INTERRUPT LABS

C Intro Pack

Section 3: C-ms it's Almost Over

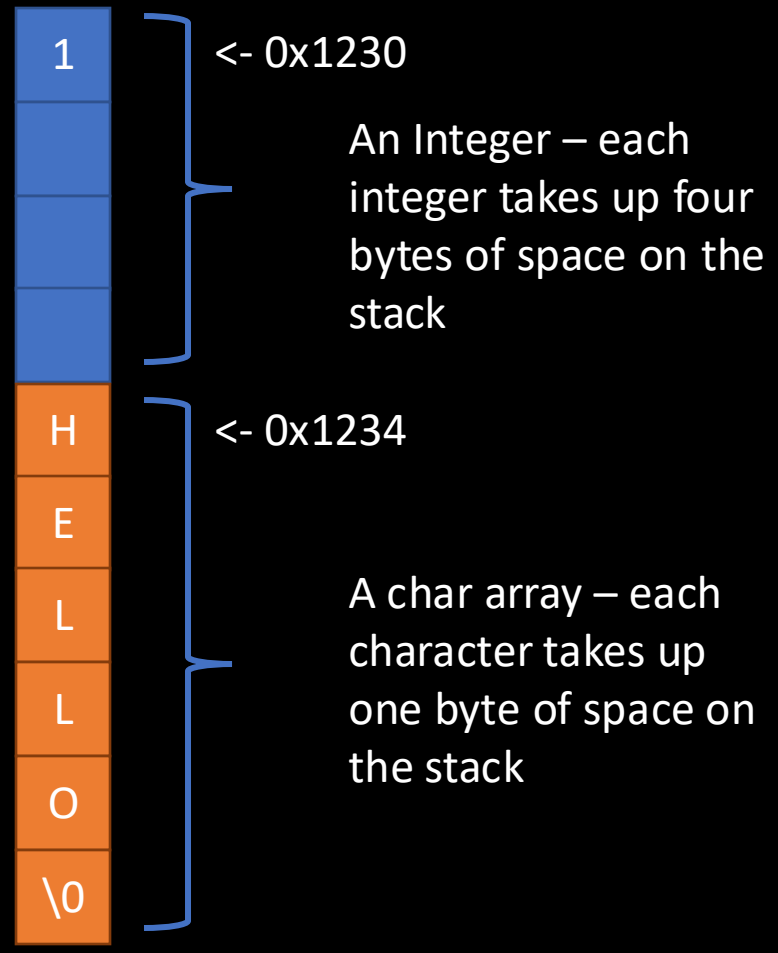
Oh God, Pointers!

- Well, we had to get here eventually. There's no escaping pointers in C.
- Pointers are well known to strike fear into even the most hardened high-level programmer, but once you've learnt their uses and functionality, you'll realise just how useful they are in C.
- The next few slides will cover just getting used to pointers and how to use them – we'll take things slowly and one step at a time.
- By the end of this you'll be far more comfortable with the concept of pointers, and you'll have used them in several programs.
- Let's dive in…

Let the Deep Dive Begin

- To explain the world of pointers, we need to start at the basics – variables.
- When we declare a variable, that variable's value must be stored somewhere in memory for us to be able to access it and change it – otherwise it's not a very good variable…
- But where is a variable stored? It's stored on the stack!

- Think of the stack like a series of Lego blocks stacked on top of each other… Hey look there's a diagram to the right!
  - The stack can grow either up or down – Computer Scientists can never agree on anything but for our sanity we'll just say the stack grows down.
  - As you can see, each variable is a different colour block, and each variable type takes up a different amount of space.
  - The location of each Lego block is represented by its memory address – this is generally a very long and scary looking hexadecimal number, but all it does is simply tell us where in memory its sitting
  - Notice how the series of characters looks – it's what a string looks like in memory!



Sizing Things Up

- Firstly, let's look at how we can get the size of a variable in memory.
- This is easily done with the `sizeof` function!
  - The `sizeof` function returns the size of a variable or variable type.
  - Copy the program to the right to see for yourself!
- Wait a second, what's going on with the size of the struct??
  - No that's not a mistake, the sizes of the struct don't match up.
  - Try a little experiment – add and remove one or two of the `int` variables.
  - You'll notice that the size of the struct goes up and down in increments of 4.
  - This is because of alignment – on the stack things need to be aligned to multiples of 4 bytes.
  - Since our array is 5 + (4 x 4) = 21, the compiler adds three bytes of padding to get it to a multiple of four!
  - It's not something you'll have to worry about with C, but it's worth knowing if you enter the dark world of assembly.

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

struct example
{
    char array[5];
    int a;
    int b;
    int c;
    int d;
};

int main()
{
    char c = 'c';
    int i = 12;
    float pi = 3.14;
    double e = 2.718281828;

    puts("*** SIZES ***\n");
    printf("Char   = %lu\n", sizeof(c));
    printf("Int    = %lu\n", sizeof(i));
    printf("Float  = %lu\n", sizeof(pi));
    printf("Double = %lu\n", sizeof(e));
    printf("Struct = %lu\n", sizeof(struct example));

    return(0);
}
```

Location, Location, Location

- Next, let's look at how we find the location of variables in C.
- This is done using the `&` operator!
  - Think of `&` as meaning "Address of".
- Huh, that was a short slide – let's copy down and run the example quick before a task appears…
  - It's the same program as before – the only thing that's changed are the `printf` statements and the struct.
  - We declare the struct to place it in memory so we can get an address!

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

struct example
{
    char array[5];
    int a;
    int b;
    int c;
    int d;
};

int main()
{
    char c = 'c';
    int i = 12;
    float pi = 3.14;
    double e = 2.718281828;

    struct example address;

    puts("*** LOCATIONS ***\n");
    printf("Char   = %p\n", &c);
    printf("Int    = %p\n", &i);
    printf("Float  = %p\n", &pi);
    printf("Double = %p\n", &e);
    printf("Struct = %p\n", &address);

    return(0);
}
```

Task Time!

