - Function Prototypes
 - Revisiting Hello World
 - Function Exercises 1
 - Passing Around Pointers
 - Function Exercises 2
- Revisiting Arrays
 - Arrays and Pointers
 - Array Decay
 - Pointer Arithmetic
 - Array Exercises
- Strings
 - String Exercises
- Reading Input
- Memory Allocation
 - Malloc
 - Realloc
 - Calloc
 - Memory Exercises
- Optional Exercises

Pointers

We are about to learn arguably the most fundamental part of **C**: pointers. They may seem scary but we'll soon see that they're not *actually* that bad!

Memory Addresses: How Variables Are Stored

We know that variables store values for us; but what does this actually look like, in the memory of our computers?

Let's think about the following variable initialisations:

```
char a = 5;  
char b = 14;
```

These variables correspond to the following address table:

Memory Address	Value Stored
6422039	5
6422040	14

We can see that the variable **a** has been stored at 6422039, and that variable **b** has been stored at 6422040 (the next **memory address**). These **memory addresses** tell you where you'd find the variables in all of the RAM your computer is currently using so they tend to be fairly large values.

Memory addresses are actually in **binary** (which uses only 2 symbols: 0, 1) and converting to **denary** (our numbering system which uses 10 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9) is a bit of a pain, so we don't usually represent addresses with denary. However, because of the large values that memory addresses tend to be, we can't exactly read them in their binary form since you get values like this: 11000011111111000010111 (6422039 in binary).

Therefore, we typically represent addresses using **hexadecimal** (which uses 16 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F) which is both more compact than binary and, unlike denary, can be easily converted to and from binary.

Our memory addresses in various different bases:

Denary	Binary	Hexadecimal (what is commonly used!)
6422039	11000011111111000010111	61FE17
6422040	11000011111111000011000	61FE18

The Point

It turns out that we can actually access the memory address of a variable in **C**, with the **address** operator (&). This returns a value which is the memory address of a variable so this value is said to **point** to that variable. This value is a **pointer**! Pointers have their own format specifier **%p** (**p** for **p**ointer) which we'll use to inspect some pointers:

```
char a = 5;
char b = 14;

printf("%p\n", &a); // 61FE17
printf("%p\n", &b); // 61FE18
```

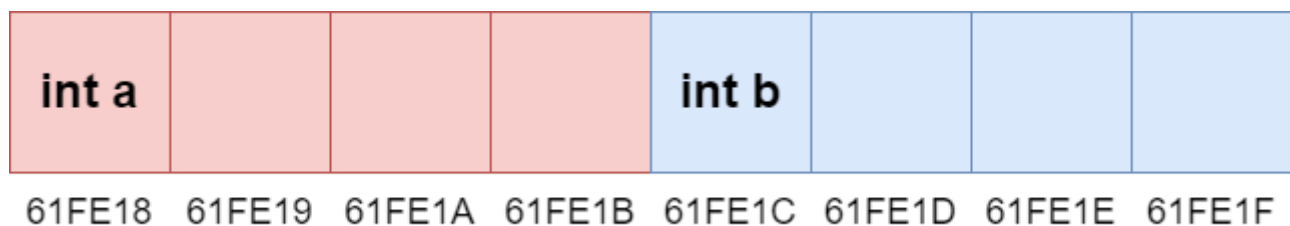
So we know variable **a** is stored at memory address **61FE17**, and variable **b** is stored at memory address **61FE18**.

So far, we've only been dealing with pointers to **char** which is convenient since each char takes up **1 byte** in most machines, and each bit of memory (that has an address) stores **1 byte** of memory; what about ints which typically take up **4 bytes**? How are these stored in memory?

```
int a = 7;
int b = 22;

printf("%p\n", &a); // 61FE18
printf("%p\n", &b); // 61FE1C
```

This can be visualised:



Red cells are taken up by int a, blue cells are taken up by int b

Since each cell (bit of memory) can store 1 byte and ints take up 4 bytes, each int covers 4 cells. The pointer contains the memory address of the first cell; when you check the contents of a variable using its pointer, you have to keep in mind the number of bytes it takes up so you know how many cells to check.

We can also initialise variables for pointers which require:

1. the **type** of value it's pointing to
2. a **memory address** (the variable's value)
3. a **name** for the variable
4. an asterisk ***** to tell **C** that it's a pointer

type* **variableName** = **&aVariable**;

Of course, you can opt to just *declare* the pointer and not give it a value but this will lead to undefined behaviour.

