
COMP1511 - Programming Fundamentals

— Week 1 - Lecture 1 —

What are we talking about today?

First Hour

- Introductions
- Welcome to UNSW and Programming
- How COMP1511 works
 - How we will be teaching you
 - What we expect from you
- How to get help and the best ways to approach learning Programming

Second Hour

- What are these amazing machines we call computers?
- A first look at C
- Working in Linux

Who's Teaching you?

- Course Convener/Lecturer **Marc Chee**
 - cs1511@cse.unsw.edu.au
- Tutors and Admin
 - Too many to mention in person!
 - You will meet (or already have met) your tutor this week in class
- Course webpage
<https://webcms3.cse.unsw.edu.au/COMP1511/20T3/>
- Course Forum (you should have received an email invite to this)



Welcome to UNSW and Programming

What is studying at University like?

- You are now in control of your own education

What is programming?

- Talking to computers . . . in a language that we can both understand
- Learning to solve problems
- We give the computer a procedure, not an answer



How is this course going to run?

We will teach you how to code, with no assumptions of prior knowledge

- How to think when programming
- The C programming language
- How to solve problems with code

Course Format

- Lectures
- Live Streams
- Tutorials
- Laboratory Sessions
- Weekly Tests and Assignments
- Help Sessions

Term Schedule and Flexibility Week

UNSW's 10 Week Term

- The subject runs from week 1 to 10
- Week 6 is the flexibility week

Flexibility Week

- Optional and Guest Lectures
- No official class content
- Help Sessions will be running for Assignment support

Lectures

Two hour sessions

- Streamed online via YouTube Live (recordings will be available)
- Tuesday 11am-1pm (AEST then AEDT after October 4th)
- Friday 11am-1pm (AEST then AEDT after October 4th)

If you have a question, feel free to ask in live chat, we often have tutors in the chat to help answer

There's only one rule: No one disrupts anyone else's education

Lecture Content

- Theory - What are we trying to understand?
- Demonstrations - Some live coding to show you how some things work
- Problem Solving - How do we decide what to code?
- Other stuff - Outside of programming, what's important?

Lecture slides (and other materials) are available from the Course Website:
[\(https://webcms3.cse.unsw.edu.au/COMP1511/20T3/\)](https://webcms3.cse.unsw.edu.au/COMP1511/20T3/)

Lecture recordings will be in the YouTube playlist and linked via the Course Website

Tutorials

A one hour classroom environment in Blackboard Collaborate

- Go further in depth into the topics we're teaching
- Actual practical working of tasks and problems we've given you
- Learning how to solve problems before you write the code!

Tutorial Questions will be available in advance of the tutorials

Tutorials are a good place for interactive learning. You'll have time to discuss and work through problems there

They will be held in an online classroom called Blackboard Collaborate

Laboratory Sessions

Two hour laboratory sessions that follow immediately from tutorials

- Continuing in Blackboard Collaborate (same session as the tutorial)
- Practical coding including working in small groups
- Time to have one on one conversations with your tutors
- Lab exercises will be marked automatically and count towards your final marks (10%)
- There are challenge exercises for earning bonus marks (not necessary and some are hard enough that they'll eat up a lot of time)

Weekly Programming Tests

Self-run practice for exam situations

- 1 hour tests under self-run exam conditions with automated marking
- Running from weeks 3-5 and 7-10
- These are ways for you to gauge your current progress
- Tests count towards your final mark (5%)

No Spoilers

- We're not going to discuss Weekly Tests publicly until after they're due
- No spoiling the surprise on the forums until everyone's seen them

Assignments

Larger scale projects

- Individual work
- These will take you a few weeks and will test how well you can apply the theory you've learnt
- There are two Assignments due in Weeks 6 and 10
 - Late penalties of 1% per hour apply (this reduces your maximum possible mark)
- Assignment 1 is worth 15% of your final mark
- Assignment 2 is worth 25% of your final mark

Help Sessions

Optional Sessions scheduled during the week

- Also held using Blackboard Collaborate
- Some one on one consultation with tutors
- Time for you to ask individual questions or get help with specific problems
- Schedule will be up on the Course Website soon
- These are particularly active around Assignments

Live Streaming

Optional Extra Content

- Extra streams outside of normal lecture times
- Marc will do some live coding
- He will also answer your questions live
- Primarily, these are to give you information about the Assignments
- These will be scheduled at Assignment release dates (mostly)

Exam

Take-home Open-book exam

- Expected workload of around 3-5 hours total
- You'll be given a series of problems to solve in C
- You will also be expected to read some C and show you understand it
- There will also be some questions covering programming ideas

Exam Hurdles

- Parts of the exam are competency hurdles
- These questions must be answered correctly to pass the course

Total Assessment

- 10% Labs
- 5% Weekly Tests
- 15% Assignment 1
- 25% Assignment 2
- 45% Exam

To pass the course you must:

- Score at least 50/100 overall
- Solve problems using arrays in the final exam
- Solve problems using linked lists in the final exam

Special Consideration & Supplementary Assessment

Special Consideration

- Support for any issues that make it difficult for you to study
- <https://student.unsw.edu.au/special-consideration>
- You can apply now if you have existing reasons (or later if something comes up)

A Supplementary exam can be offered to students granted Special Consideration for the exam

- Identical in format to the main exam
- Will be held in January (11-15th January, exact date to be confirmed)

Course Textbook

This is an optional book if you wish to use one

- *Programming, Problem Solving, and Abstraction with C*
Alistair Moffat, Pearson Educational, Australia, 2012,
ISBN 1486010970

Code of Conduct

This course and this University allows all students to learn, regardless of background or situation

Remember the one rule . . . you will not hinder anyone else's learning!

Anything connected to COMP1511, including social media, will follow respectful behaviour

- No discrimination of any kind
- No inappropriate behaviour
 - No harassment, bullying, aggression or sexual harassment
- Full respect for the privacy of others

Plagiarism

Plagiarism is the presentation of someone else's work or ideas as if they were your own.

Any kind of cheating on your work for this course will incur penalties (see the course outline for details)

Collaboration on individual assessments like Assignments and Weekly Tests is considered plagiarism

And really . . . if you don't spend the time to learn the content, the only person who loses is you

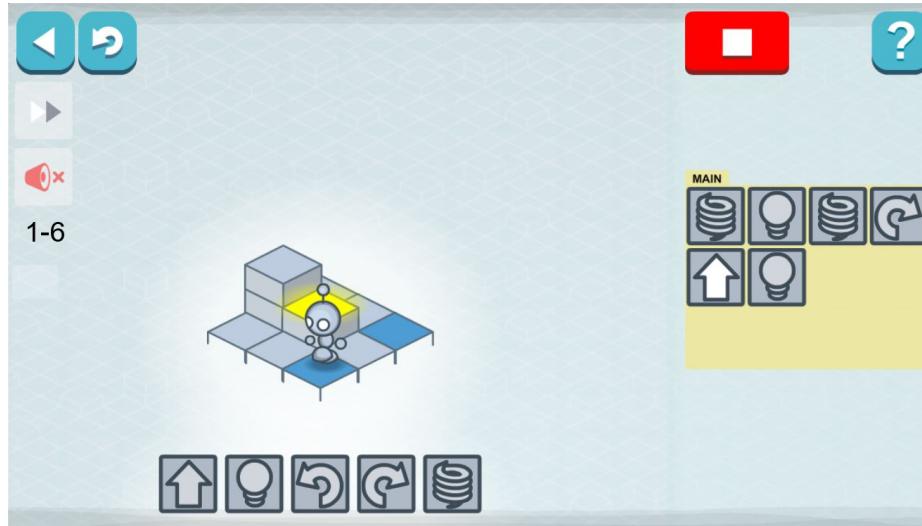
Collaboration vs Plagiarism

- Discussion of work and algorithms is fine (and encouraged)
- The internet has a lot of resources you should learn to use, just make sure you credit your sources
- No collaboration at all on individual assignments
- Your submissions are entirely your own work
 - Don't use other people's code
 - Don't ask someone else to solve problems for you (even verbally)
 - Don't provide your code to other people
- At best, you'll lose the marks for the particular assignment
- At worst, you'll be asked to leave UNSW
- And even worse . . . you won't learn what you paid all this money and time to learn

Break Time!

Let's take five minutes in between lecture sections

- There's a fun little app called **Lightbot** (<http://lightbot.com>)
- Also available on iOS and Android



How to succeed in COMP1511 (and life)

A simple idea:

The more time you spend practising something, the better you get at it

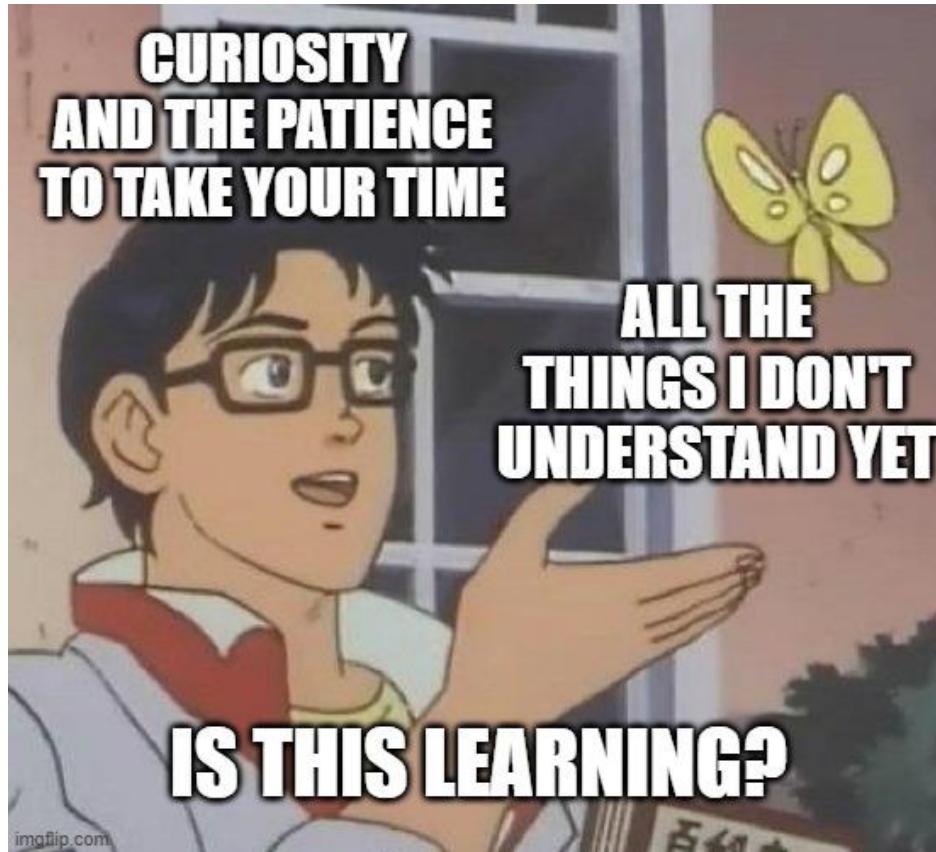
We've set up the course to ease your learning . . .

If you just follow what we're teaching, that will be a good start

- Complete all tutorial and lab work
- Complete all assignment work

Try not to focus on how good you are (or are not), but the real focus is on whether you're improving or not

Learning is a long journey ...



If you want more info ...

- Reading
 - Course webpage
 - Course forum
- Recorded Lectures (replay YouTube Streams)
- One on One
 - Help Sessions
 - Ask your tutor during lab sessions
- Serious Issues
 - Email cs1511@cse.unsw.edu.au
 - The Nucleus (student hub: nucleus.unsw.edu.au)
 - CSE Help Desk (<http://www.cse.unsw.edu.au/~helpdesk/>)

Speaking of Email

Make sure you have your UNSW email address set up

If you need to contact the course, please contact your tutor first

You may already have received an email from them!

Otherwise, use cs1511@cse.unsw.edu.au

Use your UNSW email address and make sure you include your zID so we know who we're helping

What is a Computer?

A tool . . . a machine . . .

The ultimate tool in its ability to be reconfigured for different purposes.

The key elements:

- A processor to execute commands
- Memory to store information

History of Computers

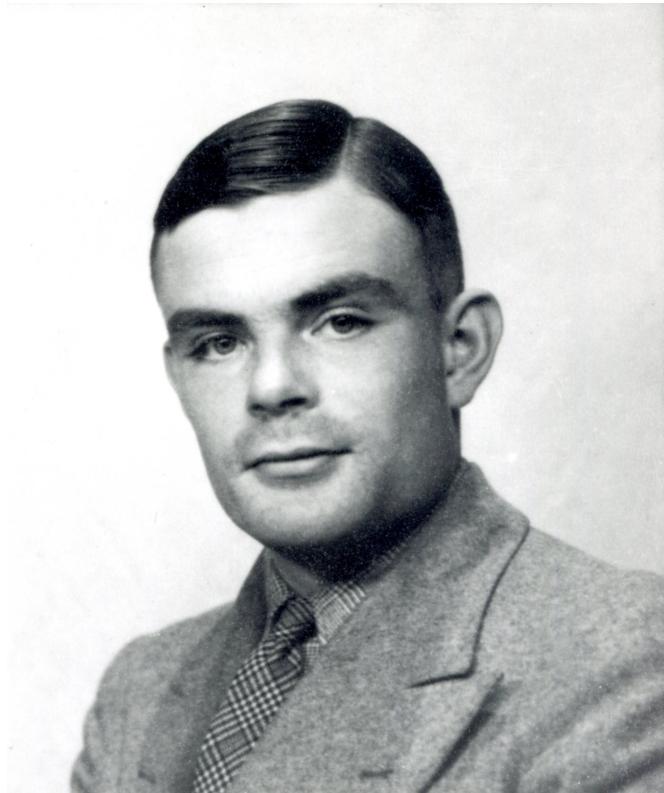
Humans have been using calculation and data storage tools for millennia

The first concepts of a programmable computer were from around 1835

- Charles Babbage designed the first “Analytical Engine”
- Ada Lovelace was the first person to write a computer program



Computers this Century



In 1944, the computer Colossus is made in the UK to break German codes

Alan Turing was instrumental in the Allied effort in World War 2

He also developed the concept of the Turing Machine, which is the basis of all our modern computers

Modern Computing

We now have much more processing capability than in the past

So what can we do with it?

- Realtime solutions to very difficult problems
- Global simulations for climate and weather patterns
- Connecting individuals globally in communication networks
- Simulating highly complex virtual environments
- Deciding what you want to watch next on Netflix . . .

And maybe you will come up with something new to do with them in the future . . .

Using CSE's Computing Resources

Our labs are running **Linux** with the basic tools necessary to get started

You will definitely want to get your own computer ready to code with:

- VLAB allows you to remotely use CSE's resources
- Visual Studio Code with SSH to connect to CSE
- or - You can set up a programming environment on your own computer
(check the course website for links to guides)

For COMP1511 we need:

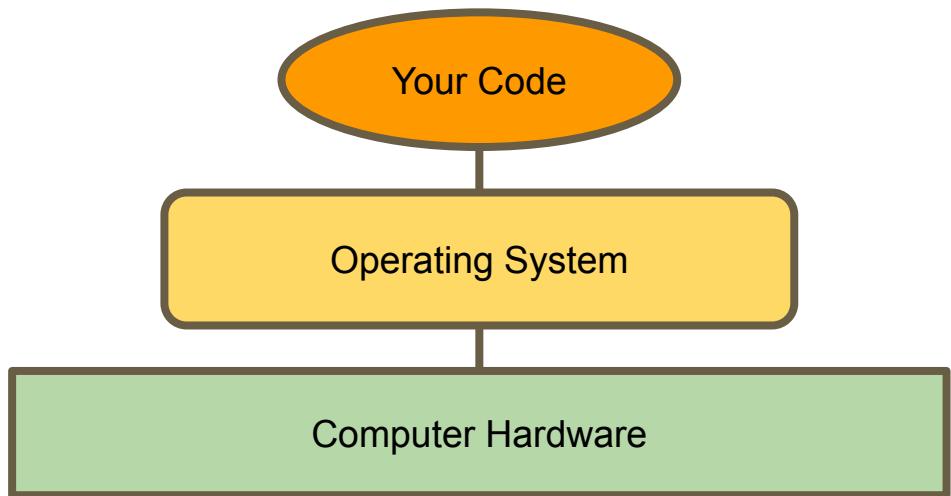
- A text editor like gedit
- A compiler (we use dcc)

Working in Linux

The first thing is to get setup with a simple programming environment

Here at CSE we use the Linux Operating System

An Operating System sits between our code and the computer, providing essential services



Using a Terminal

The main interface to Linux is a terminal

This means all our interaction is in text

Some commands:

- **ls**
 - Lists all the files in the current directory
- **mkdir *directoryName***
 - Makes a new directory called *directoryName*
- **cd**
 - Changes the current directory
- **pwd**
 - Tells you where you are in the directory structure at the moment

What the basics look like

gedit

- A basic text editor
- Helps out a little by highlighting C in different colours

gcc/gcc

- A compiler - A translator that takes our formal human readable C and turns it into the actual machine readable program
- The result of the compiler is a program we can "run"

You can use VLAB to access CSE's editor and compiler

Programming in C

Programming is like talking to your computer

We need a shared language to be able to have this conversation

We'll be looking at one particular language, **C** and learning how to write it

C is:

- A clear language with defined rules so that nothing we write in it is ambiguous
- Many modern programming languages are based on C
- A good starting point for learning how to control a computer from its roots

Let's see some C

```
// Demo Program showing output
// Marc Chee, September 2020

#include <stdio.h>

int main(void) {
    printf("Hello World.\n");
    return 0;
}
```

Comments

```
// Demo Program showing output  
// Marc Chee, September 2020
```

Words for humans

- Half our code is for the machine, the other half is for humans! (roughly)
- We put “**comments**” in to describe to our future selves or our colleagues what we intended for this code
- `//` in front of a line makes it a comment
- If we use `/*` and `*/` everything between them will be comments
- The compiler will ignore comments, so they don’t have to be proper code

#include

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

#include is a special tag for our compiler

It asks the compiler to grab another file of code and add it to ours

In this case, it's the Standard Input Output Library, allowing us to make text appear on the screen (as well as other things)

The "main" Function

```
int main(void) {  
    printf("Hello World.\n");  
    return 0;  
}
```

A function is a block of code that is a set of instructions

Our computer will run this code line by line, executing our instructions

The first line has details that we'll cover in later lectures

- `int` is the output - this stands for integer, which is a whole number
- `main` is the name of the function
- `(void)` means that this function doesn't take any input

The Body of the Function

```
int main(void) {  
    printf("Hello World.\n");  
    return 0;  
}
```

Between the { and } are a set of program instructions

`printf()` makes text appear on the screen. It is actually another function from `stdio.h` which we included.

`return` is a C keyword that says we are now delivering the output of the function. A main that returns 0 is signifying a correct outcome of the program

Editing and Compilation

We can open a terminal now and try the code we've just looked at

In the linux terminal we will open the file to edit by typing:

```
gedit helloworld.c &
```

Once we're happy with the code we've written, we'll compile it by typing:

```
gcc helloworld.c -o helloworld
```

The **-o** part tells our compiler to write out a file called "helloworld" that we can then run by typing:

```
./helloworld
```

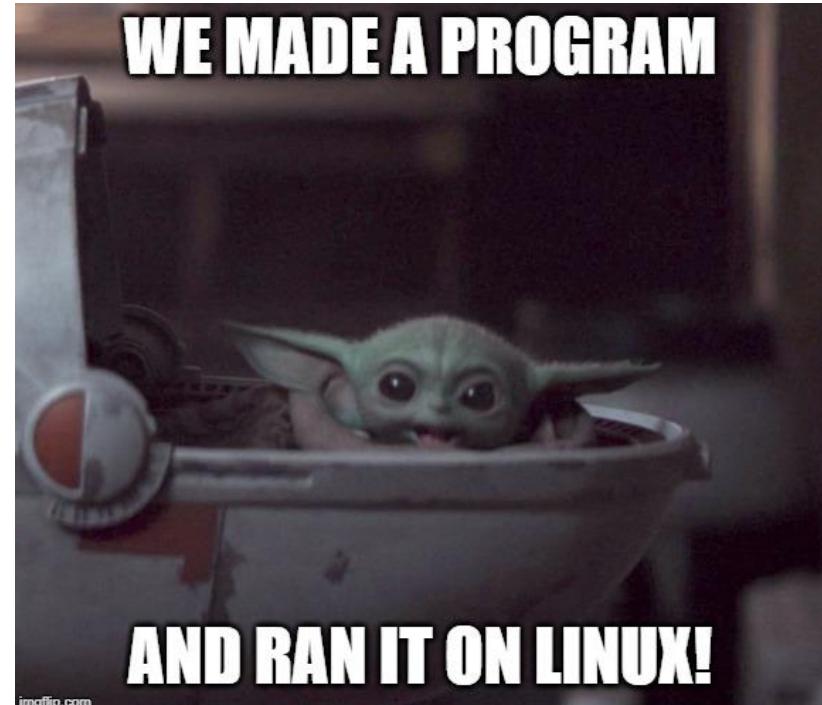
The **./** lets us run the program "helloworld" that is in our current directory

One working Program!

That's one program working!

What to do next?

- Try this yourself!
- Try it using VLAB via your own computer
- Try setting up a programming environment on your own computer (differing levels of difficulty depending on your operating system)



What did we learn today?

- COMP1511 and administration
 - Where to find resources (course webpage and forum!)
 - What your responsibilities are and how to succeed in programming
-
- A brief look at the history of computers
 - An overview of how they work
 - Your very first C program
 - Using the basics of Linux

COMP1511 - Programming Fundamentals

— Week 1 - Lecture 2 —

What did we learn last lecture?

Introductions

- Course Administration
- Do you know where to find help? (Course website and forum!)
- Tips on how to do well in Programming

A first look at programming

- A brief look at computers and history
- Our very first C program . . .
- . . . Running in Linux!

What are we covering today?

Variables

- Variables and how we store information
- Some basic maths in code

Conditionals and Branching Code

- Conditionals - Running code based on questions and answers
- The 'if' statement

A brief recap

```
// Demo Program showing output
// Marc Chee, September 2020

#include <stdio.h>

int main(void) {
    printf("Hello World.\n");
    return 0;
}
```

How does a computer remember things?

Ones and Zeros

- Computer memory is literally a big pile of on-off switches
- We call these **bits**
- We often collect these together into bunches of 8 bits
- We call these **bytes**

Bits and Bytes

- We are going to be working with chunks of memory made up of specific sizes
- You may have heard of things like 32bit and 64bit systems

Variables

A Variable is . . .

- Our way of asking the computer to remember something for us
- Called a "variable" because it can change its value
- A certain number of bits that we use to represent something
- Made with a specific purpose in mind

We're going to start out with two types of number variables:

- **int** - integer, a whole number (eg: 0,1,2,3)
- **double** - floating point number (eg: 3.14159, 8.534, 7.11)

Naming your Variables

Remember . . . Half our coding is for people!

- Names are a quick description of what the variable is
 - Eg: "answer" and "diameter"
 - Rather than "a" and "b"
- We always use lower case letters to start our variable names
- C is case sensitive:
 - "ansWer" and "answer" are two different variables
- C also reserves some words
 - "return", "int" and "double" can't be used as variable names
- Multiple words
 - We can split words with underscores: "long_answer" or capitals: "shortAnswer"

int

Integer

- A whole number, with no fractions or decimals
- Most commonly uses 32 bits (which is also 4 bytes)
- This gives us exactly 2^{32} different possible values
- This also means the integer is limited, it can't just hold any number
- Ranges from -2^{31} to $2^{31} - 1$



double

A double-sized floating point number

- A decimal value - "floating point" means the point can be anywhere in the number
 - Eg: 10.567 or 105.67 (the points are in different places in the same digits)
- It's called "double" because it's usually 64 bits, hence the double size of our integers

Code for Variables

```
int main (void) {
    // Declaring a variable
    int answer;
    // Initialising the variable
    answer = 42;
    // Give the variable a different value
    answer = 7;

    // we can also Declare and Initialise together
    int answer_two = 88;
}
```

Printing Variables

printf

- Not just for specific messages we type in advance
- We can also print variables to our display

```
// printing a variable
int number = 7;
printf("My number is %d\n", number);
```

- **%d** - where in the output you'd like to put an int
- After the comma, you put the name of the variable you want to write

Printing more variables

printf with multiple variables

```
// print two variables
int first = 5;
int second = 10;
printf("First is %d and second is %d\n", first, second);
```

The variables will match the symbols in the same order as they appear

Printing different types of variables

`%d` is for ints, `%lf` is for doubles

```
// print an int and a double
int diameter = 5;
double pi = 3.14159;
printf("Diameter is %d, pi is , %lf\n", diameter, pi);
```

The `%d` and `%lf` are symbols that are part of `printf`

- `%d` stands for “decimal integer”
- `%lf` stands for “long floating point number” (a double)

Reading Input into Variables

scanf

Reads input from the user in the same format as printf

Note the & symbol that tells scanf where the variable is

```
// reading an integer
int input;
printf("Please type in a number: ");
scanf("%d", &input);

// reading a double
double inputDouble;
printf("Please type in a decimal point number: ");
scanf("%lf", &inputDouble);
```

Constants

**Constants are like variables, only
they never change**

Can't call it a variable if it isn't . . .
variable

Sometimes we will use fixed numbers
using a special command, **#define**

We name them in all caps so that we
remember that they're not variables

```
// Variables demo
// Marc Chee, February 2019

#include <stdio.h>

#define PI 3.14159265359
#define SPEED_OF_LIGHT 299792458.0

int main (void) { ... }
```

Maths

Numbers and Arithmetic

A lot of arithmetic operations will look very familiar in C

- `+, -, *, /`
- These will happen in their normal mathematical order
- We can also use brackets to force precedence

```
// some basic maths
int x = 5;
int y = 10;
int result;
result = (x + y) * x;
printf("My maths comes out to: %d\n", result);
```

In more detail...

There are a lot of little details with the way variables are set up!

- These next three slides are not generally needed to use variables
- But they are some interesting and possibly dangerous side effects of how they're built

int issues

Some things to look out for

- **Overflow and Underflow**
- If we add two large ints together, we might go over the maximum value, which will actually roll around to the minimum value and possibly end up negative (trivia: Look up Ariane 5 explosion, a simple error like this caused a rather large problem)
- ints might not always be 32 bits . . . dependent on Operating System

double issues

Doubles also have a few things to think about . . .

- No such thing as infinite precision
- We can't precisely encode a simple number like $\frac{1}{3}$
- If we divide 1.0 by 3.0, we'll get an approximation of $\frac{1}{3}$
- The effect of approximation can compound the more you use them

Division and Variable types

Division does some interesting things in C

- If either numbers in the division are **doubles**, the result will be a **double**
- If both numbers are **ints**, the result will be an **int**
 - Eg: $3/2$ will not return 1.5, because ints are only whole numbers
- **ints** will always drop whatever fraction exists, they won't round nicely
 - $5/3$ will result in 1
- **%** is called **Modulus**. It will give us the remainder from a division between integers
 - Eg: $5 \% 3$ will result in 2 (3 goes into 5 once, with 2 left over)

Break Time

We'll take 5 minutes here before launching into some more C

What's a program?

- A series of instructions
- Some are: Store this information (using variables)
- Some are: Perform this task
- Usually these will be performed in order from top to bottom
- It's a little bit like a cooking recipe!

Some More Linux

File operations

- `cp source destination` - copy
- `mv source destination` - move (can also be used to rename)
- `rm filename` - remove a file (delete)

The `-r` tag can be added to `cp` or `rm` to recursively go through a directory and perform the command on all the files

Eg: “`cp -r COMP1511 COMP1511Backup`” will copy all files from my `COMP1511` directory to my `COMP1511Backup` directory

Letting our Computer Make Decisions

- Sometimes we want to make decisions based on what information we have at the time
- We can let our program branch between sets of instructions
- In C this is the **if** statement



The if statement

An **if** is like a question and an answer

- First we ask a question
- If we get the right answer, we run some code

```
// the code inside the curly brackets
// runs if the expression is true (not zero)
if (expression) {
    code statement;
    code statement;
}
```

else - Adding to if statements

We can expand beyond the simple **if** by adding the **else** statement

```
if (expression) {  
    // this runs if the expression results in  
    // anything other than 0 (true)  
} else {  
    // this runs if the earlier expression  
    // results in 0 (false)  
}
```

Chaining ifs and elses

This code shows ifs and elses joined together

```
if (expression1) {
    // this runs if expression1 is true
    // (anything other than 0)
} else if (expression2) {
    // this runs if expression1 is false (results in 0)
    // and expression2 is true (results in anything
    // other than 0)
} else {
    // this runs if both expression1 and
    // expression2 result in false (0)
}
```

Asking the right Question

For our **if** to perform correctly, we need to know how to ask questions

Relational Operators work with pairs of numbers:

- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to
- == equals (we've already used "=" to assign values, so we use "==" to ask a question)
- != not equal to

All of these will result in 0 if false and a 1 if true

Chaining Questions Together

We use **Logical Operators** to compare expressions

The first two are used between two expressions

- `&&` *AND* equates to 1 if both sides equate to 1
- `||` *OR* is 1 if either side is 1

This is used in front of an expression

- `!` *NOT* is the opposite of whatever the expression was

Some examples

Expression	Result
5 < 10	1
8 != 8	0
12 <= 12	1
7 < 15 && 8 >= 15	0
7 < 15 8 >= 15	1
!(5 < 10 6 > 13)	0

Let's create a program with our new skills

This program will help us in our games of "Catacombs and Large Reptiles"

1. A user will roll two dice and tell us the result of each die
2. Our program will add them together and check them against a target number that only the program knows.
3. It will then report back whether the total of the dice was higher, equal or lower than the secret number.

What does our program need?

All recipes need ingredients

- A way to “talk” to our user
 - We know about **printf**
- A way to receive input
 - We learnt about **scanf** today
- A way to compare numbers . . .
 - **Relational Operators**
- . . . Against a secret number
 - A **variable** or a **constant**
- A way to run different code depending on the number comparisons
 - **If** and **else** conditional statements

Describe our program

We'll write some comments and include stdio.h for printf and scanf

```
/* The D&D Dice checker example
Marc Chee, September 2020

This small example will ask the user to input the
result of two dice rolls.
It will then check the sum of them against its
secret number.
It will report back:
    success (higher or equal)
    or failure (lower)
*/
#include <stdio.h>
```

Define our Secret Number

`#define` is nice for things that we know aren't going to change. Note the use of all caps to signify a constant and underscores to show different words

This will go after our `#include`, but before our main

```
#define SECRET_TARGET 7
```

Main function

We always need a main function for C to know where our program starts

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

Setting up some variables

We know we're going to be getting dice values from our user, so we can start by declaring them

```
// set up some dice variables so we can store numbers
int dieOne;
int dieTwo;
```

Talk to our user and ask them for their dice rolls

Using printf and scanf, we can print words to the screen and read numbers

We store the two die rolls in the variables we set up earlier

```
// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then read in a number they type in the terminal
scanf("%d", &dieOne);
// repeat for the second die
printf("Please enter your second die roll: ");
scanf("%d", &dieTwo);
```

Total the dice

Using some basic arithmetic, we calculate our total

We then save that value in a “**total**” integer variable

```
// calculate the total and report it
int total = dieOne + dieTwo;
printf("Your total roll is: %d\n", total);
```

Test the Total against the Target

We now use an if statement to test for success

```
// Now test against the secret number
if (total >= SECRET_TARGET) {
    // success
    printf("Skill roll succeeded!\n");
}
```

What about the failure?

The other option, which we actually don't have to test for, because it's all that's left.

```
// Now test against the secret number
if (total >= SECRET_TARGET) {
    // success
    printf("Skill roll succeeded!\n");
} else {
    // the same as total < SECRET TARGET
    // but we don't have to test it because
    // we've already checked all other
    // possibilities
    printf("Skill roll failed!\n");
}
```

We have a dice check program!

The finished code will be on the course website for you to peruse

Challenges:

Can you modify this code to:

- Detect exact ties as well as success and failure?
- What about the idea of a "critical double"? Can you detect when the player rolled the same number on both dice and report it as a "critical" success, tie or failure?

What did we learn today?

Variables

- They come in different sizes and types
- Printing from variables and reading user input into variables
- Using maths to manipulate variables

Conditions

- **if** and **else** statements
- Relational Operators and Logical Operators

COMP1511 - Programming Fundamentals

— Week 2 - Lecture 3 —

What did we learn last week?

COMP1511 as a subject

C as a programming language

- Basic Syntax
- `printf` and `scanf`
- Variables (ints and doubles)
- Maths operators (+, -, *, / and %)
- Relational Operators (<, >, ==, etc)
- Logical Operators (&&, ||, !)
- `if` and `else` statements

What are we covering today?

Going slightly deeper in programming . . .

- Recap of some of the concepts introduced last week
- Continuing learning about `if` statements
- Some in-depth thinking about problems
- Processes for solving problems
- Continuing the Dice Checker, but with more nuance

Recap - Variables

- Data storage in memory
- Made up of bits (and bytes are sets of 8 bits)
- Chosen for a specific purpose
 - **int** - 32 bit integer numbers
 - **double** - 64 bit floating point numbers
- We choose the name - try to make it meaningful!
- We can change the value as we go

Recap - Reading and Writing to our Terminal

`printf()`

- Outputs text to the terminal
- We can format our variables to output them
 - `%d` - decimal integer (works with ints)
 - `%lf` - long floating point number (works with doubles)

`scanf()`

- Reads text from the user
- Uses a similar format to `printf()`

Recap - Maths Operators

- `+, -, *, /`
- These four work pretty much exactly as normal maths does
- (brackets) allow us to force some operations to run before others

`%` - Modulus

- Gives us the remainder (as an integer) of a division between integers
- Does not actually perform the division

Recap - Relational and Logical Operators

Relational Operators

- `>`, `>=`, `<`, `<=`, `==`, `!=`
- Comparisons made between numbers
- Will result in 1 for true and 0 for false

Logical Operators

- `&&`, `||`, `!`
- Comparisons made between true and false (0 and 1) results
- Used to combine Relational Operator Questions together

Recap - if and else statements

- Branching control of a program

```
// One of the challenges from last week
if (total > TARGET_VALUE) {
    // this runs if the test above is true
    printf("Skill roll succeeded!\n");
} else if (total == TARGET_VALUE) {
    // otherwise if this test is true
    printf("Skill roll tied!\n");
} else {
    // if neither of the others are true
    printf("Skill roll failed!\n");
}
```

Problems and Solutions

What's our problem?

- This is always a good question!
- Spending some time figuring out exactly what we aim to do (or what's stopping us from getting there) is important
- Keeping the problem in mind keeps you focused on a solution

**IF YOU ONLY
FOCUS ON THE PROBLEM**



**YOU MIGHT
MISS THE EASY SOLUTION**

A process for problem solving

We can develop a way to approach all problems

1. Figure out what's wrong (or what we need to solve)
2. Find out what our options are (what code could we write or change?)
3. Assess those options
 - a. How well do they solve the problem?
 - b. Can we make them work?
4. Pick an option to try
5. Did it work?
 - a. If it didn't or even if it did, we can get more information for our next attempt

Back to the Dice Checker

We created a program that:

- Asked the user to input their dice values
- Reported back whether the total was above or below a target value

Let's look at one problem that might occur

- What if the user enters incorrect values?
- Too high or too low?