» Darn, not fast enough! These should be simple though so crack on with them!

1. Create a program which reads in a string given by the user and tells them the following:
   - Where in memory its stored.
   - The size of the string in memory.

   Run it a few times with a same string and note what happens to the address each time.
2. Modify your program so the address of each character in the array is printed out.

Bonus: Modify the program so only the addresses of the first and last characters of the string are printed out.

What's the Point-er?

- Before we continue, read and memorise this sentence.
  - A pointer is a variable which stores a memory location.
- As stated above, despite the misleading name, a pointer is simply a variable type which holds a memory location.
- Let's look at an example: Copy that code there, compile, and run.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      int *pointer; // Create a pointer variable
7
8      int number = 1234;  // Create a variable
9      pointer = &number;  // Store the address of that variable
10
11     puts("*** INFO ABOUT 'name' ***");
12     printf(" Size:\t%ld\n", sizeof(number));
13     printf("Value:\t%d\n", number);
14     printf("Addr.:\t%p\n\n", &number);
15
16     puts("*** INFO ABOUT 'pointer' ***");
17     printf("\t Value: %p\n", pointer);
18     printf("Value at addr.:\t%d\n", *pointer);
19
20     return(0);
21 }
```

- Just like any other variable, the first step is to declare the pointer variable.
  - We do this using the * symbol – look at line 6.
  - It goes between the variable type and name, just before the name.
- Next, you assign it a value – just like a normal variable.
  - We do this by taking the address of a variable using &.
  - Then like any other variable, we use = to set the pointer to the address.
- We can read two values from the pointer.
  - To read the address it holds, simply type the pointer variable name.
  - To read the value it holds, put an asterisk before the pointer variable name.

Point-er of Order

- Pointers can also be used to set values at addresses of variables, just like variables can set the values of pointers.
- Have a quick look at the example code over there. Notice how we declare the value of the pointer in two different ways:
- The first print statement sets a pointer to the address of the char variable storing the letter '`A`'. It prints out the value at that address.
- The second print statement sets a pointer to the address of an empty char variable.
  - This section sets the value at the address held by the pointer to the character the user enters.
  - Since this address is the address of the empty variable '`b`', line 15 is the same as writing `scanf("1c", &b);`.
  - That's why we can use the variable `b` in the print statement after!

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char *pointer;
7      char a = 'A', b;
8
9      pointer = &a;
10     printf("Variable 'a' stores %c\n\n", *pointer);
11
12     pointer = &b;
13     puts("The variable 'b' is currently empty.");
14     printf("Enter a character: ");
15     scanf("%1c", pointer);
16     printf("Now b is equal to %c\n", b);
17
18     return(0);
19 }
```

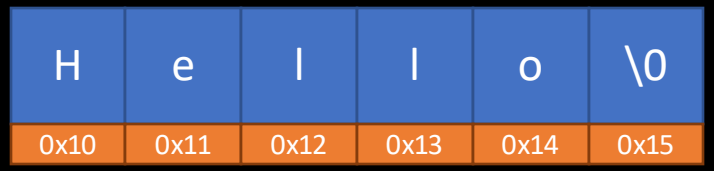
Point-er in the Right Direction

- Remember in the last slide pack how we said that arrays were pointers in disguise? You know, the slide with the meme on it?
- It's time to prove that arrays are simply pointers in disguise: Copy the code to the right, compile, and run.
- Notice how both print statements produce the same output – but wait, only one of them is getting the address with &!
  - You can only print the address of something without the & if the variable is a pointer… Which means…
  - The array is considered a pointer by C!
- We can prove this further with the commented line – uncomment it and see what happens.
  - The compiler will complain as soon as you try and print the address of a variable without the &.
  - Again, this must mean the array is a pointer!

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

int main()
{
    char array[5] = {'H', 'e', 'l', 'l', 'o'};
    int random = 3;

    printf("The array is at address: %p\n", &array);
    printf("Equally it's at address: %p\n", array);

    // Uncomment the line below for an angry compiler!
    //printf("The variable is at address %p\n", random);

    return(0);
}
```

Getting my Point-er cross

- So cool, arrays are pointers, but what does this mean? Well, now we can start getting very funky with data and variables!
- Because pointers hold memory addresses, and memory addresses are just numbers in hexadecimal format, we can start using maths to access array elements.
- Look at the diagram below: It shows a character array in memory, starting at `0x10`. The variable `p` holds the address of the first element.



- If we add one to `p` by running `p++;`, and `p = 0x10`, what address will `p` now point to? The next element in the array!
  - It might sound daft, but if you're struggling, try following through the array with your finger adding and subtracting different amounts.

Task Time!

» Getting used to arrays being pointers and manipulating them is important, so let's do an example to solidify everything we've covered.

1. Create a program which uses pointers to fill a char array with all the uppercase letters of the alphabet.
   - Your array should have a size of 27
   - BIG HINT: If you're really struggling, your program should have a line similar to `*pointer = i + 'A';` sat in some form of loop.
2. Print out the array once populated using pointer notation.

Printing Things Like Crazy

- During the last task slide, you will have had to increment your pointer and read the value it was pointing to. You probably did this in two steps…
- Well, we could've done it in one step, and to prove it I'm going to get you printing things in funky ways – copy that code over there, compile, and run!

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

int main()
{
    char *string = "Printing this was odd...\n";

    while(putchar(*string++));

    return(0);
}
```

- Wait a second – a string with no array notation?!
  - Well…
  - Remember how an array is a pointer in disguise, and a string is a char array in disguise?
  - Combining these two points leads us to the fact that a string is a char pointer in disguise!
- You might be scratching your head as to why that printed out the string with no issues – the magic lies in the statement in the `putchar` function.
  - `*string` will print out whatever value is stored at the address in `string` .
  - The `++` will increment the address in string to the next one – the next letter in the string.
  - When it gets to the end of the string, the while function will evaluate to `\0` – or `NULL`. This quits the `while` loop as it evaluates to `while(0)` !

