COMP1511 – Course Notes

Contents

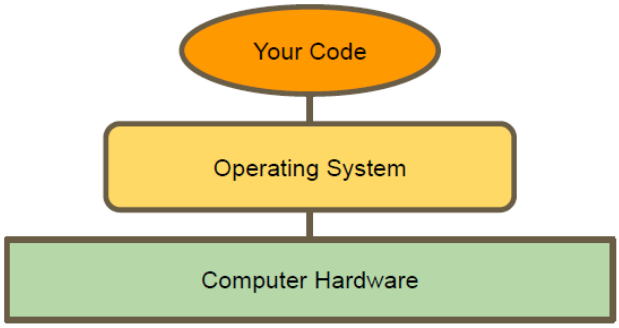
Week 1

Lecture 1:

- ✓ ***Introductions***
- ✓ ***Welcome to UNSW and Programming***
- ✓ ***How COMP1511 works***
- ✓ ***How to get help and the best ways to approach learning Programming***
- ✓ ***What are these amazing machines we call computers?***
- ✓ ***A first look at C***
- ✓ ***Working in Linux***

- Course Convenor/Lecturer **Marc Chee** cs1511@cse.unsw.edu.au
- Course format:
  - Lectures (Tuesday 9 am – 11 pm, Thursday 4 pm – 6 pm)
  - Tutorials (Thursday 6 pm – 7 pm)
  - Laboratory Sessions (10%) (Thursday 7pm – 9 pm)
  - Weekly Tests (5%) (Weeks 3-5 and 7-10)
  - Assignments (15% and 25%) (Week 6 and 10)
  - Exam (45%) 24 hour take-home exam
  - Live Streams
  - Help Sessions

- CSE computers use the **Linux** operating system.
- Use VLAB to remotely access CSE's resources.
- For COMP1511 we need:
  - A text editor like **gedit**
  - A compiler (we use **dcc**)



Linux

- The main interface to Linux is a terminal – all our interaction is in text.
- Commands:
  - `ls` – (listing) lists all the files in the current directory
  - `mkdir directoryName` – (make directory NAME) makes a new directory called directoryName
  - `cd directoryName` – (change directory NAME) changes the currect directory
  - `pwd` – tells you where you are in the directory structure at the moment
  - `gedit helloWorld.c` – open gedit, file named hellowWorld, c is the language
  - `gedit helloWorld.c &` – open gedit, file named hellowWorld, c is the language and keeps the terminal window open
  - `dcc helloWorld.c -o helloWorld` – compiling, use the dcc program for helloWorld.c to write out (-o) a file called helloWorld that we can run by then typing
  - `./helloWorld` – run the program after helloWorld.c is written out

Basics

- gedit (gee-edit)
    - A basic text editor
    - Helps out a little by highlighting C in different colours
- dcc/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, June 2019

#include <stdio.h>

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

- Comments

```
// Demo Program showing output
// Marc Chee, June 2019
```

    - For humans to understand!
    - Use either:
        - `// <comment>`
        - `/* <comment> */`
- `#include`

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

    - "standard input output"
    - 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

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

- A function is a block of code that is a set of instructions
- The computer runs this code, line by line
- The first line tells us:
  - `int` is the output – this stands for integer which is a whole number
  - `main` is the name of the function
  - `(void)` means that the function does not take any input
- Between the `{` and `}` are a set of program instructions
- `printf()` is actually another function from stdio.h which we included. It makes text appear on the screen
- `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. If it doesn't return 0 then there is an error.

Lecture 2:

- ✓ ***Variables and how we store information***
- ✓ ***Some basic maths in code***
- ✓ ***Conditionals - Running code based on questions and answers***
- ✓ ***The 'if' statement***

How computer's remember things

- Computer memory is a big pile of on-off switches (binary system), called bits. 8 bits make up a byte.
- We work with chunks of memory made of specific sizes – 32bit and 64bit systems.

Variables

- Asking the computer to remember something for us
- A "variable" can change its value
- Each variable has a certain number of bits depending on how much information we store
- Starting 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). Has digits and a point.
- To name the variables:
  - ALWAYS start with lowercase, otherwise it has a different purpose
  - They are CASE SENSITIVE
  - Some words CANNOT be variables, e.g. `return`, `int`, `double`
  - Multiple words can be written in:
    - Snakecase: `long_answer`
    - Camelcase: `shortAnswer`

- `int`

- A whole number, no fractions or decimals
- Commonly 32 bits (4 bytes)
- 2^{32} different possible values, meaning the integer is limited and cannot just hold any number
- Ranges from -2147483648 (-2^{31}) to 2147483647 ($2^{31} - 1$). The zero is considered positive.
- Issues:
  - If it exceeds the maximum number, it loops back to the lowest possible number (in the negatives) and vice versa. Look up the Ariane 5 explosion. This is called underflow and overflow.
  - `int` may not always be 32 bits, dependent on Operating System

