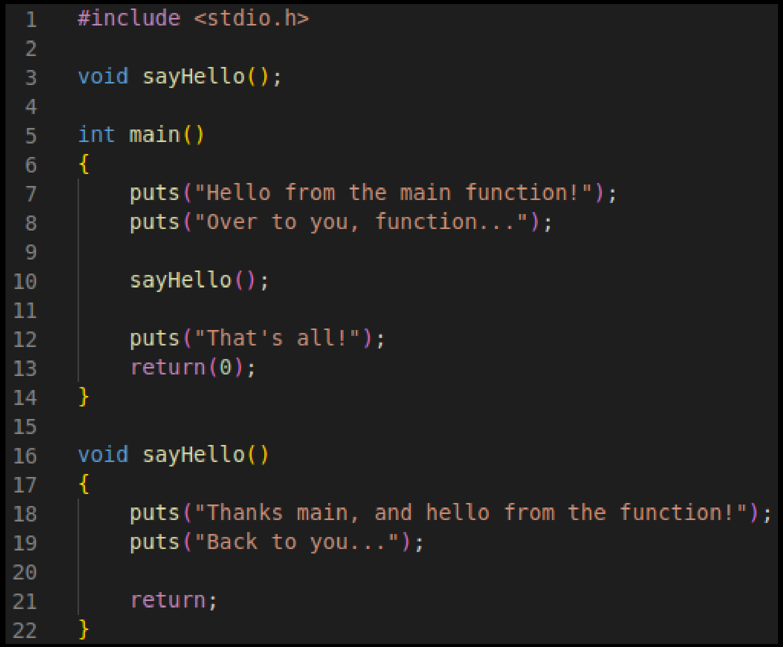
INTERRUPT LABS

C Intro Pack

Section 2: I was blind but now I C

Things Are Getting Func-y

- Where functions provided by C fall short, we can create our own very easily.
- Every function follows the form `type funcName(<vars>)`, where:
  - `type` = The type of the value returned or generated by the function. These are the same as the types you've come across before!
  - `funcName` = The name of the function – you can call functions whatever you want, just as long as the name is unique.
  - `<vars>` = Any values that are passed to the function. A function doesn't need to have any values passed to it.
- Every function must return back to the block that called it – this is done via two ways:
  - `return;` - This is used when you don't have a value to return.
  - `return(value);` - This is used when you have a value to return.
- Exercise: Copy the code and run it. Have a guess at what `void` means.
  - Be careful to include line 3 – we'll explain its importance in the next slide.

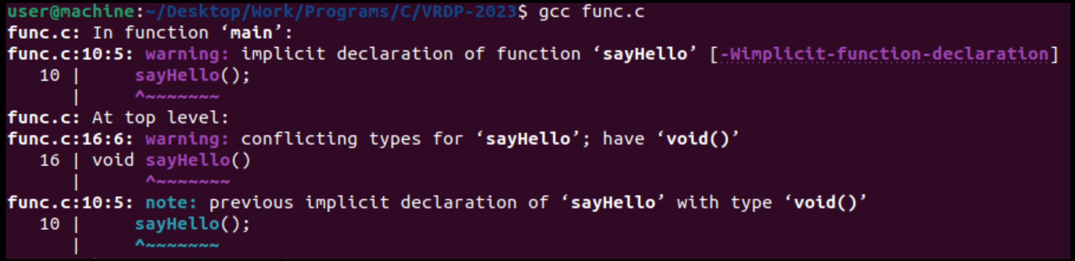


```
1   #include <stdio.h>
2
3   void sayHello();
4
5   int main()
6   {
7       puts("Hello from the main function!");
8       puts("Over to you, function...");
9
10      sayHello();
11
12      puts("That's all!");
13      return(0);
14  }
15
16  void sayHello()
17  {
18      puts("Thanks main, and hello from the function!");
19      puts("Back to you...");
20
21      return;
22  }
```

Cake or Prototype? (A: Prototype)

» You may be wondering why the third line of that program is so important it got its own line in the previous slide.

» Time for an experiment - comment out that line and compile the program again. You should see something like this:



```
user@machine:~/Desktop/Work/Programs/C/VRDP-2023$ gcc func.c
func.c: In function ‘main’:
func.c:10:5: warning: implicit declaration of function ‘sayHello’ [-Wimplicit-function-declaration]
   10 |     sayHello();
      |     ^~~~~~~~
func.c: At top level:
func.c:16:6: warning: conflicting types for ‘sayHello’; have ‘void()’
   16 | void sayHello()
      |      ^~~~~~~~
func.c:10:5: note: previous implicit declaration of ‘sayHello’ with type ‘void()’
   10 |     sayHello();
      |     ^~~~~~~~
```

» Don’t get too put off by the warnings – the compiler is telling us that it has no idea what the function “`sayHello`” is, so running the code without fixing it is going to be risky.

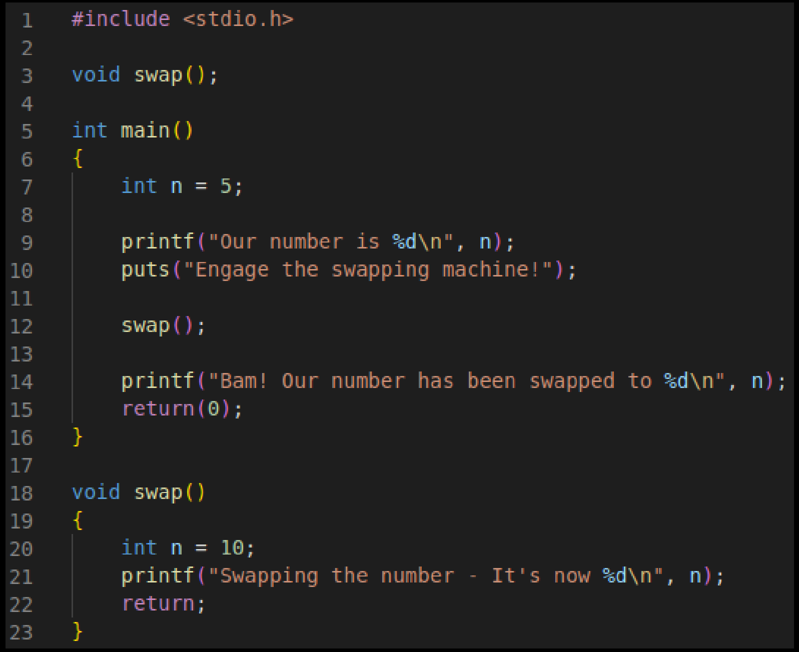
» There are two ways of fixing this sort of error:

1. Prototyping the function like we did before – this is like giving the compiler a heads up that the function exists, and to not be scared when it encounters it.
   - To achieve this, simply copy the definition of the function and paste it above `main` with a semi-colon at the end!
2. Placing any function definitions before they are used – i.e. writing the definition of the function above where you use it. This has the same effect as prototyping.

» It doesn’t matter which prototyping method you use, just be consistent!

Getting Function-al With Variables

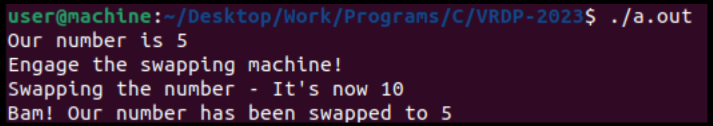
- Functions can use variables just like the `main` function can, and they're declared the same way too!
- BUT, and this is a big but…
- You need to be aware of the idea of function locality.
  - In other words, functions don't like to share variables.
- Exercise: Let's make an amazing program that takes a number and swaps it for another number. Copy the code below, compile it, and run it.



```c
#include <stdio.h>

void swap();

int main()
{
    int n = 5;

    printf("Our number is %d\n", n);
    puts("Engage the swapping machine!");

    swap();

    printf("Bam! Our number has been swapped to %d\n", n);
    return(0);
}

void swap()
{
    int n = 10;
    printf("Swapping the number - It's now %d\n", n);
    return;
}
```

Our Program – It’s Broken!