Handy Table – Pointer Increments

Expression	Address `p`	Value `*p`
`*p++`	Incremented after being read	Unchanged
`*(p++)`	Incremented after being read	Unchanged
`(*p)++`	Unchanged	Incremented after being read
`*++p`	Incremented before being read	Unchanged
`*(++p)`	Incremented before being read	Unchanged
`++*p`	Unchanged	Incremented after being read
`++(*p)`	Unchanged	Incremented after being read

Point me to a Pointer

- Much like how arrays can be multidimensional, pointers can be multi-pointermensional!
- This is generally done with the dreaded double pointer notation, but before we go into that copy the example over there, compile, and run.

- In the first section, we declare an array of char pointers – whilst it does contain strings, remember that strings are simply character arrays.
- The first `for` loop uses notation we've seen before.
  - We loop through the array one pointer at a time.
  - We then use puts to print the string there!
- The second `for` loop is where things get a little funky – let's break down line 22!
  - `pointers + x` – The variable pointers contains a pointer! `x` increments along the string the pointer points to.
  - `*(pointers + x)` – This is the content of the address `pointers + x`. You've seen this before!
  - `**(pointers + x)` – This is a pointer to the previous pointer. In other words, it points to a character within the pointer `pointers + x`.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      char *pointers[] = {
7          "German",
8          "English",
9          "Wiredhaired",
10         "Weimaraner",
11         "Italiano",
12         "Vizsla"
13     };
14
15     for(int x = 0; x < 6; x++)
16         puts(pointers[x]);
17
18     puts("\n**********\n");
19
20     for(int x = 0; x < 6; x++)
21     {
22         putchar(**(pointers + x));
23         putchar('\n');
24     }
25
26     return(0);
27 }
```

Handy Table – Pointer Pointers

- Double pointer notation is a scary topic, but thankfully you're very unlikely to encounter it other than when dealing with pointer arrays.
- Nonetheless, here's a table that should explain things – give it a read through before moving to the next slide

Pointer Notation	Array Notation	Description
`**ptr`	`*array[]`	Declares an array of pointers
`*ptr`	`array[0]`	The address of the first element in the array
`(*ptr + 0)`	`array[0]`	The address of the first element in the array
`**ptr`	`array[0][0]`	The first element of the first pointer in an array / The first character of the first string
`**(ptr + 1)`	`array[1][0]`	The first element of the second pointer in an array / The first character of the second string
`*(*(ptr + 1))`	`array[1][0]`	The first element of the second pointer in an array / The first character of the second string
`*(*(ptr + a) + b)`	`array[a][b]`	Element b of pointer a
`**(ptr + a) + b`	`array[a][0] + b`	The same as the above, but in a more confusing notation – only use this to obfuscate your code

Pointer Goes In…

- Like any variable, you can pass a pointer to a function – this is often even more useful than passing a measly variable into the function!
- Because you pass an address in, information can be manipulated without being returned.
- Copy the small source code over there, compile, and run to see what I mean…

- We declare a pointer and variable as normal, then print the original number.
- We then pass the pointer to the function like any other variable.
  - Remember how variables need somewhere to go once in a function?
  - We declare a pointer in the function to hold the address passed in!
- The function manipulates the value at the address rather than locally – this means the change is global!
- This is why we can reuse the variable in print statements, and it changes.

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

void half(float *addr);

int main()
{
    float num;
    float *ptr = &num;

    printf("Enter a number: ");
    scanf("%f", ptr);

    printf("You entered the number %.2f\n", num);
    half(ptr);
    printf("The number divided by two is %.2f\n", num);

    return(0);
}

void half(float *addr)
{
    *addr = *addr / 2;
}
```

…Pointer Comes Out!

- What goes in can come out, and the same applies for pointers in functions too!
- Copy that example there, compile, and have a play around with it – we'll explain it below!

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define BUFF 32
5
6  char *reversi(char *input);
7
8  int main()
9  {
10     char string[BUFF];
11
12     printf("Type some text: ");
13     fgets(string, BUFF - 2, stdin);
14
15     puts(reversi(string));
16
17     return(0);
18 }
19
20 char *reversi(char *input)
21 {
22     static char output[BUFF];
23     char *i = input, *o = output;
24
25     while(*i++ != '\n');
26     i--;
27
28     while(i >= input)
29         *o++ = *i--;
30     *o = '\0';
31
32     return(output);
33 }
```

- Firstly, we prototype the function. Since it's going to return a char pointer, we must set it to be of the type `char *`.
- The program starts by scanning in some user input into a buffer – it's vital we copy two less than the size of the buffer for reasons which will be explained later…
- We then pass the pointer to the string to the function `reversi`.
  - The function starts by declaring a static output variable – it must be static so that its value remains after the function has returned. If we don't, the computer will consider that memory free again and may overwrite our variable – yikes!
  - We then copy the pointers to our input and output to preserve their original values.
  - Line 25 increments the pointer `i` until it holds the address of the newline of the string we entered. We then subtract one, so it points to the last character of our input.
  - Lines 28 – 29 do two main things. It sets the value at address `o` to the value at address `i`, then increments `o` and decrements `i`, repeating until `i` reaches the start of the string.
  - It then sets the last character of our output array to a null character – remember all strings must be terminated with a null character!
- We finally return the address of the start of the output string, then print with puts!

Task Time!

- Hopefully pointers are starting to make a bit more sense now – time for some tasks to really solidify your knowledge!

1. Modify the program from slide 14 so that the string is printed out using a `putchar` function. Bonus points if you can get it all contained in a single `while` statement!
2. Create a program which takes in a string from the user. The string will then be passed to a function, which takes the string and capitalises all vowels. The function should return a pointer to the modified string. Print out the modified string.

Bonus: Create a program which takes a pointer array and sorts the elements by length, from shortest to longest. Print out the pointer array using a single `while(putchar( ... ))` style statement.

Extra Bonus: Create your own versions of the `strcmp` and `strcpy` functions. Call them `strcmpPtr` and `strcpyPtr`. Brownie points for doing this one.

File-nally, Easy Stuff!

- Reading and writing to files is commonplace in programming languages, and C is no different in its love for permanent storage.
- We'll start with writing to a file – copy the code there, compile, and run.

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

int main()
{
    FILE *fh;

    fh = fopen("example.txt", "w");
    if(!fh)
    {
        puts("ERROR - Can't open file!");
        return(1);
    }

    fprintf(fh, "Woah a text file\n");
    fclose(fh);

    puts("All done!");
    return(0);
}
```