- `double`
  - A decimal value – "floating point" means the point can be anywhere in the number
  - It's called a "double" because it is usually 64 bits, hence the double size of integers
  - Issues:
    - No such thing as infinite precision, cannot encode a number like 1/3
    - If 1.0 is divided by 3.0, we'll get an approximation of 1/3
    - The effect of approximation can compound the more you use them

Code for Variables

```c
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;
}
```

- Variables can be printed, for example:

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

  - Use %d for integers (`int`)

- Multiple variables can also be printed, for example:

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

- `%d` (decimal integer) is for ints, `%lf` (long floating point number) is for doubles, for example:

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

Reading input into variables

```
// 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);
```

- Note the `&` symbol tells `scanf` where the variable is

Constants

- Constants never change, so they are not variables
- They do not take up memory
- They are named in ALL CAPS so we remember that they're not variables
- Use `#define`

```
// Variables demo
// Marc Chee, February 2019
#include <stdio.h>
#define PI 3.14159265359
#define SPEED_OF_LIGHT 299792458.0
int main (void) { ...
```

Maths

- A lot of arithmetic operations will look very familiar in C
- `+`, `-`, `*`, `/`
- They use normal mathematical order, 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);
```

- Other issues:
    - 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
    - ints will always drop whatever fraction exists, they won't round nicely
        - For example, 5/3 will result in 1
    - % is called Modulus. It will give us the remainder from a division between integers
        - For example, 5 % 3 will result in 2

The if statements

- First we ask a question, then 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;
}
```

- We can expand 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)
}
```

- This code shows if and else statements 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)
}
```

Rational Operators

<	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

Logical Operators

- Used to compare expressions

&&	AND
||	OR
!	NOT

- 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

Assignment Operators

Operator	**Example**	**Same as**
=	`a = b`	`a = b`
+=	`a += b`	`a = a + b`
-=	`a -= b`	`a = a – b`
*=	`a *= b`	`a = a * b`
/=	`a /= b`	`a = a / b`
%=	`a %=b`	`a = a % b`

Increment and Decrement Operators

If `i = 5`	**Output**
`i++`	6
`i--`	4

Dice check program

```
/* Dice Checker Lecture Demo
   Marc Chee (cs1511@cse.unsw.edu.au), June 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:
       a success (higher or equal)
       a failure (lower)
*/

#include <stdio.h>

#define SECRET_TARGET 19

int main(void) {
    // read in and store the value of the first die
    int dieOne;
    printf("Please input the value of the first die: ");
    scanf("%d", &dieOne);

    // read in and store the value of the second die
    int dieTwo;
    printf("Please input the value of the second die: ");
    scanf("%d", &dieTwo);

    int total = dieOne + dieTwo;
    printf("Total of the dice roll is: %d\n", total);

    // Check the total against the secret number
    if (total >= SECRET_TARGET) { // Success
        printf("Success!\n");
    } else { // Failure
        printf("Failure!\n");
    }

    return 0;
}
```

Week 2

Lecture 3:

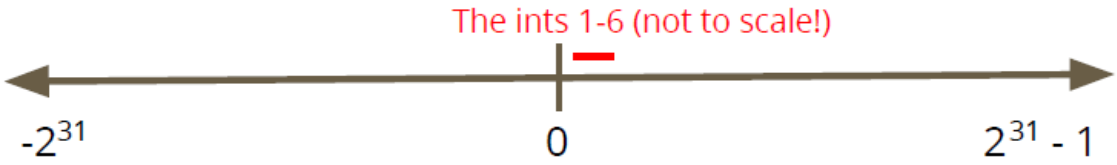
- ✓ ***Recap of some of the concepts introduced last week***
- ✓ ***A look at some more if statements***
- ✓ ***Some extra problems and solutions***
- ✓ ***Continuing the Dice Checker, but with more nuance***

Back to the dice checker

- We created a program that asks to user to input their dice values and it reported back whether the total was above or below a target value
- But what if the user enters can incorrect value? Too high or too low?

Input

- A six sided die has a input from 1 to 6, but ints have a much wider range!




```
// 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
}
```

- We could use constants instead of 1 and 6 in our code, such as:

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

  This makes our program much easier to modify for different dice sizes and makes the code more readable!