Testing our Input

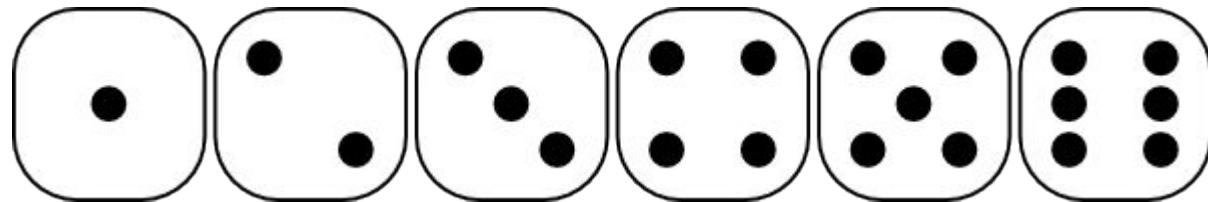
Let's assume we have this input code:

```
// Setup dice variables
int dieOne;
int dieTwo;

// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then scan their input
scanf("%d", &dieOne);
// repeat for the second die
printf("Please enter your second die roll: ");
scanf("%d", &dieTwo);
```

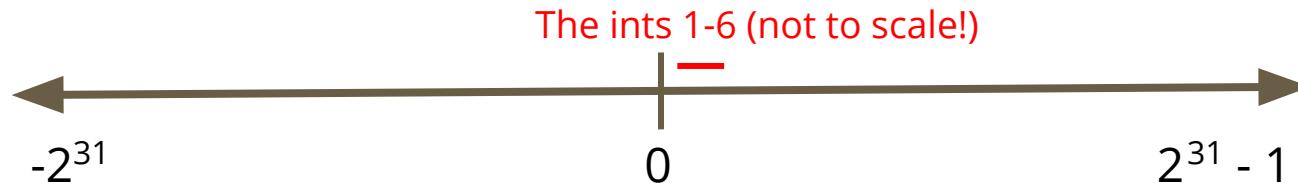
Testing Input Range

A six sided die has a specific range of inputs



We will only accept inputs in this range

But ints have a much wider range!



Testing Input Range in Code

```
// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then scan their input
scanf("%d", &dieOne);

// test for proper input range
if (1 <= dieOne && dieOne <= 6) {
    // if this succeeds, we are in the right range
} else {
    // number was outside the die's range
}
```

Using Constants in code

The 1 and 6 are the minimum and maximum values of the dice

- We could use something like this at the start of our program:

```
#define MIN_VALUE 1  
#define MAX_VALUE 6
```

Using `MIN_VALUE` and `MAX_VALUE`:

- Makes the program much easier to modify for different dice sizes
- Also makes things much more readable by using English words instead of numbers

Break Time

Dice

- The Egyptians were using flat sticks to randomise movement in Senet
- That dates games with randomisation back past 3000BC
- Six sided dice have been excavated in Iran from 2800-2500BC
- Nowadays we usually use dice ranging from 4 to 20 sides
- We're going to look at random number generation later in the course so you'll be able to simulate your own dice

What are our options?

If we know we have incorrect input, what do we do?

We have several options . . .

- PANIC!!!!!
- Reject the input, end the program
- Let the user know what the correct input is
- Correct the input
- Ask for new input

Reject the input

We can just end the program if the input is incorrect

```
// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then scan their input
scanf("%d", &dieOne);

// test for proper input range
if (MIN_VALUE <= dieOne && dieOne <= MAX_VALUE) {
    // if this succeeds, we are in the right range
} else {
    // number was outside the die's range
    return 1;
}
```

Assessing This Option

Is it a good idea to have the program just end?

- What's a good way for the program to reject incorrect input?
- If we're testing or using the program, what do we want to see?

Reporting Failure

Information from the program helps the user

```
// test for proper input range
if (MIN_VALUE <= dieOne && dieOne <= MAX_VALUE) {
    // if this succeeds, we are in the right range
} else {
    // number was outside the die's range
    printf("Input for first die, %d was out of
range. Program will exit now.\n", dieOne);
    return 1;
}
```

Can we do better?

Exploring other options

Let's give the user information that helps them correct the input issues

```
// test for proper input range
if (MIN_VALUE <= dieOne && dieOne <= MAX_VALUE) {
    // if this succeeds, we are in the right range
} else {
    // number was outside the die's range
    printf("Input for first die, %d was out of the
range 1-6. Program will exit now.\n", dieOne);
    return 1;
}
```

Correcting the input without exiting

If we want the program to finish executing even with bad input

Imperfect, but sometimes we want the program to finish

What are our options?

- Clamping - anything outside the range gets “pushed” back into the range
- Modulus - a possibly elegant solution

Clamping Values

Correcting the values - a brute force approach

```
// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then scan their input
scanf("%d", &dieOne);

// clamp any values outside the range
if (dieOne < MIN_VALUE) {
    dieOne = MIN_VALUE;
} else if (dieOne > MAX_VALUE) {
    dieOne = MAX_VALUE;
}
```

Any Issues with Clamping?

- Definitely end up with input that works
- But is it correct?
- What are the issues with correcting data without the user knowing?

Modulus

A reminder of what it is

- `%` - A maths operator that gives us the remainder of a division

How can we use it?

- Any number “mod” 6 will give us a value from 0 to 5
- If we change any 0 to a 6, we get the range 1 to 6
- This means the user could type in completely random numbers and be given a 1-6 dice roll result

Using Modulus in code

```
// we start by asking the user for their dice rolls
printf("Please enter your first die roll: ");
// then scan their input
scanf("%d", &dieOne);

// mod forces the result to stay within 0-5
dieOne = dieOne % MAX_VALUE;
// make any 0 into MAX_VALUE
if (dieOne == 0) {
    dieOne = MAX_VALUE;
}
```

Pros and Cons of using Modulus for dice

Pros

- We guarantee a number between 1 and 6 (or whatever the max value is)
- We don't shut down unexpectedly due to incorrect input
- We give a very dice-like randomish result (as opposed to clamping)

Cons

- We might accept incorrect input silently
- We might make a change that affects the user's expectations

A Range of Solutions

Which one to use?

- No single answer
- The original purpose of the program can help us decide
- What's our priority?
- Exact correctness?
- Failure on any kind of incorrect data?
- Usability and randomisation over correctness?

The Upgraded Dice Checker

- The programmer can set the size of the dice in `#define` constants
- The user can enter any number and it will produce a valid roll
- The program will still report back success or failure

Starting from our previous Dice Checker program, we can make some modifications to give it some new capabilities

Setting up

We'll start with our description of the program

```
// The Dice Checker v2
// Marc Chee, February 2019

// Allows the user to set dice size
// Tests the rolls of two dice against a target number
// Able to deal with user reported rolls outside the range
// Will report back Success, Tie or Failure

#include <stdio.h>

#define MIN_VALUE 1
#define MAX_VALUE 6
```

Variables and Constants

Set up the Target constant and some variables

```
// The secret target number
#define SECRET_TARGET 7

int main (void) {
    int dieOne;
    int dieTwo;
```

Taking user input

Two rolls will be taken as input (only one is shown here)

```
// Process the first die roll
printf("Please enter your first die roll: ");
scanf("%d", &dieOne);

// Check and fix the die roll
if (dieOne < MIN_VALUE || dieOne > MAX_VALUE) { // dieOne is invalid
    printf("%d is not a valid roll for a D%d.\n", dieOne, MAX_VALUE);
    dieOne = (dieOne % MAX_VALUE);
    if (dieOne == 0) {
        dieOne = MAX_VALUE;
    }
}
```

Calculate and report the total

This is identical to last week's code

```
// calculate the total and report it
int total = dieOne + dieTwo;
printf("Your total roll is: %d\n", total);

// Now test against the secret number
if (total > SECRET_TARGET) {
    // success
    printf("Skill roll succeeded!\n");
} else {
    // failure
    printf("Skill roll failed!\n");
}
```

We have a new Dice Check Program

We've added:

- Some measures against user mistakes
- Some modifiability

We made some decisions:

- We will report any user errors
- But we're also delivering a die roll regardless

What we learnt today

- A recap of the technical programming we've done so far
- Some discussion of how to approach problem solving
- A walkthrough of a technical problem and its many possible solutions
- Some thinking about why we choose a particular solution

- Some use of logical operators
- Some use of modulus
- Some slightly more complex code (if statements inside if statements)

COMP1511 - Programming Fundamentals

— Week 2 - Lecture 4 —

What did we learn last lecture?

- **EVERYTHING!** - A recap of the C we've seen so far
- Problem Solving
- Continuing work with if and else statements
- Showing use of **#define** constants
- Using some Relational and Logical Operators
- Showing some use of Modulus

What are we covering today?

Looping

- Repetitive tasks shouldn't require repetitive coding
- "while" loops
- How to start and stop loops
- Some interesting things we can do with them

Executing the same code more than once

Sometimes we need to repeat our work

- C normally executes in order, line by line
- `if` statements allow us to “turn on or off” parts of our code
- But up until now, we don’t have a way to repeat code
- Copy-pasting the same code again and again is not a feasible solution

While Loops

```
// expression is checked at the start of every loop
while (expression) {
    // this will run again and again
    // until the expression is evaluated as false
}
// When the program reaches this }, it will jump
// back to the start of the while loop
```

While Loops

“while” is a C keyword that lets us loop code

- Format is very similar to an **if** statement
- The “question” in the (brackets) functions very similarly
- If it’s true, the body of the **while** loop will run
- If it’s false, the body won’t run and the program will continue
- Once a while reaches the end of its { } it will start again

While Loop Control

We can use a variable to control how many times a while loop runs

- We call this variable a "loop counter"
- It's an `int` that's declared outside the loop
- Its "termination condition" can be checked in the while expression
- It will be updated inside the loop

We can also use a variable to decide to exit a loop at any time

- We call this variable a "sentinel"
- It's like an on/off switch for the loop

While Loop with a Loop Counter

```
// an integer outside the loop
int counter = 0;

while (counter < 10) {
    // this code has run counter number of times

    counter = counter + 1;
}

// When counter hits 10 and the loop's test fails
// the program will exit the loop
```

While Loops and Termination

It's actually very easy to make a program that goes forever

Consider the following while loop:

```
while (1 < 2) {  
    // Never going to give you up  
    // Never going to let you down . . .  
}
```

Using a Sentinel Variable with While Loops

A sentinel is a variable we use to decide when to exit a while loop

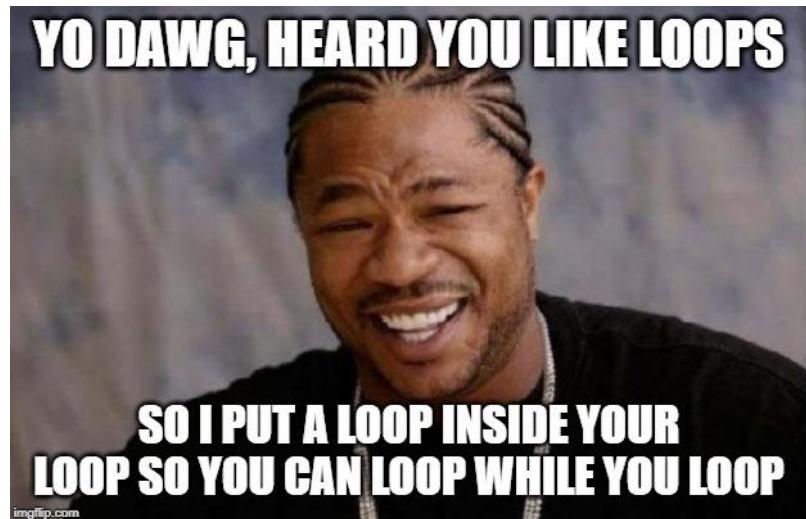
```
// an integer outside the loop
int endLoop = 0;

// The loop will exit if it reads an odd number
while (endLoop == 0) {
    int inputNumber;
    printf("Please type in a number: ");
    scanf("%d", &inputNumber);
    if (inputNumber % 2 == 0) {
        printf("Number is even.\n");
    } else {
        printf("Number is odd.\n");
        endLoop = 1;
    }
}
```

While Loops inside While Loops

If we put a loop inside a loop . . .

- Each time a loop runs
- It runs the other loop
- The inside loop ends up running a LOT of times



A loop within a loop

```
int y = 0;
// loop through and print multiple rows
while (y < 10) { // we have printed y rows
    // print a single row
    int x = 0;
    while (x < 10) { // we have printed x stars in this row
        printf("*");
        x = x + 1;
    }
    // the row is finished, start the next line
    printf("\n");
    y = y + 1;
}
```

Drawing a grid of stars

The previous slide's code:

- Sets up a loop using `y`
- In each loop of `y`, sets up a loop using `x`
- The `x` loop writes multiple `*`s to the terminal
- Then the `y` loop finishes, writing `\n` so the line ends

What do the curly braces do?

What goes on inside the curly braces stays inside the curly braces.

- Look closely at the declaration of `int x` in the grid drawing code
- The use of `x` is contained inside a set of curly braces `{ }`
- This means that `x` will only exist inside those braces
- The variable `x` will actually disappear each time the `y` loop finishes!

Curly braces create the "scope" of a program

- Anything created inside them only lasts as long as they do!

Break Time!

While loops, if statements etc, it's all code!

- An `if` statement is some code
- A `while` loop is also some code

This means that you can:

- Put `ifs` inside `while` loops
- Put `while` loops inside `ifs` or `elses`
- Put `while` loops inside `while` loops inside `if` statements etc etc etc!
- Just watch out for confusing ourselves!

Dice Statistics, a Looping Program

The following program:

I need a program that will show me all the different ways to roll two dice

If I pick a number, it will tell me all the ways those two dice can reach that total

It will also tell me what my odds are of rolling that number

Break it down

What components will we need?

- We need all possible values of the two dice
- We need all possible totals of adding them together
- Seems like we're going to be looping through all the values of one die and adding them to all the values of the other die

Let's start with this simple program then go for our bigger goals later

Code for all dice rolls

```
int main (void) {
    int diceOneSize;
    int diceTwoSize;

    // User decides the two dice sizes
    printf("Please enter the size of the first die: ");
    scanf("%d", &diceOneSize);
    printf("Please enter the size of the second die: ");
    scanf("%d", &diceTwoSize);

    // Then loop through both dice
```

Code for all dice rolls continued

```
// loop through and see all the values that the two dice can roll
int die1 = 1;
while (die1 <= diceOneSize) { // seen die1 - 1 values
    int die2 = 1;
    while (die2 <= diceTwoSize) { // seen die2 - 1 values
        printf(
            "%d, %d. Total: %d\n",
            die1, die2, die1 + die2
        );
        die2++;
    }
    die1++;
}
```

Quick Pause for new C syntax

Incrementing just got a little easier

```
int die1 = 0;
int die2 = 0;

// The following two lines have the
// same effect on their variables
die1 = die1 + 1;
die2++;

// both variables now == 1
```

We can now see all possible dice rolls

- We have all possibilities listed
- We know all the totals
- We could also count how many times the dice were rolled

Let's try now isolating a single target number

- Check the targets of the rolls and output only if they match our target value

Now with a target number

```
int main (void) {
    int diceOneSize;
    int diceTwoSize;
    int targetValue;

    // User decides the two dice sizes and target
    printf("Please enter the size of the first die: ");
    scanf("%d", &diceOneSize);
    printf("Please enter the size of the second die: ");
    scanf("%d", &diceTwoSize);
    printf("Please enter the target value: ");
    scanf("%d", &targetValue);
```

Output the rolls that match the target

```
// loop through and output rolls with totals that match the target
int die1 = 1;
while (die1 <= diceOneSize) { // seen die1 - 1 values
    int die2 = 1;
    while (die2 <= diceTwoSize) { // seen die2 - 1 values
        int total = die1 + die2;
        if (total == targetValue) {
            printf(
                "%d, %d. Total: %d\n",
                die1, die2, total
            );
        }
        die2++;
    } // die2 = diceTwoSize + 1
    die1++;
} // die1 = diceOneSize + 1
```

Getting there!

We now have a program that can identify the correct rolls

- If we want the odds, we just compare the target rolls vs the rest
- If we count the number of rolls that added to the target value
- And we count the total number of rolls
- We can do some basic maths and divide the successful rolls by the total
- That should give us our chances of getting that number

How do we keep track of success vs failure?

We can count using ints

- We can keep a counting variable outside the loop
- This will increment only on successes
- We can either calculate or count our total
- Dividing them will give us the fraction chance of rolling our target number

Measuring Successes

Adding some variables to count results

- **integers** (`diceOneSize`, `diceTwoSize`) for the two dice sizes
- **integer** (`targetValue`) for the target value
- **integer** (`numSuccesses`) for the number of successes
- **integer** (`numRolls`) for the number of rolls

Making sure our loop records results

```
// loop through and output rolls with totals that match the target
int die1 = 1;
while (die1 <= diceOneSize) { // seen die1 - 1 values
    int die2 = 1;
    while (die2 <= diceTwoSize) { // seen die2 - 1 values
        numRolls++;
        int total = die1 + die2;
        if (total == targetValue) { // target match
            numSuccesses++;
            printf("%d, %d. Total: %d\n",
                   die1, die2, total);
        }
        die2++;
    }
    die1++;
}
```

Output our Percentage

```
// Calculate percentage chance of success
int percentage = numSuccesses/numRolls * 100;
printf("Percentage chance of getting your target number is: %d\n",
      percentage);
```

There's an issue with the previous code ...

Did you notice the issue?

- Our code outputs 0 percent a lot more than it should
- This is even after we know it's counting the successes correctly

Integers do weird things with division in C

- After a division, the integers will throw away any fractions
- Since our "`numSuccesses/numRolls`" will always be between zero and 1
- Our result can only be the integers 0 or 1
- And anything less than 1 will end up having its fraction dropped!

Doubles to the rescue

Luckily we have a variable type that will store a fraction

- Result of a division will be a double if one of the variables in it is a double
- We could change one of the variables in our division to a double
- This could be done in the declaration of the variable
- But we can also just do it at the point it is used!

```
// int * double = double
// The second number will appear as a double to the division!
int percentage = numSuccesses / (numRolls * 1.0) * 100;
```

The Challenge ... did we need to do all this work?

This program didn't actually need everything we did today

- There's a much simpler way to list the rolls that sum to a target number
- There's also a much simpler way to find the total number of rolls
- If we just use a bit more maths and less raw coding . . .

See what you can come up with!

What did we learn today?

While Loops

- Repeating execution of code
- We've made some loops
- We've shown how to loop inside other loops
- We've shown different ways to end loops

COMP1511 - Programming Fundamentals

— Week 3 - Lecture 5 —

What did we learn last week?

- **if statements** - branching code
- **Problem solving** - thinking carefully while programming
- **while loops** - repeating code

What are we covering today?

Code Style

- What is Code Style? Why does it matter?

Code Reviews

- What is a Code Review?
- What can we learn from Code Reviews?

Functions

- An introduction to what a function is
- How we use functions in C

While Loops Recap

What do we know about While Loops?

- They have a specific syntax
- They test an expression and run repeatedly while it's true
- We can make them stop after a specific number of iterations
- We can make them stop after a certain condition is met
- We can run any other code inside a while loop

Will it ever stop? I don't know...

It's easy to make it start, but make sure you can stop it!

- Create every loop with the idea of how it stops
- Let's review how we stop loops

While Loop with a Loop Counter

How to make a loop run an exact number of times

```
// an integer outside the loop
int i = 0;

while (i < 10) { // loop has run i times
    // Code in here will run 10 times

    i++;
}
// When i hits 10 and the loop's test fails
// the program will exit the loop
```

Using a Sentinel Variable with While Loops

A sentinel is a variable we use to intentionally exit a while loop

```
// an integer outside the loop
int endLoop = 0;

// The loop will exit if it reads an odd number
while (endLoop == 0) {
    int inputNumber;
    scanf("%d", &inputNumber);
    if (inputNumber % 2 == 0) {
        printf("Number is even.\n");
    } else {
        printf("Number is odd.\n");
        endLoop = 1;
    }
}
```

Code Style

Why do we write code for humans?

- Easier to read
- Easier to understand
- Less mistakes
- Faster overall development time



Good Coding Practices

What is good style?

- Indentation and Bracketing
- Names of variables and functions
- Repetition (or not) of code
- Clear comments
- Consistency

The easier it is to read and understand, the less mistakes we'll make

Poor Code Style

Can we work with code that's hard to read?

- I'd like to show you something I prepared earlier . . .
- `CodeStyleBad.c` is functionally our Dice Checking program

Let's have a look at the code . . .

What went wrong?

We want more than: “Oh wow, that’s a mess”

What are the specific improvements that can make this better?

In the face of disaster, keep a clear head and focus on what can be fixed

Specific Issues

- Header comment doesn't show the program's intentions
- No blank lines separating different components
- Multiple expressions on the same line
- Inconsistent indenting
- Inconsistent spacing
- Variable names don't make any sense
- Comments don't mean anything
- Inconsistent bracketing of if statements
- Bracketing is not indented
- Inconsistent structure of identical code blocks
- The easter egg - there's actually incorrect code also!

Keeping your house (code) clean

Regular care is always less work than a big cleanout

- Write comments before code
- Name your variables before you use them
- { everything inside gets indented 4 spaces
- } line up your closing brackets vertically with the line that opened them
- One expression per line
- Maintain consistency in spacing

Comments before code

Comments before code. It's like planning ahead

- Making plans with comments
- You can fill them out with correct code later
- Some of these comments can stay even after you've written the code

```
// Checking against the target value
if () {
    // success
} else if () {
    // tie
} else {
    // failure (all other possibilities)
}
```

Naming Variables

Variable names are for humans

- Can you describe what a variable is in a word or two?
- If your lab partner was to read this name, would it make sense?
- Does it distinguish it well against the other variables?

Indentation

A common convention is to use 4 spaces for indentation

```
int main (void) {
    // everything in here is indented 4 spaces
    int total = 5;
    if (total > 10) {
        // everything in here is indented 4 more
        total = 10;
    }
    // this closing curly bracket lines up
    // vertically with the if statement
    // that opened it
}
// this curly bracket lines up vertically
// with the main function that opened it
```

One expression per line

Any single expression that runs should have its own line

```
int main (void) {  
    // NOT LIKE THIS!  
    int numOne; int numTwo;  
    numOne = 25; numTwo = numOne + 10;  
    if (numOne < numTwo) { numOne = numTwo; }  
}
```

```
int main (void) {  
    // Like this :)  
    int numOne;  
    int numTwo;  
    numOne = 25;  
    numTwo = numOne + 10;  
    if (numOne < numTwo) {  
        numOne = numTwo;  
    }  
}
```

Spacing

Operators need space to be easily read

```
int main (void) {  
    // NOT LIKE THIS!  
    int a;  
    int b;  
    int total=0;  
    if(a<b&&b>=15) {  
        total=a+b;  
    }  
}
```

```
int main (void) {  
    // Like this :)  
    int a;  
    int b;  
    int total = 0;  
    if (a < b && b >= 15) {  
        total = a + b;  
    }  
}
```

More Information about Coding Style

- The course webpage has a Style Guide
- Wherever you end up coding, there will be different styles
- Our style is only one of them, but a good place to start!

Your assignments have coding style marks (more on this when they release)

The Exam has some style marks also

Break Time

Code Style isn't just to make it look nice

- Reduces errors later in development
- Makes it easier to test and modify
- Overall, speeds up development
- Makes your co-workers hate you less



Weekly Tests

Self Invigilated Weekly Tests start this week

- A mini exam you run yourself for one hour
- (You can continue working after the one hour if you want to go back over things with less pressure)
- The detailed rules are in the test itself
- Releases on **Thursday** and you will have one week to complete it

- Use it as a way to test your progress so far
- Great practice for coding with time pressure and limited resources (exams or job interviews)

Code Review

What is a code review?

- Having other coders look over your code
- Having an active discussion about the code
- Automated testing can test functionality, but not necessarily usability
- Humans can help you improve as a human!
- Similar to proof-reading a document
- Super valuable to discuss different approaches to the same problem

Why do we review code?

As the code writer

- Get feedback on how easy it is to understand our code
- Hear about other people's ideas on solving the same problem

As the code reviewer

- Get to see how someone else writes code
- Learn more about different ways to solve problems

Different ways to review code

Pair Programming

- Lab partners actively discussing solutions
- Live reviewing and discussion while in development

More formal review

- Finish a section of code, then ask people to review it
- Sometimes in person, sometimes using software tools

How to do Pair Programming well

Also, how to learn the most from 1511 labs

- One person on the keyboard (sharing screen in a breakout room)
 - Thinking about how to structure the C and syntax
- One person over the shoulder (watching the shared screen)
 - Thinking about how to solve the problem
- Active discussion between the two of you as you go (mics open)
- This means the code is constantly under review

Programming with others is one of the best ways to learn!

Conducting a Formal Code Review

Reviewing a finished piece of code

- Reviewers will read the code and help with it
- Remember, we're judging the code, not the coder!
- We're all learning . . . this is not about picking at mistakes

Points to Discuss

- Where is it easy or hard to understand the code?
- What are the different possible ways the code can solve the problem?
- Any little issues we can help solve?

What not to do in a Code Review

These things will not help us learn better code:

- “You did this wrong”
- “Your code is bad”
- “Here are all the mistakes in this code”

We’re doing this to help ourselves and others learn more!

No judgement, only help!

What to do in a Code Review

How does one help someone else learn?

- Understand that it's very hard to put your work up for review
- We're not here to judge the code's standard
- We're here to help everyone learn more
- There is no single right way to solve a problem
- If your way and someone else's way are different, you can both be right
- Try to learn from other styles of coding that you review
- Letting people know what you don't understand is one of the most valuable things you can do in a code review

Next week's Tutorial will have a demo Code Review

Your tutor will do the first review so you can see what it's like

- After this, every code review will be lead by students
- You can also get together with other students to review your Lab work
- (just don't do it with Assignments or Weekly Tests!)

Functions

Let's introduce functions

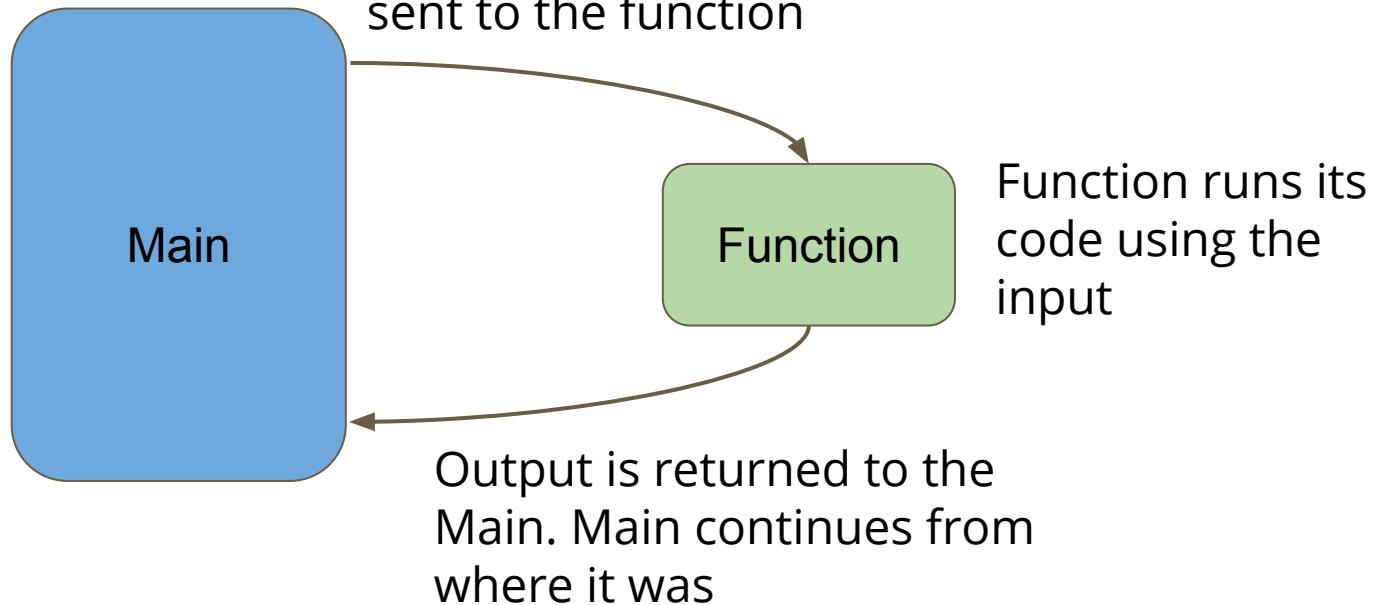
- We've already been using some functions!
- `main` is a function
- `printf` and `scanf` are also functions

What is a function?

- A separate piece of code identified by a name
- It has inputs and an output
- If we "call" a function it will run the code in the function

Functions

How do they work?



Function Syntax

We write a function with (in order left to right):

- An output (known as the function's type)
- A name
- Zero or more input(s) (also known as function parameters)
- A body of code in curly brackets

```
// a function that adds two numbers together
int add (int a, int b) {
    return a + b;
}
```

Return

An important keyword in a function

- `return` will deliver the output of a function
- `return` will also stop the function running and return to where it was called from

How is a function used?

If a function already exists (like printf)

- We can use a function by calling it by name
- And providing it with input(s) of the correct type(s)

```
// using the add function
int main (void) {
    int firstNumber = 4;
    int secondNumber = 6;
    int total;

    total = add(firstNumber, secondNumber);
    return 0;
}
```

Compilers and Functions

How does our main know what our function is?

- A compiler will process our code, line by line, from top to bottom
- If it has seen something before, it will know its name

```
// An example using variables
int main (void) {
    // declaring a variable means it's usable later
    int number = 1;

    // this next section won't work because the compiler
    // doesn't know about otherNumber before it's used
    int total = number + otherNumber;
    int otherNumber = 5;
}
```

Functions and Declaration

We need to declare a function before it can be used

```
// a function can be declared without being fully
// written (defined) until later
int add (int a, int b);

int main (void) {
    int firstNumber = 4;
    int secondNumber = 6;
    int total = add(firstNumber, secondNumber);
    return 0;
}

// the function is defined here
int add (int a, int b) {
    return a + b;
}
```

Void Functions

We can also run functions that return no output

- We can use a void function if we don't need anything back from it
- The return keyword won't have a variable or value if it is used at all

```
// a function of type "void"
// It will not give anything back to whatever function
// called it, but it might still be of use to us
void printAdd (int a, int b) {
    int total = a + b;
    printf("The total is %d", total);
}
```

What did we learn today?

Code Style

- Making your code understandable and reusable

Code Reviews

- Reviewing your's and other people's code can help you learn and share your skills

Functions

- Separating code to make it easier to read and reuse

COMP1511 - Programming Fundamentals

— Week 3 - Lecture 6 —

What did we learn last lecture?

Code Style and Code Reviews

- Coding for and with humans

Introduction of Functions

- Separating code for reuse

What are we covering today?

Computers as theoretical tools

- Fundamentals of what a computer is
- How we use memory in C

Arrays

- Using multiple variables at once

What is a computer?

At the most fundamental level . . .

- A processor that executes instructions
- Some memory that holds information

The Turing Machine

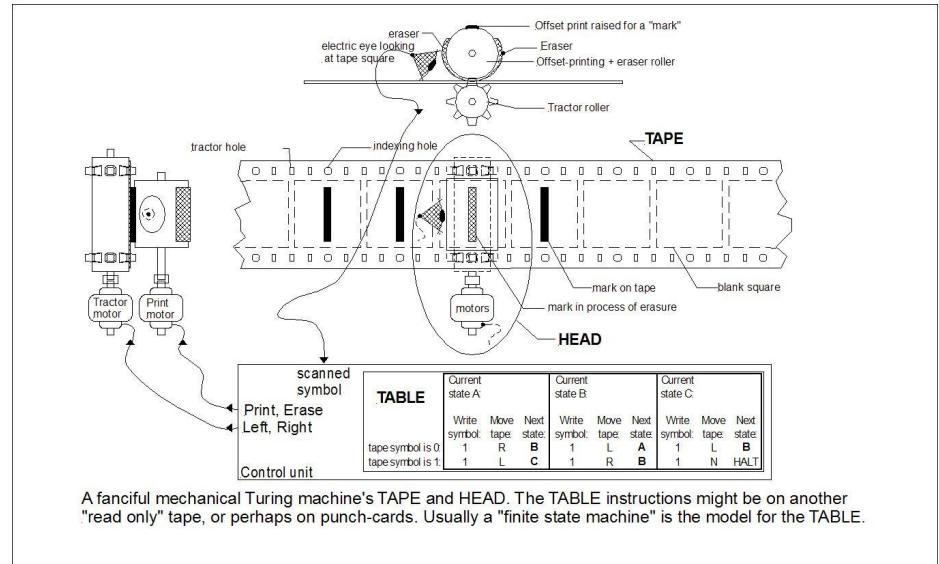
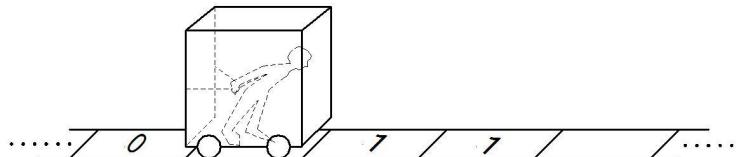
Originally a theoretical idea of computation

- There is a tape that can be infinitely long
- We have a “head” that can read or write to this tape
- We can move the head along to any part of the tape
- There’s a “state” in which the machine remembers its current status
- There’s a set of instructions that say what to do in each state

Turing Machines

Some images of Turing Machines

- A tape and a read/write head
- Some idea of control of the head



*Images from Wikipedia
(https://en.wikipedia.org/wiki/Turing_machine_gallery)*

The Processor

We also call them Central Processing Units (CPUs)

- Maintains a “state”
- Works based on a current set of instructions
- Can read and write from/to memory

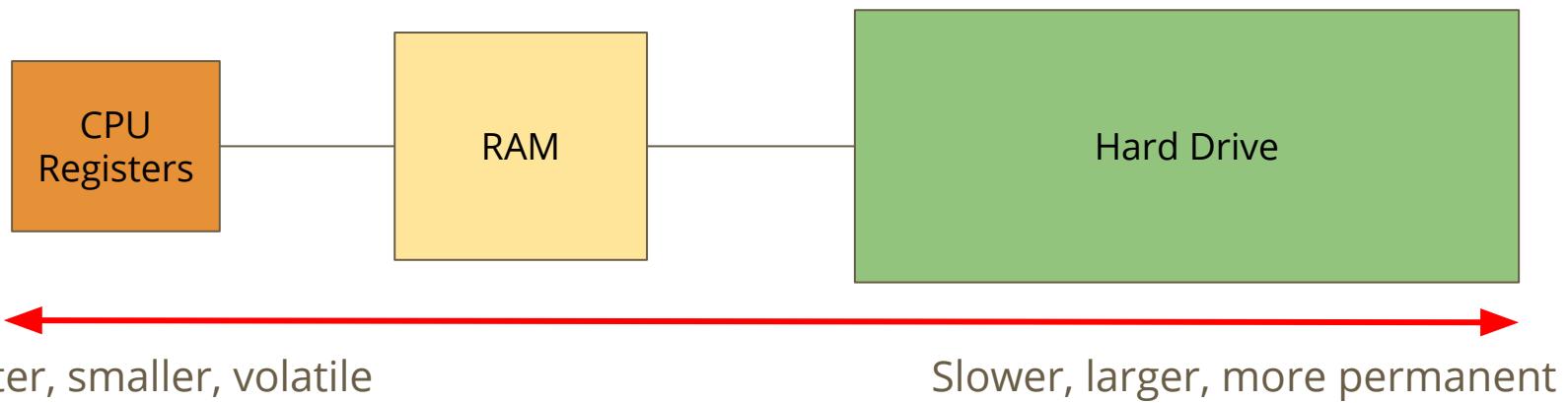
In our C Programming

- State - where are we up to in the code right now
- Instructions - compiled from our lines of code
- Reading/Writing - Variables

Memory

All forms of Data Storage on a computer

- From registers (tiny bits of memory on the CPU) through Random Access Memory (RAM) and to the Hard Disk Drive. All of these are used to store information



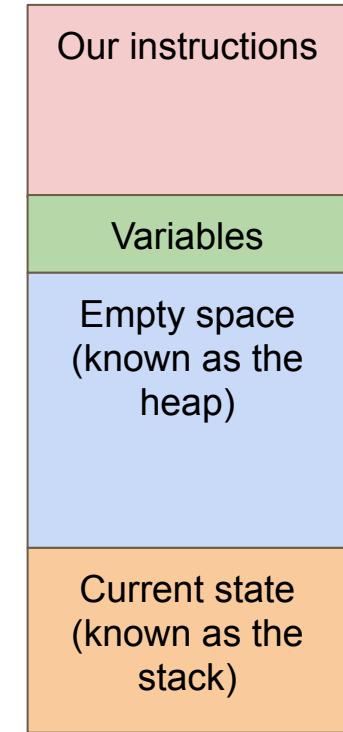
How does C use memory

- On the **Hard Drive**
- Our C source code files are stored on our Hard Drive
- Dcc compiles our source into another file, the executable program
- In **Random Access Memory**
- When we run our program, all the instructions are copied into RAM
- Our CPU will work through memory executing our instructions in order
- Our variables are stored in RAM as well
- Reading and writing to variables will change the numbers in RAM

A snapshot of a program in memory

What happens in memory when we run a program?