- We start every file manipulation by creating a `FILE` pointer – generally we call this `fh` but you can call it whatever you'd like.
- We then set the file pointer to point at our file with `fopen`. It has the form `fopen(<file>, <access_mode>)`.
  - `file` = The name/path of the file we want to open.
  - `access_mode` = How we want to access the file. Since we want to write to it, we use `w`!
  - There's a handy dandy table coming up explaining the different access modes…
- We do a quick check to see if the file opened correctly, then print text to the file using `fprintf`.
  - The first argument is the file header – the file we want to write to.
  - The second argument is the text we want to print to the file!
- Finally, we close the file using `fclose`.

Handy Dandy Table: File Access Codes

Mode	File Open For…	Creates File?	Notes
`"a"`	Appending	Yes	Adds to the end of an existing file – creates a new file if the one specified doesn't exist
`"a+"`	Appending & Reading	Yes	Data is added to the end of the file
`"r"`	Reading	No	If the file doesn't exist, `fopen` throws an error
`"r+"`	Reading & Writing	No	If the file doesn't exist, `fopen` throws an error
`"w"`	Writing	Yes	Any data in the file is overwritten if the file already exists
`"w+"`	Writing & Reading	Yes	Any data in the file is overwritten if the file already exists

Nothing Like a Good Read

- Obviously, where we have written a file, we often want to read what's inside a file too.
- This follows a very similar set of steps to writing to a file, so give that code a copy, compile, and a run!

- This time around, we use a file access mode of `r` – for read!
- `fgetc` reads one character at a time from a file – we store the character in an int variable.
  - REMEMBER: Characters in C can be either characters or numbers. This is why the int variable works!
  - For each character we read, we print it to the screen with `putchar` .
  - We read until we reach the end of the file, written as EOF.

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

int main()
{
    FILE *fh;
    int ch;

    fh = fopen("example.txt", "r");
    if(!fh)
    {
        puts("ERROR - Can't open file!");
        return(1);
    }

    while((ch = fgetc(fh)) != EOF)
        putchar(ch);
    putchar('\n');

    puts("All done!");
    return(0);
}
```

Reading by Line

- Whilst getting text from a file character by character works, wouldn't it be more efficient to read in a line at a time?
- This is where `fgetc`'s sibling comes in – `fgets`!
- Copy down the example there, compile, and run. Make sure your file has multiple lines in it.
- We start by declaring a char buffer to store our strings in – this needs to be long enough to store the longest string in the file!
- After the whole file opening routine, we then read in each line using `fgets` .
  - You've used `fgets` before to read text in from the terminal.
  - The syntax is the same, just this time we're reading from a file instead of `stdin`!
  - After each line is read, it is saved to the buffer and read. The line in the buffer is overwritten with the next line in the text file with each loop .

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

#define BUFF 256

int main()
{
    FILE *fh;
    char input[BUFF];

    fh = fopen("example.txt", "r");
    if(!fh)
    {
        puts("FILE ERROR");
        return(1);
    }

    while(fgets(input, BUFF, fh))
        printf("%s", input);

    fclose(fh);
    return(0);
}
```

Task Time!

- File manipulation is a very useful thing in C, so let's make sure you've got the basics down with two simple tasks!

1. Create a program which asks a user to enter their name. A greeting should be printed to them, and their inputted name should be appended to a file.
2. Add code to your program so that, if the user has entered their name before, the text "We've met before!" is displayed before the greeting is printed.

Get In the Bin

- Often, we'll want to write data to files which isn't just text. In most languages, this is accomplished through writing to the file in binary – and C is no different!
- We'll be using a simple example for this slide and the next, so get that example written down.