- Uh oh… If you’ve run the code, you’ll have probably seen that something is up with our program. Here’s my output:



```
user@machine:~/Desktop/Work/Programs/C/VRDP-2023$ ./a.out
Our number is 5
Engage the swapping machine!
Swapping the number - It's now 10
Bam! Our number has been swapped to 5
```

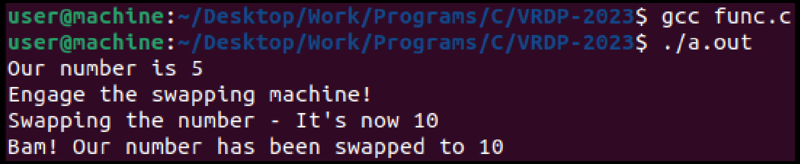
- What’s gone on here? Is there no hope for this amazing program?
- Variables declared in functions are unique to that function alone, even if they share the same name.
- This means that we can’t simply change the value of a variable between functions…
- Thankfully, there’s an easy fix to this – there is still hope for this amazing program.

Fixing Our Mistakes

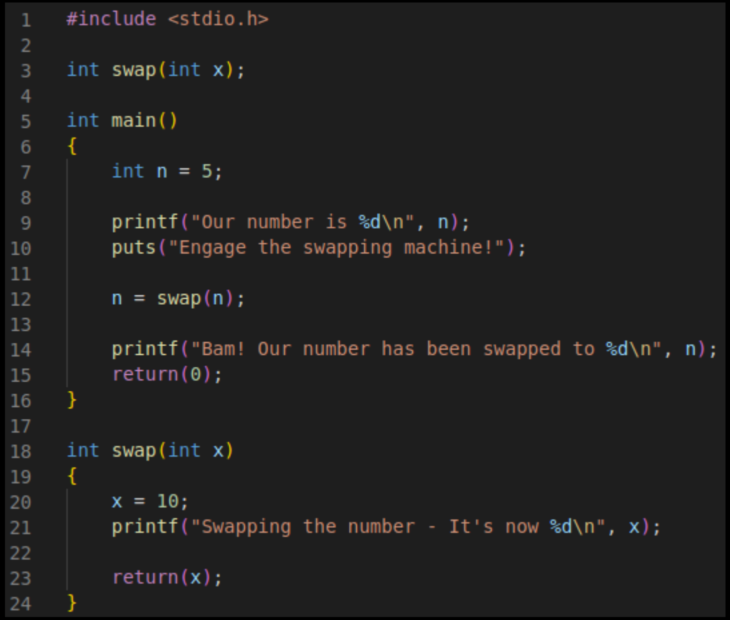
» Let's fix the mistakes line by line – we'll refer to the example code to the right to fix it.
  - Line 3 – We'll change the prototype to match the new definition of the function – the change will be explained below!
  - Line 12 – Here, we pass the variable `n` to the function `swap` – think of this like passing a string to `puts`. We also set `n` to be the value that this function returns.
  - Line 18 – Here's where things get interesting.
    - We need to be very clear to the compiler about what type of argument is required – since the function returns a value of type `int`, we declare the function as an `int`.
    - Any values that are passed to the function need somewhere to go. In this case, we declare an `int` argument called `x` – this will serve as the variable holding `n` in the function.
  - Line 23 – Once we've modified the value, we want to return it to main so we can set it to `n`. This is done by simply returning the value.

» That's our fixes done! Compile and run the program again and see what happens.

» This is the output I get from running it – looks like it works!:



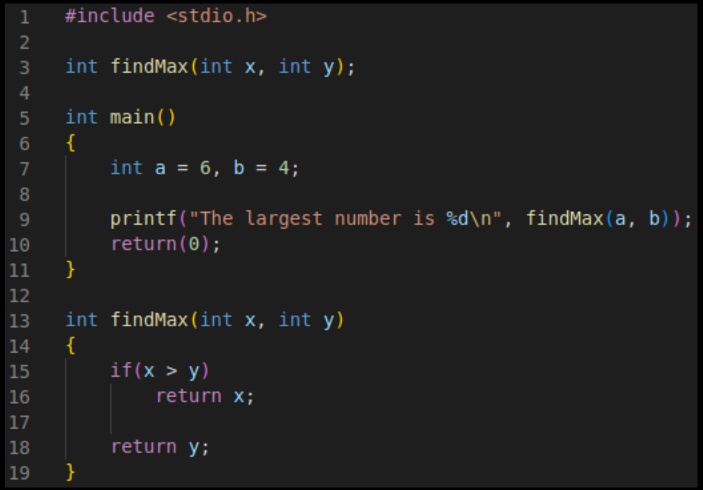
```
user@machine:~/Desktop/Work/Programs/C/VRDP-2023$ gcc func.c
user@machine:~/Desktop/Work/Programs/C/VRDP-2023$ ./a.out
Our number is 5
Engage the swapping machine!
Swapping the number - It's now 10
Bam! Our number has been swapped to 10
```



```
1   #include <stdio.h>
2
3   int swap(int x);
4
5   int main()
6   {
7       int n = 5;
8
9       printf("Our number is %d\n", n);
10      puts("Engage the swapping machine!");
11
12      n = swap(n);
13
14      printf("Bam! Our number has been swapped to %d\n", n);
15      return(0);
16  }
17
18  int swap(int x)
19  {
20      x = 10;
21      printf("Swapping the number - It's now %d\n", x);
22
23      return(x);
24  }
```

More Than One Var

- Functions can take more than one argument, and more than one type of argument at that!
- We can demonstrate this with a simple example, and some shameless improvement of one of the programs from the previous pack!
- Exercise: Copy the code below. If you did the previous slide pack you should recognise it as our number comparison program! Compile and run it.



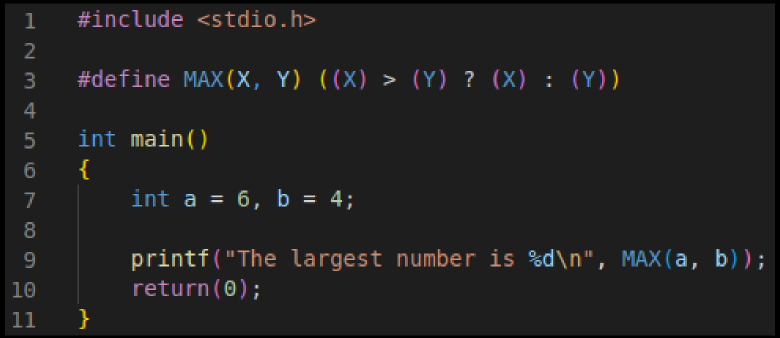
```c
#include <stdio.h>

int findMax(int x, int y);

int main()
{
    int a = 6, b = 4;

    printf("The largest number is %d\n", findMax(a, b));
    return(0);
}

int findMax(int x, int y)
{
    if(x > y)
        return x;

    return y;
}
```

- As you've probably seen in the `printf` statement, we can use the return value of the function without needing it to be assigned to a variable – convenient!

Define me a Function

- There's one more type of function you should know about – the Macro function.
- Like how we can define constants with `#define`, we can also define functions!
  - These are commonly used for single line functions – if you've come across lambda functions in Python these are very similar.
  - Functions written as a `#define` are copied into the code when its compiled. This can result in the compiled copy of the code being larger in size than the C file!
  - They're useful for simpler functions, but for anything complicated I'd recommend sticking to a regular function.
  - Any variables used in the function must be surrounded by brackets.

- Exercise: Copy the code to the right, compile, and run.
  - As you can see, our function has been replaced with a Macro function.
  - You'll also notice the weird `?` thing…
    - This is called a ternary operator – it's like an if statement but on one line.
    - It follows the format `condition ? ifTrue : ifFalse`.
    - If `condition` is true, `ifTrue` is run. If not, `ifFalse` is run. Simple!



```c
#include <stdio.h>

#define MAX(X, Y) ((X) > (Y) ? (X) : (Y))

int main()
{
    int a = 6, b = 4;

    printf("The largest number is %d\n", MAX(a, b));
    return(0);
}
```

Task Time - Functions

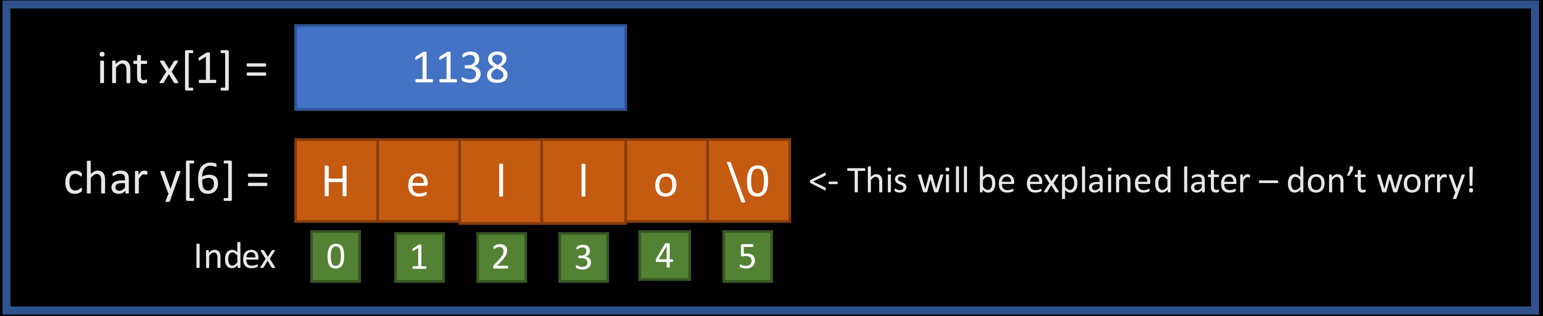
» That was a lot of learning at once – let’s solidify your knowledge with some exercises!