- Our Operating System gives us a chunk of memory
- Our program copies its instructions there
- Some space is reserved for declared variables
- The **Stack** is used to track the current state
- The stack grows and shrinks as the program runs
- The **Heap** is empty and ready for use
- We can use the heap to store data while the program is running



There's more ... later

Computers and programs are highly complex

- This was just an overview
- As you go through your learning, you will unlock more information
- For now, we have enough understanding to continue using C

Arrays

When we need a collection of variables together

- Sometimes we need a bunch of variables of the same type
- We also might need to process them all
- Our current use of ints and doubles might not be able to handle this

Let's take a look at our current capability (and why we need arrays) . . .

An Example

Let's record everyone's marks at the end of the term

- We could do this as a large collection of integers . . .

```
int main (void) {
    int marksStudent1;
    int marksStudent2;
    int marksStudent3;
    int marksStudent4;
    // etc
```

If we want to test all these ints

We'd need a whole bunch of nearly identical if statements

In this situation

- There's no way to loop through the integers
- Having to rewrite the same code is annoying and hard to read or edit
- So let's find a better way . . .

```
int main (void) {
    int marksStudent1;
    int marksStudent2;
    int marksStudent3;
    int marksStudent4;
    // etc

    if (marksStudent1 >= 50) {
        // pass
    }
    if (marksStudent2 >= 50) {
        // pass
    }
    // etc
```

An Array of Integers

If our integers are listed as a collection

- We'll be able to access them as a group
- We'll be able to loop through and access each individual element

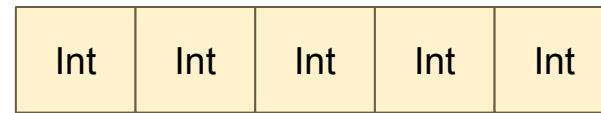
Arrays

What is an array?

- A variable is a small amount of memory
- An array is a larger amount of memory that contains multiple variables
- All of the elements (individual variables) in an array are the same type
- Individual elements don't get names, they are accessed by an integer index



A single integer
worth of memory



An array that holds 5 integers

Declaring an Array

Similar, but more complex than declaring a variable

```
int main (void) {  
    // declare an array  
    int arrayOfMarks[10] = {0};
```

- `int` - the type of the variables stored in the array
- `[10]` - the number of elements in the array
- `= {0}` - Initialises the array as all zeroes
- `= {0,1,2,3,4,5,6,7,8,9}` - Initialises the array with these values

Array Elements

- An element is a single variable inside the array
- They are accessed by their index, an int that is like their address
- Indexes start from 0
- Trying to access an index outside of the array will cause errors

	0	1	2	3	4	5	6	7	8	9
arrayOfMarks	55	70	44	91	82	64	62	68	32	72

In this example, element 2 of arrayOfMarks is 44 and element 6 is 62

Accessing elements in C

C code for reading and writing to individual elements

```
int main (void) {
    // declare an array, all zeroes
    int arrayOfMarks[10] = {0};

    // make first element 85
    arrayOfMarks[0] = 85;
    // access using a variable
    int accessIndex = 3;
    arrayOfMarks[accessIndex] = 50;
    // copy one element over another
    arrayOfMarks[2] = arrayOfMarks[6];
    // cause an error by trying to access out of bounds
    arrayOfMarks[10] = 99;
```

Reading and Writing

Printf and scanf with arrays

- We can't printf a whole array
- We also can't scanf a line of user input text into an array
- We can do it for individual elements though!

The trick then becomes looping to access all individual elements one by one

User input/output with Arrays

Using printf and scanf with Arrays

```
int main (void) {
    // declare an array, all zeroes
    int arrayOfMarks[10] = {0};

    // read from user input into 3rd element
    scanf("%d", &arrayOfMarks[2]);
    // output value of 5th element
    printf("The 5th Element is: %d", arrayOfMarks[4]);

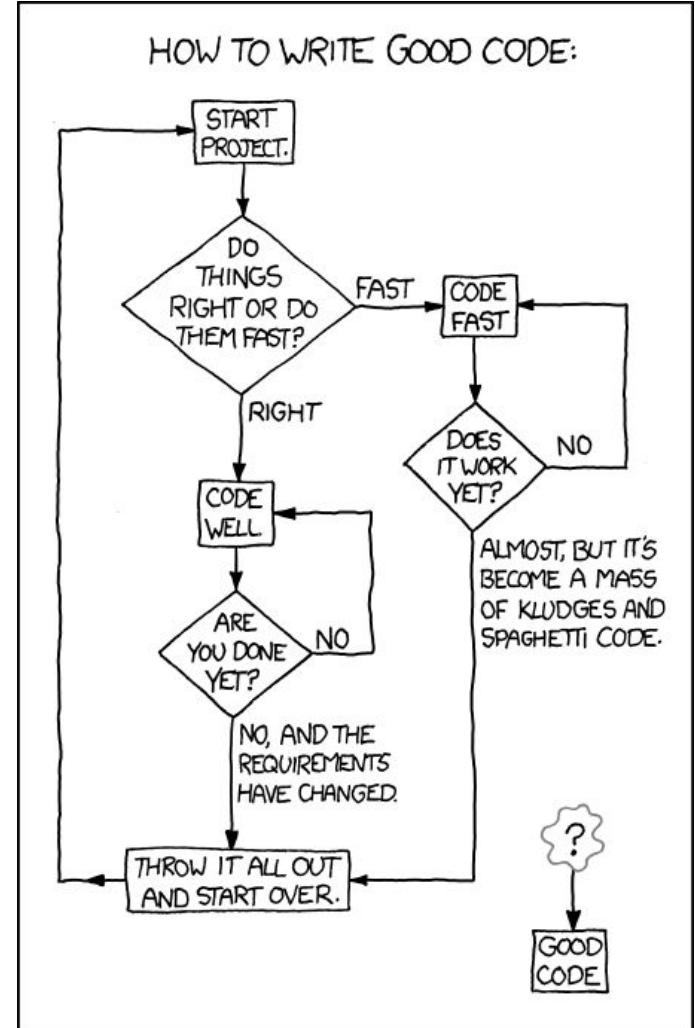
    // the following code DOES NOT WORK
    scanf("%d %d %d %d %d %d %d %d %d", &arrayOfMarks);
```

Break Time

Writing "good" code

- It's never going to be easy!
- It's an ongoing struggle

<https://xkcd.com/844/>



A Basic Program using Arrays

Let's make a program to track player scores in a game

- We have four players that are playing a game together
- We want to be able to set and display their scores
- We also want to be able to see who's winning and losing the game
- The game needs to know how many points have been scored in total, so we'll also want some way of calculating that total

Break down the program

What are the individual elements we need to make?

- First we create an array
- Then we use indexes to access the individual players and enter scores
- We're going to need while loops to step through the array
- Most of the extra functionality we want will be done by looping through the array

Create the Array and populate it

Setting the elements using indexes (manually for now)

```
#include <stdio.h>

#define NUM_PLAYERS 4

int main(void) {
    int scores[NUM_PLAYERS] = {0};
    int counter;

    // assigning values directly to indexes
    scores[0] = 55;
    scores[1] = 84;
    scores[2] = 32;
    scores[3] = 61;
```

Let's loop through and see those values

Accessing all array elements by looping

This is a pretty good candidate for code to put in a function!

```
// continued from last slide
// loop through and display all scores
int counter = 0;
while (counter < NUM_PLAYERS) {
    printf(
        "Player %d has scored %d points.\n",
        counter,
        scores[counter]
    );
    counter++;
}
```

Now that we have our array

It will look a bit like this:

	0	1	2	3
scores	55	84	32	61

Next, we can loop through to find:

- The lowest
- The highest
- And the total

Finding particular values in an array

If we see all the values, we can easily find the highest

- We'll loop through all the values in the array
- We'll save the highest value we've seen so far
- Then replace it if we find something higher
- By the time we reach the end, we will have the highest value

Finding the highest score

We could put this in a separate function also!

```
int highest = 0;
int indexHighest = -1;
counter = 0;
while (counter < NUM_PLAYERS) {
    if (scores[counter] > highest) {
        highest = scores[counter];
        indexHighest = counter;
    }
    counter++;
}
printf(
    "Player %d has the highest score of %d.\n",
    indexHighest, highest
);
```

Finding the Total

This is even easier than the highest!

We just add all the values to a variable we're keeping outside the loop

```
int total = 0;
counter = 0;
while (counter < NUM_PLAYERS) {
    total += scores[counter];
    counter++;
}
printf("Total points scored across the players is %d", total);
```

Wait, what was that new syntax?

`+=` is another shorthand operator

It's used for accumulating values in a variable

```
int a = 0;
int b = 0;

// These two lines of code will do the same thing
a += 5;
b = b + 5;

// both a and b are now equal to 5
```

What about input into an array

Remember, we can't access the whole array, only individual elements

But we can definitely loop through the array entering values!

```
// assigning scores using user input
counter = 0;
while (counter < NUM_PLAYERS) {
    printf("Please enter Player %d's score: ", counter);
    scanf("%d", &scores[counter]);
    counter++;
}
```

A Score Tracker

We've built our first program using an array (and maybe some functions)

- We've accessed elements by index to set their values
- We've looped through to access values to output
- We've looped through to find highest and lowest
- We learnt about accumulating values
- We've also looked at reading values into the array
- We've seen how we can separate code into a function

What did we learn today?

Theory of a Computer

- A processor - carries out operations
- Some memory - stores information

Arrays

- Collections of identical variables
- Individual elements are accessed by indexes

COMP1511 - Programming Fundamentals

— Week 4 - Lecture 7 —

What did we learn last week?

Code Style and Code Reviews

- Making our code understandable

Functions

- Separating code for reuse

Arrays

- Collections of identical variables

What are we covering today?

Functions and Libraries

- A recap of functions
- Using functions from other files

Arrays and 2D Arrays

- A recap of arrays
- Arrays inside arrays!

Recap of Functions

Code outside of our main that we can use (and reuse)

- Has a name that we use to call it
- Has an output type and input parameters
- Has a body of code that runs when it is called
- Uses return to exit and give back its output

Functions in Code

```
// a function declaration
int add (int a, int b);

int main (void) {
    int firstNumber = 4;
    int secondNumber = 6;
    // use the function here
    int total = add(firstNumber, secondNumber);
    return 0;
}

// the function is defined here
int add (int a, int b) {
    return a + b;
}
```

Why use functions?

Why do we separate code into functions?

Saves us from repeating code

- Instead of replicating code, we can write it once
- This also makes the code much easier to modify

Easier to organise code

- Complex functionality can be hidden inside a function
- The flow of the program can be read easily with clear function names

C Libraries

We've already used `stdio.h` several times

- C has other standard libraries that we can make use of
- The simple C reference in the Weekly Tests has some information
- `math.h` is a useful library of common maths functions
- `stdlib.h` has some useful functions
- Look through the references (including `man` manuals in linux)
- Don't worry if you don't understand the functions yet, some of them have no context in the programming we've done so far

Using Libraries

```
// include some libraries
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

int main (void) {
    int firstNumber = -4;
    int secondNumber = 6;

    // change a number to its absolute value
    firstNumber = abs(firstNumber);

    // calculate a square root
    int squareRoot = sqrt(firstnumber);
    printf("The final number is: %d", squareRoot);
    return 0;
}
```

Libraries

More complexity means more use of functions

- We've already seen `printf()` and `scanf()`
- We'll be using libraries with memory management
- And in other upcoming topics . . .

Using the output of Functions

Functions return their result to wherever they were called from

- Functions are a bit like other expressions
- They'll be run just like an expression is evaluated
- The result of the function can be used like any other value

Scanf as an example

`scanf()` is a function we've used already

- It returns the number of values that it has successfully read and stored
- This means we can use the return value of the function
- It will tell us whether or not we've received a valid input
- Check out `scanfDemo.c` in this week's lecture section for some examples

Recap of Arrays

A collection of variables

- Contains multiple variables all of the same type
- Declared using a variable type and a size
- Individual variables are accessed using an index

Indexes	0	1	2	3	4
An Array	63	88	43	55	67

Using Arrays in C

Some example code of an array

```
int main (void) {
    // declare an array of doubles, size 4, initially all 0
    double myArray[4] = {0.0};

    // assign a value
    myArray[1] = 0.95;
    // test a value
    if (myArray[2] < 1.0) {
        // print out a value
        printf("Third element is: %lf", myArray[2]);
    }
}
```

Accessing multiple values at once

Loops and Arrays go together perfectly

- Accessing all members is a reasonably simple while loop

```
int main (void) {
    // declare an array of doubles, size 4, initially all 0
    double myArray[4] = {0};

    // loop through the array and output the elements
    int i = 0;
    while (i < 4) {
        printf("%lf\n", myArray[i]);
        i++;
    }
}
```

Creating Arrays with certain sizes

Arrays start at an exact size and don't change

- When we create an array, we give it a size and a type
- Both of those are fixed and won't change

```
int main (void) {  
    // declare an array of doubles,  
    // size 4  
    double myArray[4] = {0};  
}
```

```
int main (void) {  
    // This declaration is not  
    // possible!  
    int arraySize = 4;  
    double myArray[arraySize] = {0};  
}
```

We can't declare an array with a variable size like this!

Using Constants for Array Sizes

If we do want to be able to change the size in code . . .

- We can use a constant to set the size
- Unlike a variable, this cannot change after it is compiled
- It does make our lives much easier if we need a change mid-project

```
#define ARRAY_SIZE 4

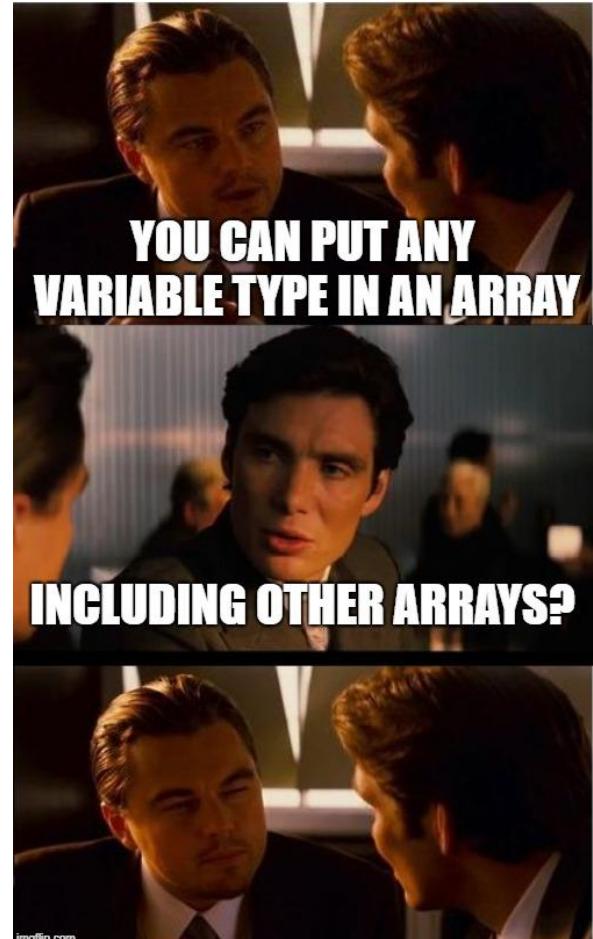
int main (void) {
    // This declaration allows us to change the
    // array size while coding
    double myArray[ARRAY_SIZE] = {0};
}
```

Arrays inside Arrays

An Array is a type of variable

An Array can contain any type of variable

- Arrays can be put inside other arrays!
- We call these multi-dimensional arrays
- Think of them as a grid, two or more dimensions



Two Dimensional Arrays

Arrays inside arrays

- Can be thought of like a grid
- The outer array contains arrays
- Each of those is a row of the grid
- Addressed using a pair of integers like coordinates
- All inner arrays are of the same type

Indexes	0	1	2	3	4
0	63	88	43	55	67
1	54	52	91	21	32
2	77	58	1	61	79

A 2D Array

Two Dimensional Arrays in Code

```
int main (void) {
    // declare a 2D Array
    int grid[4][4] = {0};

    // assign a value
    grid[1][3] = 3;
    // test a value
    if (grid[2][0] < 1) {
        // print out a value
        printf("The bottom left square is: %d", grid[3][0]);
    }
}
```

Break Time

Let's take five minutes break

- We're building up to much harder (and maybe frustrating) problems
- Remember that most hard problems are made up of smaller, easier problems
- Look for ways to break things down into parts that you can manage with the coding skills you've learnt!



Problems by Parallel Studio

Let's work with 2D Arrays

I would like to make a simple game called "The Tourist"

- The world is a square grid
- The tourist can move up, down, left or right
- Be able to print out the world, including the location of the tourist
- The tourist likes seeing new things . . .
- Track where they've been
- And lose the game if we revisit somewhere we've been

Starter Code

Working with code that's already got some functionality

- We're going to start with some code called tourist.c
- This file already has the ability to print a 2D array
- It's a bit similar to using starting code in the Assignment

Print Map

Here's a handy function that we'll be reusing

```
void printMap(int map[N_ROWS] [N_COLS], int posR, int posC) {  
    int row = 0;  
    while (row < N_ROWS) {  
        int col = 0;  
        while (col < N_COLS) {  
            if (posR == row && posC == col) {  
                printf("T ");  
            } else {  
                printf("%d ", map[row][col]);  
            }  
            col++;  
        }  
        row++;  
        printf("\n");  
    }  
}
```

Break the problem down into parts

What do we need to do?

- We need to set up our grid and the tourist's position
- The tourist needs to move one step at a time
- Each time the tourist visits a location, we set it to 1
- We also check each location to make sure it's new

The Square Grid World

Variables for the grid and the tourist's position

```
#include <stdio.h>

// The dimensions of the map
#define N_ROWS 10
#define N_COLS 10

int main (void) {
    int map[N_ROWS][N_COLS] = {0};
    int posR = 0, posC = 0;
```

Controlling the Tourist

Next Steps

- Let's add movement
- Then track where the Tourist has been, using the map
- After that, we'll check for places we've already been

Looping

- We can loop repeatedly for “turns” to allow the user to input directions

Movement - this code will loop

```
printf("Please enter a numpad direction or 0 to exit: ");
int input;
scanf("%d", &input);
if (input == 4) {
    posC--;
} else if (input == 8) {
    posR--;
} else if (input == 6) {
    posC++;
} else if (input == 2) {
    posR++;
} else if (input == 0) {
    exit = 1;
} else {
    printf("Input is not a numpad direction, please use 2,4,6 or 8\n");
}
```

Tracking the Tourist using the Map

Set each location we visit to 1

```
// loop and let the user control the Tourist's movement
int exit = 0;
while (!exit) {
    // mark the location as having been visited by incrementing
    map[posR][posC] = 1;

    // show the current status
    printMap(grid, posR, posC);

    printf("Please enter a numpad direction or 0 to exit: ");

    // Movement code from previous slide goes here . . .
```

Have we been here before?

We want the game to end if the tourist revisits a location

- If the location we visit is already 1
- Then we're going to exit the game
- We can add this check after our movement

```
// Check if we've been here before
if (map[posR][posC] == 1) {
    printf("We've already been here! How boring!\n");
    exit = 1;
}
```

1 isn't as helpful as "EXPLORED"

Let's swap out the number for a more readable #define

```
#include <stdio.h>

// The dimensions of the map
#define N_ROWS 10
#define N_COLS 10

// Has the square been explored before?
#define UNEXPLORED 0
#define EXPLORED 1
```

The Tourist Game

This is now roughly complete

- We can move the tourist
- We can track where we've been
- We can display where we've been as well as current location
- We can exit if we revisit a location

But how safe is it?

- Try different inputs
- Try moving around a bit

What happens if...

Moving around and seeing what works

- Use the controls to move around the map
- Try entering some integers that aren't the movement

What issues do we find?

Walking off the edge of the map

Our Tourist can walk outside of the bounds of our arrays!

Let's add some code to check if we're outside the map and stop that movement

```
// Check if we've walked off the map
if (posR < 0) {
    posR = 0;
} else if (posR >= N_ROWS) {
    posR = N_ROWS - 1;
}
if (posC < 0) {
    posC = 0;
} else if (posC >= N_COLS) {
    posC = N_COLS - 1;
}
```

Where else can we take this code?

What about scoring?

- Could we give the player a score based on the number of places they visited?
- How would we calculate that?
- Also . . .
- Some of this code might be useful in understanding the first assignment

What did we learn today?

Functions and Libraries

- We can use functions that we didn't write ourselves
- We can include libraries that have many functions that can help us

Multi-Dimensional Arrays

- We can work with arrays in arrays to make things like grids

COMP1511 - Programming Fundamentals

— Week 4 - Lecture 8 —

What did we learn last lecture?

Functions and Libraries

- Using functions we haven't written
- `#include` to use C standard libraries

Multi-Dimensional Arrays

- Arrays of arrays
- Like a 2D (or more than 2 dimensions) map or grid

What are we covering today?

Memory

- Looking at computer memory in more detail

Pointers

- A C variable that lets us access memory directly

Libraries recap

We can use functions from the C standard libraries

- These are added to our code using `#include`
- Once they're included, they provide access to their functions
- An example is `<stdio.h>` giving access to `printf` and `scanf`
- We'll see more of these today!

Multi Dimensional Arrays Recap

We can store any variables in arrays

- Arrays are variables!
- We can store arrays in arrays
- In 2 dimensions, this can build a grid

Indexes	0	1	2	3	4
0	63	88	43	55	67
1	54	52	91	21	32
2	77	58	1	61	79

A 2D Array

Accessing 2D Arrays

Two coordinates to access single elements

- We use two dimensions to create the 2D array
- We also use two coordinates to get access to a single element

```
int main (void) {
    // declare a 2D Array
    int grid[4][4] = {0};

    // test a value
    if (grid[2][0] < 1) {
        // print out a value
        printf("The bottom left square is: %d", grid[3][0]);
    }
}
```

Memory and addressing

More detail about how memory works in our computer

- Let's start with an idea of a neighbourhood
- Each house is a piece of memory (a byte or more, depending)
- Every house has a unique address that we can use to find it

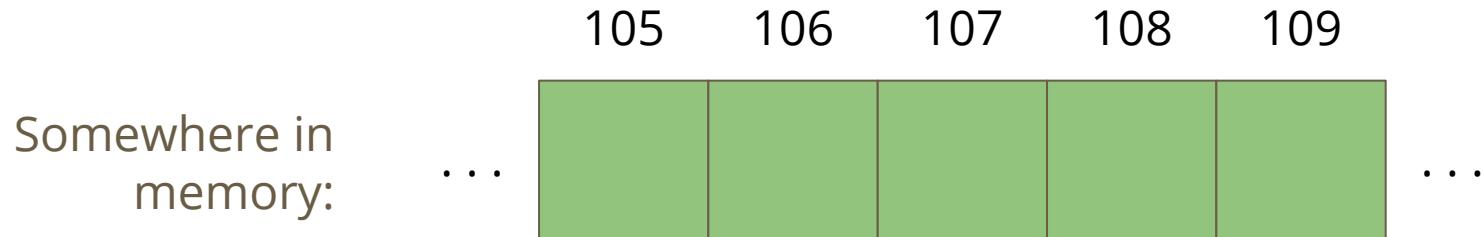
Arrays work a bit like this . . .

- We've already seen indexing into arrays to find elements
- We could think of our entire computer's memory as a big array of bytes

A neighbourhood of memory

Every block of memory has an address

- The address is actually an integer
- If I have that address, it means I can find the variable wherever it is in memory
- Just like if I have an address to a house, I'll be able to find it



Houses and addresses

Continuing the idea . . .

- A variable is a house
- That house is in a certain location in memory, its address
- The house contains the bits and bytes that decide what the value of the variable is

The address is an integer

- In a 64 bit system, we'll usually use a 64 bit integer to store an address
- We can address 2^{64} bytes of memory

Introducing Pointers

A New Variable Type - Pointers

- Pointers are variables that hold memory addresses
- They are created to point at the location of other variables
- If a variable was a house, the pointer would be the address of that house
- In C, the pointer is like an integer variable that stores a memory address
- Pointers are usually created with the intention of "aiming at" a variable (storing a particular variable's address)

Pointers in C

Pointers can be declared, but slightly differently to other variables

- A pointer is always aimed at a particular variable type
- We use a ***** to declare a variable as a pointer
- A pointer is most often "aimed" at a particular variable
- That means the pointer stores the address of that variable
- We use **&** to find the address of a variable

```
int i = 100;
// create a pointer called ip that points at i
// it stores the address of i
int *ip = &i;
```

Pointer Types

Different pointers to point at different variables

```
// some variables
int i;
double d;

// some pointers to particular variables
// * declares a pointer variable
// & finds the address of a variable
int *ip = &i;
double *dp = &d;
```

Initialising Pointers

Pointers should be initialised like other variables

- Generally pointers will be initialised by pointing at a variable
- "**NULL**" is a **#define** from most standard C libraries (including stdio.h)
- If we need to initialise a pointer that is not aimed at anything, we will use **NULL**

Using Pointers

If we want to look at the variable that a pointer “points at”

- We use the `*` on a pointer to access (dereference) the variable it points at
- Using the address analogy, this is like following the address to actually get to the house, then looking inside

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

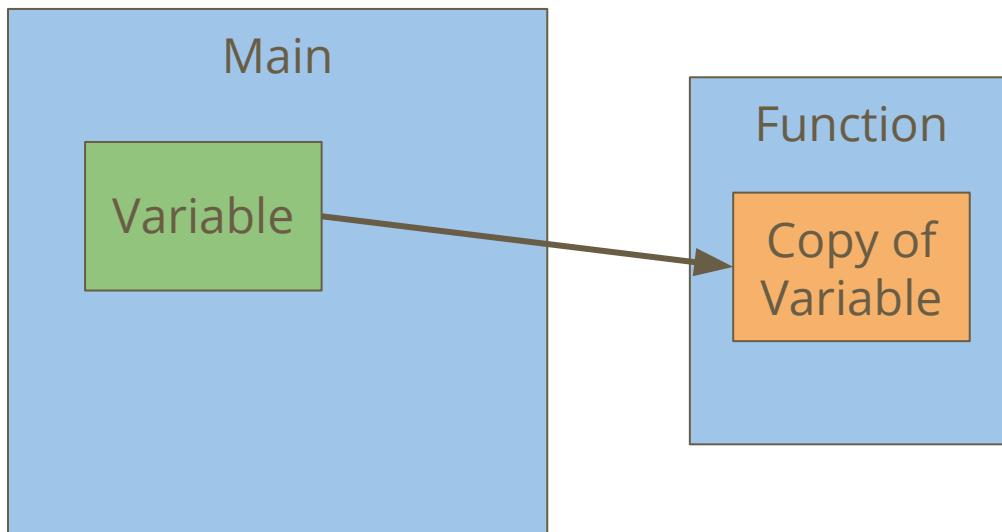
- `%p` in `printf` will print the address stored in a pointer

Pointers and Functions

Pointers allow us to pass around an address instead of a variable

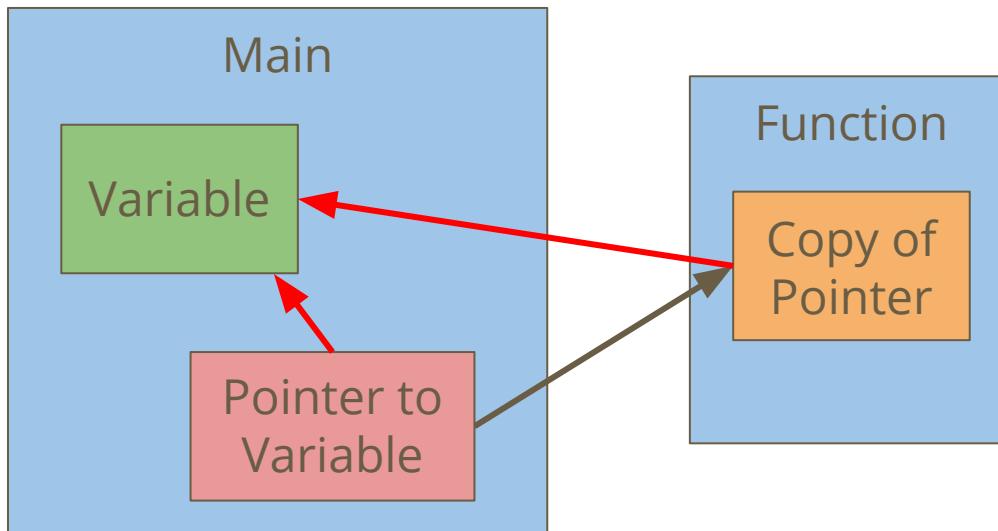
- We can create functions that take pointers as input
- All function inputs are always passed in "by value" which means they're copies, not the same variable
- But if I have a copy of the address of a variable, I can still find exactly the variable I'm looking for

Function variables pass in "by value"



In this case, the copy of the variable can't ever change the value of the variable, because it's just a copy

Pointers pass in "by value" also



The function has a copy of the pointer.

However, even a copy of a pointer contains the address of the original variable, allowing the function to access it.

Pointers and Functions in code

The following code illustrates the two examples

- A variable passed to a function is a copy and has no effect on the original
- A pointer passed to a function gives us the address of the original

```
// this function will have no effect!
void incrementInt(int n) {
    n = n + 1;
}
// this function will affect whatever n is pointing at
void incrementPointer(int *n) {
    *n = *n + 1;
}
```

Pointers and Functions

We can now do more with functions

- Pointers mean we can give a function access to multiple variables
- This means one function can now change multiple variables at once

```
// This function is now possible!
void swap(int *n, int *m) {
    int tmp;
    tmp = *n;
    *n = *m;
    *m = tmp;
}
```

Pointers and Arrays

Arrays are blocks of memory

- An array variable actually stores the memory address of the start of the array!
- This is why arrays as input to functions let you change the array

```
int numbers[10];
// both of these print statements
// will print the same address!
printf("%p\n", &numbers[0]);
printf("%p\n", numbers);
```

Pointers to Pointers

- Pointers are variables
- Pointers can point at variables
- uh oh . . .
- For now, we will not use pointers aimed at other pointers, but in the future you may find uses for them



Break Time

Making Art with Computers

- Microsoft Paint is one of the simplest computer art programs
- The theme of our first assignment (CS Paint)
- What follows from something simple like colouring pixels . . .
- Ends up with 3D games, VR and blockbuster movies
- You can run a pretty version of CS Paint in a CSE terminal by using the command: `1511 canvas solution`

Let's make a program using functions and pointers

This program is called The Jumbler

- It will take some numbers as inputs
- It will jumble them a little, changing their order
- Then it will print them back out

- We'll make some use of functions to separate our code
- We'll show how pointers let us access memory in our program

What functions do we want?

Deciding how to split up our functionality

- A function that reads the inputs as integers
- A function that swaps two numbers
- A function that swaps several numbers
- A function that prints out our numbers

Reading Input

A function to read inputs into an array

- We're also going to want to know how many numbers are being entered!

```
int read_inputs(int nums[MAX_NUMS]) {
    int i = 0;
    int inputCount = 0;
    printf("How many numbers? ");
    scanf("%d", &inputCount);
    while (i < MAX_NUMS && i < inputCount) { // have processed i inputs
        scanf("%d", &nums[i]);
        i++;
    } // have processed i inputs in total
    return inputCount
}
```

Printing our numbers

This is a trivial function

- The only issue is that we might have to work with an array that isn't full
- So we use numCount to stop us early if necessary

```
void print_nums(int nums[MAX_NUMS], int numCount) {
    int i = 0;
    while (i < MAX_NUMS && i < numCount) {
        printf("%d ", nums[i]);
        i++;
    }
}
```

Using Pointers to swap variable values

A simple swap function

- This function doesn't even know whether the ints are in arrays or not
- It sees two memory locations containing ints
- and uses a temporary int variable to swap them

```
void swap_nums(int *num1, int *num2) {  
    int temp = *num1;  
    *num1 = *num2;  
    *num2 = temp;  
}
```

Jumble performs some swaps

This function just loops through and swaps a few numbers

- This is a good candidate for a function that could be changed or written differently and just used by our main without thinking about it

```
void jumble(int nums[MAX_NUMS], int numCount) {
    int i = 0;
    while (i < MAX_NUMS && i < numCount) {
        int j = i * 2;
        if (j < MAX_NUMS && j < numCount) {
            swap_nums(&nums[i], &nums[j]);
        }
        i++;
    }
}
```

Using all the functions in the main

A nice main makes use of its functions

- It's very easy to read this main!
- It shows its steps using its function names
- There isn't much code to dig through

```
int main(int argc, char *argv[]) {
    int numbers[MAX_NUMS];
    int numInputs = read_args(numbers);
    jumble(numbers, numInputs);
    print_nums(numbers, numInputs);
    return 0;
}
```

It's a simple program, but what's different?

Using functions, we have much more readable code

- Large sections of code are outside of the main
- The main itself is now very readable
- Each separate piece of functionality is on its own

Pointers give us access to other parts of memory

- We can give access to our variables via pointers

What did we learn today?

Memory and Pointers

- Pointers are variables that contain memory addresses
- We can use them to get access to variables anywhere in our program
- Functions operate in their own memory "space"

Using Functions

- A practical example of how functions can separate code
- Makes our code very readable
- Also means that all of the code for a specific purpose is collected together

COMP1511 - Programming Fundamentals

— Week 5 - Lecture 9 —

What did we learn last week?

Functions

- Libraries

Arrays

- 2D Arrays

Pointers

- Variables that store memory addresses
- How we use memory addresses to access variables

What are we covering today?

Pointers recap

Debugging

- What's a bug?
- How do we find them and remove them?

Characters

- Variables for letters

Recap - Pointers and Memory

What is a pointer?

- It's a variable that stores the address of another variable of a specific type
- We call them pointers because knowing something's address allows you to "point" at it

Why pointers?

- They allow us to pass around the address of a variable instead of the variable itself

Using Pointers

Pointers are like street addresses . . .

- We can create a pointer by declaring it with a `*` (*like writing down a street address*)
- If we have a variable (*like a house*) and we want to know its address, we use `&`

```
int i = 100;
// create a pointer called ip that
// stores the address of i
int *ip = &i;
```

Using Pointers

If we want to look at the variable that a pointer “points at”

- We use the * on a pointer to access the variable it points at
- Using the address analogy, this is like navigating to the house at that address and looking inside the house

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

Pointers in Functions

We'll often use pointers as input to functions

- Pointers give a function access to a variable that's in memory
- They also allow us to affect multiple variables instead of only having one output

```
void swap_nums(int *num1, int *num2) {
    int temp = *num1;
    *num1 = *num2;
    *num2 = temp;
}
```

Pointers and Arrays

These are very similar

- Arrays are sections of memory that contain multiple identical variables
- The variable of the array itself stores the memory address of the start of the array
- Pointers are also memory addresses
- This gives both pointers and arrays access to memory

Debugging

It's going to take up most of your time as a programmer!

- What is a bug?
- Different types of bugs
- How to find bugs

Debugging is the process of finding and removing software bugs



What is a Software Bug?

Errors in code are called “bugs”

- Something we have written (or not written) in our code
- Any kind of error that stops the program from running as intended

Two most common types of bugs

- Syntax Errors
- Logical Errors

Syntax Errors

C is a specific language with its own grammar

- **Syntax** - the precise use of a language within its rules
- C is much more specific than most human languages
- Slight mistakes in the characters we use can result in different behaviour
- Some syntax errors are obvious and your compiler will find them
- Some are more devious and the error message will be the consequence of the bug, rather than the bug itself

Logical Errors

We can write a functional program that still doesn't solve our problem

- Logical errors can be syntactically correct
- But the program might not do what we intended!

Human error is real!