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

int main()
{
    FILE *fh;
    int cord[5];

    fh = fopen("cords.db", "w");
    if(!fh)
    {
        puts("FILE ERROR");
        exit(1);
    }

    for(int i = 0; i < 5; i++)
    {
        printf("Enter co-ordinate %d/5: ", i + 1);
        scanf("%d", &cord[i]);
    }

    fwrite(cord, sizeof(int), 5, fh);
    fclose(fh);
    return(0);
}
```

- We create an integer array, into which we read five integer values given by the user.
- We then use a new function called fwrite – this writes binary data to a file!
  - Form = `fwrite(<var_ptr>, <size_var>, <num_elems>, <file_header>)`
  - `var_ptr` = A pointer/address of the data you want to write to the file.
  - `size_var` = The size of the variable type you want to write – use `sizeof` for this!
  - `num_elems` = The number of items you want to write.
  - `file_header` = A file pointer to the file you want to write to.
- Take a look at the `cords.db` file – it should contain some random characters, nothing like the nice numbers we entered.
  - Sneaky bonus task: Change the array notation in the `scanf` to use pointer notation instead.

Get Out the Bin

- It's nice to write binary data, but how do we get it back out again?
- Well, much like reading a regular file, we can read binary files too! Copy the code there – it's a slight modification of the previous program.
- Like last time, we modify the `fopen` to read the binary file rather than write to it.
- We then use `fread` to read in the file – if you look closely the arguments don't change! They're very similar.
- We then simply write out the contents of the array one at a time to get a nice score list.

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

int main()
{
    FILE *fh;
    int cord[5];

    fh = fopen("cords.db", "r");
    if(!fh)
    {
        puts("FILE ERROR");
        exit(1);
    }

    fread(cord, sizeof(int), 5, fh);
    puts("*** CO-ORDINATES ***");
    for(int i = 0; i < 5, i++)
        printf("%d: %d\n", i + 1, cord[i]);

    fclose(fh);
    return(0);
}
```

A Structured File

- Because we can now write binary data, we can now start writing more interesting things to files, like structures!
- There's an example over there, give it a copy and give it a run.

- We declare a struct as normal, then assign each field a value.
- Saving the struct is super easy, thanks to the power of `fwrite`!
  - For the arguments, we start with the address of the struct we just made – `jeb` .
  - We then give it a size of the struct, and state that there's only one of them.
  - Finally, we specify the file we want to save it to!
- Equally, reading from structs is easy too!
  - WARNING: Make sure the struct you are reading into has the same structure as the one you saved, otherwise you'll get problems.
  - WARNING 2: Always make sure you're reading into a struct variable, not a struct pointer! Otherwise, you'll be reading in addresses and things won't go well…
  - We simply use `fread` to read into the empty struct we made… How convenient!

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFF 32

int main()
{
    struct kerbal
    {
        char name[BUFF];
        char mood[BUFF];
        int id;
    };

    struct kerbal jeb;
    strcpy(jeb.name, "Jebediah");
    strcpy(jeb.mood, "Excited");
    jeb.id = 1;

    FILE *fh;
    fh = fopen("kerb.db", "w");

    if(!fh) exit(1);

    fwrite(&jeb, sizeof(struct kerbal), 1, fh);
    fclose(fh);

    struct kerbal newJeb;

    fh = fopen("kerb.db", "r");
    if(!fh) exit(1);

    puts("** KERBAL DATA **");
    while(fread(&newJeb, sizeof(struct kerbal), 1, fh))
        printf("Name: %s\nMood: %s\n  ID: %d\n",
        newJeb.name, newJeb.mood, newJeb.id);

    fclose(fh);
    return(0);
}
```

Task Time!

» Being able to save structures and binary data is super useful – time for a quick task!

1. Create a program which initialises, saves, and loads three structures. You can make the structures contain whatever elements you'd like, but they must be saved to a file.

Big Bonus: Modify the program so that it saves and loads a linked list.

Grabbing Memory

- Unlike higher level languages, C allows us to manually request memory from the Omniscient Operating System using a handy little function called `malloc`.
- The format is like so: `*p = (type *)malloc(size)`.
  - `type` = A typecast to make sure the memory returned matches our variable type.
  - `size` = The amount of storage you want to request. Generally, you'll use `sizeof(type)` to accurately get the right amount of memory!
- `malloc` will return a pointer to the memory you requested… Except if it can't allocate you memory for whatever reason.
  - If it can't allocate memory, it'll return `NULL` – always check for a `NULL` return when using malloc!
- Take a look at the program there – identify where each stage of the malloc process takes place. Give it a quick compile and run to see it in action.
  - It's highly unlikely you'll trigger the exit condition. Modern computers have so much memory that you'll rarely not have enough.
  - If you are working on an old IBM PC from the 1980s, this check is more likely to trigger… In which case your computer is about 3 seconds away from crashing…

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

int main()
{
    int *input;
    input = (int *)malloc(sizeof(int));

    if(!input) return(1);

    printf("Please enter a number: ");
    scanf("%d", input);
    printf("You entered the number %d\n", *input);

    return(0);
}
```

Grabbing Memory for Strings

- Rather than using stinky array notation for strings, we can use the elegant malloc to create space for our strings instead!
  - This is where the `sizeof` comes into its own.
  - We can use `sizeof(char) * n` to allocate memory – this will allocate n bytes of memory and is useful when you want to be very specific about how much memory is needed!

- Just a quick slide this one – copy down the example over there, compile, and run!
  - In this example, we allocate 1Kb of space – 1,024 bytes.
  - Feel free to use more or less than this!

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

#define BUFF 1024

int main()
{
    char *string;

    string  = (char *)malloc(sizeof(char) * BUFF);
    if(!string) return(1);

    printf("Enter some text: ");
    fgets(string, BUFF - 1, stdin);
    printf("\n** YOUR INPUT **\n");
    puts(string);

