# What did we learn this term?

# Programming in C


Meme: Hussain Nawaz

# Programming in C

## DPST1091 C Language Techniques in the order they were taught

- Input/Output
- Variables
- If statements
- While statements (looping)
- Arrays
- Functions
- Pointers
- Characters and Strings
- Command Line Arguments
- Structures
- Memory Allocation
- Multi-File Projects
- Linked Lists
- Recursion

# C as a programming language

- A compiled language
- We use **dcc** as our compiler here, but there are others
  - clang
  - gcc
  - and others . . .
- Compilers read code from the top to the bottom
- They translate it into executable machine code
- All C programs must have a **main()** function, which is their starting point
- Compilers can handle multiple file projects
- We compile C files while we **#include** H files

# C and Compilation


Meme: Edward Ambrogio

Me: Mom, can I have kangaroo?
Mum: No, we have kangaroo at home
kangaroo at home:

```
printf("\n");
printf("  /\\   ___\n");
printf(" <__\\_/    \\\n");
printf("    \\_  /  _\\\n");
printf("      \\_\\ / \\\\\\n");
printf("        //   \\\\\\n");
printf("      =//     \\==\n");
printf("\n");
```

Meme: Khoi Nguyen

# Input/Output

**Scanf and Printf allow us to communicate with our user**

- **scanf** reads from the standard input
- **printf** writes to standard output
- They both use pattern strings like **%d** and **%s** to format our data in a readable way

```
// ask the user for a number, then say it back to them
int number;
printf("Please enter a number: ");
scanf("%d", &number);
printf("You entered: %d ", number);
```

# Alternatives for input/output

**We can get and put lines and characters also**

- **getchar** and **putchar** will perform input and output in single characters
- **fgets** and **fputs** will perform input and output with lines of text
- We can also use handy functions like **strtol** to convert characters to numbers so we can store them in integers

# Command Line Arguments

**When we run a program, we can add words after the program name**

- These extra strings are given to the main function to use
- **argc** is an integer that is the total number of words (including the program name)
- **argv** is an array of strings that contain all the words

# Command Line Arguments in use

```c
int main(int argc, char *argv[]) {
    printf("The %d words were ", argc);
    int i = 0;
    while (i < argc) {
        printf(" %s ", argv[i]);
        i++;
    }
}
```

When this code is run with: `$ ./args hello world`

It produces this: `The 3 words were ./args hello world`

# Hello World :P


Meme: James Comninos

# Variables

- **Variables**
- Store information in memory
- Come in different types:
  - **int, double, char, structs, arrays** etc
- We can change the value of variables
- We can pass the value of variables to functions
- We can pass variables to functions via pointers
- **Constants**
- **#define** allows us to set constant values that won't change in the program

# Simple Variables Code

```c
// GOKU will be treated as if it's 9001 in our code
#define GOKU 9001
int main(void) {
    // Declaring a variable
    int power;
    // Initialising the variable
    power = 7;
    // Assign the variable a different value
    power = GOKU;
    // we can also Declare and Initialise together
    int power_two = 88;
}
```

# if statements


Meme:Ronish Karmacharya

# if statements

**Questions and answers**

- Conditional programming
- Evaluate an expression, running the code in the brackets
- Run the body inside the curly brackets if the expression is true (non-zero)

```
if (x < y) {
    // This section runs if x is less than y
}
// otherwise the code skips to here if the
// expression in the () equates to 0
```

# While loops

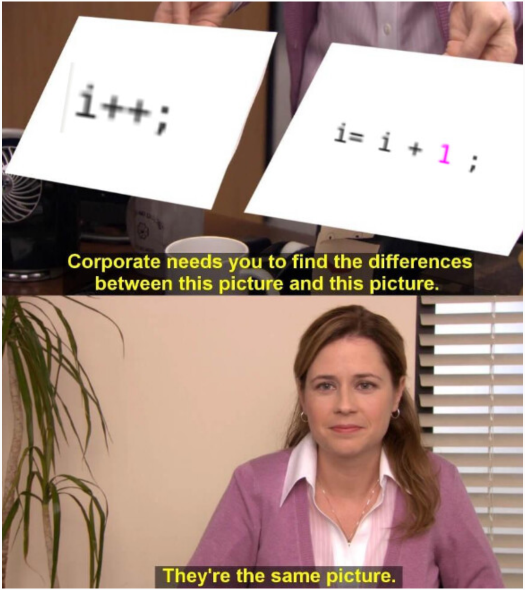
**Looping Code**

- While loops allow us to run the same code multiple times
- We can stop them after a set number of times
- Or we can stop them after a certain condition is met