- Now, if we have the incorrect input WHAT COULD WE DO?
  - PANIC!
  - Reject the input and end the program
  - Let the user know what the correct input is
  - Correct the input
  - Ask for a new input

Rejecting the input and ending the program

- If the input is incorrect, we could shut down the program by using `return 1`
- However, this isn't a very user-friendly way, so we should explain why the program will end
- 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 0;
}
```

Correct the input instead of abruptly ending the program

- We could do this in two ways:
    - Clamping – anything outside the range gets "pushed" back into the range
    - Modulus – a possibly elegant solution

Clamping

- Correcting the values, this is a brute force approach
- If the value of the die is above the range, the value becomes 6
- If the value of the die is below the range, the value become 1

```
// 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;
}
```

- Issue? You are correcting data without the user knowing!

Modulus

- Modulus, `%`, is a maths operator that gives us the remainder of a division
- Any number "mod" 6 gives us a value from 0 to 5
- 0 is not a number on a die so we should change the 0 to a 6, to get a 1 to 6 range
- This means the user can type in completely random numbers and be given a 1-6 dice roll result

```
// 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;
}
```

- This means, we guarantee a number between 1 and 6 we won't shut down unexpectedly due to incorrect input and we give a very dice-like randomish result (as opposed to clamping).
- But, we might accept incorrectly input silently and we might make a change that affects the user's expectations.

So there are a range of solutions, what should we do though?

- No single answer
- The original purpose of the program can help us decide, e.g. if this a case where precision is necessary such as in medicine
- What's our priority?
- Exact correctness?
- Failure on any kind of incorrect data?
- Usability and randomisation over correctness?

The new Dice Checker program

- We've added:
  - Some measures against user mistakes
  - Some modifiability
- We made some decisions:
  - We will report any user error
  - But we're also delivering a die roll regardless

```
/* Dice Checker Lecture Demo Continued
Marc Chee (cs1511@cse.unsw.edu.au), June 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:
       a success (higher or equal)
       a failure (lower)

In this extended example, we worked through what
the program could do if it received input values
outside the range it expected. We looked at different
options, ranging from exiting the program to
correcting input in different ways.
*/

#include <stdio.h>

#define SECRET_TARGET 7
#define MIN_VALUE 1
#define MAX_VALUE 6

int main(void) {
       // read in and store the value of the first die int
       dieOne; printf("Please input the value of the first die: ");
       scanf("%d", &dieOne);

       // This error checking is an example of us reading incorrect // input and
       correcting it.
       if (dieOne >= MIN_VALUE && dieOne <= MAX_VALUE) {
              // valid roll
       } else {
              // invalid roll
              printf(
              "Die One, %d, was outside the range %d-%d.\n",
              dieOne, MIN_VALUE, MAX_VALUE
```

```
        );
        /* correct the invalid input using clamping
        if (dieOne < MIN_VALUE) {
            dieOne = MIN_VALUE;
        } else if (dieOne > MAX_VALUE) {
            dieOne = MAX_VALUE;
        } */

        // correct invalid input using modulus
        dieOne = dieOne % MAX_VALUE;
        printf("dieOne after modulus is: %d\n", dieOne);
        if (dieOne == 0) {
            dieOne = MAX_VALUE;
            printf("corrected value from 0 to %d\n", dieOne);
        } else if (dieOne < 0) {
            // Two options for dealing with negative mod outcomes here
            //dieOne = dieOne + MAX_VALUE;
            dieOne = dieOne * -1;
            printf("corrected negative value to %d\n", dieOne);
        }
}

// read in and store the value of the second die
int dieTwo;
printf("Please input the value of the second die: ");
scanf("%d", &dieTwo);

// This error checking is an example of us rejecting incorrect
// input and just exiting the program.
if (dieTwo >= MIN_VALUE && dieTwo <= MAX_VALUE) {
        // valid roll
} else {
        // invalid roll
        printf("Die Two, %d, was incorrect. Program will exit now.\n", dieTwo);
        return 1;
}

int total = dieOne + dieTwo;
printf("Total of the dice roll is: %d\n", total);

// Check the total against the secret number
if (total >= SECRET_TARGET) { // Success
        printf("Success!\n");
        } else { // Failure
        printf("Failure!\n");
        }

        return 0;
}
```

Lecture 4:

- ✓ ***"While" loops***
- ✓ ***How to start and stop loops***
- ✓ ***Some interesting things we can do with them***

Looping

- Repetitive tasks shouldn't require repetitive coding
- We will be using `while` loops (note: you cannot use while as a variable name)
- C normally executes in order, line by line
- if statements allow us to "turn on or off" parts of our code
- But we can repeat code, through loops; copy and pasting – not a feasible solution.

`while` loops

- If the expression is true, the loop will run
- If the expression is false, the loop will stop
- Once a `while` reaches the end of its `{ }` it will start again

```
// 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
```

Then how does the loop stop?

- We can stop them in two ways:
    - Loop counter
        - Use an int that's declared outside the loop
        - It's "termination condition" can be check in the while expression
        - It will be updated inside the loop
    - Sentinel
        - On and off switch for the loop
- If your loop runs forever upon run, press CTRL + C to exit the program. For example:

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

Example of a `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
```

Example of a while loop with a sentinel variable

- 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

- Each time a loop runs
- It runs the other loop
- The inside loop ends up running a LOT of times
- Here is an example of a loop within a loop, which draws a grid of stars
    - 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

```
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; // can be written as x++
      }
      // the row is finished, start the next line
      printf("\n");
      y = y + 1;
}
```

What goes on inside the curly brackets stays inside the curly brackets

- 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!