    return(0);
}
```

Letting That Memory Go

- Whilst it's not much of a problem on modern systems, it's good practice to return the memory you used to the Omniscient Operating System once you've finished with it.
- This is very easy to do using a function called `free`.
  - `free` takes a pointer as an argument – it then frees this memory and returns it back to the Omniscient Operating System.
- Huh, not so bad after all! Try adding the line `free(string)` after the `puts` in the previous program and see what happens.
  - Hopefully nothing happens – this means the memory was returned correctly!

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

#define BUFF 1024

int main()
{
    char *string;

    string  = (char *)malloc(sizeof(char) * BUFF);
    if(!string) return(1);

    printf("Enter some text: ");
    fgets(string, BUFF - 1, stdin);
    printf("\n** YOUR INPUT **\n");
    puts(string);

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

Task Time!

» Just a quick one! Time to make sure if you know about memory munching functions.

1. Create a program which uses `malloc` to allocate enough space for a string input from the user.
2. Create a second char buffer using `malloc`. Copy the user input into that buffer. Print out the string from the copied buffer.

The Struct-ure of Malloc

- We can use malloc to great effect with variables, but where malloc truly shines is with struct variables.
- We can create struct pointers just like ordinary struct variables – what changes is their notation…
  - `struct.variable` = The OG way – you've used this before if you've done the other slide packs.
  - `struct->variable` = The pointer way – if your struct is a pointer, you reference elements using the funky arrow notation!
- Take a look at the example to the right – copy, compile, and run.
  - We start off by defining the `kerbal` struct like normal – nothing new here.
  - On Line 16, we declare a new struct pointer called `jeb` – we then allocate it memory and check if it allocated it properly.
  - Notice how, because we're using a struct pointer variable now, we reference things in the struct using `->` .

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define BUFF 64
6
7  int main()
8  {
9      struct kerbal
10     {
11         char name[BUFF];
12         char mood[BUFF];
13         int id;
14     };
15
16     struct kerbal *jeb;
17
18     jeb = (struct kerbal *)malloc(sizeof(struct kerbal));
19     if(!jeb) return(1);
20
21     strcpy(jeb->name, "Jebediah");
22     strcpy(jeb->mood, "Excited");
23     jeb->id = 3;
24
25     puts("*** KERBAL DATA ***");
26     printf("Name: %s\n", jeb->name);
27     printf("  ID: %d\n", jeb->id);
28     printf("Mood: %s\n", jeb->mood);
29
30     return(0);
31 }
```

Linking Everything Together

- Things are about to get a little complex – it's time to talk about linked lists.
- Remember this definition: A linked list is a list of structures that contain pointers to the next structure in the list.

- Have a look at the diagram above – we can see how with each structure, there's an element at the end which points to the next structure.
- We could also include an element which points to the previous list too, but for our concerns we just need to focus on moving forward through a linked list.

Oh Boy That's a Big Program (1/3)

- Copy the code over there – don't compile anything yet as it'll take three slides to complete this one…

- We start off by defining the libraries we need, plus any constants that will be useful.
- We then define the `kerbal` structure – this time with an extra entry!
  - This element will store the address of the next list in the linked list element.
  - Look back at the last slide if you're confused!
- We then define three `kerbal` structures – generally with linked list you define these three structures.
  - `*first` - holds the address of the first linked list.
  - `*current` - holds the address of the linked list element being worked on.
  - `*new` - holds the address of any newly made structures.
- Finally, we define the functions that we'll use – notice how we can define a function as returning a structure!

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFF 64
#define NUM_KERBS 3

struct kerbal
{
    char name[BUFF];
    char mood[BUFF];
    int id;
    struct kerbal *next;
};

struct kerbal *first;
struct kerbal *current;
struct kerbal *new;

struct kerbal *make();
void fill(struct kerbal *input);
void printStruct(struct kerbal *input);
```

Oh Boy That's a Big Program (2/3)

- Here's the next section to copy down – we'll go through it so just hang in there.

- We start by making a loop to create the structures one after the other.
  - If we're making the first structure, we set the pointer `first` to the struct being created.
  - For every other structure, we carry out three steps:
    - 1) Make a new structure.
    - 2) Set the `next` pointer of the current structure to the address of the new structure.
    - 3) Set the current structure to the newly made structure.
- Once all the structures have been made, we set the next pointer of the last structure to `NULL`.
- We then print out the linked list elements one by one – this is why we save the address of the first structure!
  - We start from the `first` structure – we print out the structure using the function.
  - We then set the pointer `current` to the address of the next structure.
  - This loops until we hit the `NULL` of the last structure!

```
24  int main()
25  {
26      for(int x = 0; x < NUM_KERBS; x++)
27      {
28          if(x == 0)
29          {
30              first = make();
31              current = first;
32          }
33
34          else
35          {
36              new = make();
37              current->next = new;
38              current = new;
39          }
40          fill(current);
41      }
42      current->next = NULL;
43
44      puts("--------------------");
45      puts("*** KERBAL DATA ***");
46      current = first;
47
48      while(current)
49      {
50          printStruct(current);
51          current = current->next;
52          puts("--------------------");
53      }
54
55      return(0);
56  }
```

Oh Boy That's a Big Program (3/3)

- Last section! You're almost there – get it copied down and you'll have your first linked list!

- The first function – `make` – creates a structure in memory and returns the pointer to it.
  - This is very similar to the last example you did, just in function form!
- The second function – `fill` – is responsible for filling in each struct.
  - We create some temporary buffers to store the data the user inputs.
  - We then use a series of functions to copy the data the user inputs into the structure elements.
  - See, it may look complicated, but it's actually quite simple!
- The final function – `printStruct` – is responsible for printing out the structure.
  - This simply takes in a struct and prints out each element.

- And that's it! Make sure you've copied it accurately, compile, and run!