1. Create a program which converts a temperature from Celsius to Fahrenheit. The program should use a function to perform the conversion and return the answer. You can include user input if you’d like.
2. Create a function that prints out a user inputted string to the terminal. The function should take the string input as an argument and print out a formatted message.
   - (HINT: Your function prototype will look something like this: `void myFunc(char myArr[n]);`).
   - If you get stuck, leave this one for now and come back to it – arrays in functions will be explained later.
3. Create a calculator program that uses macro functions to perform mathematical operations on two numbers.
   - The user should be able to select the mathematical operation from a list.
   - The user should then be prompted to enter two numbers.
   - The program should then print the solution to the user.

Make the Array-ngements

» Arrays are very useful things – they allow us to store many values of the same type in a format that is easy to access!

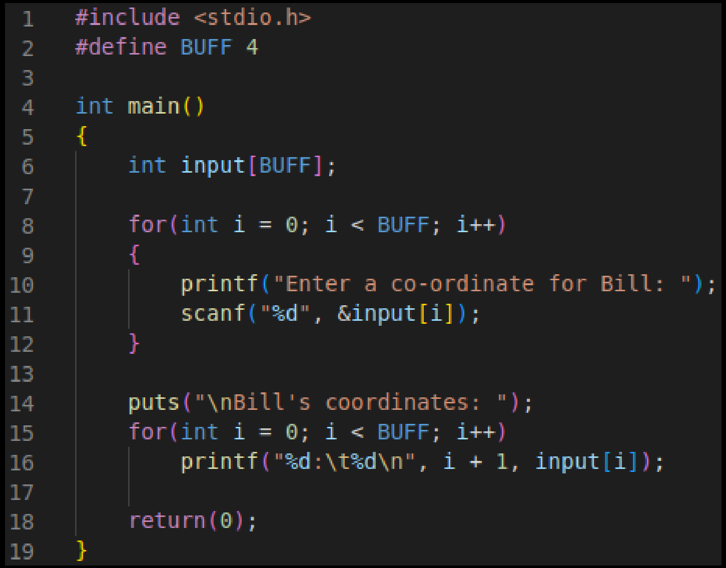
- Think of an array as a long string of boxes, where each box can only hold one type of thing, called an element.



- Once an array's size has been declared in C, it's very difficult to increase/decrease it.
  - Always be sure your array will have enough space to store what you want!
- We can access an element by using the square brackets.
  - For example: `x[2] = 3, y[0] = H`
  - Indexing in C starts from 0 – this may not make sense to you, but to a computer this is better than indexing from 1. (The reason is complex, but a good thing to research if you're interested!)

Int-eresting Arrays

» Let's go through an example – copy the code below, compile, and run. Be careful to copy it correctly!



```c
#include <stdio.h>
#define BUFF 4

int main()
{
    int input[BUFF];

    for(int i = 0; i < BUFF; i++)
    {
        printf("Enter a co-ordinate for Bill: ");
        scanf("%d", &input[i]);
    }

    puts("\nBill's coordinates: ");
    for(int i = 0; i < BUFF; i++)
        printf("%d:\t%d\n", i + 1, input[i]);

    return(0);
}
```

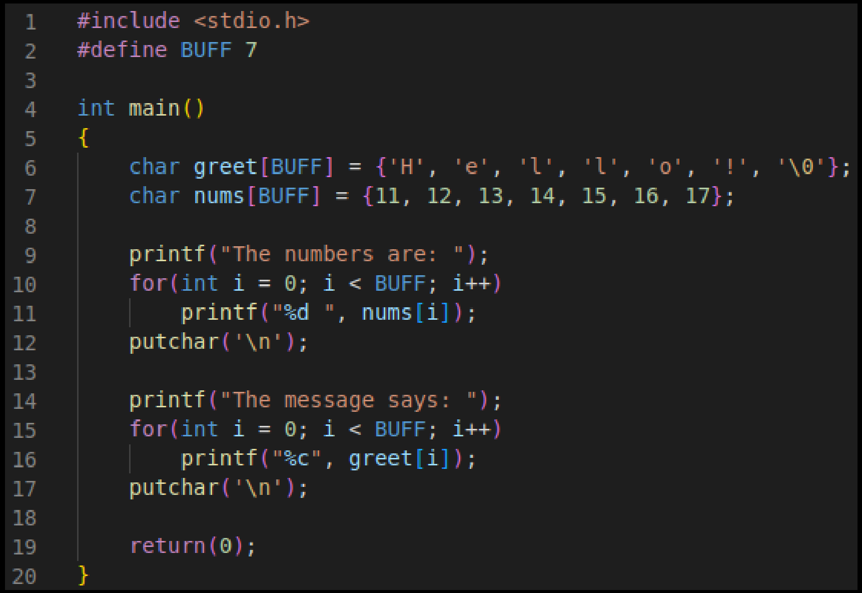
» We declare an empty array called input with a size of 4, then use a `for` loop to place four numbers into it.

» We then use another for loop to access each element of the array in order.

- This example shows how useful for loops can be, especially when dealing with arrays.
- Just remember – arrays start indexing from 0, not 1!

Declare That Array!

- As well as adding elements to an array, we can also declare an array with elements in it already.
- Exercise: Copy the code below, compile, and run it.

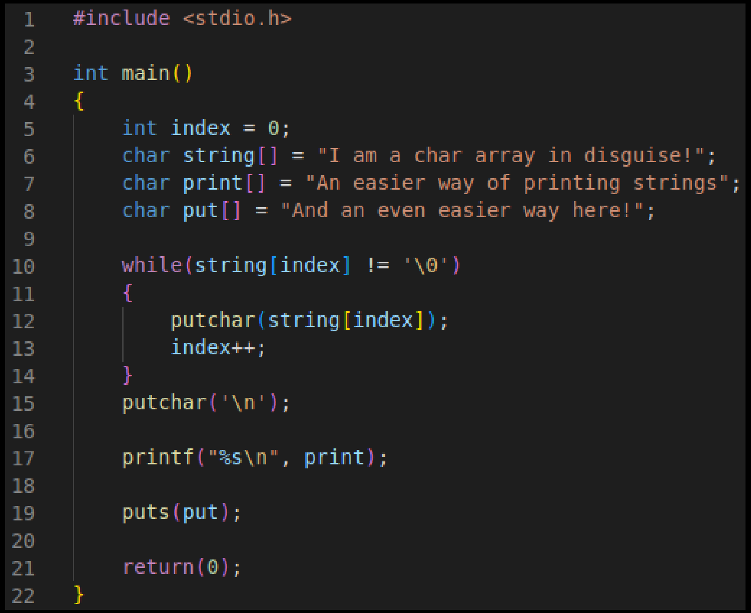


```c
#include <stdio.h>
#define BUFF 7

int main()
{
    char greet[BUFF] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
    char nums[BUFF] = {11, 12, 13, 14, 15, 16, 17};

    printf("The numbers are: ");
    for(int i = 0; i < BUFF; i++)
        printf("%d ", nums[i]);
    putchar('\n');

    printf("The message says: ");
    for(int i = 0; i < BUFF; i++)
        printf("%c", greet[i]);
    putchar('\n');

    return(0);
}
```

- Here we declare two arrays, with elements in them.
  - Array elements are declared inside curly brackets, with each element separated by a comma.
  - With character arrays, if you're storing a string as individual characters it needs to be terminated with a '\0'.
  - This will be explained in the next slide!