- Sometimes we read the problem specification wrongly
- Sometimes we forget the initial goal of the program
- Sometimes we solve the wrong problem
- Sometimes we forget how the program might be used

How do we find bugs?

Sometimes they find us . . .

- **Compilers** can catch some syntactical bugs
- We'll need to learn how to use compilers to find bugs
- Code Reviews and pair programming help for logical bugs
- **Testing** is always super important!
- Learning how to test is a very valuable skill

Using our compiler to hunt syntax bugs

The Compiler can be trusted to understand the language better than us

- The simplest thing we can do is run **dcc** and see what happens

What to do when **dcc** gives you errors and warnings

- Always start with the first error
- Subsequent errors might just be a consequence of that first line
- An error is the result of an issue, not necessarily the cause
- At the very least, you will know a line and character where something has gone wrong

Solving Compiler Errors

Compiler Errors will usually point out a syntax bug for us

- Look for a line number and character (column) in the error message
- Sometimes knowing where it is is enough for you to solve it

- Read the error message and try to interpret it
- Remember that the error message is from a program that reads code
- It might not make sense initially!
- Sometimes it's an expectation of something that's missing
- Sometimes it's confusion based on something being incorrect syntax

Break Time

Testing is hard!

- You'll never know all the possible ways your program might be used!



Brenan Keller

@brenankeller

Follow

A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

1:21 PM - 30 Nov 2018

Let's look at some code and fix some bugs

`debugThis.c` is a program with some bugs in it . . .



What errors did we find?

Just focusing on fixing compiler errors, let's read and fix some code

What did we discover? (*spoilers here . . . try debugging before reading this slide!*)

- Single = if statement.
 - = is an assignment of a value
 - == is a relational operator
- An extra bracket causes a lot of issues (and a very odd error message)

Testing

We'll often test parts of all of our code to make sure it's working

- Simple - Run the code
- Try different types of inputs to see different situations (autotest is your teaching staff writing these tests!)
- Try using inputs that are not what is expected

How do you know if the tests are succeeding?

- Use output to show information as the program runs
- Check different sections of the code to see where errors are

Simple Testing

Let's use a good process here that we can apply to all code testing

- Write your program to give you a lot of information
- Test with intention. It's valuable to test with specific goals
- Be able to find out what the code is doing at different points in the code
- Be able to separate different sections of code

Finding a needle in a haystack gets easier if you can split the haystack into smaller parts!

Let's try some information gathering

Some of the tricks we'll use, continuing with our `debugThis.c`

- What is the program supposed to do?
- Decide on some ranges of inputs to test
- Modify the code to give useful information while it's running

What did we test?

What techniques did we use?

- Try different input ranges, including 0 and negative numbers
- Try outputting x and y values to make sure they're working
- Try outputting loop information so that we can see our structure

When we do good testing, we will be able to find our logical errors

Logical errors can be hard to find because the code looks correct syntactically

Characters

We've only used ints and doubles so far

- We have a new type called **char**
- Characters are what we think of as letters, like 'a' , 'b' , 'c' etc
- They can also represent numbers, like '0' , '1' , '2' etc
- They are actually **8 bit** integers!
- We use them as characters, but they're actually encoded numbers
- ASCII (American Standard Code for Information Interchange)
- We will rarely be using **char** for individual characters, but we will in arrays

ASCII and Characters as numbers

We make use of ASCII, but we don't need to know it

- ASCII specifically uses values 0-127 and encodes:
 - Upper and Lower case English letters
 - Digits 0-9
 - Punctuation symbols
 - Space and Newline
 - And more . . .
- It's not necessary to memorise ASCII, rather it's important to remember that characters can be treated like numbers sometimes

Characters in code

```
#include <stdio.h>

int main (void) {
    // we're using an int to represent a single character
    int character;
    // we can assign a character value using single quotes
    character = 'a';
    // This int representing a character can be used as either
    // a character or a number
    printf("The letter %c has the ASCII value %d.\n", character,
character);
    return 0;
}
```

Note the use of %c in the printf will format the variable as a character

Helpful Library Functions

`getchar()` is a function that will read a character from input

- Reads a byte from standard input
- Usually returns an int between 0 and 255 (ASCII code of the byte it read)
- Can return a -1 to signify end of input, "EOF" (which is why we use an int, not a char)
- Sometimes `getchar()` won't get its input until enter is pressed at the end of a line

`putchar()` is a function that will write a character to output

- Will act very similarly to `printf("%c", character);`

Use of getchar() and putchar()

```
// using getchar() to read a single character from input
int inputChar;
printf("Please enter a character: ");
inputChar = getchar();
printf("The input %c has the ASCII value %d.\n", inputChar, inputChar);

// using putchar() to write a single character to output
putchar(inputChar);
```

Invisible Characters

There are other ASCII codes for "characters" that can't be seen

- Newline(\n) is a character
- Space is a character
- There's also a special character, **EOF** (End of File) that signifies that there's no more input
- EOF has been **#defined** in **stdio.h**, so we use it like a constant
- We can signal the end of input in a Linux terminal by using Ctrl-D

Working with multiple characters

We can read in multiple characters (including space and newline)

This code is worth trying out . . . you get to see that space and newline have ASCII codes!

```
// reading multiple characters in a loop
int readChar;
readChar = getchar();
while (readChar != EOF) {
    printf(
        "I read character: %c, with ASCII code: %d.\n",
        readChar, readChar
    );
    readChar = getchar();
}
```

More Character Functions

`<ctype.h>` is a useful library that works with characters

- `int isalpha(int c)` will say if the character is a letter
- `int isdigit(int c)` will say if it is a numeral
- `int islower(int c)` will say if a character is a lower case letter
- `int toupper(int c)` will convert a character to upper case
- There are more! Look up `ctype.h` references or `man` pages for more information

What did we learn today?

Pointers Recap

- A quick recap of the details of pointers

Debugging

- Different types of bugs (software errors)
- Syntax and Logical Errors
- Using testing to find bugs

Characters

- Variables representing letters

COMP1511 - Programming Fundamentals

— Week 5 - Lecture 10 —

What did we cover last lecture?

Debugging

- How to think about different bugs (code errors)
- Some tricks and techniques to remove bugs from our code

Characters

- A new variable type!
- Letters and other symbols

What are we covering today?

Characters

- Continuing characters

Strings

- Words that contain multiple characters

Command Line Arguments

- Input at the moment the program starts running

Characters recap

```
#include <stdio.h>

int main (void) {
    // we're using an int to represent a single character
    int character;
    // we can assign a character value using single quotes
    character = 'a';
    // This int representing a character can be used as either
    // a character or a number
    printf("The letter %c has the ASCII value %d.\n", character,
character);
    return 0;
}
```

Note the use of %c in the printf will format the variable as a character

Helpful Functions

getchar() is a function that will read a character from input

- Reads a byte from standard input
- Usually returns an int between 0 and 255 (ASCII code of the byte it read)
- Can return a -1 to signify end of input, EOF (which is why we use an int, not a char)
- Sometimes **getchar** won't get its input until enter is pressed at the end of a line

putchar() is a function that will write a character to output

- Will act very similarly to **printf("%c", character);**

Use of getchar() and putchar()

```
// using getchar() to read a single character from input
int inputChar;
printf("Please enter a character: ");
inputChar = getchar();
printf("The input %c has the ASCII value %d.\n", inputChar, inputChar);

// using putchar() to write a single character to output
putchar(inputChar);
```

Invisible Characters

There are other ASCII codes for “characters” that can’t be seen

- Newline(\n) is a character
- Space is a character
- There’s also a special character, **EOF** (End of File) that signifies that there’s no more input
- **EOF** has been **#defined** in **stdio.h**, so we use it like a constant
- We can signal the end of input in a Linux terminal by using Ctrl-D

Working with multiple characters

We can read in multiple characters (including space and newline)

This code is worth trying out . . . you get to see that space and newline have ASCII codes!

```
// reading multiple characters in a loop
int readChar;
readChar = getchar();
while (readChar != EOF) {
    printf(
        "I read character: %c, with ASCII code: %d.\n",
        readChar, readChar
    );
    readChar = getchar();
}
```

More Character Functions

`<ctype.h>` is a useful library that works with characters

- `int isalpha(int c)` will say if the character is a letter
- `int isdigit(int c)` will say if it is a numeral
- `int islower(int c)` will say if a character is a lower case letter
- `int toupper(int c)` will convert a character to upper case
- There are more! Look up `ctype.h` references or `man` pages for more information

Strings

When we have multiple characters together, we call it a string

- Strings in C are arrays of **char** variables
- Strings are like words (or sentences), while chars are single letters
- Strings have a helping element at the end, a character: '\0'
- It's often called the 'null terminator' and it is an invisible character
- This marks the end of the string
- It helps us because we know we won't read any further into the array

Strings in Code

Strings are arrays of type `char`, but they have a convenient shorthand

```
// a string is an array of characters
char word1[] = {'h','e','l','l','o','\0'};
// but we also have a convenient shorthand
// that feels more like words
char word2[] = "hello";
```

Both of these strings will be created with 6 elements. The letters `h,e,l,l,o` and the null terminator `\0`



Reading and writing strings

`fgets(array[], length, stream)` is a useful function for reading strings

- It will take up to `length` number of characters
- They will be written into the `array`
- The characters will be taken from a stream
- Our most commonly used stream is called `stdin`, “standard input”
- `stdin` is our user typing input into the terminal

Reading and writing strings in code

```
// reading and writing lines of text
char line[MAX_LINE_LENGTH];
while (fgets(line, MAX_LINE_LENGTH, stdin) != NULL) {
    fputs(line, stdout);
}
```

- **fputs(array, stream)** works very similarly to printf
- It will output the string stored in the array to a stream
- We can use **stdout** which is our stream to write to the terminal

Helpful Functions in the String Library

`<string.h>` has access to some very useful functions

Note that `char *s` is equivalent to `char s[]` as a function input

- `int strlen(char *s)` - return the length of the string (not including `\0`)
- `strcpy` and `strncpy` - copy the contents of one string into another
- `strcat` and `strncat` - attach one string to the end of another
- `strcmp` and variations - compare two strings
- `strchr` and `strrchr` - find the first or last occurrence of a character
- And more ...

Command Line Arguments

Sometimes we want to give information to our program at the moment when we run it

- The "**Command Line**" is where we type in commands into the terminal
- **Arguments** are another word for input parameters

```
$ ./program extra information 1 2 3
```

- This extra text we type after the name of our program can be passed into our program as strings

Main functions that accept arguments

`int main` doesn't have to have `void` input parameters!

```
int main(int argc, char *argv[]) {  
}
```

- **argc** will be an "argument count"
- This will be an integer of the number of words that were typed in (including the program name)
- **argv** will be "argument values"
- This will be an array of strings where each string is one of the words



An example of use of arguments

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i = 1;
    printf("Well actually %s says there's no such thing as ", argv[0]);
    while (i < argc) {
        fputs(argv[i], stdout);
        printf(" ");
        i++;
    }
    printf("\n");
}
```

Arguments in argv are always strings

But what if we want to use things like numbers?

- We can read the strings in, but we might want to process them

```
$ ./program extra information 1 2 3
```

- In this example, how do we read 1 2 3 as numbers?
- We can use a library function to convert the strings to integers!
- `strtol()` - "string to long integer" is from the `stdlib.h`

Code for transforming strings to ints

Adding together the command line arguments

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

int main(int argc, char *argv[]) {
    int total = 0;

    int i = 1;
    while (i < argc) {
        total += strtol(argv[i], NULL, 10);
        i++;
    }
    printf("Total is %d.\n", total);

}
```

Break Time

We're roughly halfway through COMP1511

- This time can sometimes be rough
- Sometimes, we're just holding on until the end of the year
- Remember that you only have to take one step at a time
- Your goals might be so far away that you can't think of how to reach them
- But you only have to move a little bit towards them at a time
- And you'll get there eventually!

Whooaaah We're Halfway There ...

We're going to use a bit of everything we've seen so far in COMP1511

This program is a rhyming helper

- It will read in a string from the command line
- It will then read in another string from the user and tell us whether it thinks they might rhyme
- It does this by checking the input string against the last word in the command line and seeing how similar they are
- This will use nearly all the topics we've covered so far in COMP1511

Where will we start?

A simple version to begin with

- Let's read in a string from the command line
- Then read in a single character from standard input and see whether it's in the string or not

Then we complicate things

- We'll try to compare two strings and see if they're similar

Read in strings from the command line

We're expecting these on the command line, so let's check there

- `argc` should tell us how many strings there are

```
int main(int argc, char *argv[]) {
    if (argc <= 1) {
        // there's no extra input on the command line!
        printf("You can't rhyme with nothing!\n");
        return 1;
}
```

Read in a single character

Starting simple, we can take a character as input

- `getchar()` will read a single character from standard input
- Remember that we'll be using `int` as our type for individual characters

```
// starting with inputChar = EOF lets us know later
// whether getchar() replaced it with a character
// or not
int inputChar = EOF;
inputChar = getchar();
if (inputChar != EOF) {
    // we know we've read a character
}
```

A Function to find a character in a string

Looping through a string until the null terminator

```
int check_letter(char letter, char word[]) {
    int found_index = -1;
    int i = 0;

    // The while loop check will loop through
    // until the string is terminated.
    while (word[i] != '\0') {
        if (word[i] == letter) {
            found_index = i;
        }
        i++;
    }
    return found_index;
}
```

We're interested in the last word

How do we know what the last word is?

- `argc` tells us how many words there are!
- So the index of the last word is `argc - 1`
- We can check for the letter in the last word

```
// argv[argc - 1] is the last word of the command line
int found_letter = check_letter(input_char, argv[argc - 1]);
```

Testing a whole word

We could loop `getchar()` to grab multiple characters

- Or we can try another library function that grabs a whole line of text!
- `fgets()` will read a line from standard input

```
// read a line of input
char input_line[MAX_LENGTH];
printf("Please enter a word to test for rhyming.\n");
fgets(input_line, MAX_LENGTH, stdin);
```

How well do two words rhyme?

How many letters appear in the other word (not a great test for rhyming)

```
double rhyming_amount(char word1[], char word2[]) {  
    // Loop through word1 and check if the letter is in word2  
    int match_count = 0;  
    int i = strlen(word1) - 1;  
    while (i >= 0) {  
        int found_letter = check_letter(word1[i], word2);  
        if (found_letter >= 0) {  
            // found the same letter in the final word  
            match_count++;  
        }  
        i--;  
    }  
    return (match_count * 1.0)/strlen(word1);  
}
```

Using Library Functions

Where does the `strlen()` come from?

- This function will tell us how long a string is
- We need to `#include <string.h>` to use it

Are we sure our program is working?

What tests should we run at this point?

- Look for syntax errors using our compiler (dcc)
- Look for logical errors by testing with different inputs

We might need to add in some extra outputs

- If we're getting strange behaviour, we can confirm our guesses
- We might learn more about what's going on in our program

Are there more characters than we intended?

Maybe we need to check what those characters are

- Some temporary print statements can help here

```
int check_letter(char letter, char word[]) {  
    printf("Checking for %c", letter);  
    printf("in word %s.\n", word);
```

```
double rhyming_amount(char word1[], char word2[]) {  
    printf("Checking %s", word1);  
    printf("against %s.\n", word2);
```

Dealing with little issues

We're reading newlines (\n) as characters!

- Let's remove the newlines from our `fgets()` result
- We'll look for \n at the end of the string
- We'll then replace the \n with \0 which will end the string early

Removing a suspected newline

Removing a \n at the end of a string:

```
// read a line of input
    char input_line[MAX_LENGTH];
    printf("Please enter a word to test for rhyming.\n");
    fgets(input_line, MAX_LENGTH, stdin);

// check for a \n at the end of the input and remove it
    int last_letter = strlen(input_line) - 1;
    if (input_line[last_letter] == '\n') {
        input_line[last_letter] = '\0';
    }
```

A simple rhyming helper

What coding concepts have we used here that we want to remember?

- Characters and Strings (note that we'll never need to memorise the ASCII table to work with characters)
- Using libraries and provided functions
- Loops on strings (using the Null Terminator \0)
- Writing multiple functions and using functions within functions
- A lot of our basic C concepts like if, while and array indexing

Challenge?

You may have noticed that `rhyming_amount()` loops backwards . . .

- A challenge . . . for bonus Marcs (no actual marks)
- Rhyming amount is a bit simplistic, just checking letter matches
- Can you extend it so that it specifically starts at the end of the words and works backwards and tests the matches for the exact ordering of letters?
- Eg: "light" rhymes with "tonight" because they both end in the same letters
- There are also more standard library functions that might be able to replace some of our code . . . see if you can discover them

What did we learn today?

Characters and Strings

- Expanding our variables to letters and words
- A code example to show some of the use of strings
- Using libraries to make strings easier

Command Line Arguments

- How to take information from the same line that runs the program

COMP1511 - Programming Fundamentals

— Week 7 - Lecture 11 —

What did we cover last time?

Characters and Strings

- Using letters as variables
- Using arrays of letters
- Some useful library functions

Command Line Arguments

- Reading strings from the command line

What are we covering today?

Memory

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

Structs

- Making custom variables
- Collections of variables that aren't all the same type

Functions and Memory - a recap

What actually gets passed to a function?

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

What happens to variables passed to functions?

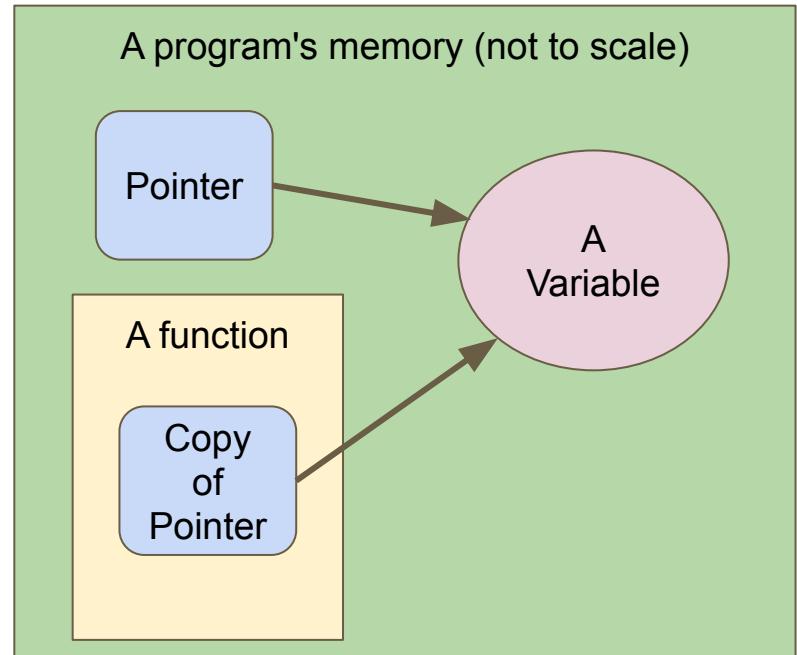
```
int main (void) {
    int x = 5;
    doubler(x);
    printf("x is %d.\n", x,);
    // "x is 5"
    // this is because the doubler function takes the value 5 from x
    // and copies it into the variable "number" which is a new variable
    // that only lasts as long as the doubler function runs
}

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

Functions and Pointers

What happens to pointers that are passed to functions?

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



Functions and Pointers

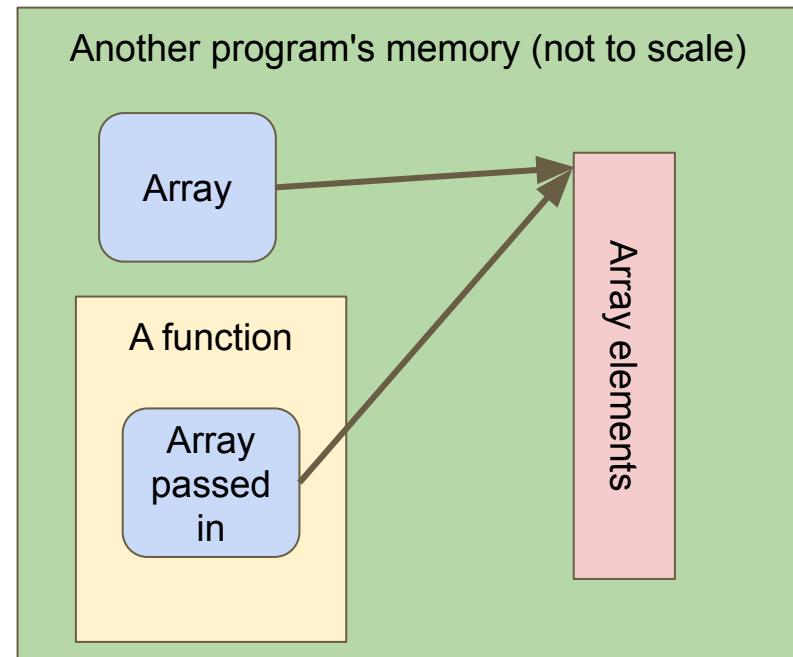
```
int main (void) {
    int x = 5;
    int *pointerX = &x;
    doublePointer(pointerX);
    printf("x is %d.\n", x);
    // "x is 10"
    // This is because doublePointer gets given access to x via its
    // copied pointer . . . since it changes what's at the other end of
    // that pointer, it affects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

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

Arrays are represented as memory addresses

Arrays and pointers are very similar

- An array is a variable
- It's not actually a variable containing all the elements
- When we use the array variable (no []), it's actually the memory address of the start of the elements
- Arrays and pointers are nearly identical when passed to functions



Functions and Arrays

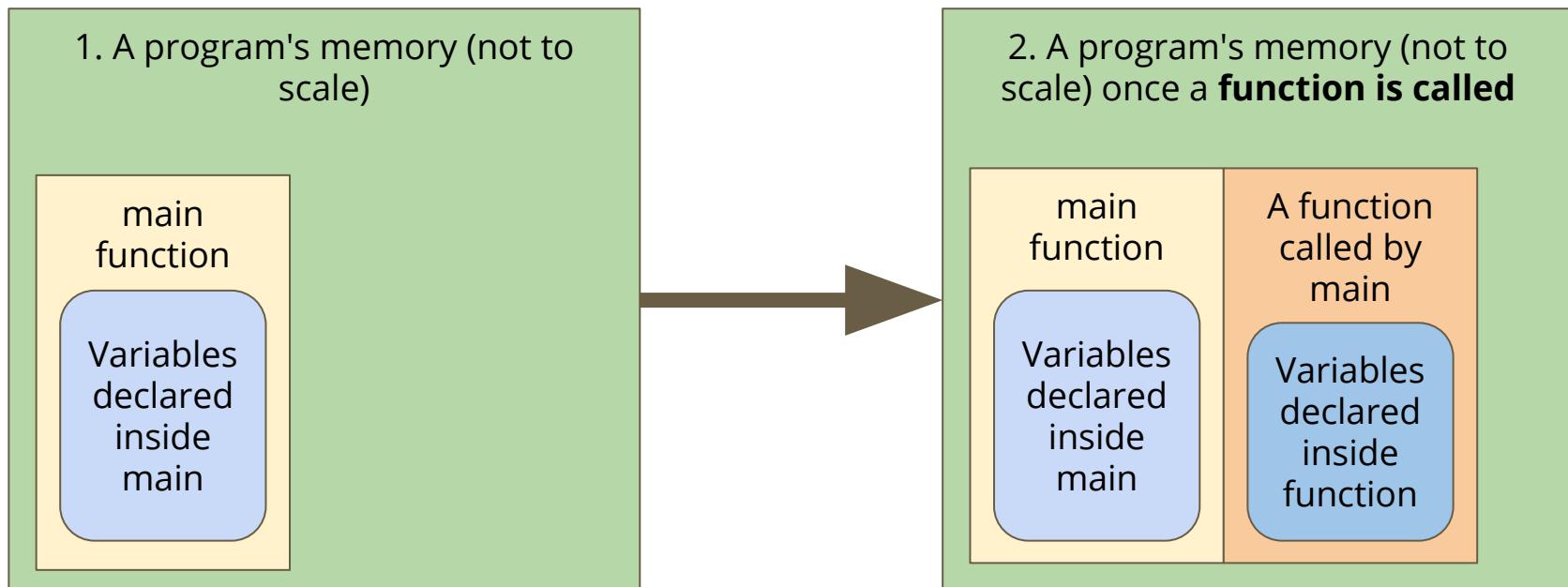
```
int main (void) {
    int myNums[3] = {1,2,3};
    doubleAll(3, myNums);
    printf("Array is: ");
    int i = 0;
    while(i < 3) {
        printf("%d ", myNums[i]);
        i++;
    }
    printf("\n");
    // "Array is 2 4 6"
    // Since passing an array to a function will pass the address
  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

Functions and Arrays continued

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

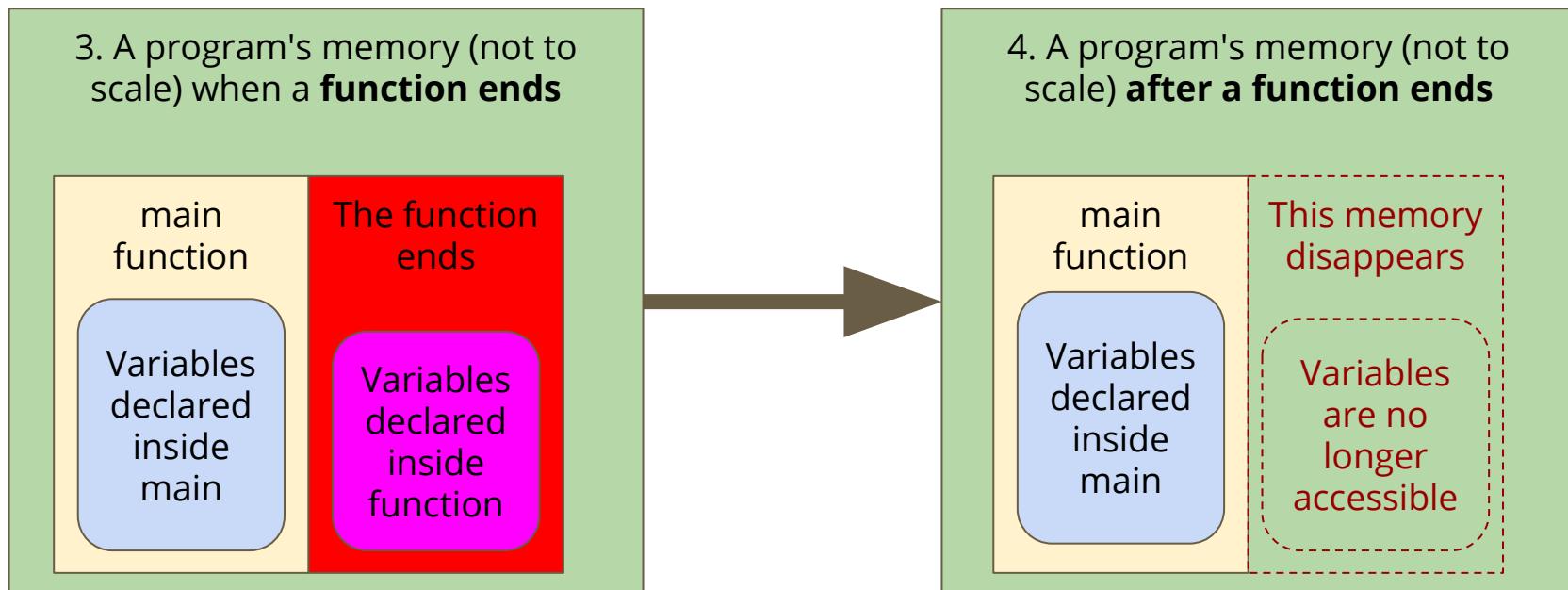
Memory in Functions

What happens to variables we create inside functions?



Memory in Functions

What happens to variables we create inside functions?



Keeping memory available

What if we want to create something in a function?

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

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

Memory Allocation

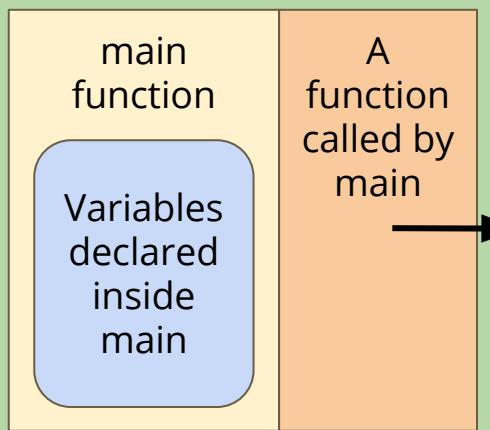
C has the ability to allocate memory

- A function called **malloc (bytes)** returns a pointer to memory
- Allows us to take control of a block of memory
- This won't automatically be cleaned up when a function ends
- To clean up the memory, we call **free (pointer)**
- **free ()** will use the pointer to find our previous memory to clean it up

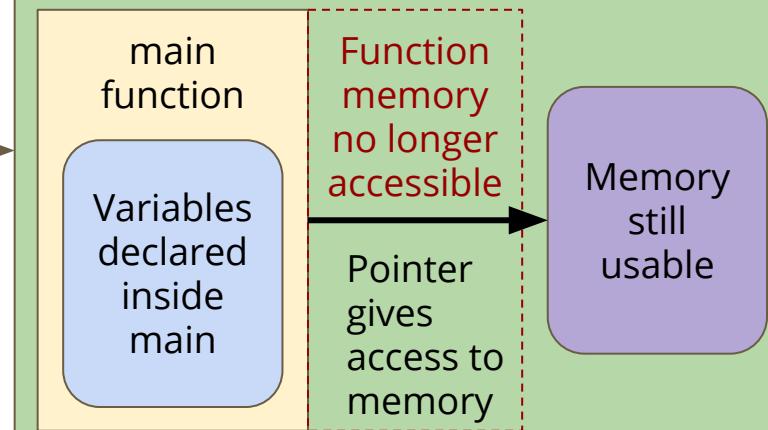
What malloc() does

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

1. A program's memory (not to scale) when a **function allocates memory**



2. A program's memory (not to scale) **after the function returns**



Malloc() in code

We can assign a particular amount of memory for use

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

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

Cleaning up after ourselves

Allocated memory is never cleaned up automatically

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

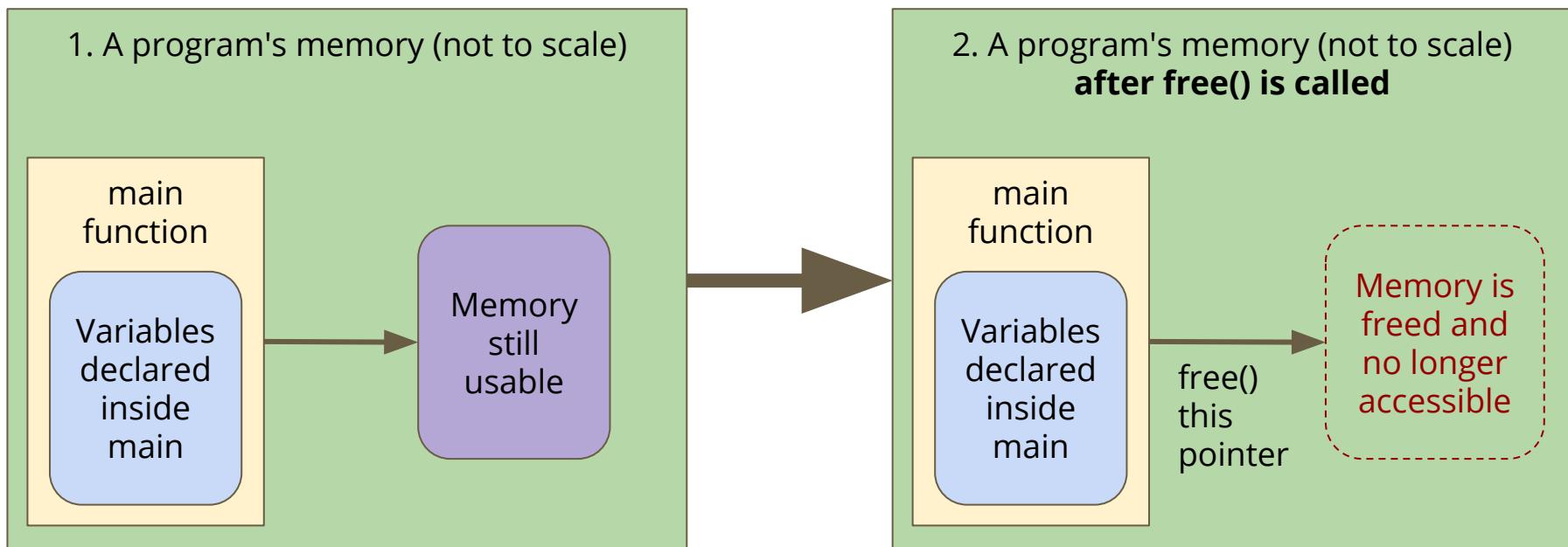
```
// Use an allocated variable via its pointer then free it
int main(void) {
    int *iPointer = mallocNumber();

    *iPointer += 25;

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

Freeing up memory

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



Using memory

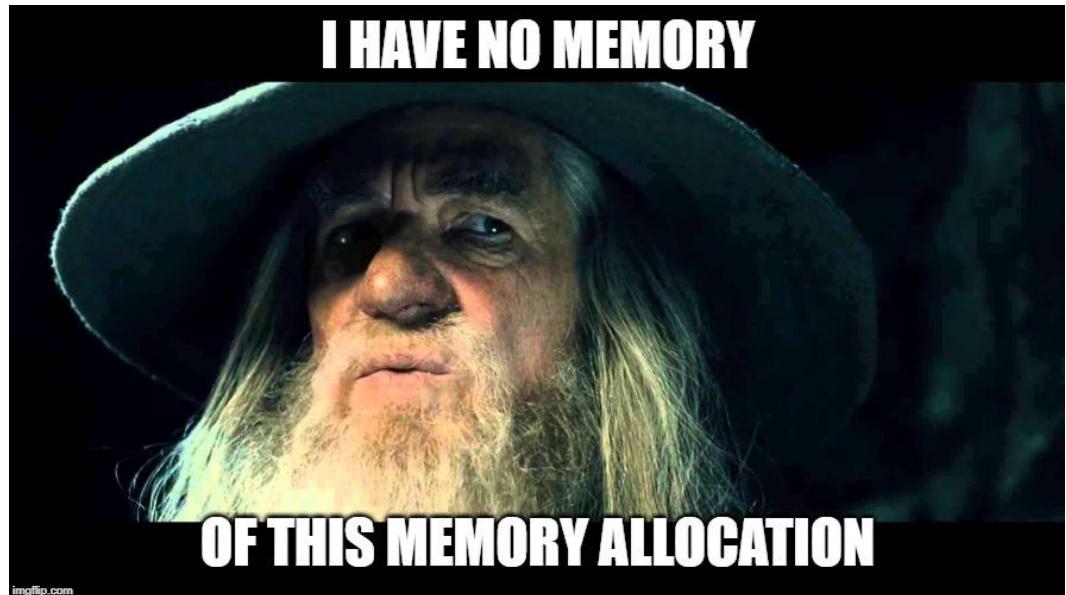
Some things to think about with `malloc()` and `free()`

- You can use `sizeof()` to figure out how many bytes something needs
- We can malloc arrays as well as variables
- In general, always use `sizeof()` with `malloc()`
- Anything allocated with `malloc()` must be `free()` after you've finished with it
- Otherwise we get what's known as memory leaks!
- `gcc --leak-check` can be used to tell you if you have any memory leaks

Break Time

Memory allocation is tricky

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



Structs

A new way of collecting variables together

- Structs (short for structures) are a way to create custom variables
- Structs are variables that are made up of other variables
- They are not limited to a single type like arrays
- They are also able to name their variables
- Structs are like the bento box of variable collections



Before we can use a struct ...