```
58  struct kerbal *make()
59  {
60      struct kerbal *tmp;
61
62      tmp = (struct kerbal *)malloc(sizeof(struct kerbal));
63      if(!tmp) exit(1);
64
65      return(tmp);
66  }
67
68  void fill(struct kerbal *input)
69  {
70      char nameBuff[BUFF];
71      char moodBuff[BUFF];
72      int idBuff;
73
74      printf("Enter kerbal name: ");
75      scanf("%63s", nameBuff);
76      while(getchar() != '\n');
77      strcpy(input->name, nameBuff);
78
79      printf("Enter kerbal mood: ");
80      scanf("%63s", moodBuff);
81      while(getchar() != '\n');
82      strcpy(input->mood, moodBuff);
83
84      printf("Enter kerbal ID num: ");
85      scanf("%d", &idBuff);
86      input->id = idBuff;
87
88      putchar('\n');
89      return;
90  }
91
92  void printStruct(struct kerbal *input)
93  {
94      printf("Name: %s\n", input->name);
95      printf("  ID: %d\n", input->id);
96      printf("Mood: %s\n", input->mood);
97
98      return;
99  }
```

Task Time!

» Linked lists are quite confusing, and I appreciate your mind is probably soup by now. This is just one task to make sure you've really got the hang of linked lists. Give it a go!

1. Add an int element to the structure called "missions". Add to the `make` function to read in the number of completed missions from the user. Print this information along with the rest of the information.

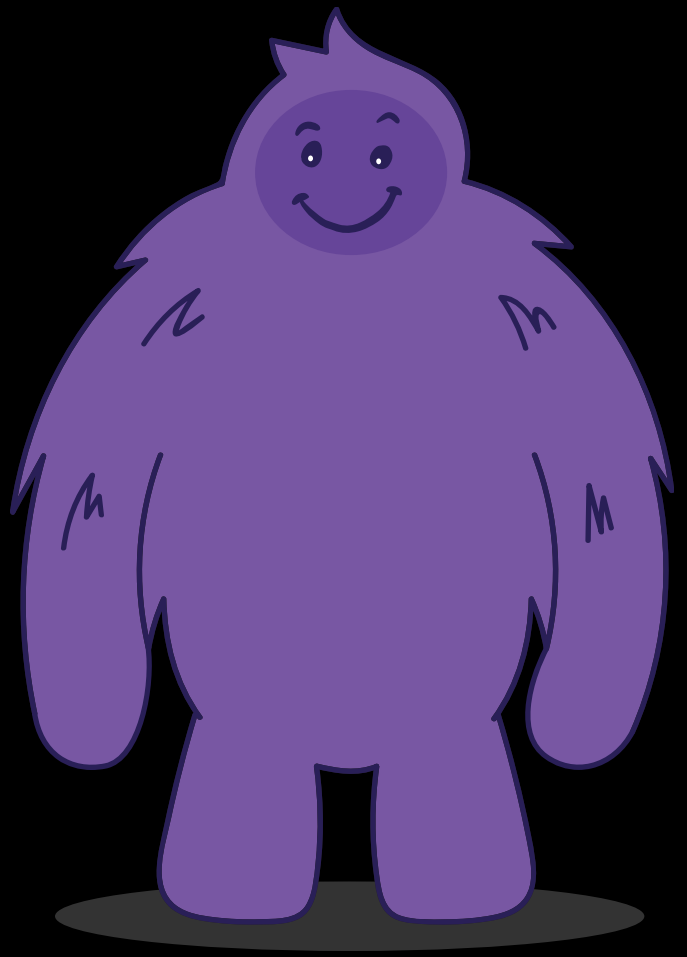
I Command You, Args!

- A lot of programs contain the ability to pass arguments via the command line to their programs. C is no different, but it does it in an interesting way…
- You may have realised that main is a function by now, and as such it can take arguments – two key ones in fact!
  - `int argc` – This holds a number. What number? The number of arguments passed, including the program name!
  - `char *argv[]` – This holds the command line parameters passed. It will always include the name of the executable file used to run the program.
- Take a look at the example there – copy it down, compile, and try running with some command line arguments!
  - If you're confused, I mean something like `./a.out hello world cool`.
  - As we can see, `argc` holds the number of arguments – it'll always be at least one.
  - In the `for` loop, we iterate through the list of arguments in `argv` – don't get scared by the pointer notation, you've seen it before!
  - Note that `*argv[0]` or `*(argv + 0)` is always the command used to run the program with!

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

int main(int argc, char *argv[])
{
    if(argc == 1)
    {
        puts("Hey, try running me with some arguments!");
        exit(0);
    }

    printf("Number of arguments passed: %d\n\n", argc);

    for(int i = 0; i < argc; i++)
        printf("Arg %d = %s\n", i + 1, *(argv + i));

    return(0);
}
```

Challenge Three

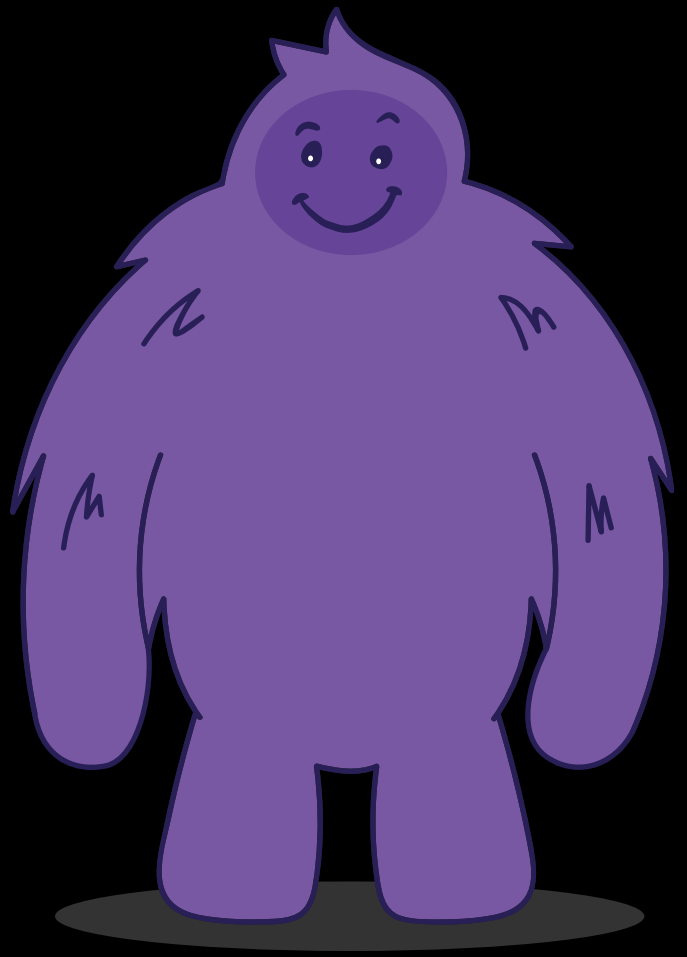
- Oh boy, it's the big one! Time to dust down your keyboard and put everything you've learnt to the test.
- This is not an easy program, and as such we've broken the challenge down.
- Hints are available – try and do as much as you can without them, but if you need them, they're at the end of this slide pack!
- Create a simple password management database which uses linked lists in C.