Char Array == String == Char Array

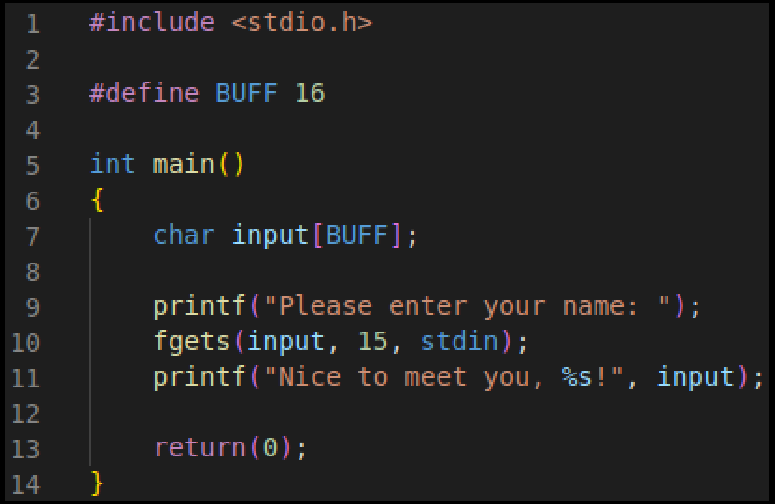
- We mentioned briefly in the last slide pack that there's no such thing as a string type in C. In fact, all strings in C are secretly character arrays in disguise!
- You can declare a string as either an initialised or an uninitialized array.
  - In the last slide, you used an initialised array to store a string as an array of characters!
  - To remind you – to declare a string, use `char string[] = “myString”;`.
- All strings in C are terminated with what's called a null terminator.
  - This is represented by ‘`\0`’.
  - It provides an indication that this is the end of the string.
- Using this knowledge, we can use a different method to print out character arrays!
  - Copy down the example and run it – it shows three ways of printing a string.
  - The easiest way is with puts, yes, but it's good to know the other methods!



```c
#include <stdio.h>

int main()
{
    int index = 0;
    char string[] = "I am a char array in disguise!";
    char print[] = "An easier way of printing strings";
    char put[] = "And an even easier way here!";

    while(string[index] != '\0')
    {
        putchar(string[index]);
        index++;
    }
    putchar('\n');

    printf("%s\n", print);

    puts(put);

    return(0);
}
```

Store me a String

- Whilst reading into an array one character at a time is simple, it's not very practical. Therefore, we need a way of reading strings into arrays.
- You must be precise when declaring the size of the character array – the size you declare must be long enough to hold the maximum length of the string, plus one to account for the null byte at the end.
- `scanf` will copy into an array, but it won't care whether the array is too small, which can cause what's known as a buffer overflow.
  - The compiler REALLY won't like you exceeding the bounds of the array.
  - We can modify a `scanf` to read in a specific length with "`%ns`", where `n` is the number of bytes you want to read in.
- To get around this, we can use another method – `fgets`.
  - `fgets` has the form `fgets(dest, size, source)`.
    - `dest` = Where you want to copy the data to.
    - `size` = The length of the data you want to read in.
    - `source` = Where the data is coming from – when reading from the terminal, this is `stdin` (**ST**an**D**ard **In**put).
- Exercise: Copy the example down, compile, and run. Try and enter an input greater than 15 characters. (Remember: 16 = 15 + 1 for the null byte). Note what happens. Replace the `fgets` with a `scanf` that performs the same function.

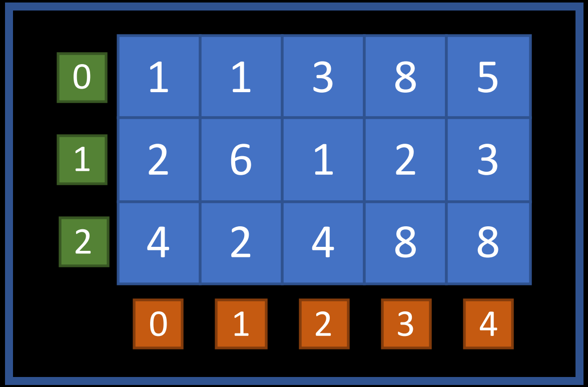


```c
#include <stdio.h>

#define BUFF 16

int main()
{
    char input[BUFF];

    printf("Please enter your name: ");
    fgets(input, 15, stdin);
    printf("Nice to meet you, %s!", input);

    return(0);
}
```

Array Array

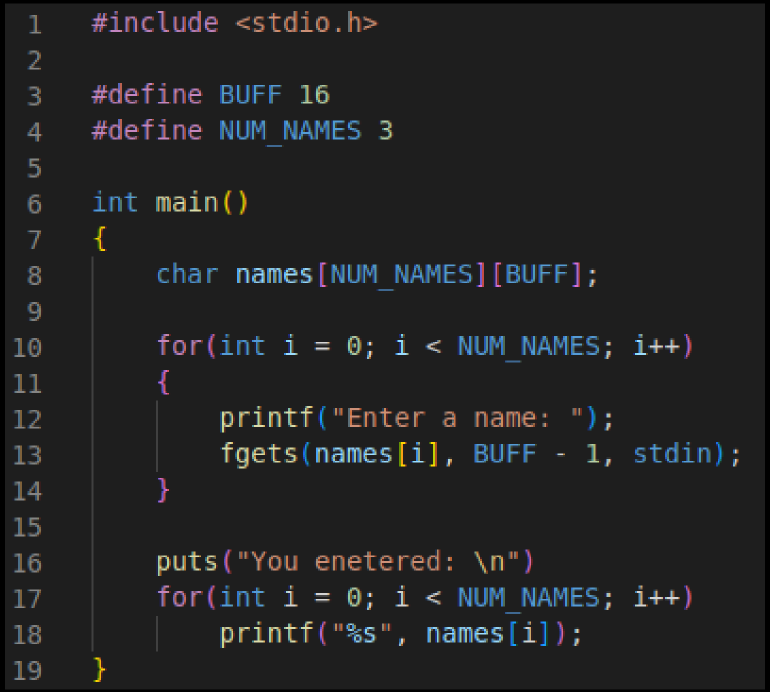
» Along with one-dimensional arrays, we can create two-dimensional arrays too!

- Think of a 2D array as a grid, a bit like a graph or a map.
- Let's use the example `int x[3][5]` – that would look like the graph below:



- To access the number 6, you would read the value `x[1][2]`.
  - Unlike co-ordinates which is x -> y, indexing in C is y -> x.
  - Rule of thumb – down then Left.
- The array must be big enough to hold the largest element in it.
  - E.g. if you have 3 strings, where one is 12 characters long and the rest are 2, you would declare the array as `arr[3][12]`.
  - Don't forget the null byte!

» As a quick exercise, copy down the code on the right, compile and run.



```c
#include <stdio.h>

#define BUFF 16
#define NUM_NAMES 3

int main()
{
    char names[NUM_NAMES][BUFF];

    for(int i = 0; i < NUM_NAMES; i++)
    {
        printf("Enter a name: ");
        fgets(names[i], BUFF - 1, stdin);
    }

    puts("You enetered: \n")
    for(int i = 0; i < NUM_NAMES; i++)
        printf("%s", names[i]);
}
```