Structs are like creating our own variable type

- We need to declare this type before any of the functions that use it
- We declare what a struct is called and what the fields (variables) are

```
struct bender {
    char name[MAX_LENGTH];
    char element[MAX_LENGTH];
    int power;
};
```

Creating a struct variable and accessing its fields

Declaring and populating a struct variable

- Declaring a struct: "struct *structname* *variablename*;"
- Use the . to access any of the fields inside the struct by name

```
int main(void) {
    struct bender avatar;
    strcpy(avatar.name, "Aang");
    strcpy(avatar.element, "Air");
    avatar.power = 10;

    printf("%s's element is: %s.\n", avatar.name, avatar.element);
}
```

Accessing Structs through pointers

Pointers and structs go together so often that they have a shorthand!

-> is a new shorthand that avoids possible mistakes in dereferencing

```
struct bender *last_airbender = &avatar;

// knowledge of pointers suggests using this
(*last_airbender).power = 100;

// but there's another symbol that automatically
// dereferences the pointer and accesses a field
// inside the struct
last_airbender->power = 100;
```

Pointers and Structs

We often use pointers and structs together

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

```
void display_person(struct bender *person) {
    printf("Name: %s\n", person->name);
    printf("Element: %s\n", person->element);
    printf("Power: %d\n", person->power);
}
```

Structs as Variables

Structs can be treated as variables

- Yes, this means arrays of structs are possible
- It also means structs can be some of the variables inside other structs
- In general, it means that once you've defined what a struct is, you use it like any other variable

We'll do some live coding with structs later!

Benders - an example of structs and malloc

We want to form a team of people with special elemental powers

- We'd like to have a struct that can represent an individual
- Then we'd like to build up a team
- We'll maintain an array of pointers
- And allocate memory for the team members

Create Structs for Characters

Create a struct to allow us to represent the characters

We'll borrow the one we created earlier

```
struct bender {
    char name[MAX_LENGTH];
    char element[MAX_LENGTH];
    int power;
};
```

Building up a team

We could actually do this with another struct!

We can make a struct that has an array of pointers to other structs

```
struct team {
    char name[MAX_LENGTH];
    int numMembers;
    struct bender *teamMembers[TEAM_SIZE];
};
```

Creating a bender with a function

A function to allocate memory for a struct and give us a pointer to it

```
struct bender *createBender(
    char *benderName,
    char *benderElement,
    int benderPower
) {
    struct bender *newBender = malloc(sizeof(struct bender));

    strcpy(newBender->name, benderName);
    strcpy(newBender->element, benderElement);
    newBender->power = benderPower;

    return newBender;
}
```

Setting up our structures in memory

We can use malloc in a very similar way to declaring a variable

```
// allocate the memory for one instance of benderTeam
struct team *benderTeam = malloc(sizeof (struct team));
strcpy(benderTeam->name, "Avatar's team");

// Assigning the result of createBender to each element
// of benderTeam's teamMembers array.
benderTeam->teamMembers[0] = createBender("Aang", "Air", 10);
benderTeam->numMembers = 1;
benderTeam->teamMembers[1] = createBender("Katara", "Water", 6);
benderTeam->numMembers++;
benderTeam->teamMembers[2] = createBender("Sokka", "None", 2);
benderTeam->numMembers++;
```

Using structs without memory allocation

We can also use structs like regular variables

- Remember that accessing fields is different depending on whether you're using a pointer or not
- Accessing through a pointer: ->
- Accessing a variable: .

```
// And an example of creating a struct without using
// memory allocation.
struct bender zuko;
strcpy(zuko.name, "Prince Zuko");
strcpy(zuko.element, "Fire");
zuko.power = 9;
```

Printing the contents

A function to print out the team. This only needs one pointer as input!

```
// printTeam will print out the details of the team members
// to the terminal. It will not change the team.
void printTeam(struct team *printTeam) {
    printf("Team name is %s\n", printTeam->name);
    int i = 0;
    while (i < printTeam->numMembers) {
        printf("Team member %s uses the element: %s\n",
               printTeam->teamMembers[i]->name,
               printTeam->teamMembers[i]->element
        );
        i++;
    }
}
```

What's left? There's memory left!

We still have allocated memory that we haven't given back!

- Every allocated piece of memory must be freed before the program ends
- This means we'd have to free all the members of the team
- And also the team itself
- `gcc benders.c -o benders --leak-check`
- This command will create a version of the program that will check for memory leaks (unfreed memory allocations)

Some code for freeing memory

We can run a function that will clean up the memory for a team

```
// freeTeam will free all the memory used for a team.  
// It will first free all members, then the team itself  
void freeTeam(struct team *fTeam) {  
    int i = 0;  
    while (i < fTeam->numMembers) {  
        free(fTeam->teamMembers[i]);  
        i++;  
    }  
    free(fTeam);  
}
```

What did we learn today?

Functions and Memory

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

Structs

- Making our own custom variable types
- These can be collections of different types of variables

COMP1511 - Programming Fundamentals

— Week 7 - Lecture 12 —

What did we learn last lecture?

Memory

- Using memory beyond what's in our functions
- Allocating memory so that it lasts beyond the lifetime of the curly brackets

Structs

- Our custom variables
- Made up of other variables

What are we covering today?

Multi-File Projects

- Spreading a program over multiple files

Linked Lists

- Like an array, contains multiple of the same type of variable
- More flexible in that it can change length
- Is also able to add and remove elements from partway through the list
- Tying together structs, pointers and memory allocation

C Projects with Multiple Files

For readability and also to separate code by subject

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

Reusable sub-projects

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

Header Files and C (Implementation) Files

Two different files for different purposes

- Header and C files usually go together in pairs

Header *.h file

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

C Implementation *.c file

- Contains the underlying implementation of the H file

File.h

Header Files show you what the code's functions are

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

File.c

Implementation Files show you how the code runs in detail

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

Main.c and other Files

Our Entry Point into our code

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

Compiling a Project with Multiple Files

How do we compile multi-file project?

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

Let's look at a multi-file project

I'm Batman!

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

person.h

What's in the Header file?

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

person.c

Our implementation file

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

main.c

The main file

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

Using the multi-file project

Compiling

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

Using Multi-file projects in COMP1511

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

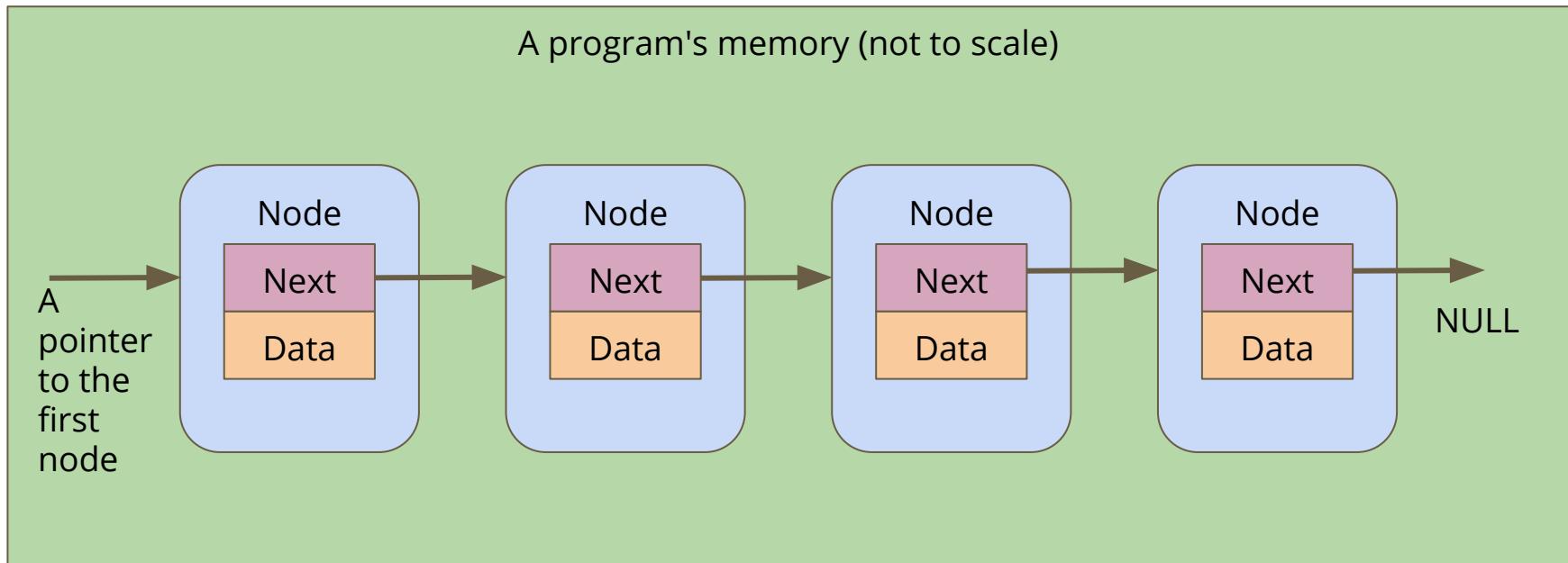
A new kind of struct

Let's make an interesting struct

- This is a node
- It contains some information
- As well as a pointer to another node of the same type!

```
struct node {  
    struct node *next;  
    int data;  
};
```

A Chain of Nodes - a Linked List



Linked Lists

A chain of these nodes is called a **Linked List**

As opposed to **Arrays** . . .

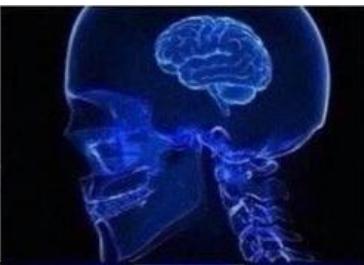
- Not one continuous block of memory
- Items can be shuffled around by changing where pointers aim
- Length is not fixed when created
- You can add or remove items from anywhere in the list

Break Time

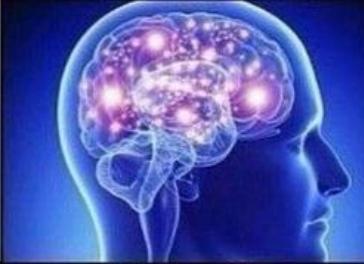
Linked Lists

- Pointers, structs and memory allocation
- Structs with pointers to their own type
- Linked Lists combine a lot of our newer code techniques

**ALLOCATING
MEMORY SO WE
DON'T LOSE THINGS**



**ALLOCATING
MEMORY
FOR STRUCTS**



**STRUCTS
WITH POINTERS
TO THEMSELVES**



**LINKED
LISTS**



Linked Lists in code

What do we need for the simplest possible list?

- A struct for a node
- A pointer to keep track of the start of the list
- A way to create a node and connect it

```
struct node {  
    struct node *next;  
    int data;  
};
```

A function to add a node

```
// Create a node using the data and next pointer provided
// Return a pointer to this node
struct node *createNode(int data, struct node *next) {
    struct node *n;
    n = malloc(sizeof (struct node));
    n->data = data;
    n->next = next;
    return n;
}
```

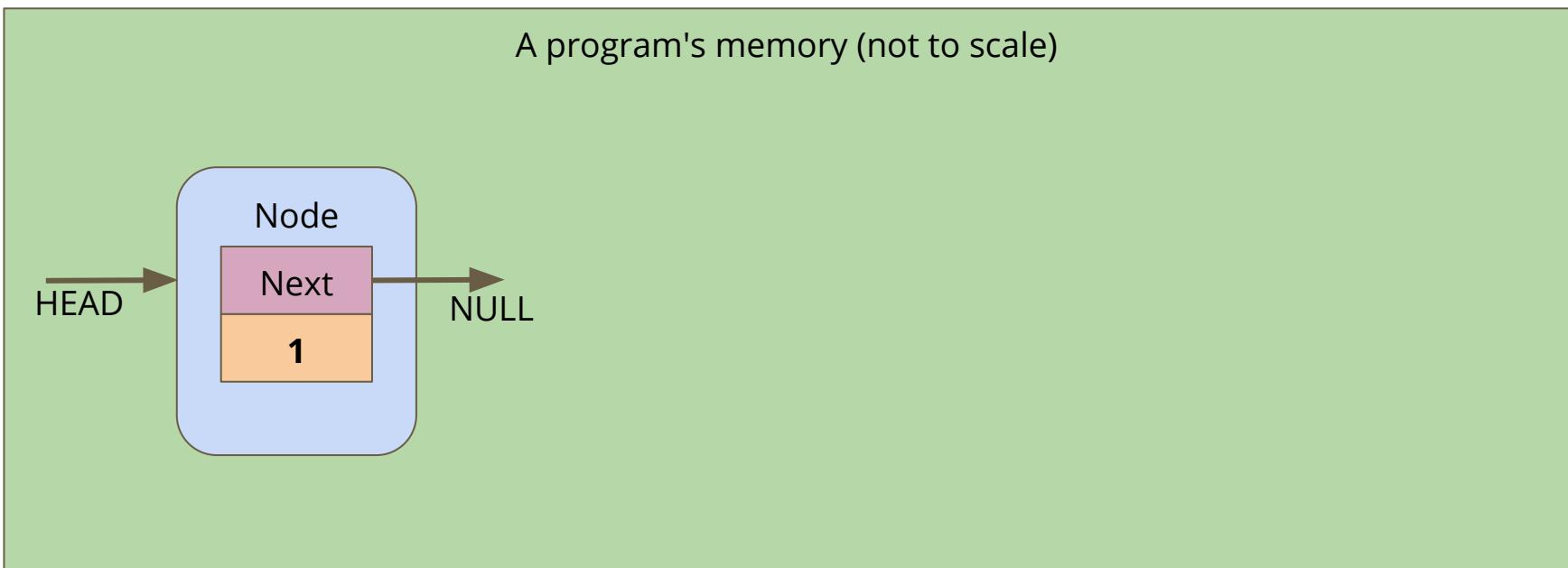
Building a list using only createNode()

```
int main (void) {
    // head will always point to the first element of our list
    struct node *head = createNode(1, NULL);
    head = createNode(2, head);
    head = createNode(3, head);
    head = createNode(4, head);
    head = createNode(5, head);

    return 0;
}
```

How it works 1

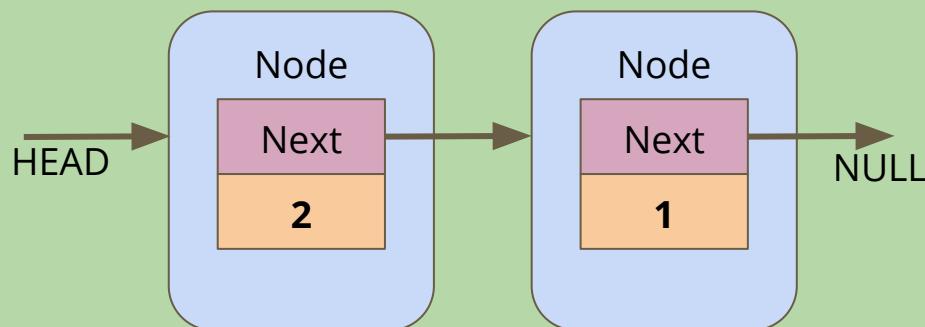
CreateNode makes a node with a NULL next and we point head at it



How it works 2

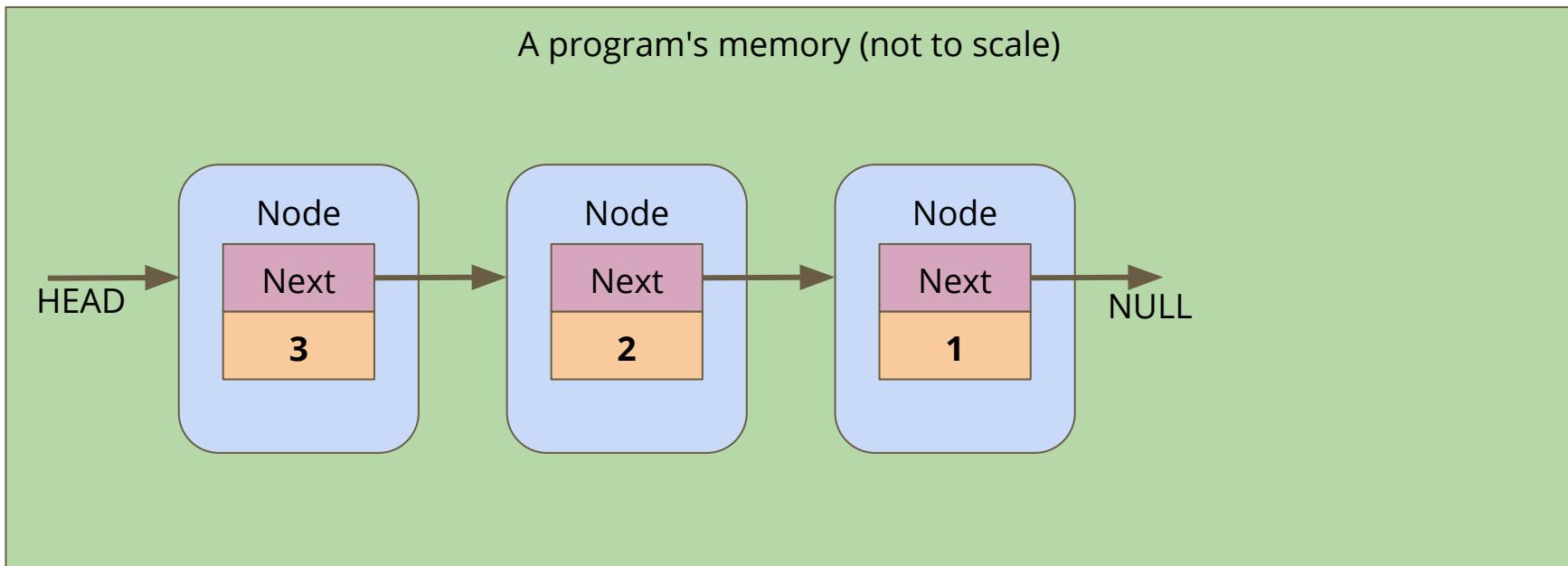
The 2nd node points its "next" at the old head, then it replaces head with its own address

A program's memory (not to scale)



How it works 3

The process continues . . .



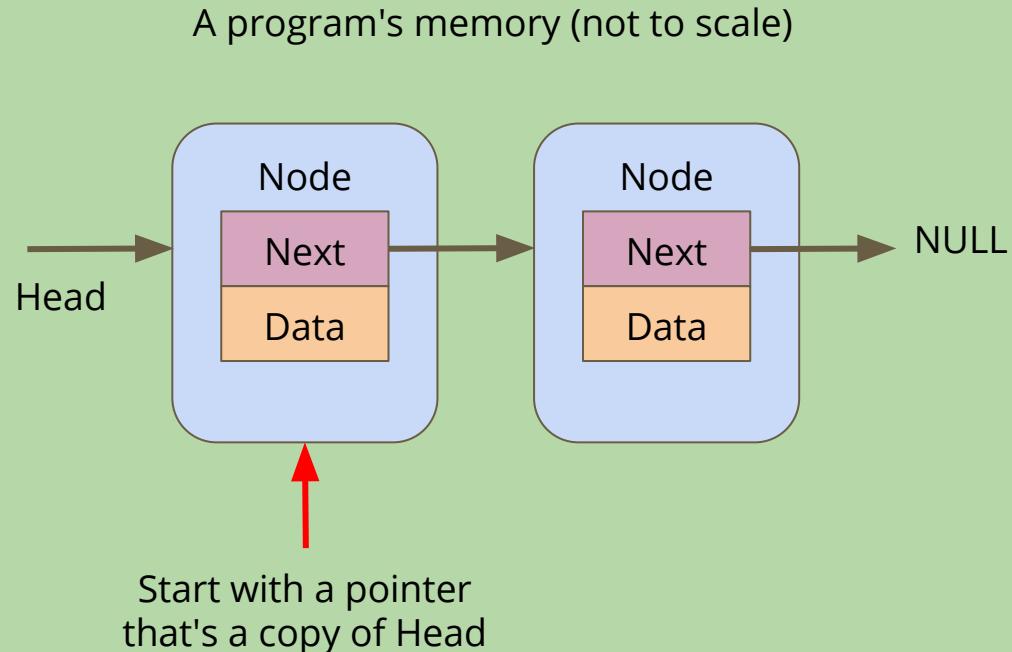
Looping through a Linked List

Linked lists don't have indexes . . .

- We can't loop through them in the same way as arrays
- We have to follow the links from node to node
- If we reach a NULL node pointer, it means we're at the end of the list

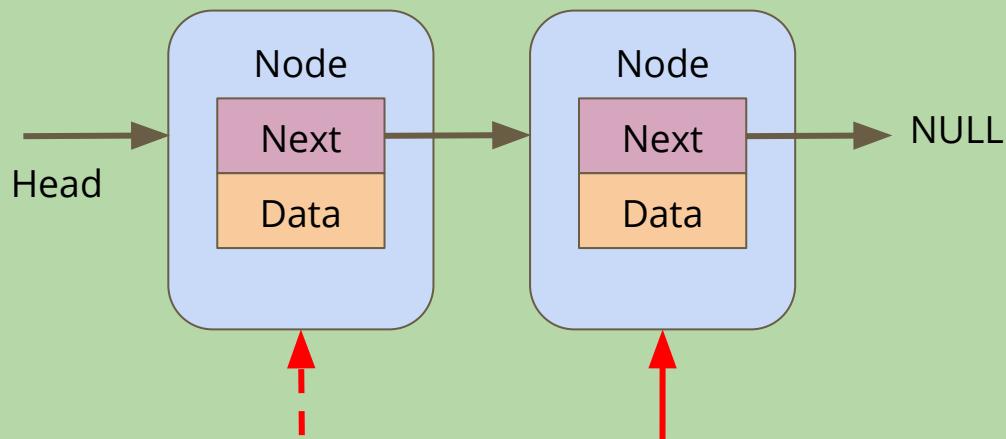
```
// Loop through a list of nodes, printing out their data
void printData(struct node *n) {
    while (n != NULL) {
        printf("%d\n", n->data);
        n = n->next;
    }
}
```

Looping through a Linked List



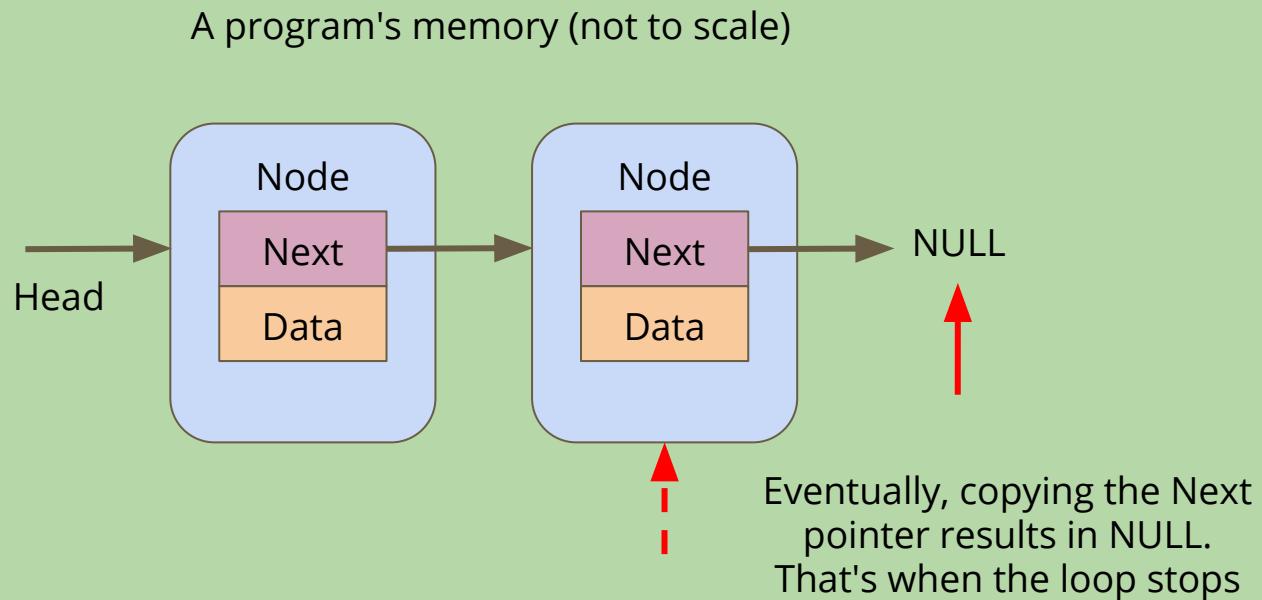
Looping through a Linked List

A program's memory (not to scale)



After you're finished with a node, copy its
Next pointer to reach the next node

Looping through a Linked List



Battle Royale

Let's use a Linked List to track the players in a game

- We're going to start by adding players to the game
- We want to be able to print all the players that are currently in the game (the list of players can change as the game goes on)
- We might want to control the order of the list, so we need to be able to insert at a particular position
- We also want to be able to find and remove players from the list if they're knocked out of the round

What will our nodes look like?

We're definitely going to want a basic node struct

- Let's start with a name
- And a pointer to the next node

```
struct node {
    char name[MAX_NAME_LENGTH];
    struct node *next;
};
```

Creating nodes

We'll want a function that creates a node

```
// Create a node using the name and next pointer provided
// Return a pointer to this node
struct node *createNode(char newName[], struct node *newNext) {
    struct node *n;
    n = malloc(sizeof(struct node));
    strcpy(n->name, newName);
    n->next = newNext;
    return n;
}
```

Creating the list itself

Note that we don't need to specify the length of the list!

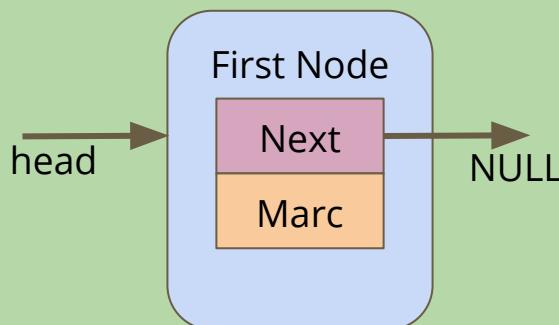
```
int main(void) {
    // create the list of players
    struct node *head = createNode("Marc", NULL);
    head = createNode("Chicken", head);
    head = createNode("Aang", head);
    head = createNode("Katara", head);

    return 0;
}
```

Using `createNode`

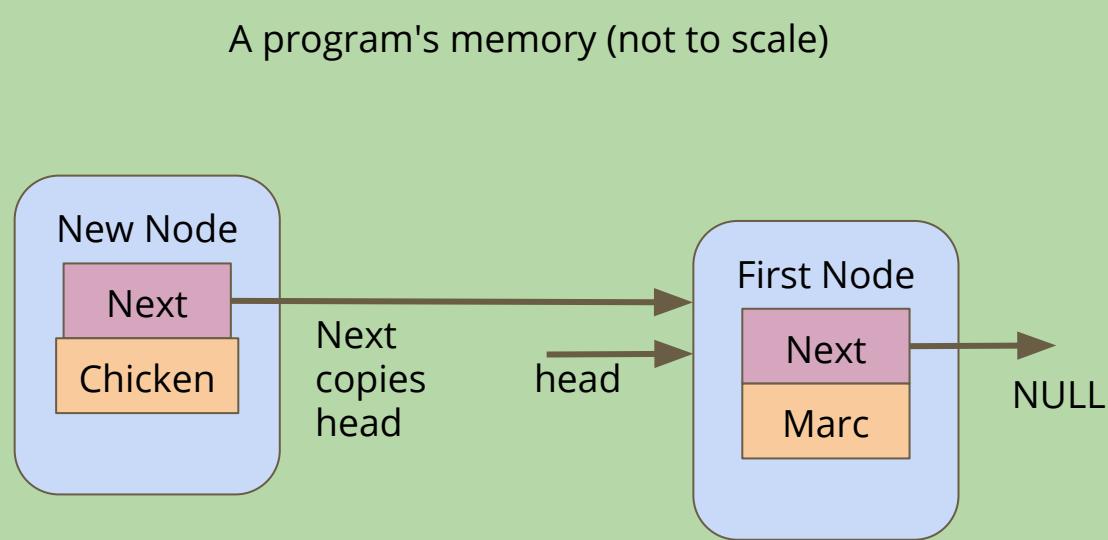
Head points at the First Node, its next is NULL

A program's memory (not to scale)



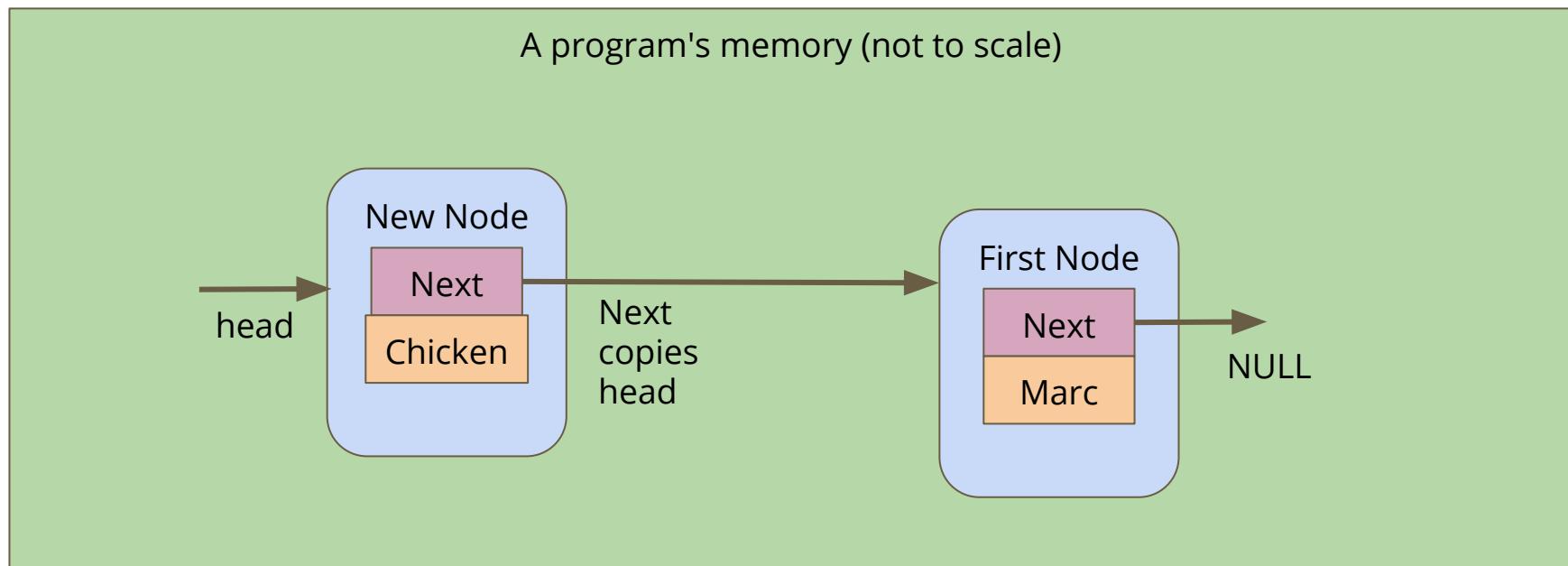
Using `createNode`

The New Node is created and copies the head pointer for its next



Using `createNode`

`createNode` returns a pointer to New Node, which is assigned to head



Printing out the list of players

How do we traverse a list to see all the elements in it?

- Loop through, starting with the pointer to the head of the list
- Use whatever data is inside the node
- Then move onto the next pointer from that node
- If the pointer is NULL, then we've reached the end of the list

```
// Loop through the list and print out the player names
void printPlayers(struct node* listNode) {
    while (listNode != NULL) {
        printf("%s\n", listNode->name);
        listNode = listNode->next;
    }
}
```

To be continued

It's a big project . . . we'll continue it later!

- We might want to insert at a different place in the list
- We still want to insert for a reason (thinking about keeping lists sorted)
- We haven't yet looked at removal from a list
- Once we have all the functionality we need, we'll actually run the game

What did we learn today?

Multi-File Projects

- Spreading out our code functionality into more than one file

Linked Lists

- A new struct that can point at its own type
- Chaining nodes together forms a list
- Nodes can have a variety of information in them
- Code for creation of nodes and lists
- Looping through the lists

COMP1511 - Programming Fundamentals

— Week 8 - Lecture 13 —

What did we learn last week?

Structs and Memory

- Our own custom variable types made up of other variables
- Allocating memory for use beyond the scope of functions

Multiple File Projects

- Separating code into different files

Linked Lists

- structs, pointers and memory allocation all together!

What are we learning today?

Linked Lists

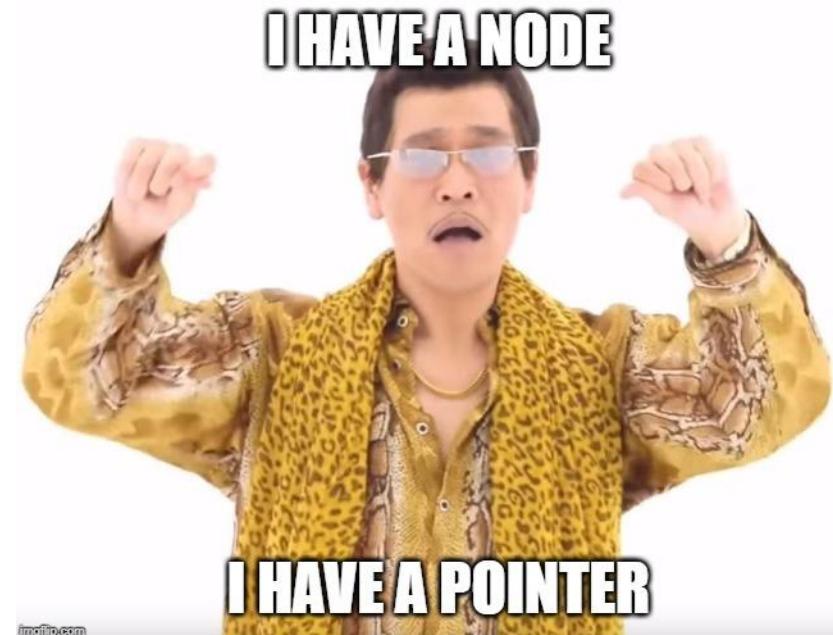
- Continuing our work from last week
- Continuing our example of a Linked List project
- Adding to Linked Lists
- Searching through a list for specific conditions

Recap - Linked Lists

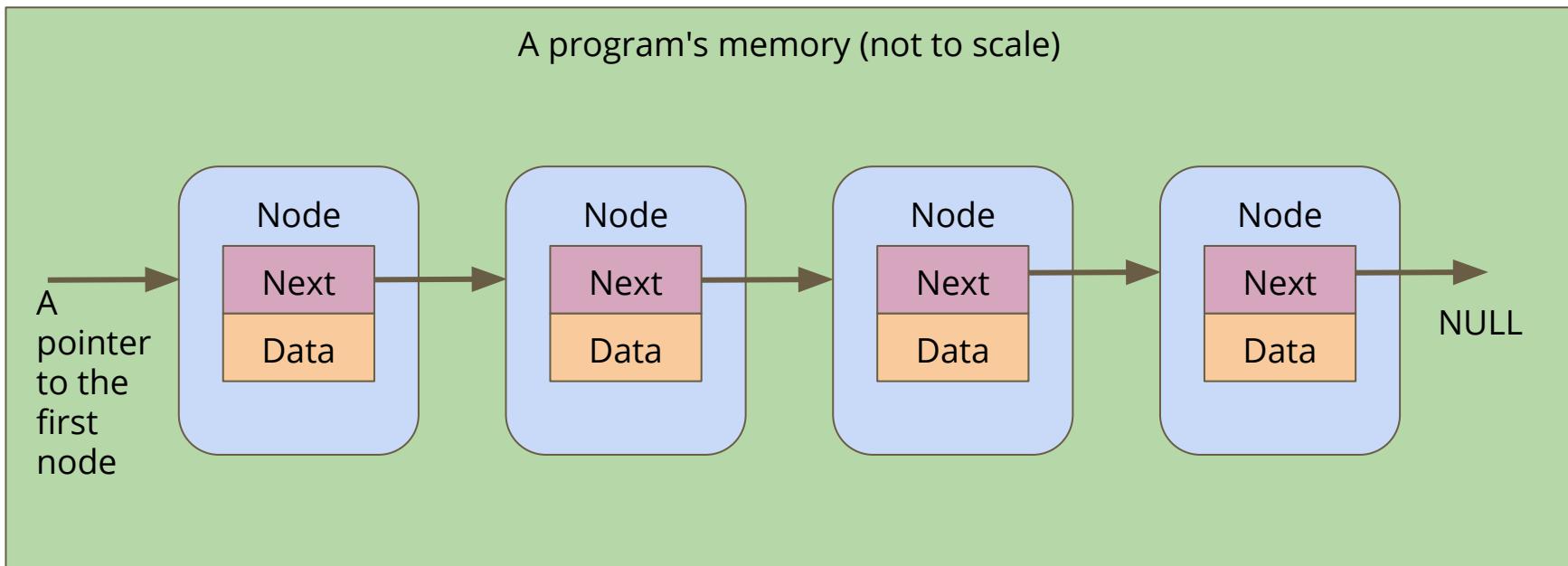
A chain of identical structs to hold information

- Pointers to the same type of struct so they can be chained together
- Some kind of information stored in the struct

```
struct node {  
    struct node *next;  
    int data;  
};
```



A Linked List



Looping through a Linked List

Loop by using the next pointer

- We can jump to the next node by following the current node's next pointer
- We know we're at the end if the next pointer is NULL

```
// Loop through a list of nodes, printing out their data
void printData(struct node *n) {
    while (n != NULL) {
        printf("%d\n", n->data);
        n = n->next;
    }
}
```

Battle Royale - Continued

What do we have so far?