Interestingly, the position of the `*` doesn't matter as long as it's between the type and the name of the pointer. All of the following are valid pointer declarations:

```
char* a, char * b, char *c;
```

I would personally recommend the first one: putting the `*` next to the type makes it obvious that it's related to the type. For example, that `char*` is a type that is different from the type `char`.

Dereferencing Pointers

We can access the value pointed to by the pointer by using the **dereferencing** operator (`*`); ***don't get confused*** by the fact that `*` is used both in the **type** when you're declaring a variable and as the dereferencing operator - they are completely different uses of the symbol!

```
int a = 51;
int* pointerToA = &a;
int b = *pointerToA; // the pointer "pointerToA" is said to be dereferenced here
```

So we can access the value stored by pointers by dereferencing the pointer, and actually manipulate it as well:

```
int a = 3;
int* pointerToA = &a;

(*pointerToA)++; // updating a

printf("%d\n", a); // 4
```

Recall that you do the operations inside the brackets first!

The Null Pointer

There's a special kind of pointer that points to no memory address called the **null** pointer. This value is special because any type of pointer can be set to **null**!

```
int* p1 = NULL;
char* p2 = NULL;
short* p3 = NULL;
```

Note that the definition for the null pointer lives inside

Pointer Exercises

- Hex and Binary exercise (explain how to use)

Functions

If you want to write good code, then you want to write organised code. You want to write code that easily clicks into place with other code like lego, and can be easily reused.

This is called **adaptability**: how easily and quickly you can modify your code to introduce new features.

We can organise our code more with **functions**, which we've already been using (**printf** is a function)! A function is a block of code that we can execute by **calling** the function, with some properties:

- you can pass values to the function by calling the function with **arguments**; think about *printf* - it wouldn't be very useful if you couldn't pass it the values you want to output
- the function can **return** a *single* value to its caller

The syntax of declaring a function:

```
typeOfReturnValue functionName(arg1, arg2, ...){  
    // Some statements  
}
```

An example of a simple function:

```
int sumTwoInts(int a, int b){  
    int result = a + b;  
    return result; // result matches return type of sumTwoInts (int)  
}
```

Or even simpler:

```
int sumTwoInts(int a, int b){  
    return a + b;  
}
```

Note that **a**, **b** here are variables and you can treat them like variables.

We can call this function like so:

```
int x = 10;  
int y = 5;  
int sum = sumTwoInts(x, y);  
  
printf("Sum of x, y: %d\n", sum); // 15  
printf("Sum of 2, 4: %d\n", sumTwoInts(2, 4)); // 6
```

Notice how we can use the return value directly (like in summing 2 and 4), or store it in a variable (like in summing 10 and 5).

Note: Sometimes people will call the variables, that take on the values of the arguments, **parameters**. So you call functions with **arguments**, and these get assigned as values to the **parameters** in the function. In other words, **parameters** are the names and **arguments** are the values.

Void

Sometimes, like in `printf`, you may not want to return something: you might want the function to just carry out some action(s) and have no reason to return anything.

You can just set the function's return type to **void**:

```
#include <stdio.h>

void shout(){
    printf("THIS FUNCTION SHOUTS THIS MESSAGE!!!!!!\n");
}

int main(){
    shout(); // trying to set a variable's value to this will error!
}
```

So we now know that `printf` is a function of return type void.

We can also use **void** to tell **C** that our function takes in no arguments; of course, we could also just leave the arguments blank:

```
void scream(){
    printf("AHHH THIS FUNCTION SHOULD HAVE NO ARGUMENTS!!!\n");
}
```

The issue with this is that we can *still* call `scream` with arguments, but **C** will discard these arguments for us:

```
int main(){
    scream(5); // still prints fine: AHHH THIS FUNCTION SHOULD HAVE NO
    ARGUMENTS!!!
}
```

If we want **C** to complain and throw a fit when we try to give it arguments, then we can use **void**:

```
void scream(void){
    printf("AHHH THIS FUNCTION SHOULD HAVE NO ARGUMENTS!!!\n");
}
```

Now we cannot compile this code if we call `scream` with arguments since **C** will complain. This is considered **good practice** because it gets the compiler to tell us when we've made a mistake, so we can catch any potential errors sooner! We already have so much hard-to-debug undefined behaviour in **C**, so let's try to make it easier on ourselves :)

Function Prototypes

Revisiting Main

Now that we know about functions, we can revisit our old friend - the **main** function.

```
int main(){  
}
```

C lets us get away with not explicitly returning an `int` for `main`, but what would it mean if we did?