**Loops are used for . . .**

- Checking all the values in a data structure ( **array** or **linked list** )
- Repeating a task until something specific changes
- and any other repetition we might need

# Looping


i++;
i= i + 1 ;
Corporate needs you to find the differences between this picture and this picture.
They're the same picture.

# While loop code - Arrays

**Very commonly used to loop through an array**

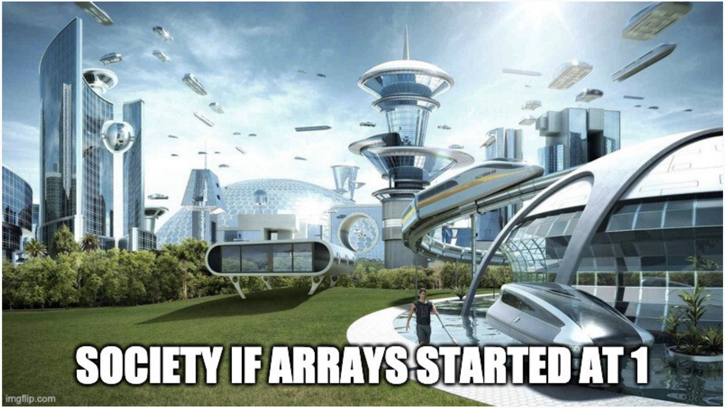
```c
int numbers[10] = { 0 };
// set array to the numbers 0-9 sequential
int i = 0;
while (i < 10) {
    // code in here will run 10 times
    numbers[i] = i;
    // increment the counter
    i++;
}
// When counter hits 10 and the loop's test fails
// the program will exit the loop
```

# While loop code - Linked Lists

**Looping through Linked Lists is also very common**

```
// current starts pointing at the first element of the list
struct node *current = head;
while (current != NULL) {
    // code in here will run until the current pointer
    // moves off the end of the list
    // increment the current pointer
    current = current->next;
}
// When current pointer is aiming off the end of the list
// the program will exit the loop
```

# Arrays




Meme: Jayden Matthews

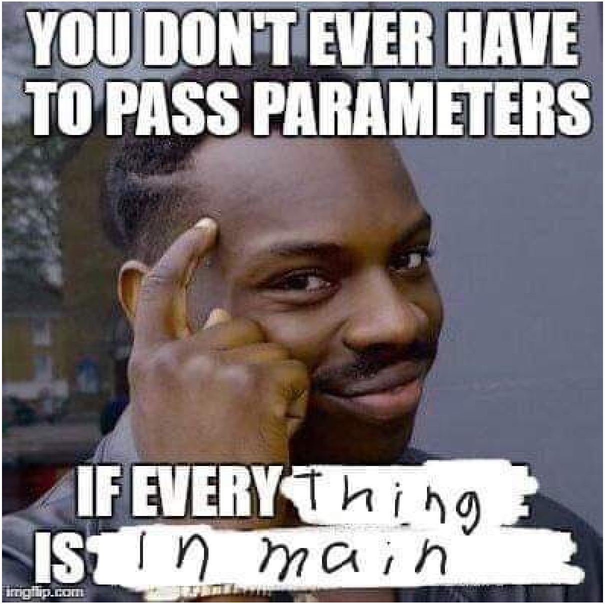
# Arrays

**Collections of variables of the same type**

- We use these if we need multiple of the same type of variable
- The array size is decided when it is created and cannot change
- Array elements arecollected together in memory
- Not accessible individually by name, but by index

# Array Code

```
int main(void) {
    // declare an array, all zeroes
    int m;
    marks[10] = {0};
    // set first element to 85
    marks[0] = 85;
    // access using an index variable
    int index = 3;
    marks[index] = 50;
    // copy one element over another
    marks[2] = marks[6];
    // cause an error by trying to access out of bounds
    marks[10] = 99;
```