- We've defined basic player structs (linked list nodes)
- We have a function to allocate and create them
- A way of building a list that just uses the create function
- A function to loop through a list and print out the names

Player nodes

A basic linked list node struct

```
struct player {  
    char name[MAX_NAME_LENGTH];  
    struct player *next;  
};
```

Creating players

A function that creates a node

```
// Create a player node using the name and next pointer provided
// Return a pointer to this node
struct player *createPlayer(char *newName, struct player *newNext) {
    struct player *newPlayer = malloc(sizeof (struct player));
    strcpy(newPlayer->name, newName);
    newPlayer->next = newNext;
    return newPlayer;
}
```

Creating the list

This is a simple way of doing this

```
int main(void) {
    // create the list of players
    struct node *head = NULL;
    head = createPlayer("Marc", head);
    head = createPlayer("Chicken", head);
    head = createPlayer("Aang", head);
    head = createPlayer("Katara", head);

    return 0;
}
```

This method basically adds a new element to the start of the list each time

Printing out the list of players

Looping through and printing out the name of each player

- Starting with the pointer to the head of the list
- Use whatever data is inside the player node
- Then move the curr pointer to the next node
- If the curr pointer is NULL, then we've reached the end of the list

```
// Loop through the list and print out the player names
void printPlayers(struct player *playerList) {
    struct player *curr = playerList;
    while (curr != NULL) {
        printf("%s\n", curr->name);
        curr = curr->next;
    }
}
```

Battle Royale - What's next?

What else does the program need?

- Add players to the game
 - Inserting into a list
- Maintain a list of players that's in order
 - Inserting into a specific position in a list

Inserting Nodes into a Linked List

Linked Lists allow you to insert nodes in between other nodes

- We can do this by simply aiming next pointers to the right places
- We find two linked nodes that we want to put a node between
- We take the **next** of the first node and point it at our new node
- We take the **next** of the new node and point it at the second node

This is much less complicated with diagrams . . .

Our Linked List

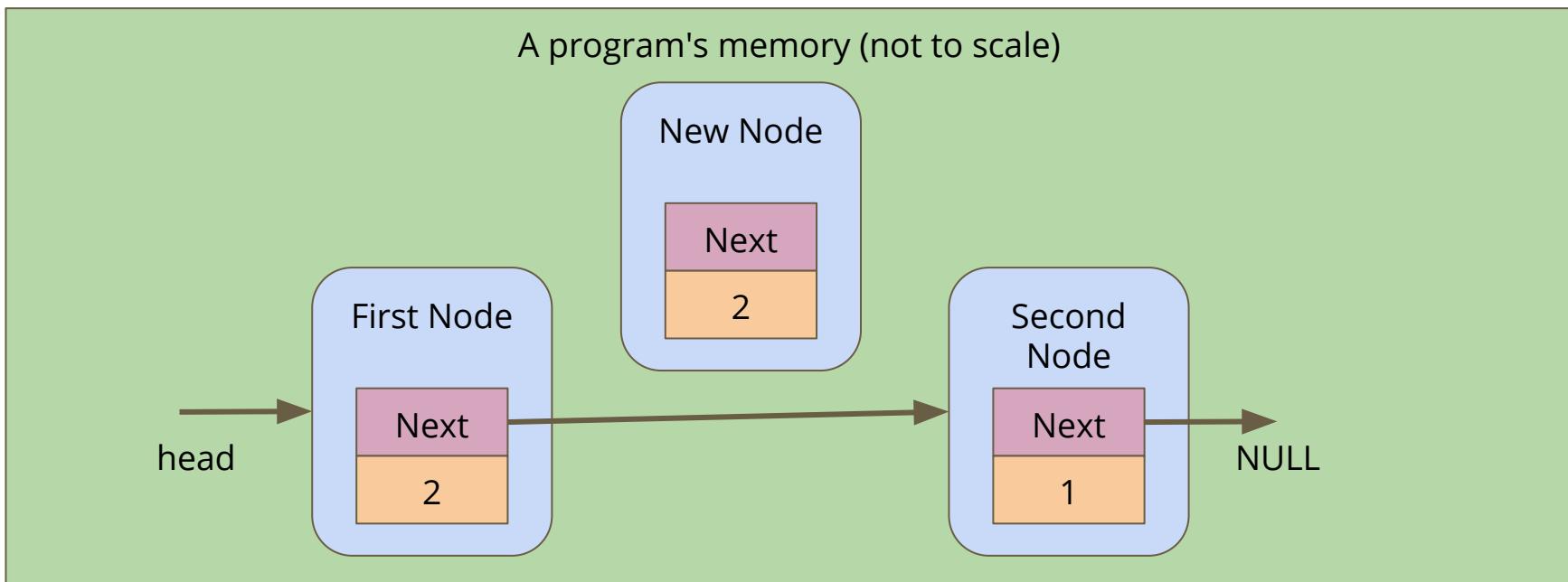
Before we've tried to insert anything

A program's memory (not to scale)



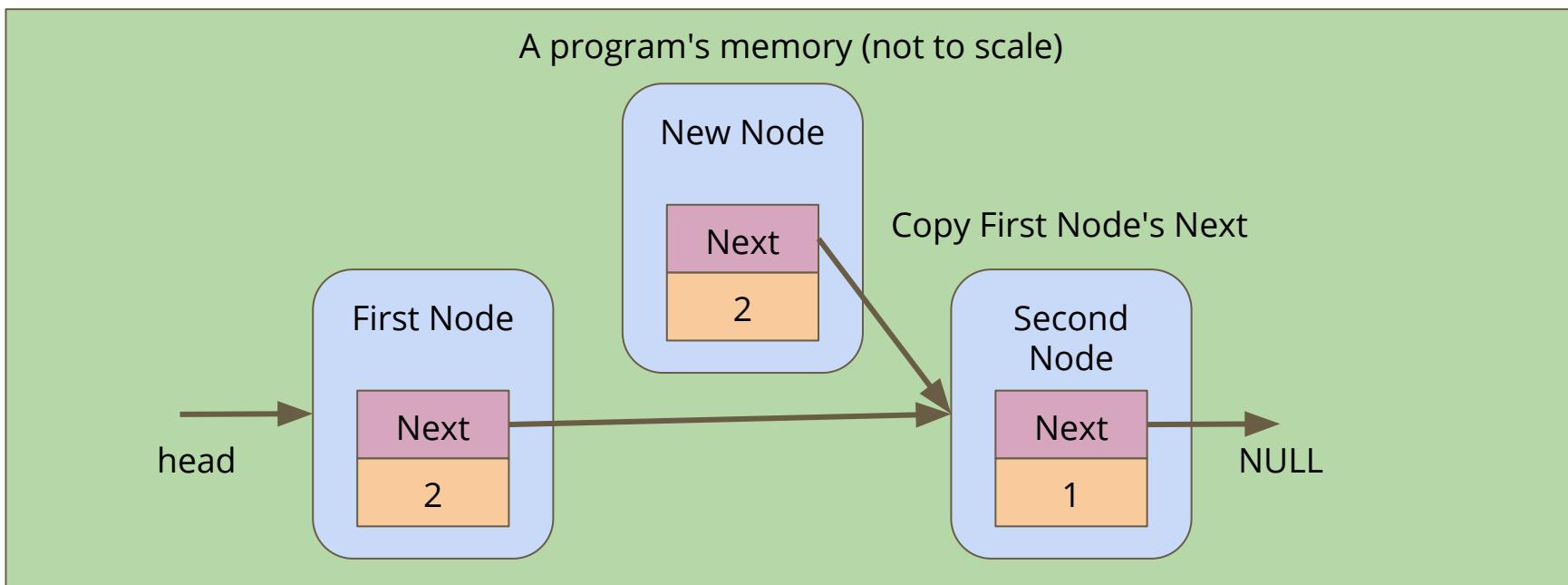
Create a node

A new node is made, it's not connected to anything yet



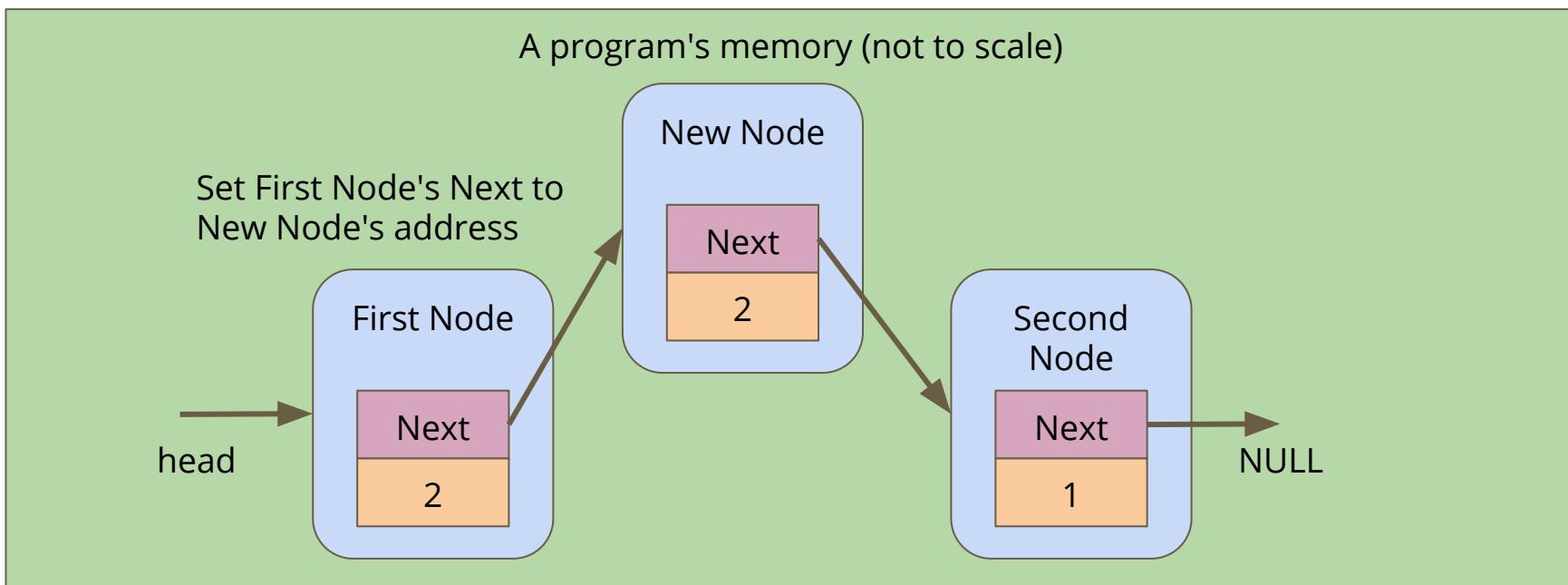
Connect the new node to the second node

Alter the **next** pointer on the New Node



Connect the first node to the new node

Alter the **next** pointer on the First Node



Break Time

Homework - it's not real homework, just things that can inspire you

- *AlphaGo* Documentary (on Netflix)
- *I, Robot* Short Stories (Isaac Asimov)
- *Snow Crash* and *The Cryptonomicon* Novels (Neal Stephenson)
- *Human Resource Machine* Game (on Steam, iOS and Android)
- *Space Alert* Board Game (Vlaada Chvátil)

Code for insertion of players

```
// Create and insert a new node into a list after a given insert position
struct player *insertAfter(struct player* insertPos, char newName[]) {
    struct player *p = createPlayer(newName, NULL);
    if (insertPos == NULL) {
        // List is empty, p becomes the only element in the list
        insertPos = p;
        p->next = NULL;
    } else {
        // Set the new player (p)'s next to after the insertion position
        p->next = insertPos->next;
        // Set the insert position node's next to now aim at p
        insertPos->next = p;
    }
    return insertPos;
}
```

Inserting Players to create a list

We can use insertion to have greater control of where players end up

In this example, Chicken is inserted after the head (Marc), then Aang is also inserted after Marc (and before Chicken)

```
int main(void) {  
    // create the list of players  
    struct node *head = createPlayer("Marc", NULL);  
    insertAfter("Chicken", head);  
    insertAfter("Aang", head);  
  
    printPlayers(head);  
  
    return 0;  
}
```

Insertion with some conditions

We can now insert into any position in a Linked List

- We can read the data in a node and decide whether we want to insert before or after it
- Let's insert our elements into our list based on alphabetical order
- We're going to use a **string.h** function, **strcmp()** for this
- **strcmp()** compares two strings, and returns
 - 0 if they're equal
 - negative if the first has a lower ascii value than the second
 - positive if the first has a higher ascii value than the second

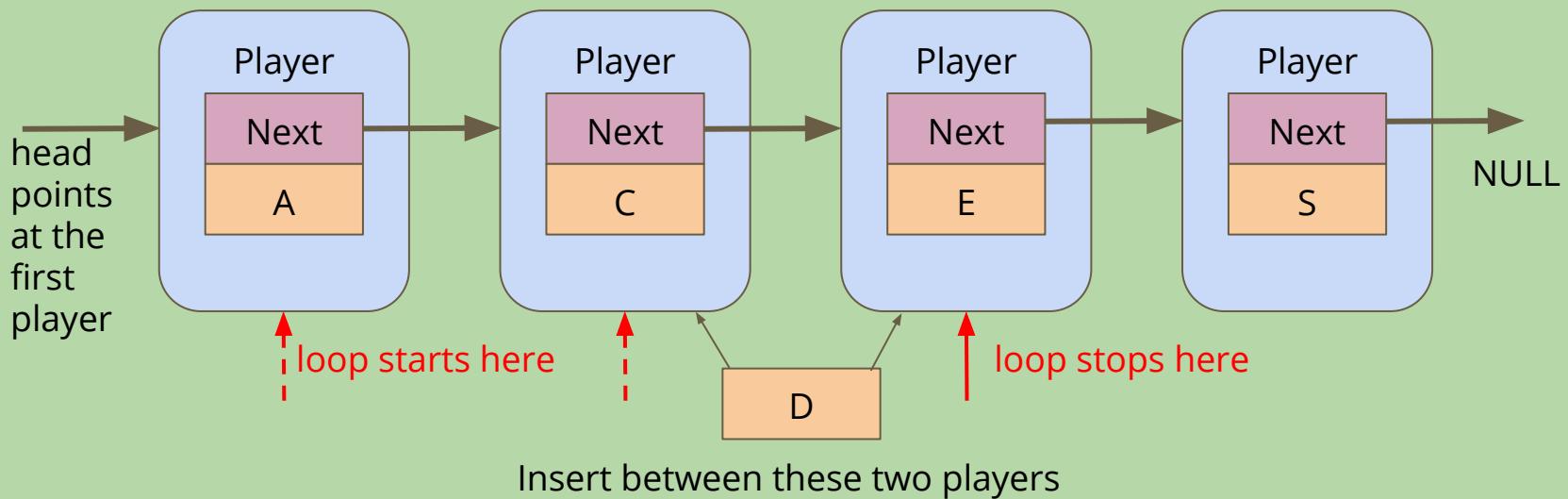
Finding where to insert

We're going to loop through the list

- This loop assumes the list is already in alphabetical order
- Each time we loop, we're going to keep track of the previous player
- We'll test the name of each player using `strcmp()`
- We stop looping once we find the first name that's "higher" than ours
- Then we insert before that player

Finding the insertion point

Attempting to insert a player with name: "D" into a sorted list while maintaining the alphabetical order



Inserting into a list Alphabetically

```
struct player *insertAlphabetical(char newName[], struct player* head) {  
    struct player *previous = NULL;  
    struct player *p = head;  
    // Loop through the list and find the right place for the new name  
    while (p != NULL && strcmp(newName, p->name) > 0) {  
        previous = p;  
        p = p->next;  
    }  
    struct player *insertionPoint = insertAfter(newName, previous);  
    // Return the head of the list (even if it has changed)  
    if (previous == NULL) { // we inserted at the start of the list  
        insertionPoint->next = p;  
        return insertionPoint;  
    } else {  
        return head;  
    }  
}
```

What did we learn today?

Linked Lists

- Recap of Linked Lists
- Building the list
- Looping through the list
- Inserting nodes at a specific location
- Inserting nodes into an ordered list

COMP1511 - Programming Fundamentals

— Week 8 - Lecture 14 —

What did we learn last lecture?

Linked Lists

- Linked List Recap
- Insertion into Lists
- Looping through Linked Lists to find specific conditions

What are we doing today?

More Linked Lists

- Linked List Removal
- Freeing our Allocated Memory
- Playing the Battle Royale game

Insertion with some conditions - recap

We can now insert into any position in a Linked List

- We can read the data in a node and decide whether we want to insert before or after it
- Let's insert our elements into our list based on alphabetical order
- We're going to use a **string.h** function, **strcmp()** for this
- **strcmp()** compares two strings, and returns
 - 0 if they're equal
 - negative if the first has a lower ascii value than the second
 - positive if the first has a higher ascii value than the second

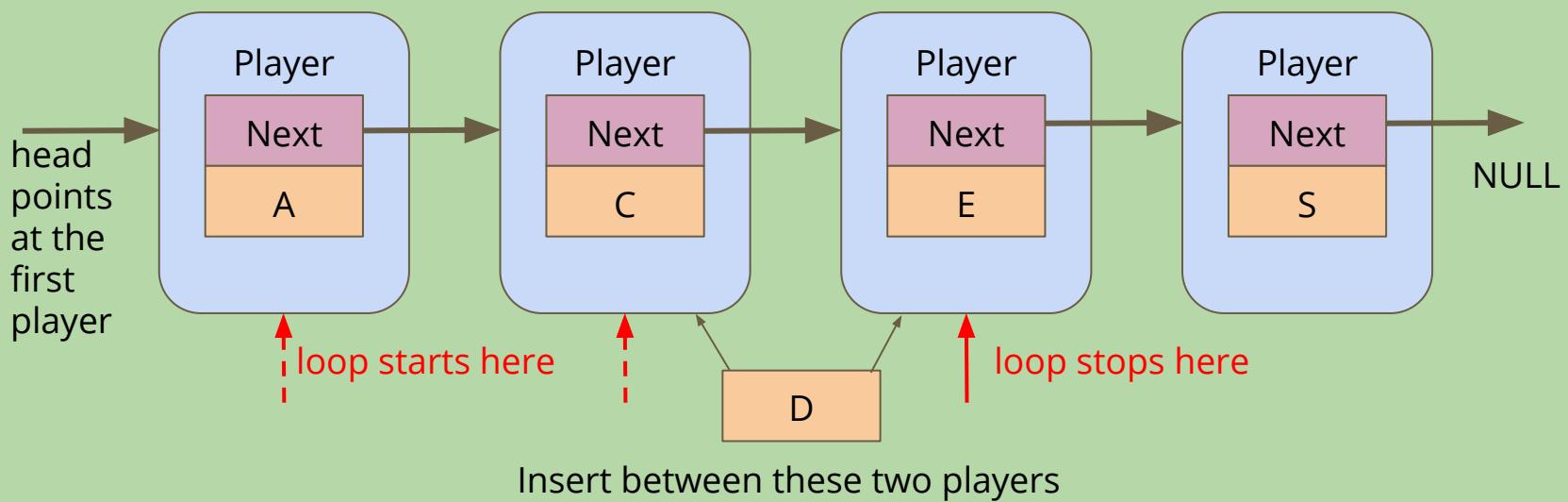
Finding where to insert

We're going to loop through the list

- This loop assumes the list is already in alphabetical order
- Each time we loop, we're going to keep track of the previous player
- We'll test the name of each player using `strcmp()`
- We stop looping once we find the first name that's "higher" than ours
- Then we insert before that player

Finding the insertion point

Attempting to insert a player with name: "D" into a sorted list while maintaining the alphabetical order



Inserting into a list Alphabetically

```
struct player *insertAlphabetical(char newName[], struct player* head) {  
    struct player *previous = NULL;  
    struct player *p = head;  
    // Loop through the list and find the right place for the new name  
    while (p != NULL && strcmp(newName, p->name) > 0) {  
        previous = p;  
        p = p->next;  
    }  
    struct player *insertionPoint = insert(newName, previous);  
    // Return the head of the list (even if it has changed)  
    if (previous == NULL) { // we inserted at the start of the list  
        insertionPoint->next = p;  
        return insertionPoint;  
    } else {  
        return head;  
    }  
}
```

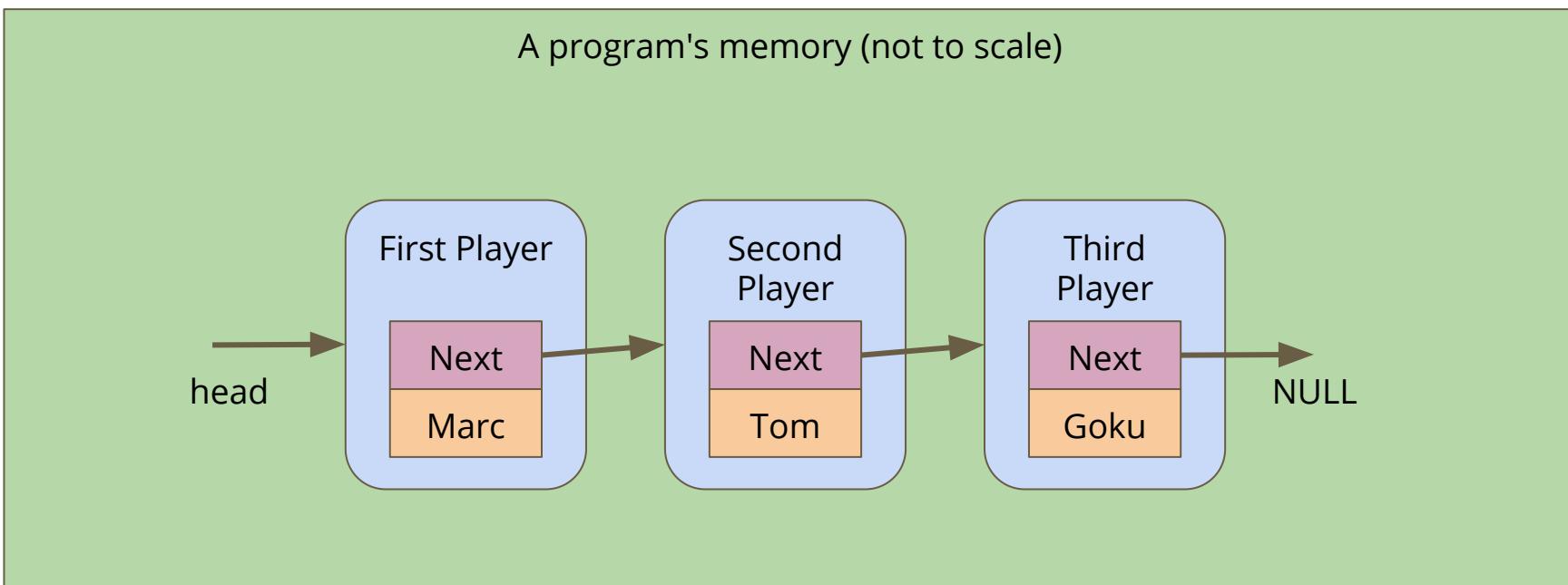
Removing a player

If we want to remove a specific player, given their name

- We need to look through the list and see if a player name matches the one we want to remove
- To remove, we'll use **next** pointers to connect the list around the player node
- Then, we'll free the node itself that we don't need anymore

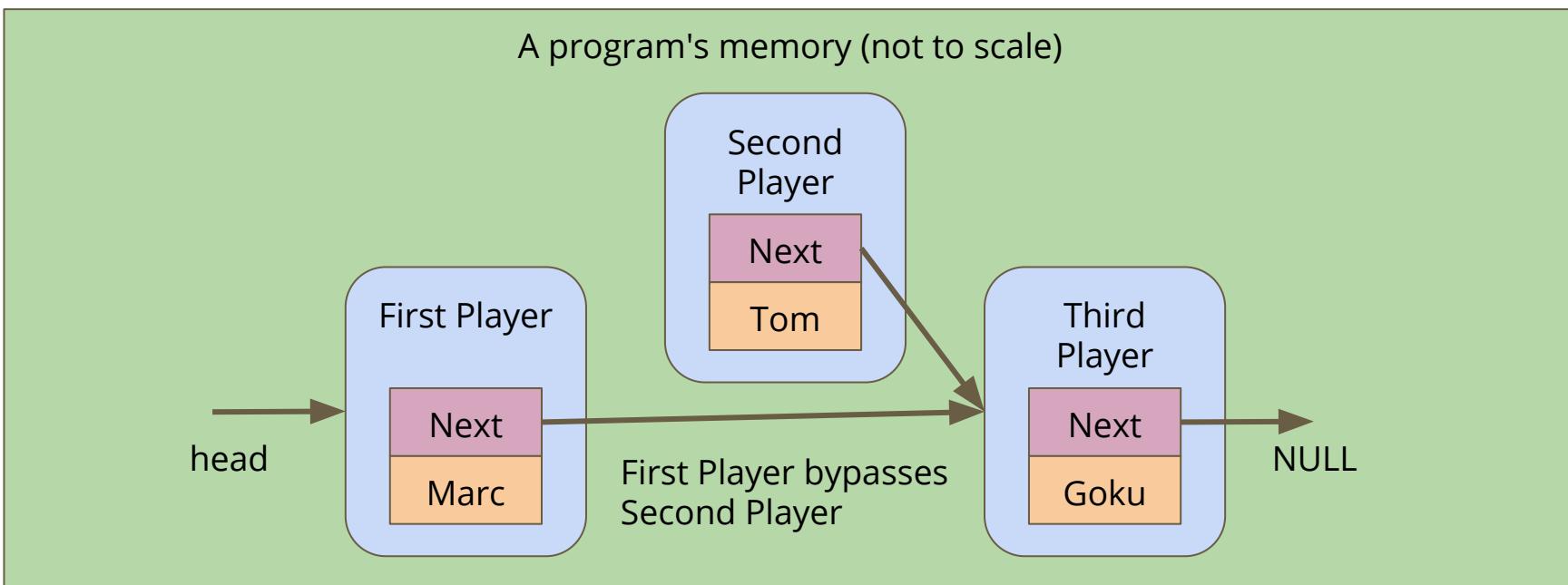
Removing a player node

If we want to remove the Second Player



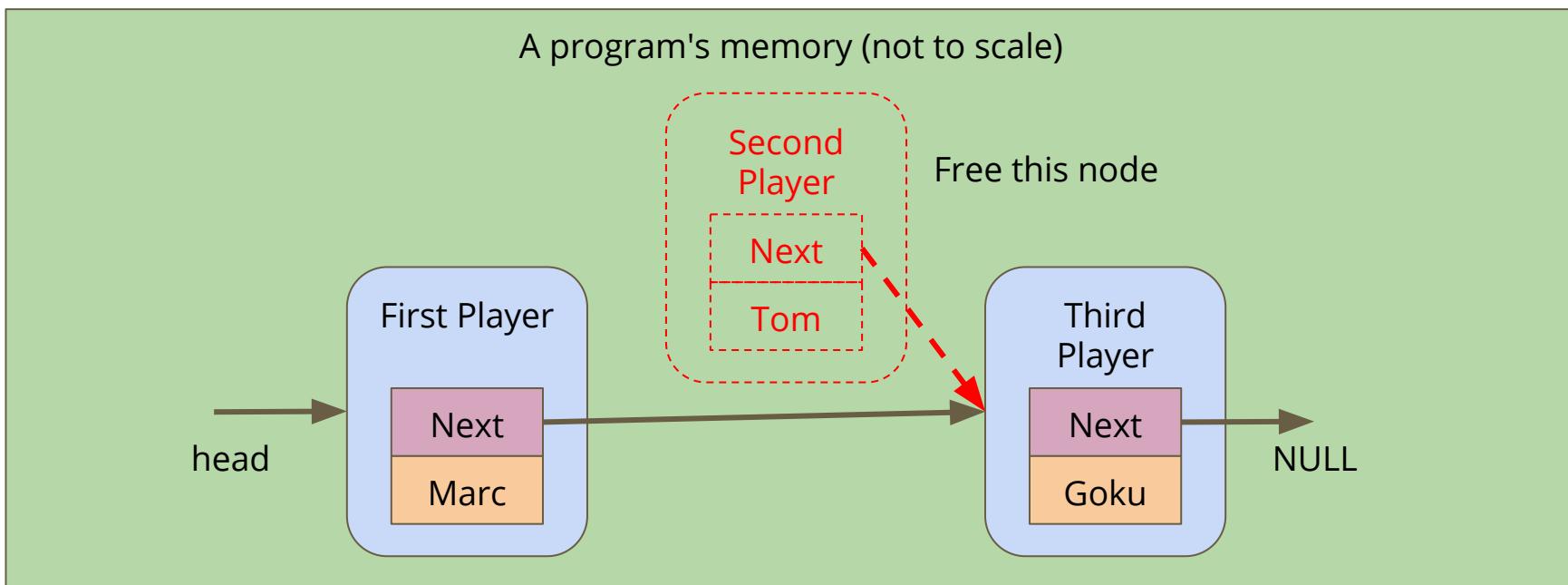
Skipping the player node

Alter the First Player's **next** to bypass the player node we're removing



Freeing the removed node

Free the memory from the now bypassed player node



Break Time

Keeping track of your own code projects

- Using **git** is a really handy way to keep backups of your work
- GitHub and BitBucket are two providers that will give you free online repositories to store your code
- Graphical Interfaces are available for git (GitHub Desktop and Sourcetree respectively)
- It takes some time to get familiar with how these work . . . but you can start practicing now!



Finding the right player

Loop until you find the right match

This is very similar to finding the insertion point earlier

```
struct player *removePlayer(char *remName, struct player *head) {
    struct player *previous = NULL;
    struct player *current = head;
    // Keep looping until we find the matching name
    while (current != NULL && strcmp(name, current->name) != 0) {
        previous = current;
        current = current->next;
    }
    if (current != NULL) {
        // if we didn't reach the end of the list,
        // we found the right player
```

Removing a player

Having found the player node, remove it from the list

```
if (current != NULL) {
    // if we didn't reach the end of the list,
    // we found the right player
    if (previous == NULL) {
        // it's the first player
        head = current->next;
    } else {
        previous->next = current->next;
    }
    free(current);
}
return head;
```

The Battle Royale

In a Battle Royale, people are removed from the game one at a time until only one person is left. They are the winner

- We can create a list of players
- We can make sure it's in a nice alphabetical order
- We can remove a single player from the list
- Now we need to remove players one at a time
- When there's only one left, they are the winner!

Game code

Once our list is created, we can loop through the game

- We print out the player list (we might want to modify that function!)
- Our user will tell us who was knocked out

```
// A game loop that runs until only one player is left
while (printPlayers(head) > 1) {
    printf("Who just got knocked out?\n");
    char koName[MAX_NAME_LENGTH];
    fgets(koName, MAX_NAME_LENGTH, stdin);
    koName[strlen(koName) - 1] = '\0';
    head = removePlayer(koName, head);
    printf("-----\n");
}
printf("The winner is: %s\n", head->name);
```

Cleaning Up

Remember, All memory allocated (malloc) needs to be freed

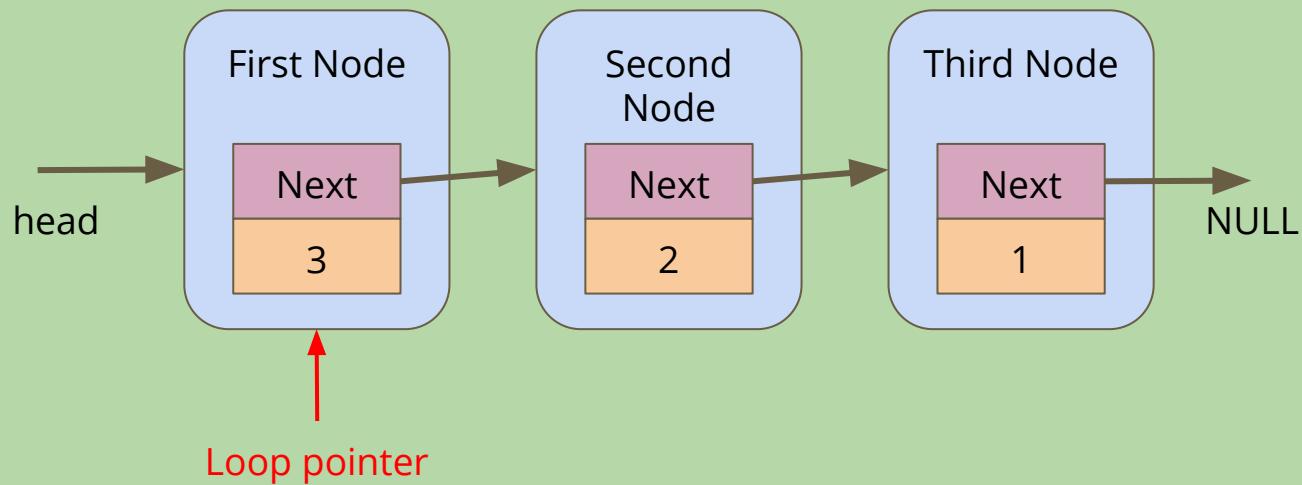
- We can run **gcc --leak-check** to see whether there's leaking memory
- What do we find?
- There are pieces of memory we've allocated that we're not freeing!

Let's write a function that frees a whole linked list

- Loop through the list, freeing the nodes
- Just be careful not to free one that we still need the pointer from!