- Returning **0** signals that the program exited correctly
- Returning a **non-zero** value signals the program exited abnormally (if something goes wrong)

Try to run the following program, which exits abnormally:

```
#include <stdio.h>  
  
int main(){  
    printf("oopsie!\n");  
    return -1;  
}
```

It should output "oopsie!" and then signal to you that something went wrong.

Note: **C** compilers used to require you to explicitly return `0` from `main` and so there will be some compilers that error if you don't, so it's considered good practice to **use return** in your `main` function. Often, `-1` is used to signal incorrect execution.

Did you know **main** can also receive arguments from the command-line?

```
int main(int argc, char* argv){  
    printf("Number of arguments: %d\n\n", argc);  
  
    for (int i = 0; i < argc; i++){  
        printf("%s\n", argv[i]);  
    }  
}
```

The **main** function gets:

1. number of arguments which we call **argc** for **argument count**
2. an array of pointers to characters, which we'll soon see is an array of strings (in the *Revisiting Arrays* section)

Right now, we won't question *how* pointers to characters are entire strings and just use **%s** to print them.

One confusing way of writing **main** is with **void** in place of arguments, because

Function Exercises 1

1. In higher-level languages, you have to return a value from a non-void function. In **C**, you can return nothing from a non-void function like so:

```
int div(int a, int b){
    if(b != 0){
        return a/b;
    }
}
```

No value is returned from div when b=0

Can you see what happens when div returns nothing? Can you output its return value?

2. Read the solution to Exercise 1 ([cFiles/exerciseSolutions/FunctionExercises1/Exercise1.c](#))
3. Using arguments to the main function, write a program that takes a first name and second name from the command-line and outputs "Your name is insert-first-name insert-last-name!"
- 4.

Passing Around Pointers

When you pass values to a function, you are just giving it the values. Editing those values inside the function will do nothing to any variables that hold those values. For example:

```
void print1GreaterN(int n){
    n++;
    printf("One greater than n: %d\n", n);
}

int main(){
    int x = 5;
    print1GreaterN(x);
    printf("%d\n", x); // still 5!

    return 0;
}
```


Inside of `print1GreaterThanN`, we edit the argument (value) we're given so that we output `6`. But the original variable `x` still holds the value `5` so what's going on here? The variable `n` and the variable `x` actually point to different places in memory, and the value that `n` holds is a copy of the value that `x` holds (at the start of the function)!

In programming, this is referred to as **passing by value**. What if we *did* want functions to edit the value of a variable? Well, we can do this by passing a pointer! This is called **passing by reference** since we're passing a pointer which points to, or **references**, the variable.

```
void print1GreaterThanN(int* n){
    (*n)++; // be very careful with brackets here! *n++ will increment the memory
    address of the pointer
    printf("One greater than n: %d\n", *n);
}

int main(){
    int x = 5;
    print1GreaterThanN(&x);
    printf("%d\n", x); // now 6!

    return 0;
}
```

Since we can get functions to edit values, we can get functions to give us data. We can already do this using **return**, but that limits us to only one value as functions can only return *one value* (or none). By passing pointers to functions, we can get them to set values.

```
int positiveNumberCheck(int n, int* isEven, int* isGreater, int* isFavourite){
    if(n <= 0){
        return 0; // could not give properties because number is not a strictly
        positive number
    }

    if(n%2 == 0){
        *isEven = 1; // recall 1 is considered 'true'
    }

    if(n > 10){
        *isGreater = 1;
    }

    if(n == 37){ // my favourite for now...
        *isFavourite = 1;
    }

    return 1; // ran successfully!
}

int main(){
```

```

int n = 37;
int isEven = 0; // automatically set to 'false'
int isGreaterThanTen = 0;
int isMyFavouriteNumber = 0;

int succ = positiveNumberCheck(n, &isEven, &isGreaterThanTen,
&isMyFavouriteNumber); // succ is short for success

if(succ){
    printf("%d %d %d \n", isEven, isGreaterThanTen, isMyFavouriteNumber);
}else{
    printf("Cannot give properties for n = %d\n", n);
}
}

```

Pointers to Functions

Function Exercises 2

1. In [Passing Pointers](#), there is an example of passing by value and we said that the variables **x**, **n** pointed to different places in memory. Can you edit the code to output the memory addresses of **x** and **n**, and run it to convince yourself?