YOU DON'T EVER HAVE TO PASS PARAMETERS
IF EVERY thing
IS in main
imgflip.com

# Functions

**Code that is written separately and is called by name**

- Not written in the line by line flow
- A block of code that is given a name
- This code runs every time that name is "called" by other code
- Functions have input parameters and an output

# Function Code

```c
// Function Declarations above the main or in a header file
int add(int a, int b);
int main(void) {
    int first = 4;
    int second = 6;
    int total = add(first, second);
    return 0;
}
// This function takes two integers and returns their sum
int add(int a, int b) {
    return a + b;
}
```

# Pointers

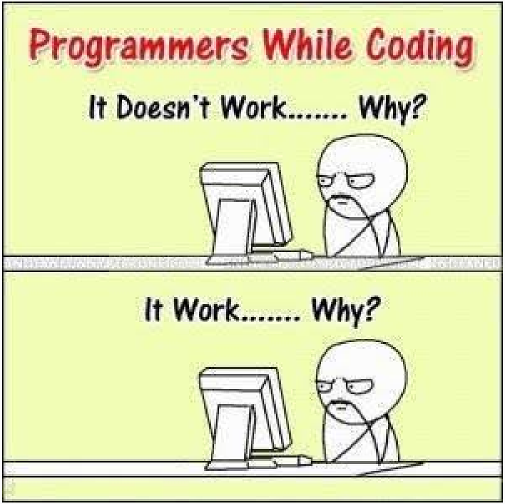
**Variables that refer to other variables**

- A pointer aims at memory (actually stores a memory address)
- That memory can be another variable already in the program
- It can also be allocated memory
- The pointer allows us to access another variable
- * dereferences the pointer (access the variable it's pointing at)
- **&** gives the address of a variable (like making a pointer to it)
- -> is used with structs to allow a pointer to access a field inside

# Simple Pointers Code

```
int main(void) {
    int i = 100;
    // the pointer ip will aim at the integer i
    int *ip = &i;
    printf("Value of variable at address %p is %d\n ", ip, *ip);
    // this second print statement will show the same address
    // but a value one higher than the previous
    increment(ip);
    printf("Value of variable at address %p is %d\n ", ip, *ip);
}
void increment(int *i) {
    *i = *i + 1;
}
```

# Problem Solving

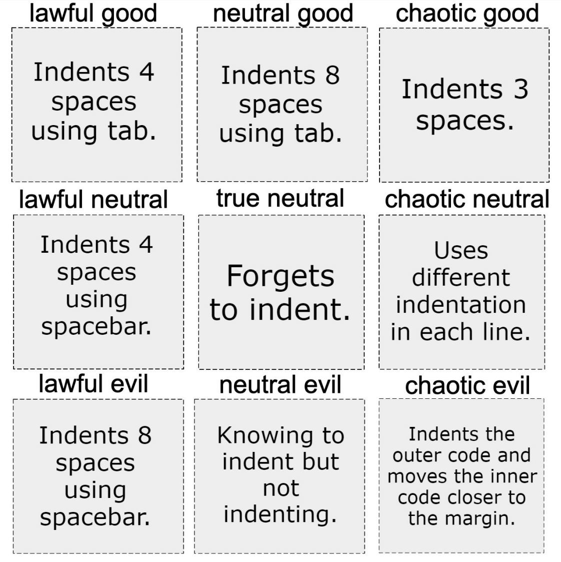


# Problem Solving

**Approach Problems with a plan!**

- Big problems are usually collections of small problems
- Find ways to break things down into parts
- Complete the ones you can do easily
- Test things in parts before moving on to other parts

# Code Style




Meme: Edison Fang

# Code Style

**Half the code is for machines, the other half for humans**

- Remember . . . readability == efficiency
- Also super important for working in teams
- It's much easier to isolate problems in code that you fully understand
- It's much easier to get help if someone can skim read your code and understand it
- It's much easier to modify code if it's written to a good style

# Code Reviews

**No one has to work without help**

- If we read each other's code . . .
- We learn more
- We help each other
- We see new ways of approaching things
- We are able to teach (which is a great way to cement knowledge)