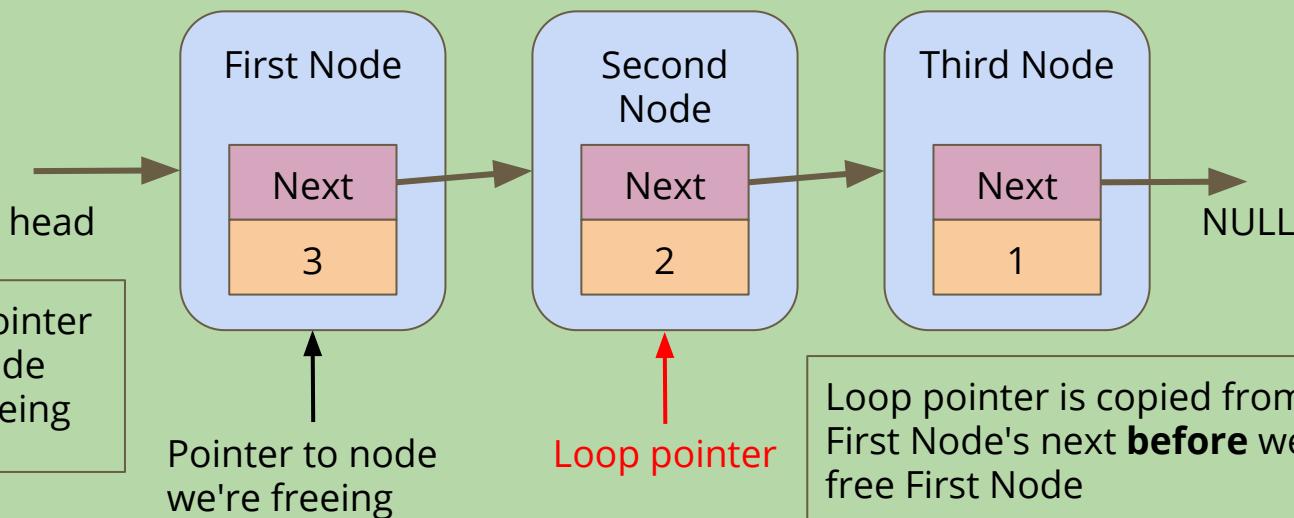
Looping to free nodes

A program's memory (not to scale)

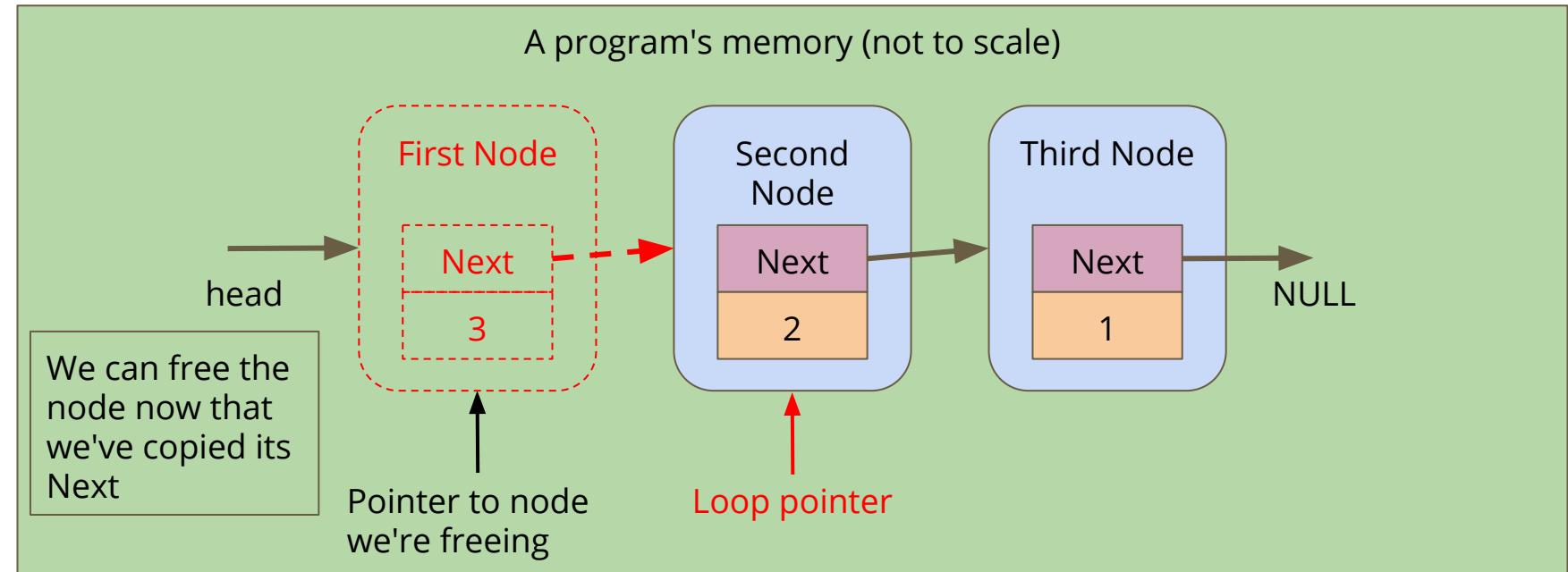


Looping to free nodes

A program's memory (not to scale)



Looping to free nodes



Code to free a linked list

```
// Loop through a list and free all the allocated memory
void freeList(struct node *n) {
    while(n != NULL) {
        // keep track of the current node
        struct node *remNode = n;

        // move the looping pointer to the next node
        n = n->next;

        // free the current node
        free(remNode);
    }
}
```

Battle Royale, the Linked Lists demo

What have we written in this program?

- Creation of nodes
- Looping through a list
- Insertion of nodes into specific locations
- Finding locations using loops
- Removal of nodes
- Managing memory

A Challenge - randomisation

Can we remove a random player from the list?

- Look up the functions `rand()` and `srand()` in the C Standard Library
- We can generate a random number and loop that many times into the list
- Then remove that player
- We will probably want to track how many items are in the list also . . .

What did we cover today?

Linked Lists

- Removal from a list
- Finding and removing a specific node
- Memory cleaning

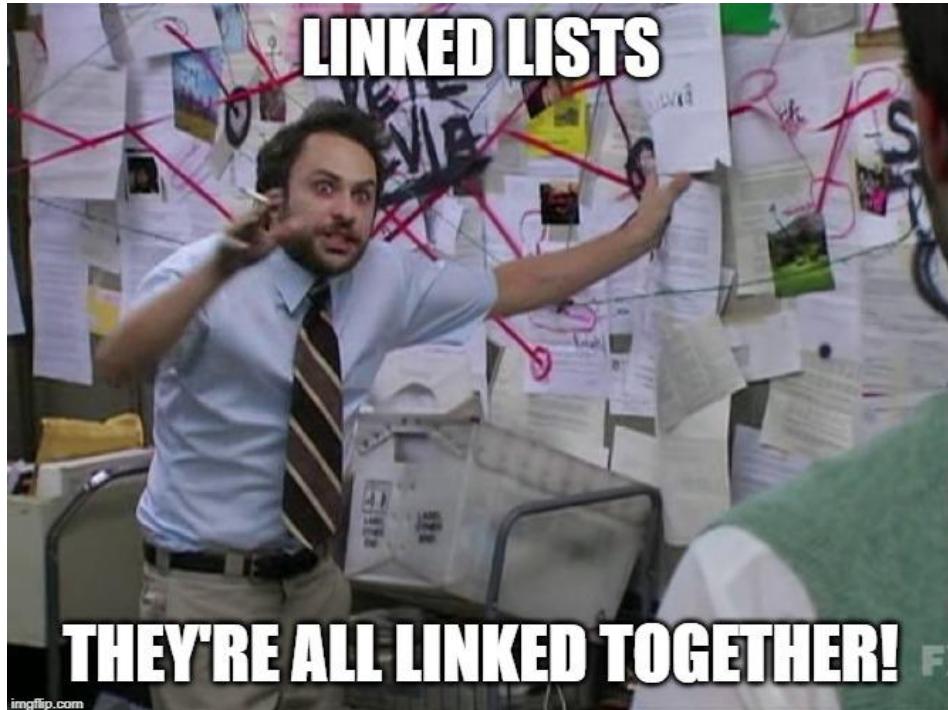
COMP1511 - Programming Fundamentals

— Week 9 - Lecture 15 —

What did we learn last week?

Linked Lists

- A complete working implementation of Linked Lists
- Inserting nodes
- Removal of nodes
- Cleaning our memory



What are we covering today?

Abstract Data Types

- A recap of Multiple File Projects
- More detail on things like `typedef`
- The ability to present capabilities of a type to us . . .
- . . . without exposing any of the inner workings

Recap - Multiple File Projects

Separating Code into Multiple files

- Header file (* .h) - Function Declarations
- Implementation file (* .c) - Majority of the running code
- Other files - can include a Header to use its capabilities

Separation protects data and makes functionality easier to read

- We don't have access to internal information we don't need
- We can't accidentally change something important
- We have a simple list of functions we can call

Using Multiple Files

Linking the Files

- A file that `#includes` the Header (* .h) file will have access to its functions
- It's own implementation (* .c) file will always `#include` it
- Implementation files are never included!

Compilation

- All Implementation files are compiled
- Header files are never compiled, they're included

An Example - CSpotify

Assignment 2 - CSpotify is a nice example

`cspotify.h`

- Contains only defines, typedefs and function declarations
- Is commented heavily so that it's easy to know how to use it

`cspotify.c`

- Contains actual structs
- Contains implementation of `cspotify.h`'s functions (once we've written them)

An Example - CSpotify

How some of the other files interact . . .

`main.c`

- `#includes cspotify.h`
- Uses the functions in `cspotify.h`

`test_cspotify.c` with `test_main.c`

- `#includes cspotify.h`
- Is mutually exclusive with `main.c` because they both have main functions

Abstract Data Types

Types we can declare for a specific purpose

- We can name them
- We can fix particular ways of interacting with them
- This can protect data from being accessed the wrong way

We can hide the implementation

- Whoever uses our code doesn't need to see how it was made
- They only need to know how to use it

Typedef

Type Definition

- We declare a new Type that we're going to use
- **typedef <original Type> <new Type Name>**
- Allows us to use a simple name for a possibly complex structure
- More importantly, hides the structure details from other parts of the code

```
typedef struct library *Library;
```

- We can use **Library** as a Type without knowing anything about the struct underlying it

Typedef in a Header file

The Header file provides an interface to the functionality

- We can put this in a **header** (*.h) file along with functions that use it
- This allows someone to see a Type without knowing exactly what it is
- The details go in the *.c file which is not included directly
- We can also see the functions without knowing how they work
- We are able to see the **header** and use the information
- We hide the **implementation** that we don't need to know about

An Example of an Abstract Data Type - A Stack

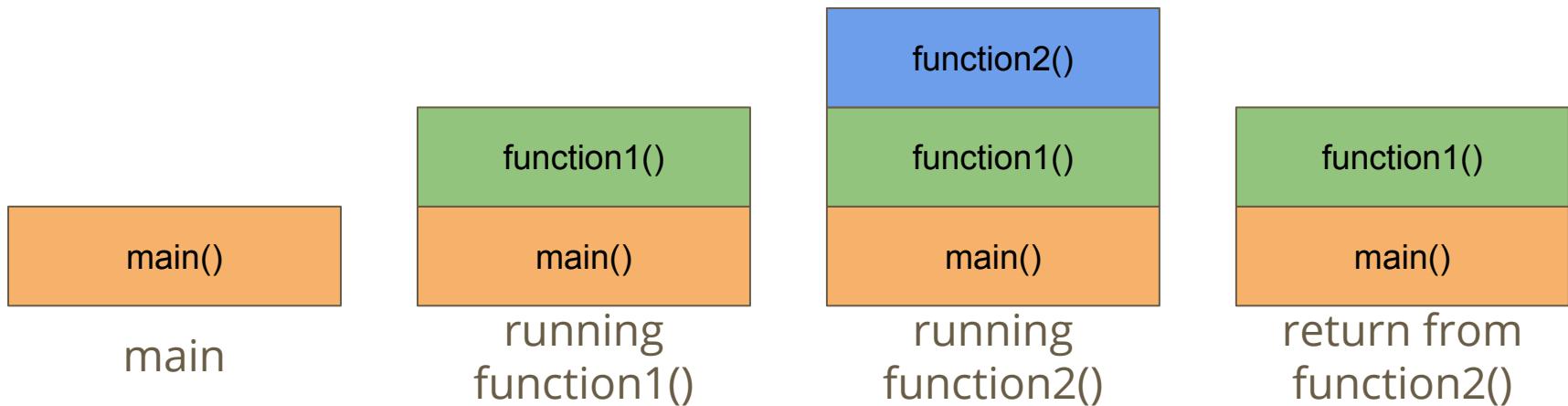
A stack is a very common data structure in programming

- It is a "Last in first out" structure
- You can put something on top of a stack
- You can take something off the top of a stack
- You can't access anything underneath

This is actually how functions work!

The currently running code is on the top of the stack

- main() calls function1() - only function1() is accessible
- function1() calls function2() - only function2() is accessible
- control returns to function1() when function2() returns



What makes it Abstract?

A Stack is an idea

- An Array or a Linked List is a very specific implementation
- A Stack is just an idea of how things should be organised
- There's a structure, but there's no implementation!

Abstract Data Type for a Stack

- We can have a header saying how the Stack is used
- The Implementation could use an Array or a Linked List to store the objects in the Stack, but we wouldn't know!

Break Time

Programming Languages

- C++, Java, C# and many others are based on C
- There are too many programming languages to count or learn!
- Remember the fundamentals!
- C syntax is not as important as your plans and thinking
- You will encounter many programming languages, some will feel very different from C in their approach
- But if you learn how you want to communicate with computers, the actual language you use will never be a barrier for you

Let's build a Stack ADT

We're only concerned with how we'll use it, not what it's made of

- Our user will see a "Stack" rather than an Array or Linked List
- We will start with a Stack of integers
- We will provide access to certain functions:
 - Create a Stack
 - Destroy a Stack
 - Add to the Stack (known as "push")
 - Remove from the Stack (known as "pop")
 - Count how many things are in the Stack

A Header File for a Stack

```
// stack type hides the struct that it is implemented as
typedef struct stackInternals *Stack;

// functions to create and destroy stacks
stack stackCreate(void);
void stackFree(Stack s);

// Push and Pop items from stacks
// Removing the item returns the item for use
void stackPush(Stack s, int item);
int stackPop(Stack s);

// Check on the size of the stack
int stackSize(Stack s);
```

What does our Header (not) Provide?

Standard Stack functions are available

- We can push or pop an element onto or off the Stack
- We are not given access to anything else inside the Stack!
- We cannot pop more than one element at a time
- We aren't able to loop through the Stack

The power of Abstract Data Types

- They stop us from accessing the data incorrectly!

Stack.c

Our *.c file is the implementation of the functionality

- The C file is like the detail under the "headings" in the header
- Each declaration in the header is like a title of what is implemented
- Let's start with a Linked List as the underlying data structure
- A Linked List makes sense because we can grow it and shrink it easily
- We can also look at how to implement this with arrays . . .

The implementation behind a type definition

We can create a pair of structs

- stackInternals represents the whole Stack
- stackNode is a single element of the list

```
// Stack internals holds a pointer to the start of a linked list
struct stackInternals {
    struct stackNode *head;
};

struct stackNode {
    struct stackNode *next;
    int data;
};
```

Creation of a Stack

If we want our struct to be persistent, we'll allocate memory for it

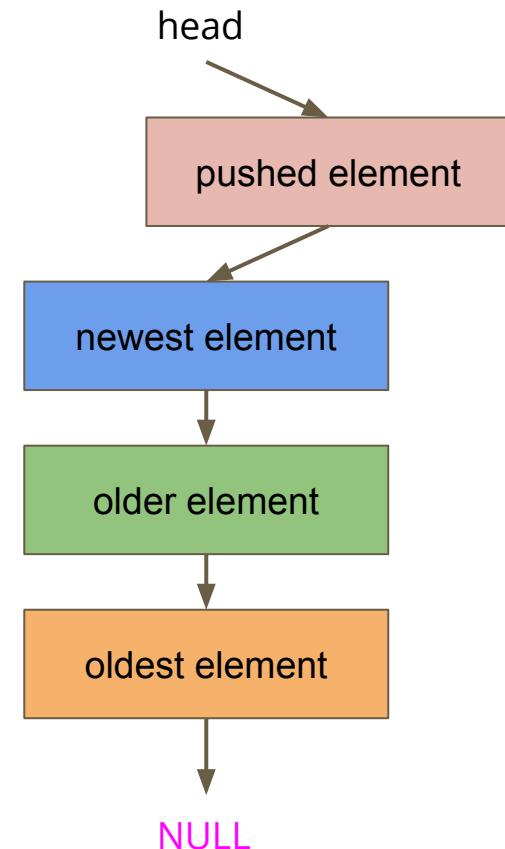
We create our Stack empty, so the pointer to the head is NULL

```
// Create an empty Stack
Stack stackCreate(void) {
    Stack newStack = malloc(sizeof(struct stackInternals));
    newStack->head = NULL;
    return newStack;
}
```

Pushing items onto the Stack

We push items onto the head of the Stack

- We can insert the new element at the head
- All the other elements will stay in the same order they were in



Code for Pushing

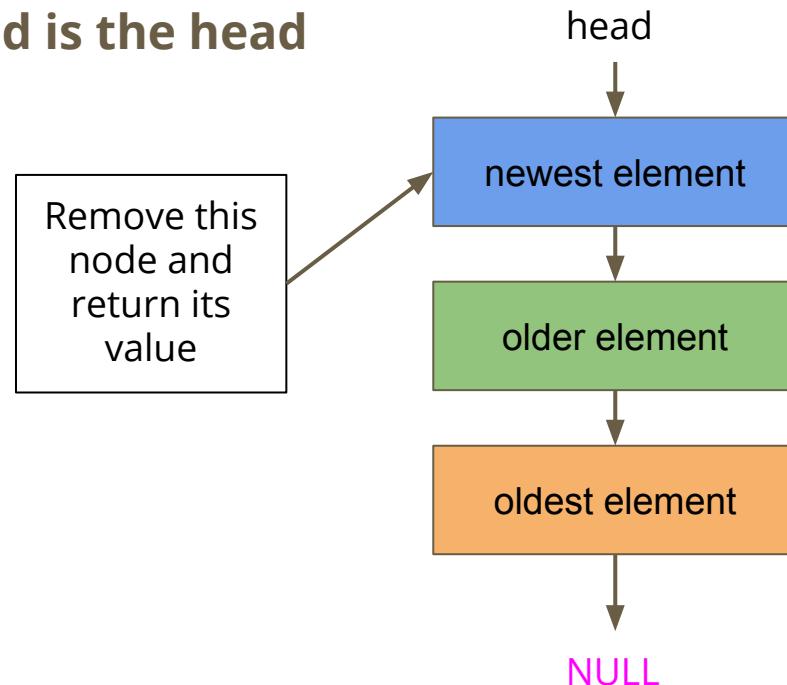
Adding to the head of a linked list is something we've done before

```
void stackPush(Stack s, int item) {
    struct stackNode *newNode = malloc(sizeof(struct stackNode));
    newNode->data = item;

    // Attach newNode to the old head and make it the new head
    newNode->next = s->head;
    s->head = newNode;
}
```

Popping (removing) a Node

The only node that can be popped is the head
(the top of the stack)



Code for Popping

```
// Remove the head from the list and free the memory used
int stackPop(Stack s) {
    if (s->head == NULL) {
        printf("Attempt to pop an element from an empty stack.\n");
        exit(1);
    }
    // Read the value from the head
    int returnData = s->head->data;
    struct stackNode *remNode = q->head;

    // move the stack head to the new head and free the old
    s->head = s->head->next;
    free(remNode);

    return returnData;
}
```

Testing Code in our Main.c

```
int main(void) {
    printf("Creating a deck of cards.\n");
    Stack deck = stackCreate();

    int card = 7;
    printf("Putting %d on top of the deck!\n", card);
    stackPush(deck, card);
    card = 10;
    printf("Putting %d on top of the deck!\n", card);
    stackPush(deck, card);

    printf("Card %d just got removed from the deck!\n", stackPop(deck));

    id = 3;
    printf("Putting %d on top of the deck!\n", card);
    stackPush(deck, card);
}
```

Other Functionality

There are some functions in the header we haven't implemented

- **Destroying and freeing the Stack**
- We're still at risk of leaking memory because we're only freeing on removal
- **Checking the Number of Elements**
- This would be very handy because it would allow us to tell how many elements we can pop before we risk errors
- You could even store an int in the Stack struct that increments every time you push and decrements every time you pop . . .

Different Implementations

Stack.c doesn't have to be a linked list . . . so long as it implements the functions in Stack.h

- We could use an array instead
- Our data can be stored in an array with a large maximum size
- We'll keep track of where the top is with an int

Array Implementation of a stack

A large array where only some of it is used

- Top is a particular index
 - Top signifies where our data ends
 - It also happens to be exactly the number of elements in the stack!



stack.c

```
// Struct representing the stack using an array
struct stackInternals {
    int stack[MAX_STACK_SIZE];
    int top;
};

// create a new stack
stack stackCreate() {
    stack s = malloc(sizeof(struct stackInternals));
    s->top = 0;
    return s;
}
```

Push and Pop

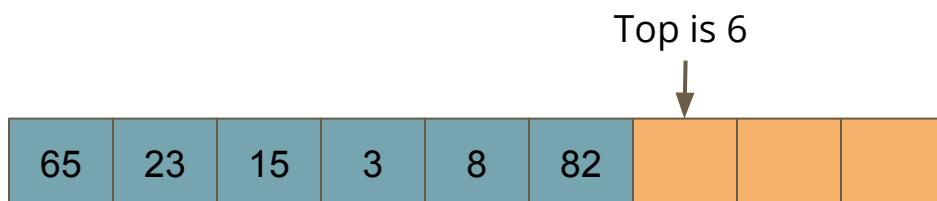
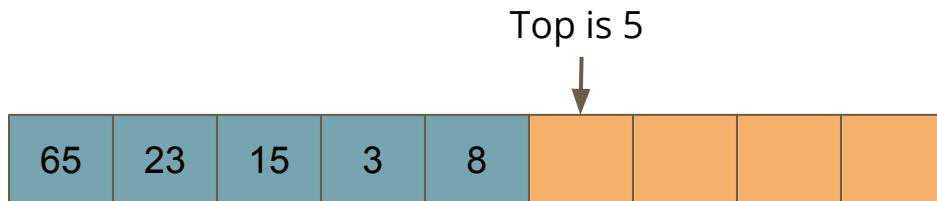
These should only interact with the top of the stack

- **Push** should add an element after the end of the stack
- It should then move the top index to that new element

- **Pop** should return the element on the top of the stack
- It should then move the top index down one

Push

Push a new element "82" onto the stack



The stack starts like this

82 is added at top's index

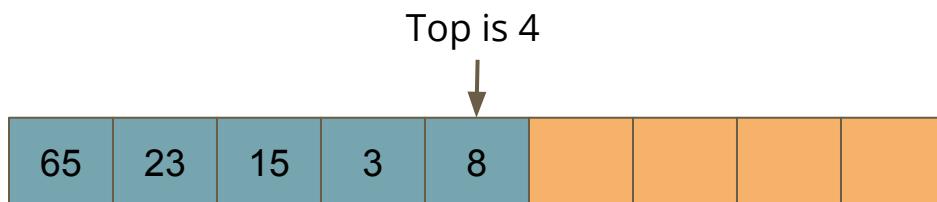
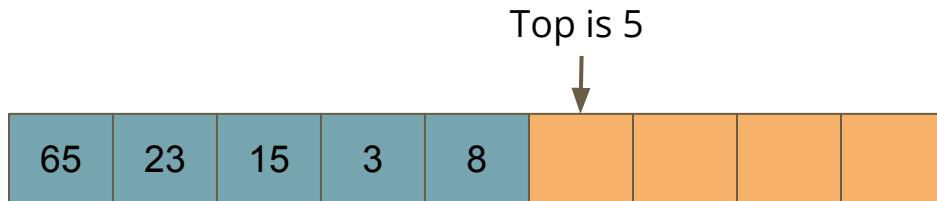
Top then moves up one

Push code

```
// Add an element to the top of the stack
void stackPush(stack s, int item) {
    // check to see if we've used up all our memory
    if(s->top == MAX_STACK_SIZE) {
        printf("Maximum stack size reached, cannot push.\n");
        exit(1);
    }
    s->stackData[s->top] = item;
    s->top++;
}
```

Pop

Pop removes the top element from the stack



The stack starts like this

Top moves down one

Read the element at top and return it

Pop code

```
// Remove an element from the top of the stack
int stackPop(stack s) {
    // check to see if the stack is empty
    if(s->top <= 0) {
        printf("Stack is empty, cannot pop.\n");
        exit(1);
    }
    s->top--;
    return s->stackData[s->top];
}
```

Hidden Implementations

Neither Implementation needs to change the Header

- The main function doesn't know the difference!
- The structures and implementations are hidden from the header file and the rest of the code that uses it
- If we want or need to, we can change the underlying implementation without affecting the main code

Other Abstract Data Types

Stacks are obviously not the only possibility here

- If we simply change the rules (last in, first out), we can make other structures
- A Queue is "first in, first out", and could be created using similar techniques
- There are many possibilities that we can create!

What did we cover today?

Abstract Data Types

- Makes use of Multi-file projects we discussed earlier
- **`typedef`** to protect a struct from open access
- Using multiple files to control how a type is used
- Hiding the implementation
- Providing a fixed interface
- Showing that different implementations can work with the same ADT

COMP1511 - Programming Fundamentals

— Week 9 - Lecture 16 —

What did we learn last lecture?

Abstract Data Types

- An extension/explanation of the use of Multiple File Projects
- Presenting an idea. A structure with a set of rules
- But hiding the implementation
- A Stack as an example

What are we covering today?

Recursion

- An interesting inversion on the order of program execution
- Functions that call themselves
- Using the program call stack to determine the order of operations

Understanding Recursion

Recursion is a little bit backwards

- Now that you understand Recursion, you can use it
- In order to understand Recursion you must already understand Recursion
- It's good that you knew Recursion before we started

We need to think a little bit in reverse here, but let's step through an example first . . .

It's easy if you
already understand it
But we haven't learnt it?



Add up all the numbers in a linked list

Loop through and add them up . . . we can already do this

```
// Loop through a list of nodes, adding their values together
int sumList(struct node *n) {
    int total = 0
    while (n != NULL) {
        total += n->data;
        n = n->next;
    }
    return total;
}
```

What about a different way?

Let's look at what might happen if we have a function that can call itself

A function that says:

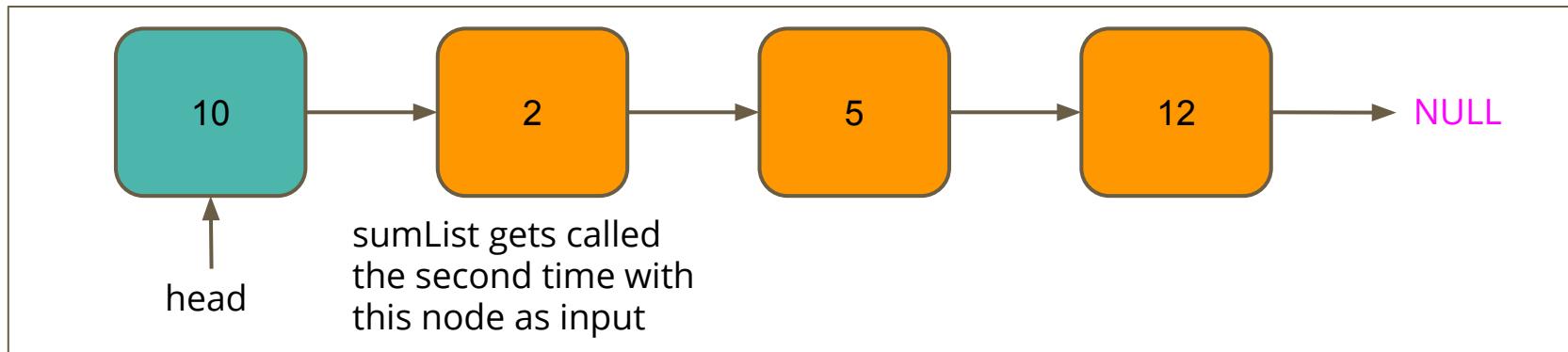
```
// sumList calls itself again, but on a different
// part of the list
int sumList(struct node *head) {
    int total = head->data + sumList(head->next);
    return total;
}
```

The function can call itself? What happens here?

Functions calling themselves

```
sumList(head) = head->data + sumList(head->next);
```

The total is equal to the value of the head added to the sumList function called on the rest of the list

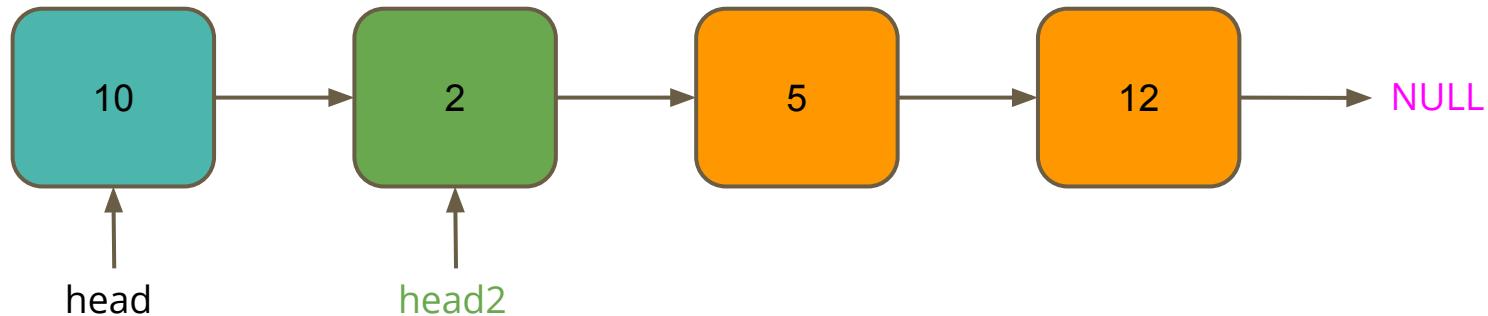


10 + whatever the rest of the list happens to add up to . . .

The second function

```
sumList(head) = head->value + sumList(head->next);
```

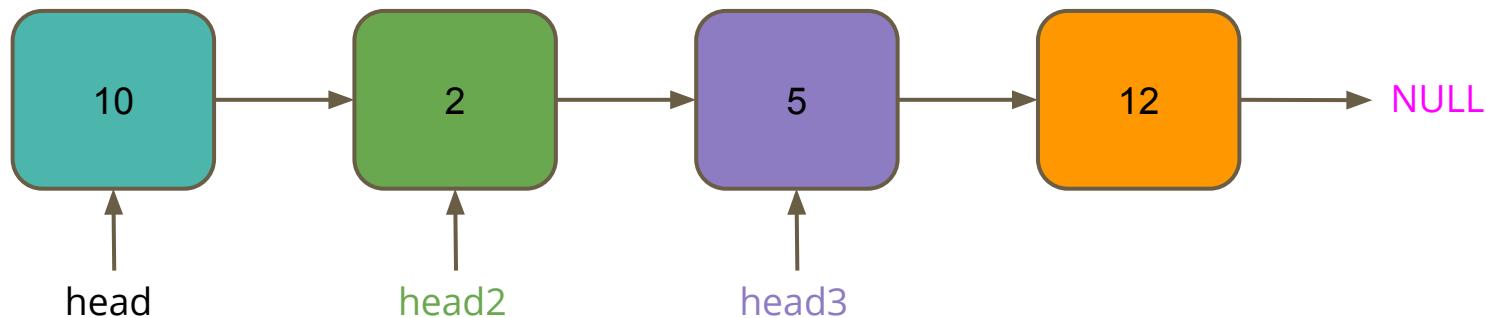
```
sumList(head) = head->value + head2->value +  
sumList(head2->next);
```



$10 + 2 + \text{whatever the rest of the list adds up to}$

It keeps going ...

```
sumList(head) = head->value + head2->value + head3->value +  
    sumList(head3->next);
```



$10 + 2 + 5 + \text{whatever the rest of the list adds up to}$

Is this endless?

Like loops, Recursive function calls still need to know when to stop

In the previous example:

- What happens if we reach the end of the list?
- What happens if the list was empty to begin with?

We need a "stopping case" where the function won't call itself again

Two Cases

Keep going or stop

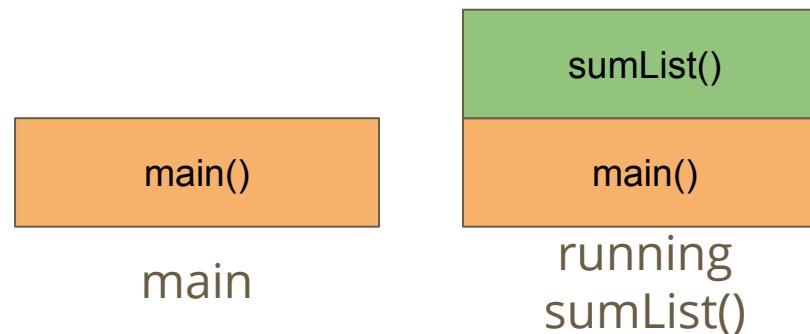
- We've already got the "keep going" case
- How do we stop?
- Let's test for the situation where we wouldn't want to add more elements

```
// sumList calls itself again, but stops if there's
// nothing to add
int sumList(struct node *head) {
    if (head == NULL) {
        return 0;
    } else {
        int total = head->data + sumList(head->next);
        return total;
    }
}
```

Functions and Stacks

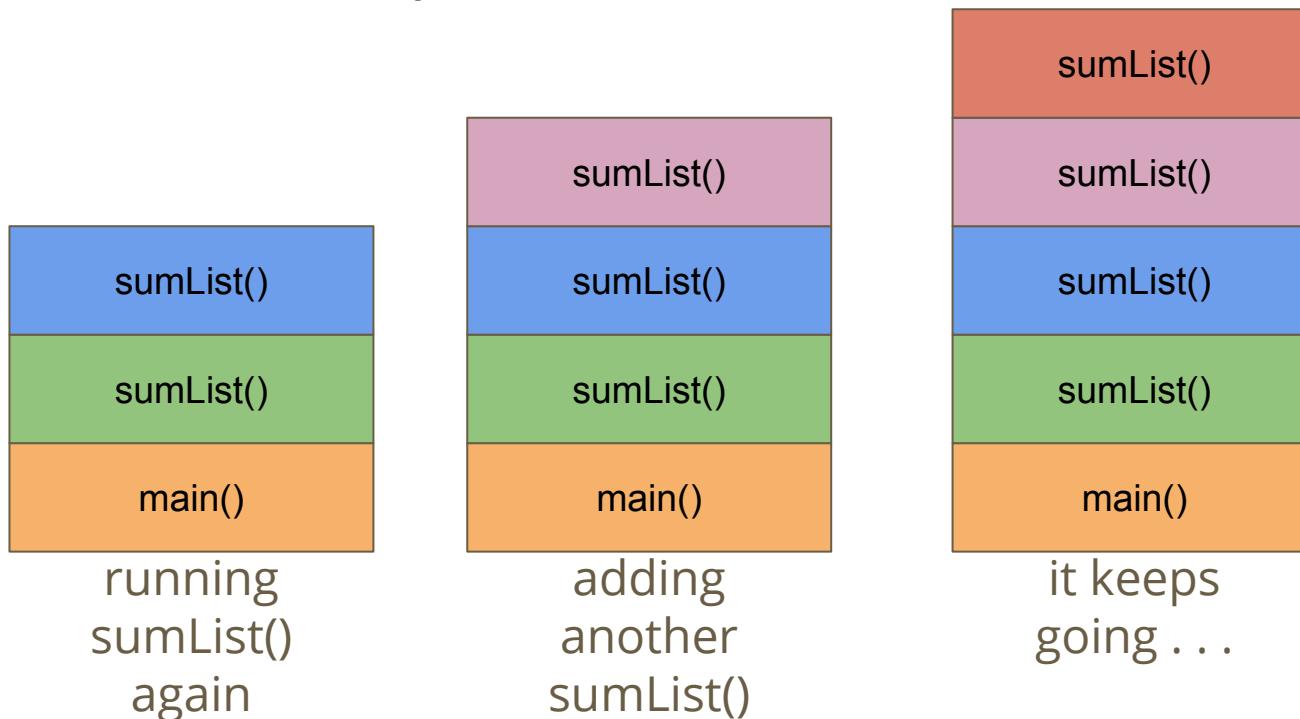
The Function Call Stack during recursion

Initially we have a main function that calls `sumList()`



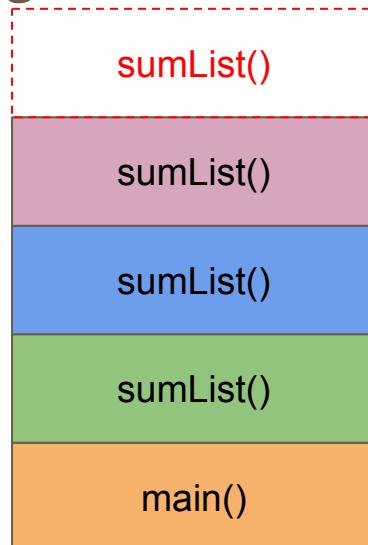
The Call Stack as Recursion continues

As the function "recurses", it adds more function calls

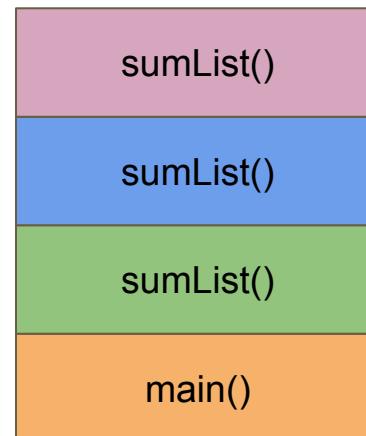


Reaching a stopping case

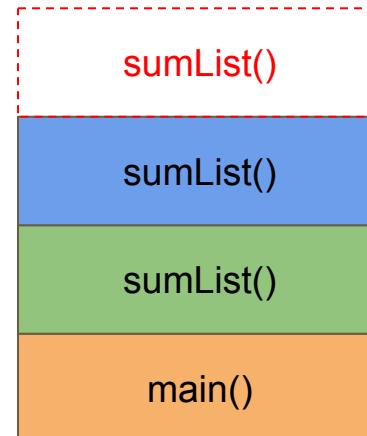
Returning from a recursive function to the previous call



Returning
because of the
stopping case



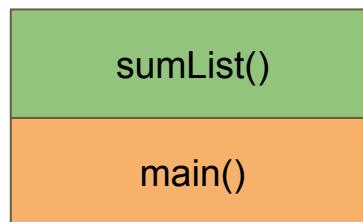
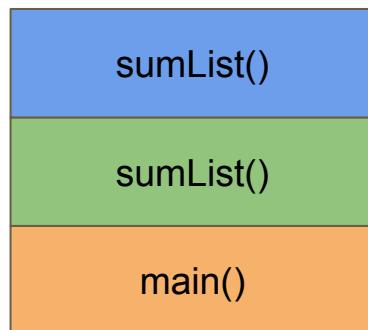
Return to
previous
function call



This also
returns

Completion of a recursive function

Eventually the chain of returns will finish



Functions return one
after the other until . . .