Revisiting Arrays

We saw last session that arrays are a collection of values that are stored in **contiguous** memory locations (aka the memory locations are right next to each other).

Arrays and Pointers

This sounds familiar... we know that a pointer to an int points to the first byte and the three (or more depending on the machine) other bytes are stored immediately after (or in contiguous memory).

In fact, what actually is an array? Let's print it!

```

int anArray[] = {2, 3, 5, 7, 11};
printf("%d\n", anArray); // 6422016

```

That's interesting, this looks like a memory location (that we've printed using **%d** so displayed as a denary number). Let's try printing it as a pointer, and also printing the memory addresses of values in the array.

```

printf("Array: %p\n", anArray); // 61FE00

for(int i = 0; i < 5; i++){
    printf("Element at location: %p\n", &anArray[i]); // 61FE00, 61FE04, 61FE08,
61FE0C, 61FE10
}

```

So the pointer to the first element points to the same address as the array! Hold on though, surely they're not the exact same because we can index arrays using `[]`.

```
int anArray[] = {2, 3, 5, 7, 11};
int* p = &anArray[0];

printf("%d\n", p[0]); // 2

for(int i = 1; i < 5; i++){
    printf("%d\n", p[i]); // 3, 5, 7, 11
}

return 0;
```

We can index pointers as well? Are arrays just pointers??? Not quite, but close. There is one **key** difference between arrays and pointers.

Array Decay

Call `sizeof` on an array, and call `sizeof` on a pointer to the first element in the array.

```
int anArray[] = {2, 3, 5, 7, 11};
int* p = &anArray[0];

// one key difference: sizeof

printf("sizeof array: %d\n", sizeof(anArray)); // 20
printf("sizeof pointer: %d\n", sizeof(p)); // 8
```

You will see that `sizeof(array)` tells us the number of bytes the array takes up, whereas `sizeof(pointer)` tells us the number of bytes that the pointer itself takes up (so how many bytes needed to store the memory address).

If we only have the pointer, then we can't find out how many elements there are in the array. So we can't index or loop through the elements of the array since we can't tell what's in the array or not, beyond the first element.

So if we wanted to use a pointer as an array, we would have to keep track of the size of the array in another variable. This is important because there is a key situation where an array becomes just a pointer, and loses information about the size of the array: this is **array decay** and it happens when you pass an array to a function.

```
void passingArray(int passedArray[]){
    printf("%d\n", sizeof(passedArray)); // 8 on my machine
}

int main(){
```

```

int ourArray[] = {1, 3, 6, 10, 15};
printf("%d\n", sizeof(ourArray)); // 20 on my machine

passingArray(ourArray);

return 0;
}

```

We can see that `ourArray` decays into `passedArray` and loses information about its size; this happens because `passedArray` is actually just the pointer to the first element, so its size is the size of a pointer on my machine (which is 8 bytes). Due to array decay, we have to provide the length of an array whenever we pass it through a function.

Note: Using `int array[]` and `int* array` is equivalent for pointers. Since an array decays into a pointer when it's passed to a function, we can use either form.

```

void passingArray(int* passedArray, int length){
    printf("Last element: %d\n", passedArray[length-1]);
}

int main(){
    int ourArray[] = {1, 3, 6, 10, 15};
    printf("%d\n", sizeof(ourArray)); // 20 on my machine

    passingArray(ourArray, 5);

    return 0;
}

```

Pointer Arithmetic

Arithmetic! The mathematicians have got us in their grasp again!