# Debugging


Meme: Malachi Wu

# Debugging

**The removal of bugs (programming errors)**

- Syntax errors are code language errors
- Logical errors are the code not doing what we intend
- The first step is always: Get more information!
- Once you know exactly what your program is doing around a bug, it's easier to fix it
- Separate things into their parts to isolate where an error is
- Always try to remember what your intentions are for your code rather than getting bogged down

# Professionalism

**There's so much more to computing than code**

- What's the most important thing for a Software Professional?
- It's not always coding!
- It's caring about what you do and the people around you!
- Even in terms of pure productivity, it's going to get more work done long term than being good at programming
- If you care about your work, you will be fulfilled by it
- If you care about your coworkers you'll teach and learn from them and you'll all grow into a great team

# Course Survey - MyExperience

**Please fill out the survey!**

- Accessible via Moodle
- Or directly via *http://myexperience.unsw.edu.au/*
- This helps us a lot to figure out what is and isn't working in the course
- A lot of the course structure and even things like marks distribution is based on feedback from previous myExperience feedback
- We love feedback!

# Characters and Strings


Meme: Jennifer Truong


Meme: Malachi Wu

# Characters and Strings

**Used to represent letters and words**

**char** is an 8 bit integer that allows us to encode characters

- Uses ASCII encoding (but we don't need to know ASCII to use them)
- Strings are arrays of characters
- The array is usually declared larger than it needs to be
- The word inside is ended by a Null Terminator **'\0'**
- Using C library functions can make working with strings easier

# Characters and Strings in code

```c
// read user input
char input[MAX_LENGTH];
fgets(input, MAX_LENGTH, stdin);
printf(" %s\n ", input);
// print string vertically
int i = 0;
while (input[i] != '\0') {
    printf("%c\n", input[i]);
    i++;
}
```

# Structures

**Custom built types made up of other types**

**structs** are declared before use

- They can contain any other types (including other structs and arrays)
- We use a **.** operator to access fields they contain
- If we have a pointer to a struct, we use -> to access fields

# Structs in code

```
struct spaceship {
    char name[MAX_NAME_LENGTH];
    int engines;
    int wings;
};
int main(void) {
    struct spaceship xwing;
    strcpy(xwing.name, "Red 5");
    xwing.engines = 4;
    xwing.wings = 4;
    struct spaceship *my_ship = &xwing;
    // my ship takes a hit
    my_ship->engines--;
    my_ship->wings--;
}
```

# Memory

**Our programs are stored in the computer's memory while they run**

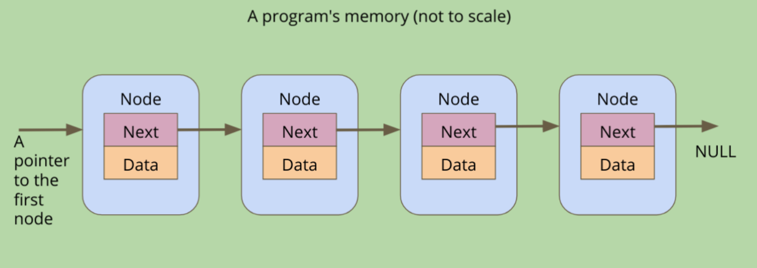
- All our code will be in memory
- All our variables also
- Variables declared inside a set of curly braces will only last until those braces close ( *what goes on inside curly braces stays inside curly braces* )
- If we want some memory to last longer than the function, we allocate it
- **malloc()** and **free()** allow us to allocate and free memory
- **sizeof** provides an exact size in bytes so malloc knows how much we need