A Functional Array

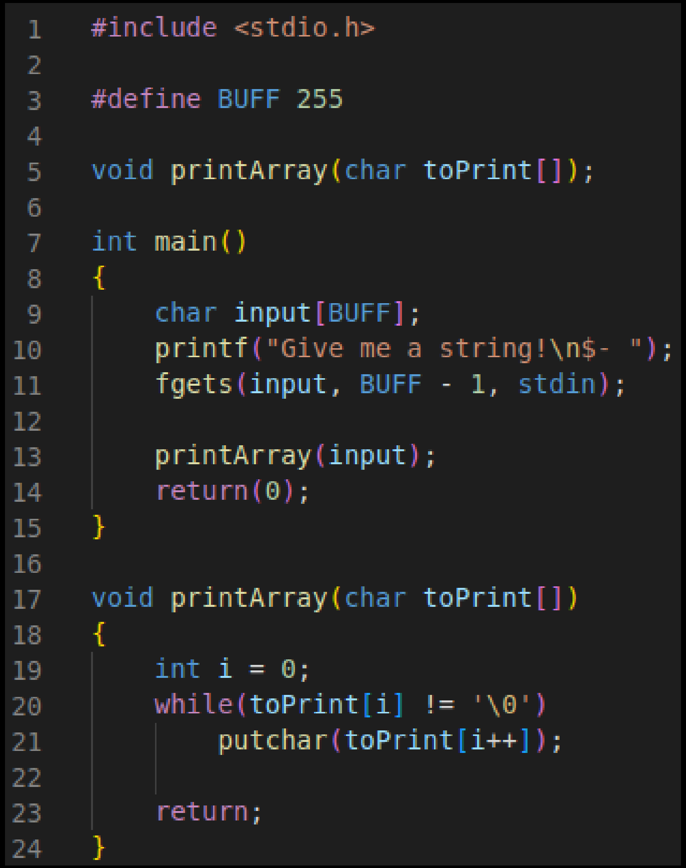
» Sometimes it's good to pass arrays to functions in order to perform operations on them, and thankfully it's pretty easy! ***
  - Your function prototype must contain an array specified as one of its arguments – for example: `void func(int myArray[])`.
  - When you pass the array to the function, simply call the function with the array as the argument – for example `func(someArray)`.

» *** Returning arrays from functions… isn't as easy.

» It requires using something called pointers, and that's not the worst part of it all.

» Eventually you'll learn that arrays in C are a lie – they're all just pointers!

» For now, just copy the example to the right and run it – pointers will be explained in the next slide pack so we're safe for now.



```c
#include <stdio.h>

#define BUFF 255

void printArray(char toPrint[]);

int main()
{
    char input[BUFF];
    printf("Give me a string!\n$- ");
    fgets(input, BUFF - 1, stdin);

    printArray(input);
    return(0);
}

void printArray(char toPrint[])
{
    int i = 0;
    while(toPrint[i] != '\0')
        putchar(toPrint[i++]);

    return;
}
```

Task Time - Arrays

- Arrays are very useful, but knowing how to use them is even more useful! Time for some tasks methinks…

1. Create a program which takes in a character array as input and prints out the following information.
   - The number of vowels in the sentence.
   - The number of upper-case characters.
   - The number of occurrences of the letter e.
2. A bubble sort is a form of sorting algorithm where the input list is repeatedly stepped through element by element. The current element’s value is compared with the one after it, and if the one after is bigger the two elements are swapped. Write a program which performs a bubble sort on a given list of integers. Your program should print out the array before and after sorting.

Getting a bit Random

» Often when programming we want to generate some random numbers.

» Whilst this might be easy in other languages, in C it's a little more complicated.

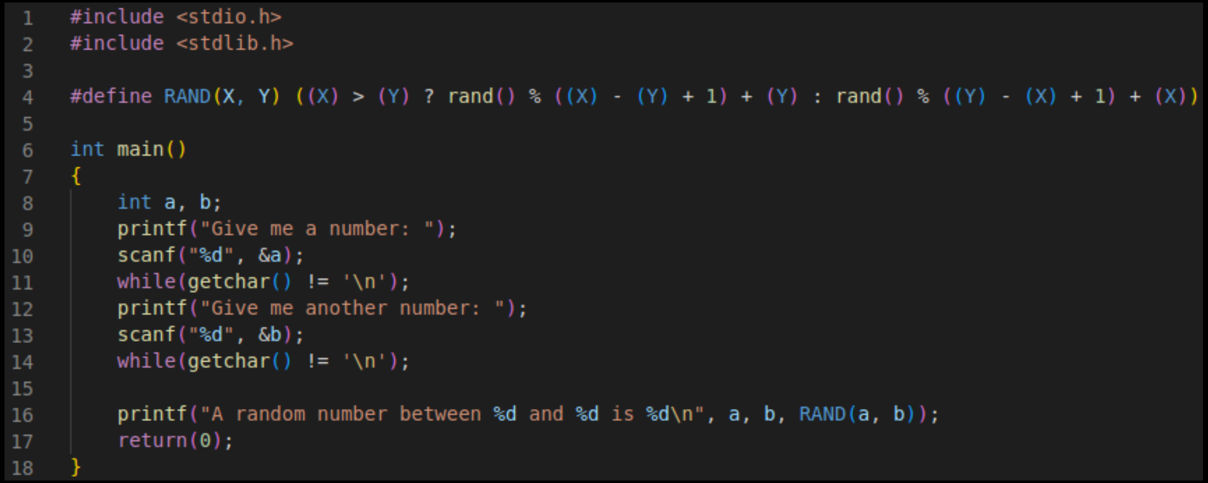
» Lucky for you, I'm going to show you how to generate random numbers, and even give you a function to generate a random number within a range!

» To begin, we must introduce a new library: `stdlib.h`.

- This library contains many useful functions relating to C, such as random generators and memory allocators.
- It's generally a good idea to always include this library along with `stdio` – we'll be doing this from now on.

» Exercise: Copy the example. You'll see I've included a macro to generate a random number in a range of given numbers – feel free to use this, it may come in useful to you!

- To use the macro, simply provide the function two numbers. It will generate a random number in that range!



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

#define RAND(X, Y) ((X) > (Y) ? rand() % ((X) - (Y) + 1) + (Y) : rand() % ((Y) - (X) + 1) + (X))

int main()
{
    int a, b;
    printf("Give me a number: ");
    scanf("%d", &a);
    while(getchar() != '\n');
    printf("Give me another number: ");
    scanf("%d", &b);
    while(getchar() != '\n');

    printf("A random number between %d and %d is %d\n", a, b, RAND(a, b));
    return(0);
}
```

Task Time!

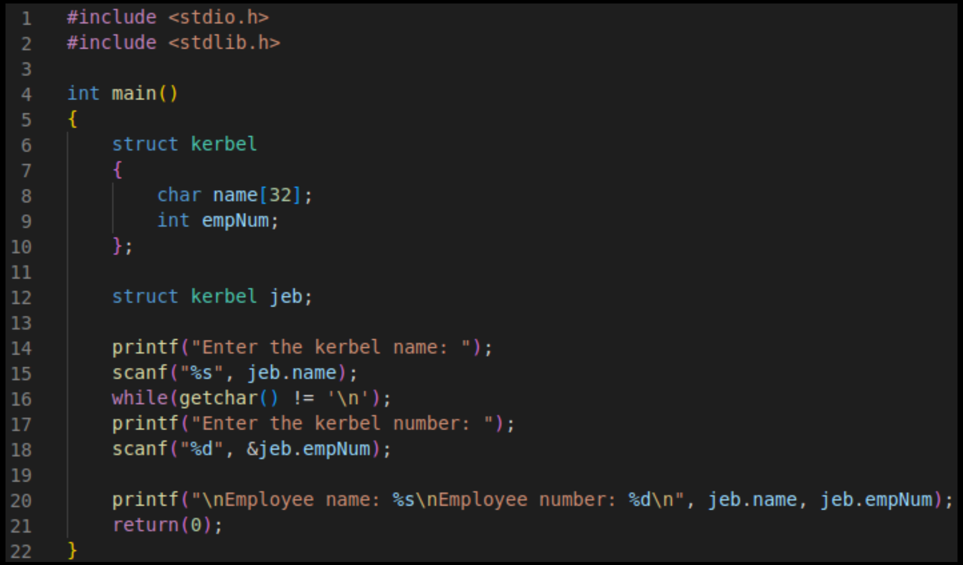
» A task already? It’s just a quick one to make sure you’ve got the hang of randomness.

1. Remember the Guess The Number program you made in the first pack? Modify that program again such that the number chosen is random between 1 and 50.

Bonus: Modify the program further to add a debug mode. This mode should be accessed by passing “debug” as an argument when executing the program. The debug mode will allow you to set the number to be guessed before the game is run.

Adding a bit of Structure

- Sometimes in programming, it'd be useful to have a way of storing several bits of information about a specific variable.
- Introducing structures – a way of storing several variables in one variable!
- Exercise: Copy down the code, compile, and run. We'll explain the code below!

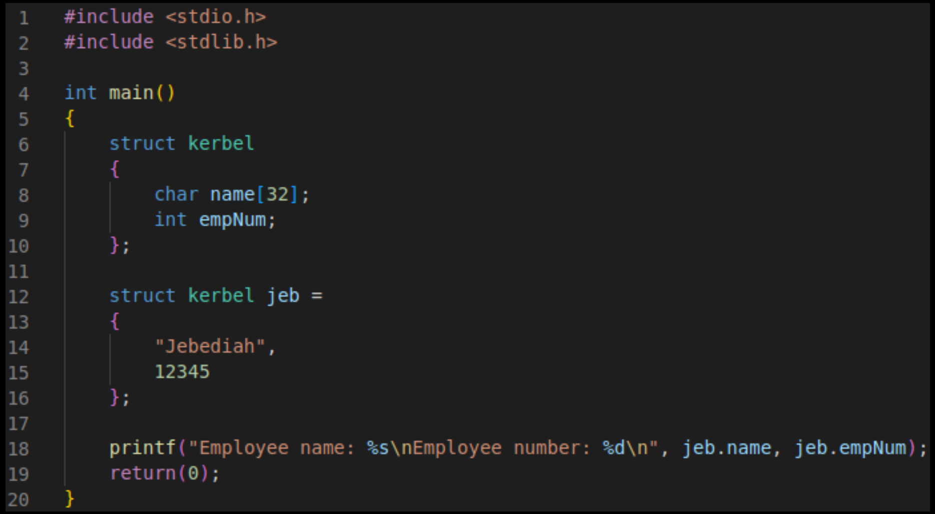