- Pointer arithmetic

Array Exercises

1. Write a function to print all the values in an array. Can you make it so it prints all the contents on one line?

Strings

String Exercises

1. Write a function to reverse a string (this function should be of type void).
2. Write a function to convert a string to all upper-case character (**hint:** look up the range of values for lower-case characters in ascii)

Input

- a mid-way (maybe end-level) exercise: create a function for processing int input safely

USEFUL ARTICLE: <https://www.geeksforgeeks.org/all-forms-of-formatted-scanf-in-c/>

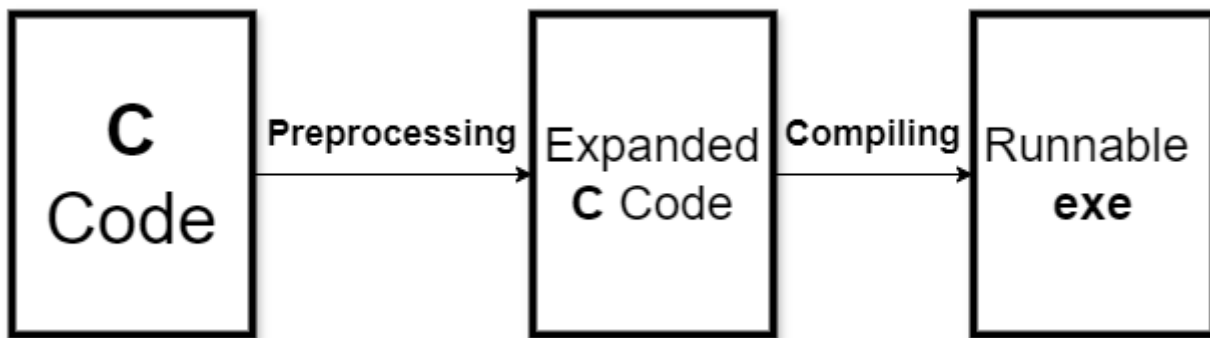
USEFUL EXERCISE: [https://www.hackerrank.com/challenges/playing-with-characters/problem?](https://www.hackerrank.com/challenges/playing-with-characters/problem?isFullScreen=true)

isFullScreen=true

- Recommend Hackerrank?

Preprocessing and Macros

In Session 1, we briefly mentioned preprocessor directives which is what `#include` is. To understand them, we need to discuss **preprocessing**.



Note that there are more stages you can split building a C file into, but this is as far as we'll break it down in this course

The **define** preprocessor directive lets us define a macro by telling the compiler to replace text in your code (the macro) with other text (the text that is defined by your macro):

#define thingToReplace whatToReplaceWith

Here 'thingToReplace' is also the name of your macro

It can be useful to define **true** and **false** in this way:

```
#define true 1
#define false 0
```

Everytime you write 'true', it will be replaced with '1' after preprocessing

You might ask: **why use macros?**

There are a couple reasons but one is **performance**

- no memory used for variable
- no cost from using the variable, or calling a function

-
- mention it'll be useful for next session (sneak peak?)

Memory Allocation

Arrays have a fixed size that we have to specify in the **C** code, but sometimes we want to grow our array according to a variable's value (dynamic size). For example, if we want to store a list of **n** numbers where **n** is inputted by a user. Luckily, the standard library (`<stdlib.h>`) has got us covered with `malloc`, `realloc`, and `calloc`!

Malloc

We can use **malloc**! We can **allocate** memory with **malloc**; we just need to tell it how many bytes we want and it'll give us a pointer to that block of memory. So if we want to store 20 characters, then we'll need to allocate (probably) 20 bytes; if we want to store 20 ints, we'll need to allocate (probably) 80 bytes. We don't have to rely on chars and ints being a byte or 4 bytes each though, since we know how to get their size from Session 1:

```
int nItems = 20;
int* p = malloc(nItems * sizeof(int));
```

Hang on though, how can `malloc` return a pointer that works for any type we want a block of memory for? This is because `malloc` returns a pointer of a special type: **void*** which begs the question, what does a pointer to **void** mean? It means that we don't specify what type of data it points to, which makes sense since `malloc` should just give us the memory address to the start of the block of memory. This gives us the flexibility to convert a **void*** pointer to an **int*** pointer, **char*** pointer, **short*** pointer etc.

We could easily do:

```
char* cP = malloc(10 * sizeof(int)); // 40 bytes on my machine
```

Which, assuming an `int` takes up 4 times the space of a `char` (it does on my machine), is a block of memory that can store 40 chars.

- `malloc`, `calloc`, `realloc`

Next Session...

Well done on completing Session 2! You're almost through to the end; just one more session to go...

If you're not tired of **C** by then, then remember there's a bonus session on the 30th October! Hopefully, I'll **C** you there!

- maybe do static keyword?
- do structs and unions!

- Makefiles (and multiple files) -> header files? More complicated macros!

Optional Exercises

Not sure what to put here yet

Acknowledgements

Thanks to ...

Originally created by Edward Denton.