# Memory code

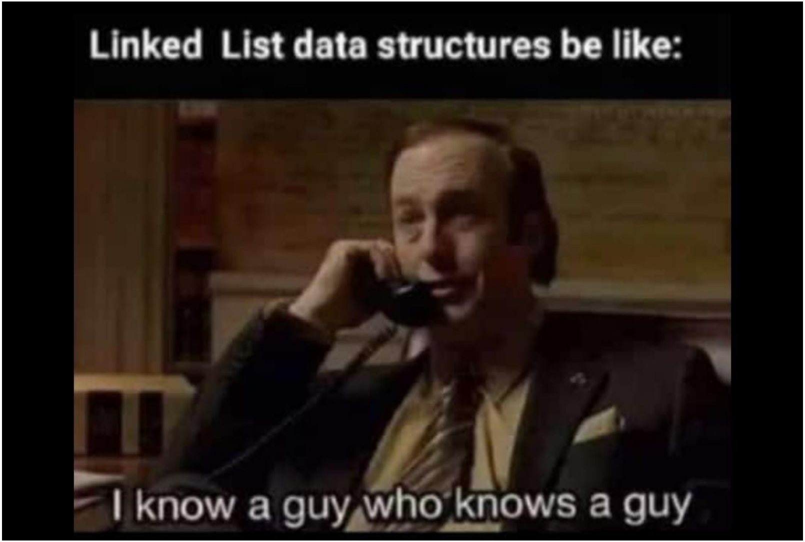
```
struct spaceship {
    char name[MAX_NAME_LENGTH];
    int engines;
    int wings;
};
int main(void) {
    struct spaceship *my_ship = malloc(sizeof(struct spaceship));
    strcpy(my_ship->name, "Millennium Falcon");
    my_ship->engines = 1;
    my_ship->wings = 0;
    // Lost my ship in a Sabacc game, free its memory
    free(my_ship);
}
```

# Linked Lists

- **Structs for nodes that contain pointers to the same struct**
- Nodes can point to each other in a chain to form a linked list
- Convenient because:
  - They're not a fixed size (can grow or shrink)
  - Elements can be inserted or removed easily anywhere in the list
- The nodes may be in separate parts of memory



# Linked Lists




Meme: Caleb Watts

# Linked Lists in code

```c
struct location {
    char name[MAX_NAME_LENGTH];
    struct location *next;
};
int main(void) {
    struct location *head = NULL;
    head = add_node("Tatooine", head);
    head = add_node("Yavin IV", head);
}
// Add a node to the start of a list and return the new head
struct location *add_node(char *name, struct location *list) {
    struct location *new_node = malloc(sizeof(struct location));
    strcpy(new_node->name, name);
    new_node->next = list;
    return new_node;
}
```

# Complications in Pointers, Structs and Memory

**What's a pointer?**

- It is a number variable that stores a memory address
- Any changes made to pointers will only change where they're aiming

**What does * do?**

- It allows us to access the memory that the pointer aims at (like following the address to the actual location)
- This is called "dereferencing" (because the pointer is a reference to something)

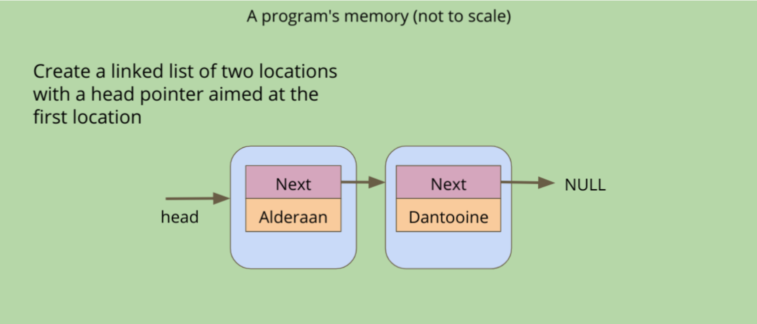
**What about -> ?**

- Specifically access a struct at the end of a pointer
- -> must point at one of the fields in the struct that the pointer aims at
- It will dereference the pointer AND access the field
- **Pointers to structs that contain pointers to other structs!**
- We can follow chains of pointers like **track->beat->note**

# Complicated Pointer Code

```c
int main(void) {
    // create a list with two locations
    struct location *head = add_node("Dantooine", NULL);
    head = add_node("Alderaan", head);
    // create a pointer to the first location
    struct location *alderaan = head;
    // set head to a newly created location
    head = malloc(sizeof(struct location));
    // What has happened to the alderaan pointer now?
    // What has happened to the variable that the head and alderaan
    // both pointed at?
}
```