```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   int main()
5   {
6       struct kerbel
7       {
8           char name[32];
9           int empNum;
10      };
11
12      struct kerbel jeb;
13
14      printf("Enter the kerbel name: ");
15      scanf("%s", jeb.name);
16      while(getchar() != '\n');
17      printf("Enter the kerbel number: ");
18      scanf("%d", &jeb.empNum);
19
20      printf("\nEmployee name: %s\nEmployee number: %d\n", jeb.name, jeb.empNum);
21      return(0);
22  }
```

- Line 6 declares the struct along with its name.
  - The name of the struct isn't a variable – it's a unique identifier for the struct itself.
- Lines 8-9 declare the variables that the structure holds – in this example it holds a character array and an integer variable.
- Line 12 declares a structure variable called `jeb`. This variable is a representation of the structure!
- To access the elements in the structure, you use a period.
  - As an example, take a look at line 15.
  - We access the name variable in the `jeb` structure variable to store the string.
  - This is accessed by using `jeb.name` - <name of struct var>.<var in struct>.

Structure off the Bat

- Structures can also be declared with data already in them!
- You simply declare a structure variable, then declare its member values in curly brackets.
- Since this is such a short slide, have a look at the example below. Write it, compile it, and run it. You should see you get the same output as before!



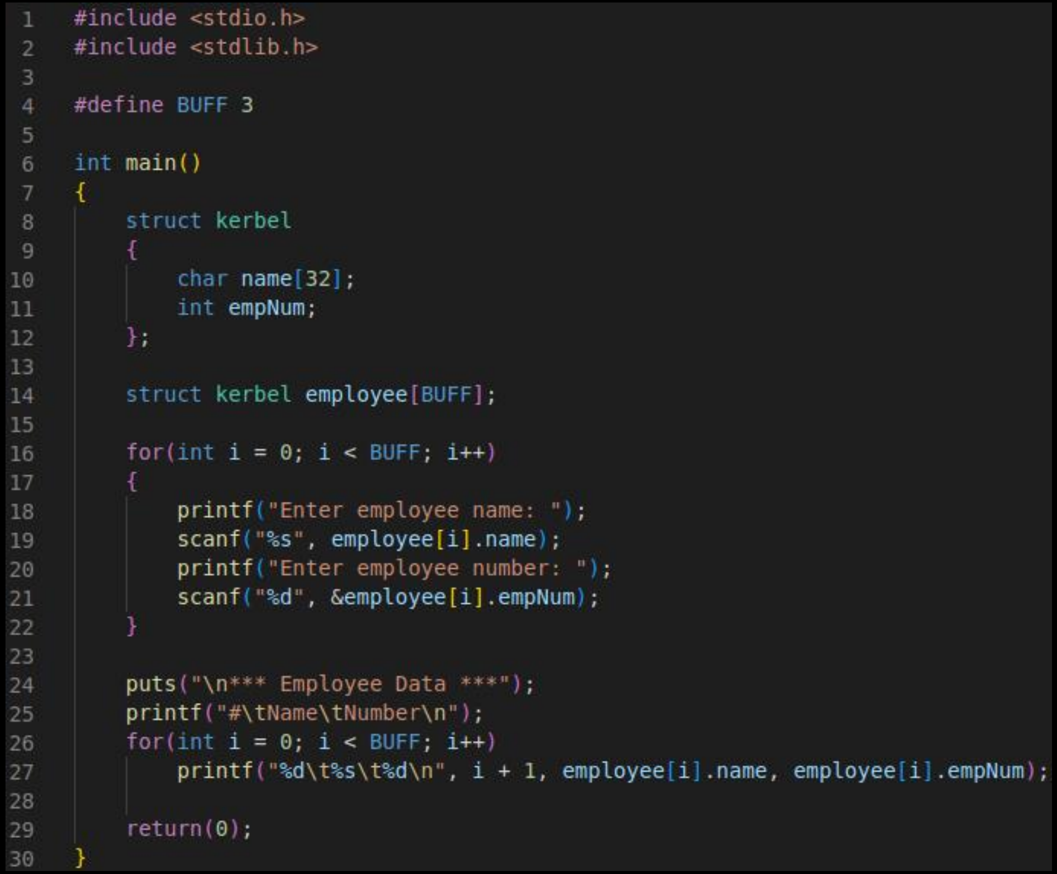
```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
    struct kerbel
    {
        char name[32];
        int empNum;
    };

    struct kerbel jeb =
    {
        "Jebediah",
        12345
    };

    printf("Employee name: %s\nEmployee number: %d\n", jeb.name, jeb.empNum);
    return(0);
}
```

Multi-Multivariable

» Like any variable, structures can be stored in arrays.
  - You declare a structure array like so: `struct kerbel employee[5];`.
  - To access a structure in the array, you simply access it like any other array: `employee[2];`.
  - To access a variable in a specific structure, you access it like a normal structure!: `employee[2].name;`.

» Exercise: Copy the code to the right – it's a modification of our previous example allowing us to record more information! Compile and run.
  - Be sure to copy the `printf` statements carefully, otherwise the output may look very odd.

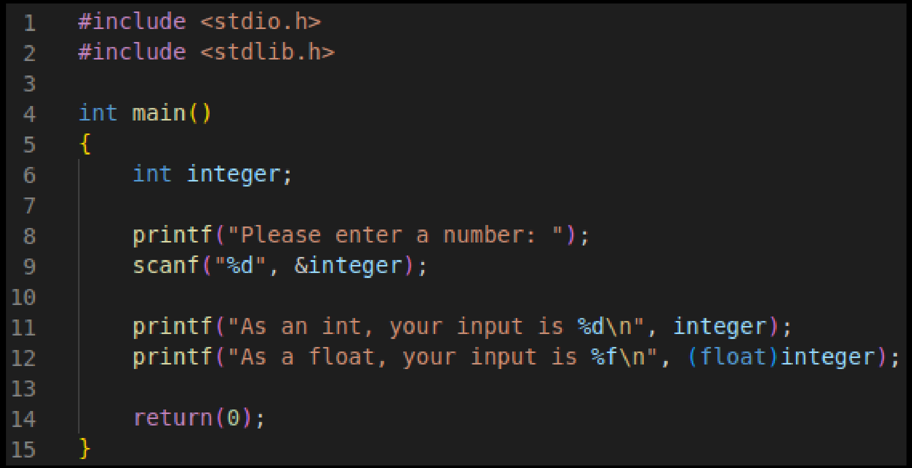
» One thing to note – there is a more effective way of storing structures known as a linked list. However, this won't be covered until the next slide pack as things get complicated.



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

#define BUFF 3

int main()
{
    struct kerbel
    {
        char name[32];
        int empNum;
    };

    struct kerbel employee[BUFF];

    for(int i = 0; i < BUFF; i++)
    {
        printf("Enter employee name: ");
        scanf("%s", employee[i].name);
        printf("Enter employee number: ");
        scanf("%d", &employee[i].empNum);
    }

    puts("\n*** Employee Data ***");
    printf("#\tName\tNumber\n");
    for(int i = 0; i < BUFF; i++)
        printf("%d\t%s\t%d\n", i + 1, employee[i].name, employee[i].empNum);

    return(0);
}
```

Type “Cast” and Win!