We're back to
the main

Code Example - Reverse Print Names

Returning to our Battle Royale Example

Say we had a list of people who had been knocked out of the game and we want to "replay" the order they were knocked out?

- We have a linked list of names
- It's currently in the reverse order of when they were knocked out
- So we want to print out their names in the opposite of their order

Our List

We have a standard linked list node

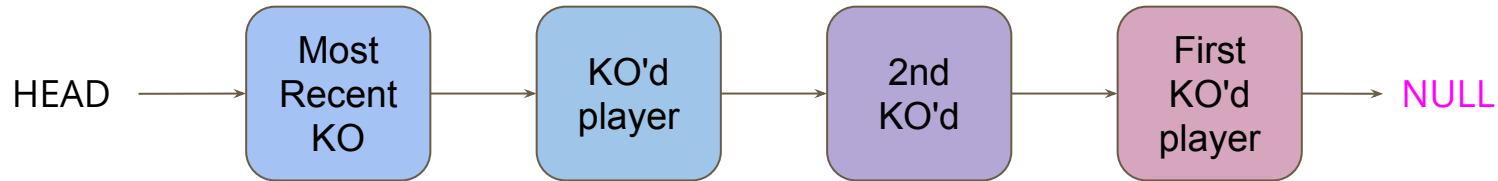
- Contains a name and a pointer

```
struct node {
    char name[MAX_NAME_LENGTH];
    struct node *next;
};
```

How it was created

Let's say that during our game, we built a list of players

- Each time a player is knocked out, we add them to the head of a list



We want to be able to print this out in the order that they were knocked out

How do we do this without Recursion?

A "procedural" implementation

1. Loop to end of the list and print out the name
2. Have some way of remembering which player we've already printed from
3. Start a new loop, going until just before the one we printed previously
4. Print out that name
5. Keep repeating until there are no names left

Code to print out an element before a pointer

```
struct player *printBefore(struct player *playerList, struct player *after)
{
    // loop until you see the after pointer
    struct player *curr = playerList;
    struct player *prev = NULL;
    while (curr != after) {
        prev = curr;
        curr = curr->next;
    }
    if (prev != NULL) {
        // element exists, print its name
        fputs(prev->name, stdout);
        putchar('\n');
    }

    return prev;
}
```

Code for a procedural reversePrint()

```
// Print out the names stored in the list in reverse order
// This is a procedural programming implementation
void reversePrint(struct player *playerList) {
    struct player *end = NULL;
    int finished = 0;

    // Loop once for each name in the list
    while (!finished) {
        end = printBefore(playerList, end);
        if (end == NULL) {
            finished = 1;
        }
    }
}
```

Break Time

Where to find further information about programming?

- There are a lot of online resources that can help with programming
- Teaching yourself can help to go beyond course content
- Stack Overflow is a question and answer site
 - It can sometimes be useful but sometimes be confusing or argumentative
- There are several free online courses that will teach you different languages
 - Too many to list!
- Experimentation will always teach you something!
- Pick an idea of something you want to make and see what you can build!

That was exhausting

What did we need to do?

Outer Loop

- Loops once for each element of the list
- Keeps track of the last element printed

`printBefore()` function

- Loops until the given element pointer
- prints out the one before that (if it exists)
- returns a pointer to the element that was printed

Recursion instead

Let's try this recursive and see how it works

Stopping case

- there are no elements, so print out nothing

Otherwise

- printReverse() the rest of the list
- After that print out the current head.

Code for reversePrintRecursive()

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

Wait is that it?

Yes.

- Recursion often takes a lot of thinking and not much code

Still, let's look deeper to get a stronger understanding

- What order are things happening?
- What happens if we change the order?

What's the order of execution?

A single call of our recursive function:

1. Check if we're stopping, if so return
2. Otherwise, call the function again with the tail (all remaining elements)
3. Then print the name of the current head of the list

Order of execution

More recursive function calls

1. Check if we're stopping, if so return
2. Otherwise, call the function again with the tail (all remaining elements)
 - a. Check if we're stopping, if so return
 - b. Otherwise, call the function again with the tail (all remaining elements)
 - i. Check if we're stopping, if so return
 - ii. Otherwise, call the function again with the tail (all remaining elements)
 - iii. Then print the name of the current head of the list
 - c. Then print the name of the current head of the list
3. Then print the name of the current head of the list

Changing the order

What happens if we change the order in a recursive function?

1. Check if we're stopping, if so return
2. Then print the name of the current head of the list
3. Otherwise, call the function again with the tail (all remaining elements)

Having swapped 2 and 3, will the function behave differently?

Changing the order in code

```
// Changing the order of operations in a recursive function
// This is a recursive programming implementation
void revPrintRec(struct player *playerList) {
    if (playerList == NULL) {
        // stopping case (there are no elements)
        return;
    } else {
        // there are element(s)
        fputs(playerList->name, stdout);
        putchar('\n');
        // the recursion is now after the print
        revPrintRec(playerList->next);
    }
}
```

Interesting results

We're now printing in order . . .

How did this happen?

Let's look at the order of execution again

Order of execution

If we change when the recursive function call is made . . .

1. Check if we're stopping, if so return
2. Otherwise, print the name of the current head of the list
3. Then call the function again with the tail (all remaining elements)
 - a. Check if we're stopping, if so return
 - b. Otherwise, print the name of the current head of the list
 - c. Then call the function again with the tail (all remaining elements)
 - i. Check if we're stopping, if so return
 - ii. Otherwise, print the name of the current head of the list
 - iii. Then call the function again with the tail (all remaining elements)

What did we learn today?

Recursion

- If you know recursion, you can return now
- Otherwise, you can learn recursion by learning recursion
- Functions calling themselves can set up interesting patterns
- It lets us use and manipulate the function call stack
- Ordering of when and how we recurse can change behaviour

COMP1511 - Programming Fundamentals

— Week 10 - Lecture 17 —

What did we learn last week?

Abstract Data Types

- Using multiple file projects
- Protecting some data from access
- Providing a nice set of functions as an interface to the code

Recursion

- Functions calling themselves
- Potentially unlimited (but with a stopping case)
- Using a call stack to control order of execution

What are we covering today?

Assessment

- The exam
- The format
- How to prepare

A recap of what we've covered in the course

- The first half of COMP1511

What's in the Exam?



Meme credit: Hyperbole and a Half

The Exam - Timing and details

12pm (Australian Eastern Daylight Savings Time) Friday 4th December for 6 hours

- Completed on your own computer at home
- Files will be provided to you when you run certain terminal commands
- Autotest and Submission will also be done with terminal commands
- Expected to take around 3-5 hours
- You can submit more than once, your final submission will be marked
- Your Week 10 labs will show you what the exam environment is like
- Before and during the exam, we will contact you via your UNSW email if/when we need to

The Exam - "Open Book"

- "Open Book" - you can use any resources from the class or online
- Still an exam! No communication between students!
- Any communication about the exam within 24 hours of the exam start will be considered plagiarism
- No external help! You cannot ask questions online or in discussion groups
- No discussion of the exam (or sharing of code) with anyone except for COMP1511 subject staff
- Effectively, you can use whatever is on the internet at 12pm at the start of the exam, but nothing that is added afterwards
- We will have an email address to contact us during the exam:
cs1511.exam@cse.unsw.edu.au

The Exam - Technical Details

- You will receive an email at your UNSW email address a few days before the exam
- **Make sure your UNSW email address is working!**
- This email will contain a link to the exam website
- The link will start working at 12pm on the day of the exam
- The commands available in the practice exam will all be available during the final exam
- Possibly with different names: 1511 `fetch-pracexam` will be 1511 `fetch-exam`

The Exam Format

The following details might change, but only slightly

- **20** short answer "theory" questions (1 mark each)
 - Some multiple choice and others a very small amount of text
- **8** practical questions (10 marks each)
 - Practical questions will involve actual programming
 - Very similar to Lab/Test questions

Exams - Marc's tips

How to survive a (take-home) exam

- Drink water. Dehydration lowers brain functions
- Eat decent meals. Blood sugar also affects your brain, especially in a stressful situation
- Take breaks! Humans can't think well for more than an hour at a time.
- You don't need to work for all 6 hours. The exam was designed for 3 hours of consistent work
- You will reach a point where you can't score any extra marks by spending more time. It's ok to hit this point and finish

Exams - Marc's tips

- If you want to prepare, you can look at the list of topics in COMP1511 and write down your own notes in advance for how you want to approach certain types of questions
- Draw Diagrams for any questions you need to think about

I'd say "*chill out, this isn't a big deal*" but no one will believe me

Short Answer Questions

Quick Questions (1 mark per question)

- These questions will be about whether you understand core coding concepts and the C programming language
- Your answers will either be multiple choice or short answers
- Some are: "What will this code do?"
- Some are: "How does this concept work?"
- Some examples are in the Week 10 Lab

Short Answer Questions - How To

- You will fetch a file called `exam_mc.txt` to answer them in
- This file is in a special format
- Type your answers within the `{{{ triple curly brackets }}}}`
- Only answers in the `{{{ triple curly brackets }}}}` will be accepted
- Some questions will have validation (so, you might only be able to answer with the letters 'A', 'B', 'C', 'D' for example)
- It will have the same structure as the file `prac_mc.txt` in the week 10 lab.

Short Answer Questions - Marc's tips

These Questions are a "warmup"

- Read through them all before answering
- Skim quickly and answer the ones you definitely know
- Then go back to the ones that take some time to think about
- Prioritise! Get the easy marks, then spend time on the ones you're reasonably sure of
- If you're unsure, go back to the lecture slides and do a text search for what you think you need
- Hit up lecture videos if you need to relearn something

Practical Questions

Less questions, more work (10 marks per question)

- Questions are similar to the Weekly Tests and Labs
- Stages of difficulty from basic to extreme challenge
- Some will have provided code as frameworks
- Each question will need to be written, compiled and tested
- You will have access to an autotest (but it's just a test!)
- Harder questions will have less autotests
- There will be no specific style marking, so you don't need to explain your code in comments

Hurdles

Hurdles must be passed to pass the course

- There's an array hurdle, question 1 or 3
- There's a linked list hurdle, question 2 or 4
- You must earn a mark of 50% or more in at least one array hurdle question
- You must also earn a mark of 50% or more in at least one linked list hurdle question
- The simplest thing is to put a serious effort into questions 1 and 2, which will cover both hurdles

Practical Questions - Marc's tips

Solving Problems

- Read all the questions before starting
- Pick the easy ones as you read. Most likely the earlier questions
- Prepare! A couple of minutes thinking and drawing a diagram will clarify how you're going to approach a question
- Use your lab/test practice! Debugging and testing will be important here
- Less questions answered completely is better than more questions partially answered
- Don't count the number of autotests. Marks are not based on number of tests passed

Questions 1-2 First Hurdles

Basic C Programming - similar to Weekly Test question 1 or 2

- Create C programs
- Use variables (ints and doubles)
- scanf and printf
- if statements and loops
- Use of arrays of ints/doubles, possibly 2D (q1)
- Use of linked lists of ints/doubles, without insertion or removal (q2)

Example Question 1

Loop through a 2D array and gather some kind of information

Eg: Edit the function even_lines so it loops through all the elements of an array. Print out every line that has exactly one even number

```
even_lines(int side_length, int numbers[SIZE][SIZE])
```

```
% ./evens  
13 14 15  
16 17 18  
21 23 25  
[Ctrl + D]  
13 14 15
```

Example Question 2

Perform some computation on a linked list

Eg: Given a linked list, print the difference between the largest and smallest values.

Edit the function: `int biggest_diff(struct node *head)`

```
% ./biggest_diff 5 4 3 2 1
```

```
4
```

```
% ./biggest_diff 3 7 3 7 12
```

```
9
```

Questions 3-4 Harder Hurdles

More advanced C - similar to a hard Weekly Test question 2

Everything from Questions 1 and 2 as well as . . .

- Looping through possibly more than once
- Testing more difficult conditions and keeping track of more than one concept
- Some insertion/removal with linked lists
- Working with Arrays (q3)
- Working with Linked Lists (q4)

Questions 5-6

Even Harder C - similar to Weekly Test question 3

- Likely using strings (q5)
- Possibly fgets, fputs, command line arguments etc
- Manipulate linked lists (adding and removing items etc) (q6)
- Potentially use malloc() and free() with structs and pointers
- Might use an Abstract Data Type
- Again, more complex combinations, and some questions requiring interesting problem solving

Question 7 and 8

Challenge Questions for people chasing HDs (10 marks)

- Everything taught in the course might be in these questions
- Think Challenge Exercises, even some of the hard ones!
- Will also test your ability to break a problem down into its parts
- This week's lab has a past Question 8 so you can see the difficulty level
- Partial completion of this question will award some marks

What to study

A little preparation goes a long way

- The basics are important!
- A basic knowledge of several topics is better than an extreme level of knowledge in just one
- Know how to use both **arrays** and **linked lists**
- Try some revision questions from the Tutorials or Labs
- The revision exercises on the course webpage are also very useful (this section will be added to the website this week)

How important are different topics?

Important

- Variables, If, Looping, Functions, Arrays, Linked Lists

Things that you will need to understand the important topics

- Characters and Strings, Pointers, Structs, Memory Allocation

Stretch Goals

- Abstract Data Types and Multi-File Projects
- Recursion

Exam Marking

Most of the marking will be automated

- Make sure your input/output format matches the specification
- Answers for hurdles will also be checked by hand
- Marks will be earnt for correct code, not for passing autotests
- Minor errors, like a typo in an otherwise correct solution, will only result in a small loss of marks
- Results should be ready by around the 18th December

Special Consideration and Supplementary Exam

- If you attend the exam, it's an indication that you are well enough to sit the exam
- If you are not well enough to sit the exam, apply for Special Consideration and do not attend the exam
- If you become sick during the exam; or you are unable to continue due to circumstances out of your control, let us know via the provided email address (**cs1511.exam@cse.unsw.edu.au**).
- A supplementary exam will be held between the 11th and 15th January 2021. If you think you will need to sit this exam, make sure you are available.

Online Exam timetable clashes

There are several exams running concurrently with COMP1511

- If you have an exam running on the same day as COMP1511
- You may have several options as to how to resolve a timing clash
- You should already have received an email about this
- Please fill out the form in the email
- Or email us at cs1511@cse.unsw.edu.au

If you are in doubt during the exam, ask!

cs1511.exam@cse.unsw.edu.au

Break Time

Human memory is based on active recall

- You can store something in your long term memory by reminding yourself of it repeatedly
- Active recall means using, not just reading
- Link your memory to things you already know (use examples in your revision code that are things you know well)
- Get some exercise! Active blood flow, even just a bit of walking, helps the brain

What did we learn this term?



Programming in C

Me:

I am good in C language.

Interviewer:

Then write "Hello World" using C.

Me:

```
c c cccccc c c ccccc  
c c ccc c c c c  
cccccc ccc c c c  
c c c c c c c  
c c ccccc ccccc ccccc  
c c cccccc ccccc c ccccc  
cc c cc c c c c c  
c c c c c c c c  
c c ccccc c c c c c  
c c cccccc c c c c c
```

Programming in C

COMP1511 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
- Abstract Data Types
- 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



Why are you running?

When my code compiles on the
first time

Input/Output

Scanf and Printf allow us to communicate with our user

- `scanf` reads from the standard input
- `printf` writes to standard input
- 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

```
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"

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

```
// 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 powerTwo = 88;
}
```

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

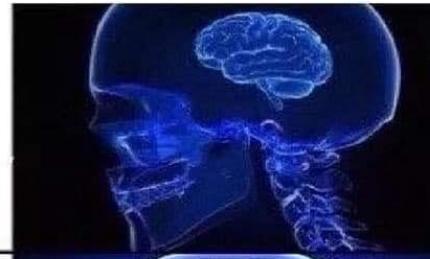
- While loops allow us to run the same code multiple times
- We can stop them after a set number of times
- Or we can stop them after a certain condition is met

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

```
3  
4  
5     x += 50;  
6  
7  
8
```



```
3  
4  
5     x = x + 50;  
6  
7  
8
```



```
3  
4  
5     for (int i = 0, i++, i<50)  
6     {  
7         x++;  
8     }  
9  
10  
11
```



```
x++; x++; x++; x++; x++;  
x++; x++; x++; x++; x++;
```



While loop code - Arrays

Very commonly used to loop through an array

```
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

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 are collected together in memory
- Not accessible individually by name, but by index

	0	1	2	3	4	5	6	7	8	9
array_of_ints	55	70	44	91	82	64	62	68	32	72

Array Code

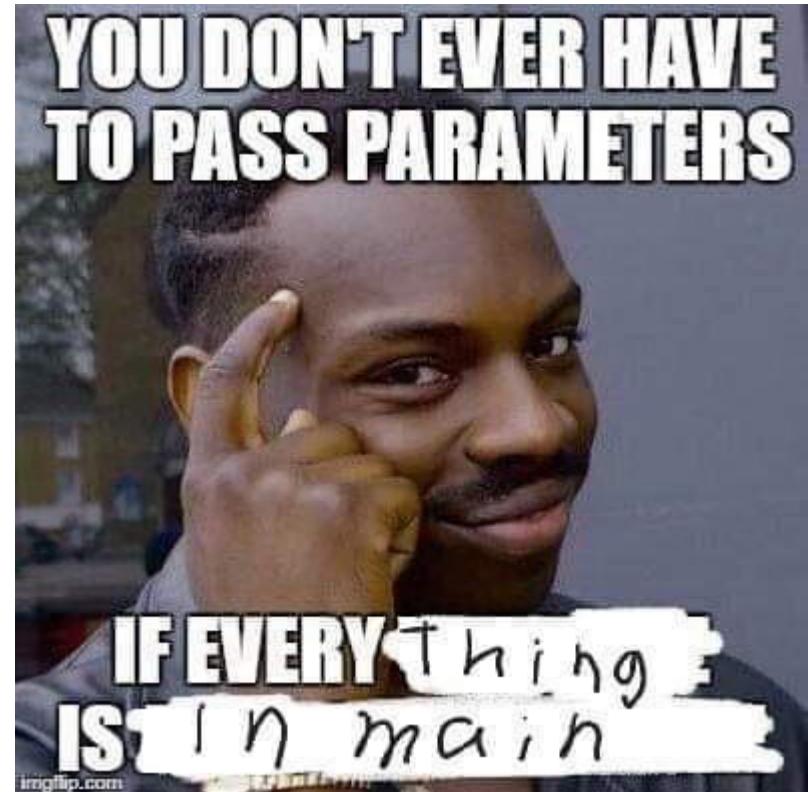
```
int main (void) {
    // declare an array, all zeroes
    int marks[10] = {0};

    // set first element to 85
    marks[0] = 85;
    // access using an index variable
    int accessIndex = 3;
    marks[accessIndex] = 50;
    // copy one element over another
    marks[2] = marks[6];
    // cause an error by trying to access out of bounds
    marks[10] = 99;
```

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

```
// Function Declarations above the main or in a header file
int add (int a, int b);

int main (void) {
    int firstNumber = 4;
    int secondNumber = 6;
    int total = add(firstNumber, secondNumber);
    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("The value of the 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("The value of the variable at address %p is %d\n", ip, *ip);
}

void increment (int *i) {
    *i = *i + 1;
}
```

What did we learn today?

Exam

- The rough format
- What to study

The first half of the course

- The technical parts of the first half of the course
- Basic C programming up to pointers

COMP1511 - Programming Fundamentals

— Week 10 - Lecture 18 —

What did we cover last lecture?

Exam

- Exam format
- Difficulty of Questions
- How to approach it

Course Recap

- The first part of COMP1511

What are we covering today?

Course Recap

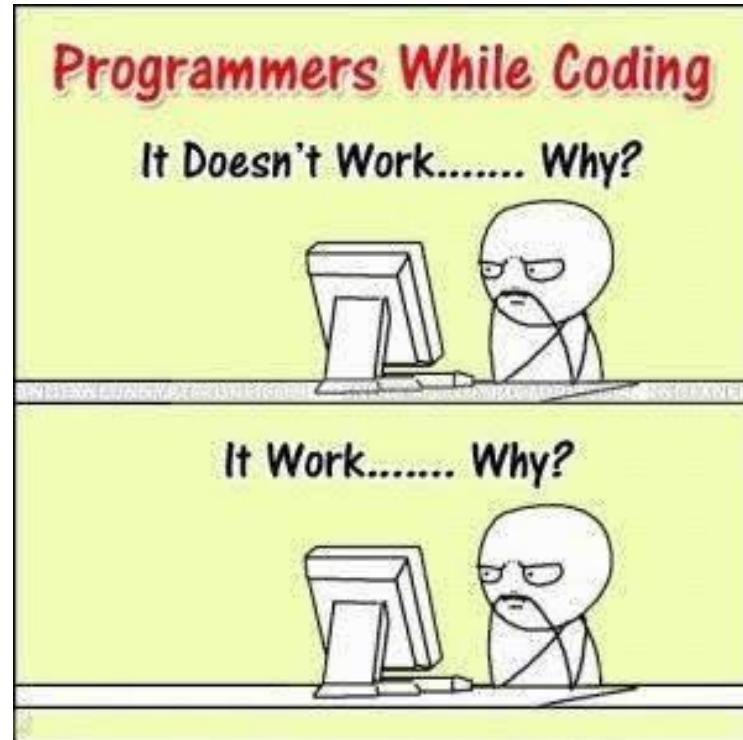
- Going through the rest of the course
- Non-technical programming skills
- The second half of the technical parts

Programming is much more than just code

COMP1511 Programming Skills Topics

- History of Computing
- Problem Solving
- Code Style
- Code Reviews
- Debugging
- Theory of a Computer
- Professionalism

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

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

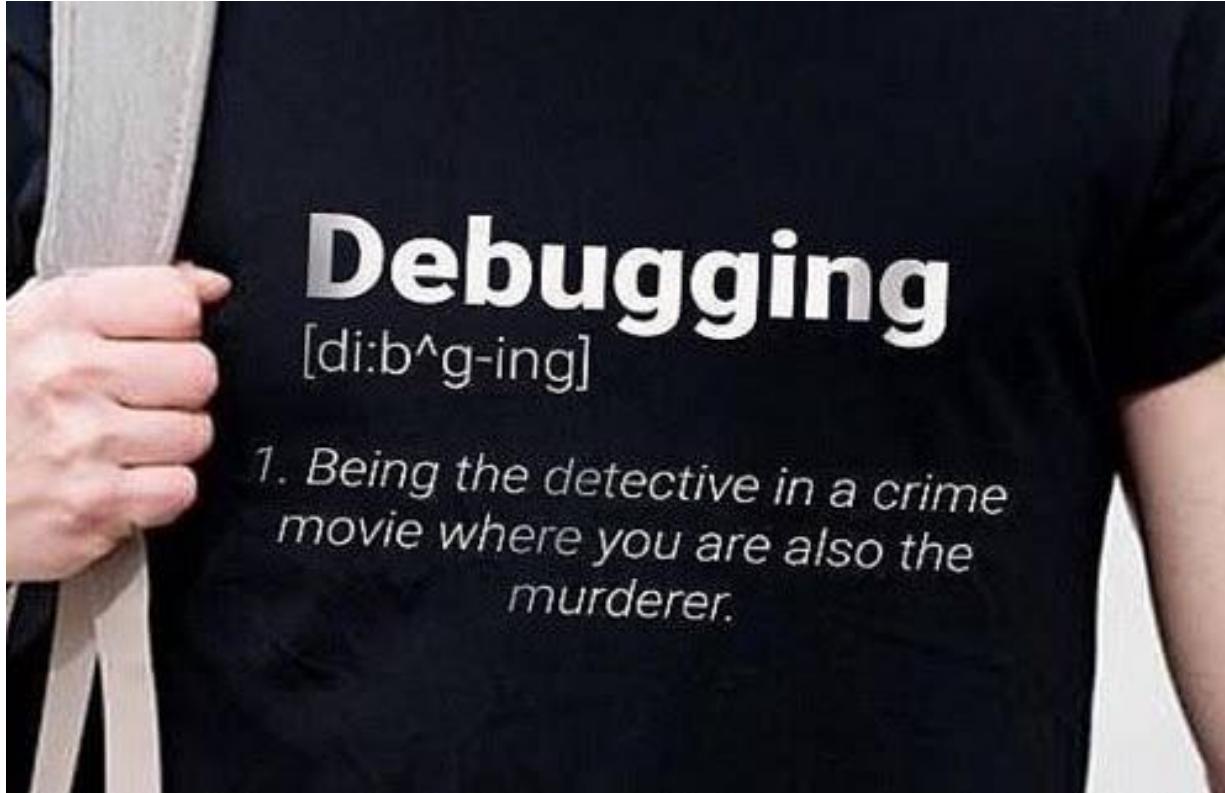
- Remember . . . readability == efficiency
- Also super important for working in teams
- It's much easier to isolate problems in code that you fully understand
- It's much easier to get help if someone can skim read your code and understand it
- It's much easier to modify code if it's written to a good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



- 1. Being the detective in a crime movie where you are also the murderer.*

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

Break Time

A thought exercise . . . the future

- Why are you doing computer science (or related field)?
- Is there something you'd like to do with these skills?
 - Jobs?
 - Research?
 - Change the World?
- How do you want to use your time at UNSW to push yourself towards your goals?
- Note: You don't need all the answers yet, but it's useful to start thinking about these things!

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
- Chicken will love you if you leave us feedback!

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

```
// 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 *myShip = &xwing;

    // my ship takes a hit
    myShip->engines--;
    myShip->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;
};

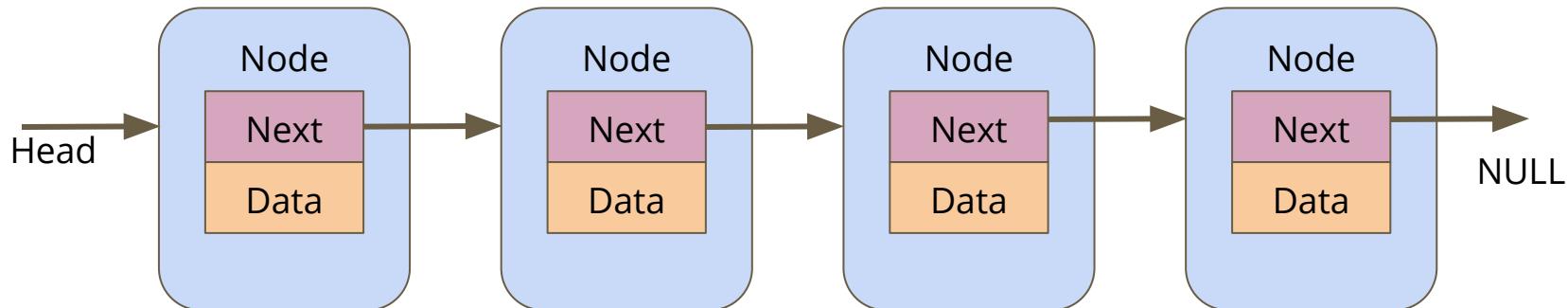
int main (void) {
    struct spaceship *myShip = malloc(sizeof (struct spaceship));
    strcpy(myShip->name, "Millennium Falcon");
    myShip->engines = 1;
    myShip->wings = 0;

    // Lost my ship in a Sabacc game, free its memory
    free(myShip);
}
```

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

Function: Removes first item from a linked list.

Second Item:



Linked Lists in code

```
struct location {
    char name[MAX_NAME_LENGTH];
    struct location *next;
};

int main (void) {
    struct location *head = NULL;
    head = addNode("Tatooine", head);
    head = addNode("Yavin IV", head);
}

// Add a node to the start of a list and return the new head
struct location *addNode(char *name, struct location *list) {
    struct location *newNode = malloc(sizeof(struct location));
    strcpy(newNode->name, name);
    newNode->next = list;
    return newNode;
}
```

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)

Complications in Pointers, Structs and Memory

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 `library->playList->track`

Complicated Pointer Code

```
int main (void) {
    // create a list with two locations
    struct location *head = addNode("Dantooine", NULL);
    head = addNode("Alderaan", head);

    // create a pointer to the first location
    struct location *alderaan = head;

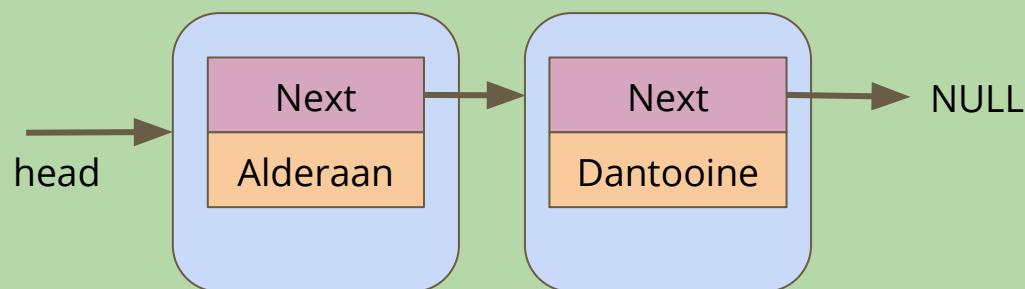
    // set head to a newly created location
    head = malloc(sizeof(struct location));

    // What has happened to the alderaan pointer now?
    // What has happened to the variable that the head and alderaan
    // both pointed at?
}
```

Pointer Arithmetic

A program's memory (not to scale)

Create a linked list of two locations
with a head pointer aimed at the
first location

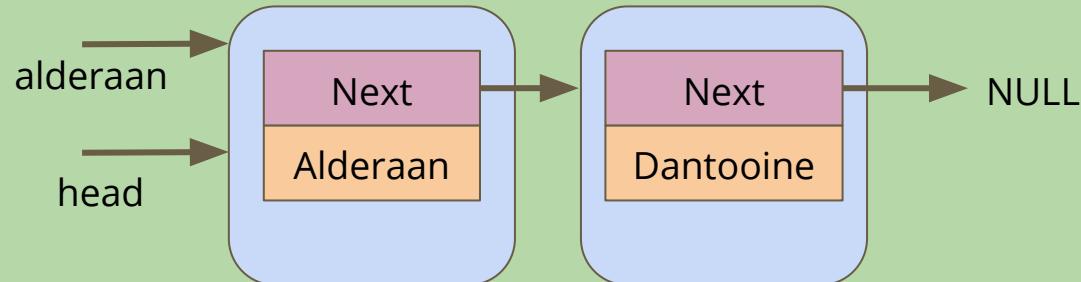


Pointer Arithmetic

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



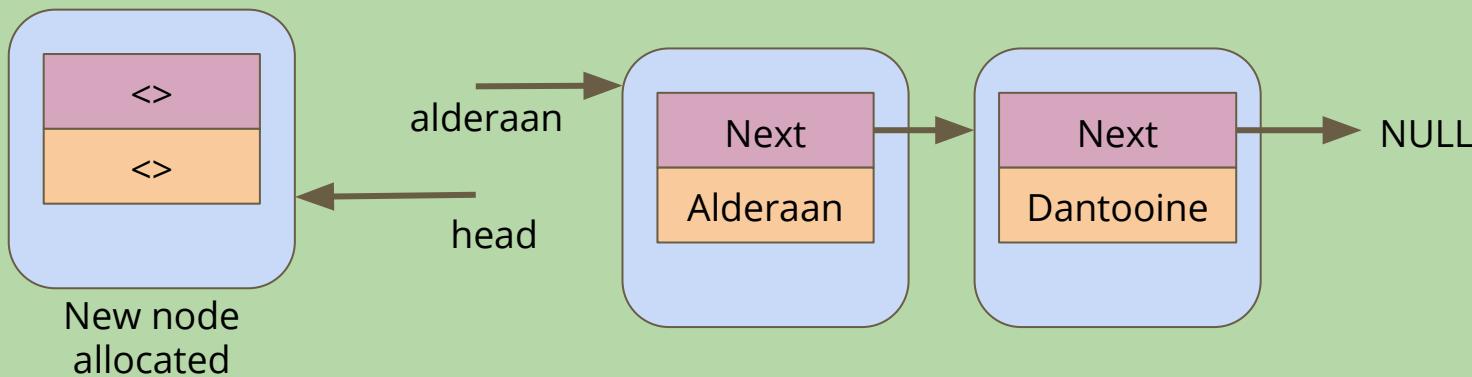
Pointer Arithmetic

A program's memory (not to scale)

```
head = malloc(sizeof(struct location));
```

This line allocates new memory and assigns the address of this new allocation to the head pointer.

Changing head doesn't change the node it was pointing at!



Keeping track of pointers

```
library->head->next->tracks->next = ????
```

- This is code that might work in most CSpotify implementations
- **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

Me: calls stack.pop()

Item at the top of the stack:



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

```
// ship type hides the struct that it is
// implemented as
typedef struct shipInternals *Ship;

// functions to create and destroy ships
Ship shipCreate(char* name);
void shipFree(Ship ship);

// set off on a voyage of discovery
Ship voyage(Ship ship, int years);
```

Abstract Data Types Implementation

```
struct shipInternals {
    char name[MAX_NAME_LENGTH];
};

Ship shipCreate(char* name) {
    Ship newShip = malloc(sizeof(struct shipInternals));
    strcpy(newShip->name, name);
    return newShip
}
void shipFree(Ship ship) {
    free(ship);
}
// set off on a voyage of discovery
Ship voyage(Ship ship, int years) {
    int discoveries = 0, yearsPast = 0;
    while(yearsPast < years) {
        discoveries++;
    }
}
```

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

```
#include "ship.h"

int main (void) {
    Ship myShip = shipCreate("Enterprise");
    myShip = voyage(myShip, 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

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

Order of execution

More recursive function calls

1. Check if we're stopping, if so return
2. Otherwise, call the function again with the tail (all remaining elements)
 - a. Check if we're stopping, if so return
 - b. Otherwise, call the function again with the tail (all remaining elements)
 - i. Check if we're stopping, if so return
 - ii. Otherwise, call the function again with the tail (all remaining elements)
 - iii. Then print the name of the current head of the list
 - c. Then print the name of the current head of the list
3. Then print the name of the current head of the list

So, you're programming now...



Coding
in C



Coding in
Python



Coding in
Scratch



Coding with
command blocks
in Minecraft



Coding

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
- Programming 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 as hard as you can and I hope to bask in your future glory

COMP1511

Good luck, have fun :)