# Keeping track of pointers

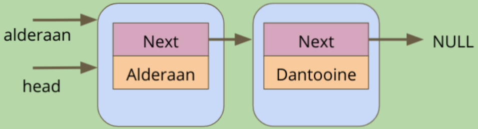


# Keeping track of pointers

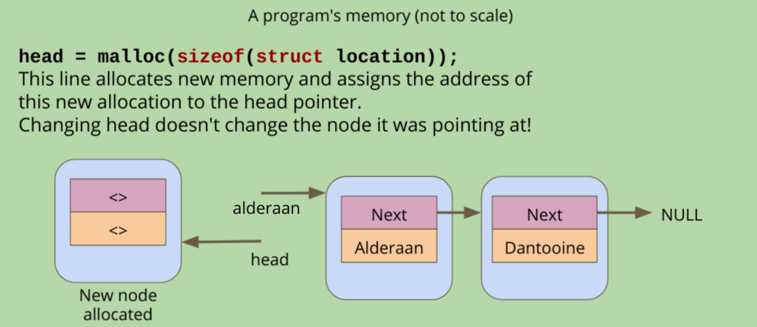
A program's memory (not to scale)

```
struct location *alderaan = head
```

This line creates a new pointer that's a copy of the head pointer. It is given the same value as head, which means it's aimed at the same memory address



# Keeping track of pointers



# Keeping track of pointers

**Remember:**

- Changing a pointer changes its value, a memory address
- Changing a pointer will change where it's aiming, nothing more!
- Once you use -> on a pointer, you're now looking at a struct field
- This means you are not changing that pointer, you have dereferenced it and accessed a field inside the struct

# Abstract Data Types

**Separating Declared Functionality from the Implementation**

- Functionality declared in a Header File
- Implementation in a C file
- This allows us to hide the Implementation
- It protects the raw data from incorrect access
- It also simplifies the interface when we just use provided functions

# Abstract Data Types Header code

```c
// Ship type hides the struct that it is
// implemented as
typedef struct ship_internals *Ship;
// functions to create and destroy Ships
Ship ship_create(char *name);
void ship_free(Ship ship);
// set off on a voyage of discovery
Ship voyage(Ship ship, int years);
```

# Abstract Data Types Implementation

```
struct ship_internals {
    char name[MAX_NAME_LENGTH];
};
Ship ship_create(char *name) {
    Ship new_ship = malloc(sizeof(struct ship_internals));
    strcpy(new_ship->name, name);
    return new_ship
}
void ship_free(Ship ship) {
    free(ship);
}
// set off on a voyage of discovery
Ship voyage(Ship ship, int years) {
    int discoveries = 0, years_past = 0;
    while (years_past < years) {
        discoveries++;
    }
    return ship;
}
```

# Abstract Data Types Main

- Including the Header allows us access to the functions
- The main doesn't know how they're implemented
- We can just trust that the functions do what they say

```c
#include "ship.h"
int main(void) {
    Ship my_ship = ship_create("Enterprise");
    my_ship = voyage(my_ship, 5);
}
```

# Recursion

**Functions calling themselves**

- A slightly inverted way of thinking about program flow
- The order of execution is determined by the Program Call Stack
- Chooses between a stopping case or a recursive case in the function

# A Recursive Function in code

```c
// Print out the names stored in the list in reverse order
// This is a recursive programming implementation
void rev_print(struct player *list) {
    if (list == NULL) {
        // stopping case (there are no elements)
        return;
    } else {
        // there are element(s)
        rev_print(list->next);
        fputs(list->name, stdout);
        putchar('\n');
    }
}
```

# Order of execution

- **More recursive function calls**
- Check if we're stopping, if so return
- Otherwise, call the function again with the tail (all remaining elements)
  - Check if we're stopping, if so return
  - Otherwise, call the function again with the tail (all remaining elements)
    - Check if we're stopping, if so return
    - Otherwise, call the function again with the tail (all remaining elements)
    - Then print the name of the current head of the list
  - Then print the name of the current head of the list
- Then print the name of the current head of the list

# So, you're programming now . . .

**Where do we go from here?**

- There's so much you can do with code now
- But there's also so much to learn
- Computing has more to offer than anyone can learn in a lifetime
- There's always something new you can discover
- It's up to you to decide what you want from it and how much of your life you want to commit to it
- Remember to care for yourselves and your work
- Enjoy yourselves, keep working on what you love and I hope to bask in your future glory

- Good luck, have fun :)