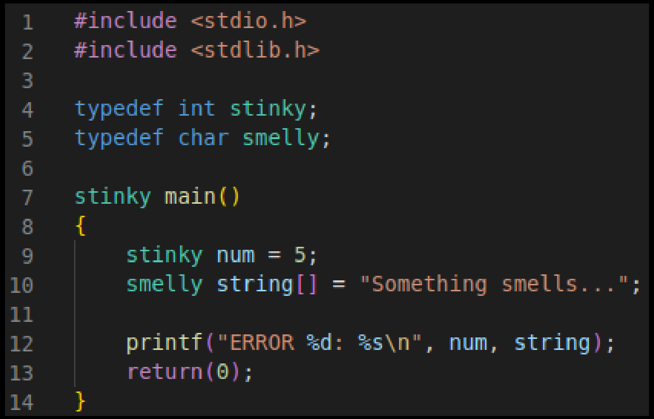
- Whilst it sounds like the start of a not so cryptic puzzle, typecasting in C is an important and useful feature.
- Typecasting is simply the process of taking one variable type, and declaring it as another type – useful if you have a variable which you don’t want to have to reassign or reprocess.
- In C, typecasting is done using this notation: `(<type>)var`:
  - `<type>` = The data type you want to reassign the variable to.
  - `var` = The variable you want to typecast.
- Huh, that wasn’t so hard. Give it a go yourself with the code example here!



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

int main()
{
    int integer;

    printf("Please enter a number: ");
    scanf("%d", &integer);

    printf("As an int, your input is %d\n", integer);
    printf("As a float, your input is %f\n", (float)integer);

    return(0);
}
```

Type “Def” and Win Also!

» Just a quick detour here to talk about `typedef` – have a guess at what it does from the example below.
- Feel free to copy it down and look at what it does!



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

typedef int stinky;
typedef char smelly;

stinky main()
{
    stinky num = 5;
    smelly string[] = "Something smells...";

    printf("ERROR %d: %s\n", num, string);
    return(0);
}
```

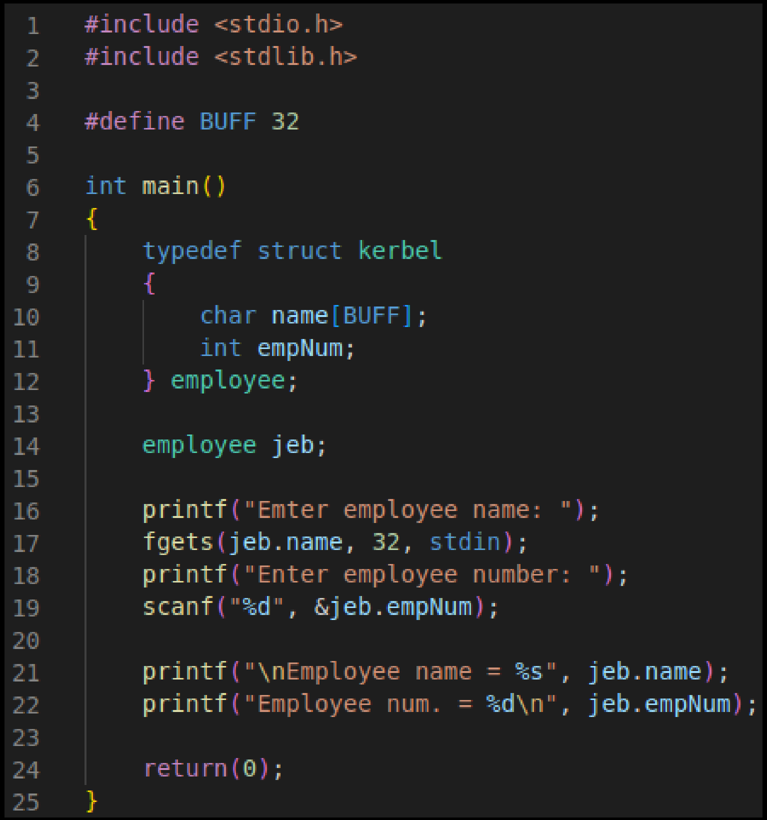
» If you didn’t get it, `typedef` allows us to redefine the names of variable types.

» Whilst we can use them to have fun with our variable names, their main use is with structures – jump over to the next slide and we’ll go over them!

Def-initely Useful

- Speedy exercise time! Copy down the example, compile, and run it.
  - We can see how typedef might come in useful now.
  - Instead of declaring the employee as `struct kerbel jeb;` , we can instead define it as simply `employee jeb;` !

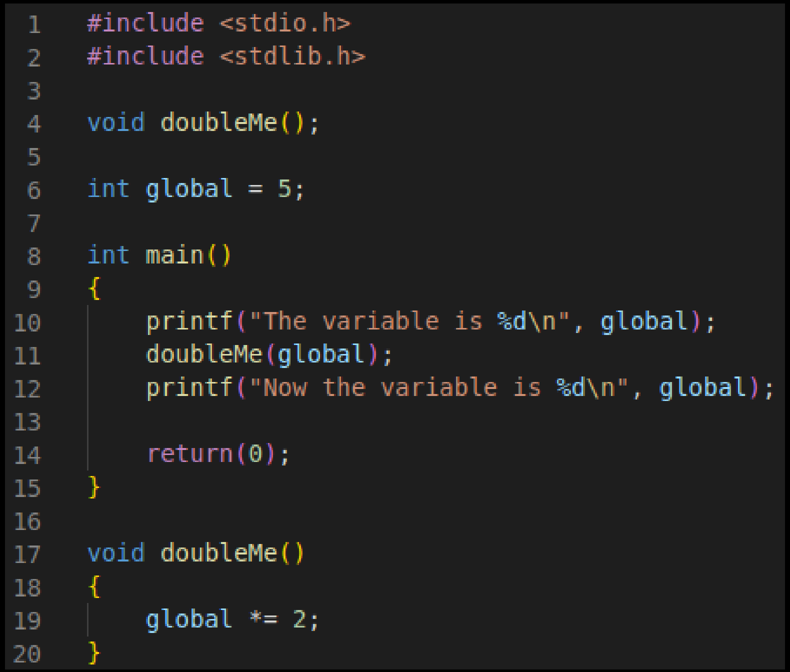
- Typedef declarations have their uses, but they're not necessary.
- You can define a structure either way – ultimately it comes down to your personal preference.



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

#define BUFF 32

int main()
{
    typedef struct kerbel
    {
        char name[BUFF];
        int empNum;
    } employee;

    employee jeb;

    printf("Emter employee name: ");
    fgets(jeb.name, 32, stdin);
    printf("Enter employee number: ");
    scanf("%d", &jeb.empNum);

    printf("\nEmployee name = %s", jeb.name);
    printf("Employee num. = %d\n", jeb.empNum);

    return(0);
}
```

Mister Worldwide

- We mentioned briefly in our function slides about the idea of local variables – well what if we want to declare a variable that we can use anywhere?
- Recall function prototyping – this allowed us to define a function that would be seen everywhere in a program.
  - I wonder if we could do the same with variables…
- Well thankfully we can! Take a look at the example and give it a run.
  - You'll notice that we don't actually pass anything to the function, yet the value of global changes as we'd expect!
  - This is because the variable is declared before the main function at the head of the program.
  - As a result, the variable can be seen by any function in the program!
  - Be warned – whilst you may think to simply declare every variable as a global for ease of programming, this is considered very bad programming practice. You will lose brownie points if you do this…



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

void doubleMe();

int global = 5;

int main()
{
    printf("The variable is %d\n", global);
    doubleMe(global);
    printf("Now the variable is %d\n", global);

    return(0);
}

void doubleMe()
{
    global *= 2;
}
```

Task Time - Structures

» Structs are fairly useful things, so let's see how well you've understood them!