- 1/5: Create a program which generates a new database if no file path is passed as an argument when running the program.
  - Each database entry must store a username and password.
  - The database must have no more than 5 entries.
  - The first entry must be a secret admin entry with an admin username and password.
  - Each other user entry must be prompted to and entered by the user.
  - Passwords must have no more than 12 characters in them.
- 2/5: Add a menu to the program, and the options to modify usernames and passwords.
  - The menu must list the following options: "Change Username", "Change Password", "View Records", "Save and Quit".
  - The menu must be able to catch bad inputs and handle them.
  - When modifying a username or password, the user should be prompted to enter the username of the entry to modify.
  - If the username is not in the database, the function should just return.
  - If the user enters a valid username, they should be prompted to enter the password associated with that username.
  - Again, if the user enters an invalid entry, the function should return.
  - If both the username and password are okay, the user should be permitted to enter a new username or password.

Challenge Three (Cont.)

- 3/5: Add another option to print out information on all the entries in the database.
  - The program should not print out the admin entry unless the user enters the correct admin password.
  - The program should then print out each entry one by one, prompting the user to advance to the next page.
  - Once all entries are shown, the program should return to the main menu.
- 4/5: Add the ability to save and quit the program.
  - If the user originally loaded the database in from a file, the database should be saved to this file.
  - If the database wasn't loaded from a file, the user should be prompted to enter the name of the file to save to.
- 5/5: Add argument parsing to the program so that the file path of a database can be passed as an argument.
  - The program should accept one argument along with the executable name – the file path of the database file.
  - The program should then load the database from this file, then present the main menu to the user once loaded.



Hints: Step 1/5

- Yikes, this program looks like it'll be a massive challenge, but don't panic! There's nothing new here – make sure you're using the previous slides as reference if you need.
- First challenge is to generate a new database if no file-path is given. Break this down into two tasks.
  - Checking if there's an argument.
  - Creating a new database.
- Generating the database shouldn't be too hard – take a look at slides 34-36 and you'll see it's nearly the same as this program here.
  - We can use a loop to generate each entry.
  - For the secret admin page, we can generate this on the first loop through, then generate the other user entries.
  - Remember to use `strcpy` when reading the strings into the struct!
- Checking for an argument… We know that if a file path is given that `argc` will equal `2`… I wonder what you could do with this knowledge. (Big hint: Use an `if` statement!)

Hints: Step 2/5

- You can make the menu with a series of print statements, and get user entry with a simple `scanf` – validation can be done with a simple `if` statement or `ctype` library function.
- As for deciding what to run based on the user input, might I recommend a `switch…case` statement?
- Hmm… Modifying the username or password should start by checking whether a username exists in the database… Why not try a `while` loop to run through each linked list element?
  - If you're super stuck, take a look at the screenshot and how I looped through each element…

```
first = my_linked_list;
current = first;

while(current)
{
    // Some program stuff here...

    current = current->next;
}
```

- We can check for matching strings with a simple `strcmp` – remember it returns `0` if the strings match!
- As for changing the username or password… I feel like `strcpy` could be of use.

Hints: Step 3/5

- To check for the admin, I'd check whether they know the admin password. Maybe another `strcmp` is in order…
- Printing out each linked list… Sounds like another loop to me!
  - If you need a big hint, look at the last slide – the loop will work in exactly the same way!
  - Have a look also at slide 35…

Hints: Step 4/5

- If the user passed a file path as an argument, then two things will be true.
  1. The argument counter (`argc`) will equal two.
  2. The argument vector (`argv`) will hold the file path.
- Remember – you can pass `argc` and `argv` to other functions from main!

- Saving seems like a scary task, but break it down:
  - We'll want to save each entry one at a time, so a loop through the linked list seems like the best idea.
  - When writing binary data to a file, what function do we use?
  - The size of each write will be the size of one structure – think `sizeof(struct …)`.
  - We'll be writing one structure at a time, so don't forget to advance to the next structure once one is written!

Hints: Step 5/5

- Don't be downtrodden if you managed to get through the other tasks but got stuck on this one – I'd argue that this step and the previous ones are the hardest of the lot so well done for even getting to this step!

- Firstly, get the file opened – this shouldn't be too difficult.
- We could do with a temporary structure to temporarily store the data in – just make sure it's a structure variable and not a structure pointer!
- I'd use a `while` loop with an `fread` inside it to read all the data in – have a think about what the arguments for the `fread` will be…
- Remember that for each entry we read in, we need to generate a structure for it!
  - The code in the while loop will be a near copy and paste of some code you wrote in step one and in slides 34-36 – you know, the one in the `main` function of slide 35…
- Once you've made a structure, copy the data from your temporary structure to the new one.
  - I'd recommend a `strcpy`.
- Remember to return the address of the first structure of the linked list!