1. Create a structure to store high scores.
   - There should be space for four high scores – each high score must contain a player name and a score.
   - Your program should initialise the first three slots with the following values:
     - Name: Jebediah | Score: 4
     - Name: Bill | Score: 8
     - Name: Bob | Score: 5
2. Create a simple maths game where you must correctly solve 10 random addition problems.
   - This code should appear after the struct you made before.
   - The two numbers to add together must be randomly selected between 1 and 100.
   - Your name and score should be saved to the remaining slot in the struct you created.
   - Once entered, the list of high scores should be displayed to the user.
3. Modify the code so that the scores are always displayed from high to low.

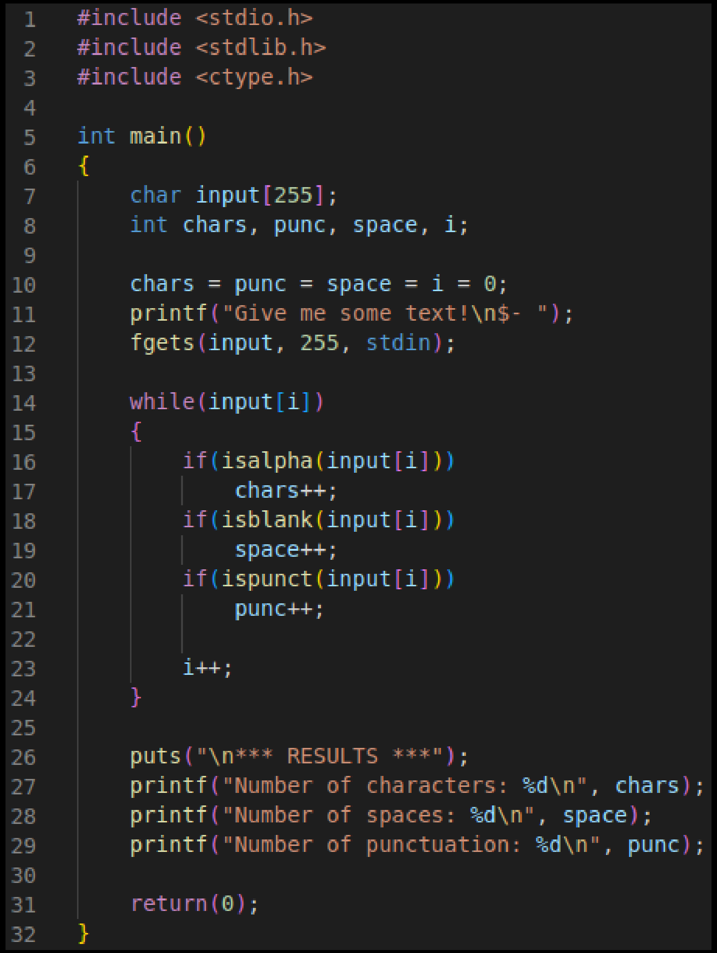
Char-med, I’m Sure

» Time to introduce a new library – the `<ctype.h>` library!
  - This library contains a bunch of macro functions which perform a series of tests and manipulations on characters.
  - All `ctype` functions accept an integer as an argument, as all ASCII characters can be represented as an integer between 0 – 255.
  - For most functions, testing functions begin with `is`, and manipulation functions begin with `to`.
  - Most `ctype` functions return either true or false.

» Copy the example down, compile, and run.

» You’ll notice it uses three functions from the ctype library:
  - `isalpha` checks if a character is a letter of the alphabet.
  - `isblank` checks for spaces.
  - `ispunct` checks for punctuation.
  - Pretty useful!

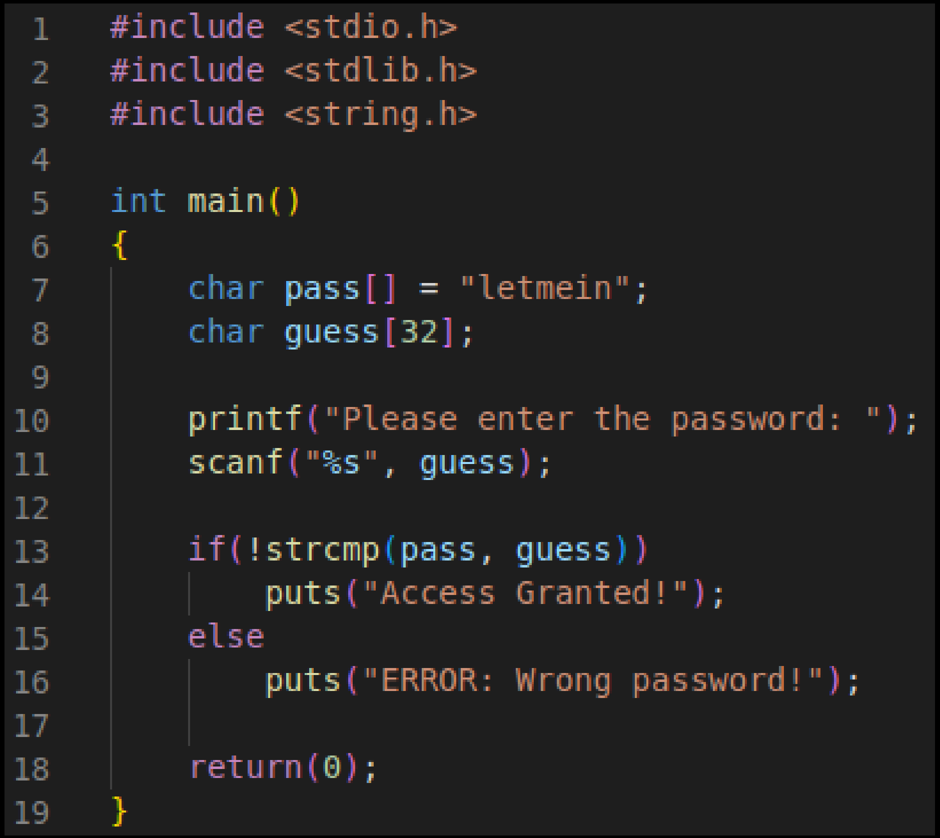
» There are a fair few more `ctype` functions – I’ll leave it up to you explore all of them…



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

int main()
{
    char input[255];
    int chars, punc, space, i;

    chars = punc = space = i = 0;
    printf("Give me some text!\n$- ");
    fgets(input, 255, stdin);

    while(input[i])
    {
        if(isalpha(input[i]))
            chars++;
        if(isblank(input[i]))
            space++;
        if(ispunct(input[i]))
            punc++;

        i++;
    }

    puts("\n*** RESULTS ***");
    printf("Number of characters: %d\n", chars);
    printf("Number of spaces: %d\n", space);
    printf("Number of punctuation: %d\n", punc);

    return(0);
}
```

String Theory

- » Another new library time! This time it's `<string.h>`.
  - This library contains a bunch of functions relating to string manipulation, comparison, copying, and more!
- » There's too many functions in `string.h` to talk about in these slides, so we'll cover the most useful and leave you to discover the rest!
  - You'll find out the existence of functions when Googling the answer to a problem.
  - This I find is often the best way of learning new functions – I'm still learning new things in C and I've been using it for five years!
- » Here's an example program for you: Copy, compile, and run.
  - The first function to explore is here: `strcmp`.
  - This function takes two strings and compares them to see if they are the same (hence **STR**ing **C**o**MP**are).
  - If they're the same, it returns 0, if they differ, it returns either a positive or negative value.
  - We use `!strcmp(pass, guess)` to ensure the if statement passes – remember that `strcmp` returns 0 if the two strings are the same!



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

int main()
{
    char pass[] = "letmein";
    char guess[32];

    printf("Please enter the password: ");
    scanf("%s", guess);

    if(!strcmp(pass, guess))
        puts("Access Granted!");
    else
        puts("ERROR: Wrong password!");

    return(0);
}
```

Challenge Two

- Create a game of Hangman which saves up to five games played in a row.

- The game should ask for your name when loaded.
- The game should pick a random word from a word list stored in an array.
- On each guess, the game should let the user know if they guessed a letter correctly or not.
  - If the user guessed a letter, every instance of it should be revealed.
  - If the user didn't guess a letter, a new part of the hangman should be revealed.
- If the user wins, their name and the number of guesses they got the word in should be saved to the struct.
  - If there is no space to add a new score, the game should just print the high scores.
  - The high scores should be printed in order from high to low.

- Bonus: If there are no spaces left in the score struct, the player's most recent score should replace the score in the score struct.