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

- Put `if`s inside `while` loops
- Put `while` loops inside `if`s 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

- 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:
    - 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

```c
// Dice Statistics Program (looping demo)

// Given the size of two dice and a target number
// Find out the chance of rolling the target number
// and report it as a percentage.

// This is a demo of looping with a little bit of
// discussion about division of integers.

// Marc Chee (cs1511@cse.unsw.edu.au), June 2020

#include <stdio.h>

int main(void) {
	// Take user input on the size of the two dice
	int dieOneSize;
	int dieTwoSize;
	int targetValue;
	printf("Please enter the size of the first die: ");
	scanf("%d", &dieOneSize);
	printf("Please enter the size of the second die: ");
	scanf("%d", &dieTwoSize);
	printf("Please enter the target value: ");
	scanf("%d", &targetValue);

	int numCombos = 0;
	int numMatches = 0;

	// loop through all the possible values of dieOne
	int dieOne = 1;
	// the command to stop a program (especially in the case
	// of an infinite loop) is Ctrl-C
	while (dieOne <= dieOneSize) { // we have seen dieOne - 1 values
		int dieTwo = 1;
		while (dieTwo <= dieTwoSize) { // we have seen dieTwo - 1 values
```

```
            int total = dieOne + dieTwo;
            numCombos++;
            // print out the dice combo if the total matches the target value
            if (total == targetValue) {
            numMatches++;
            printf(
                  "dieOne: %d, dieTwo: %d. Total: %d\n", dieOne, dieTwo, total
                  );
            }
            dieTwo++;
      }
      dieOne++; // is the same as dieOne = dieOne + 1;
}

printf(
      "Out of %d combinations, %d matched the target value.\n", numCombos,
      numMatches
      );
printf("%d divided by %d = %d\n", numMatches, numCombos, numMatches/numCombos);
printf(
      "%lf%% chance of rolling a %d.\n", (numMatches/(numCombos * 1.0)) * 100,
      targetValue
      );
}
```

Week 3

Lecture 5:

- ✓ ***Code style and why it matters***
- ✓ ***Code review and what we can learn from it***
- ✓ ***Introduction to what a function is and how we use functions in C***

Why do we write code for humans?

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

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

CodeStyleBad.c

```
// IMPORTANT NOTE:
// Before reading this code, bear in mind that it's
// an example of how poor coding style can affect
// our ability to understand or work with code

// Created by Marc Chee as example code, February 2019

#include <stdio.h>
#define constant 7
int main(void) {
int x;int a;   int b;int y;
   printf("Please enter how many sides are on your dice: ");scanf("%d", &x);
      printf("Please enter the value of the first die: ");scanf("%d", &a);
if (a<1) {    printf("Die roll value: %d is outside of the range 1 - %d.\n", a, x);
    // this bit does the dice thing
    a = a % x; if (a == 0) a = x;}
 if(  a > x) {
    printf("Die roll value: %d is outside of the range 1 - %d.\n", a, x);
    a = a %x; if (a== 0) a = x;}
printf ("Your roll is: %d\n", a);
   printf("Please enter the value of the second die: ");scanf("%d", &b);
      if (a < 1 || a > x) {
printf("Die roll value: %d is outside of the range 1 - %d.\n", b, x);b = b % x;
    if (b ==0) b = x;
}printf ("Your roll is: %d\n",b);
  y = a + b;
      printf("Total roll is: %d\n", y);
  // can't remember why but don't delete this next line
if (y>constant) {printf("Dice Check Succeeded!\n");} else if(y==constant) {
  printf("Dice Check Tied.\n");} else {printf("Dice Check Failed!\n");}}
```

- Problems:

- 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!

Keep it clean

- Comments before code. It's like planning ahead
  - Make 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)
}
```

- Variable names are for humans
  - If your lab partner was to read the variable, would it make sense?
  - Does it distinguish it well against other variables?
  - Be weary using names like n1 and n2, proper descriptions are better!

- Indentation – it is convention to use 4 spaces
  - { means 4 spaces right
  - } means 4 spaces left
  - In gedit, tab = 4 spaces. For other editors, tabs may be actual characters
  - It is better to do indentation manually than getting an editor to do it for you! This allows you to easily change editors when working in different environments

```
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.
- The wrong example below is like reading a book!

```
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;
      }
}
```

- The course webpage has a style guide. BUT, wherever you end up coding there will be different styles.
- The assignment will have coding style marks.
- The style checker is a good option to use, but won't check for proper variable names

Weekly tests

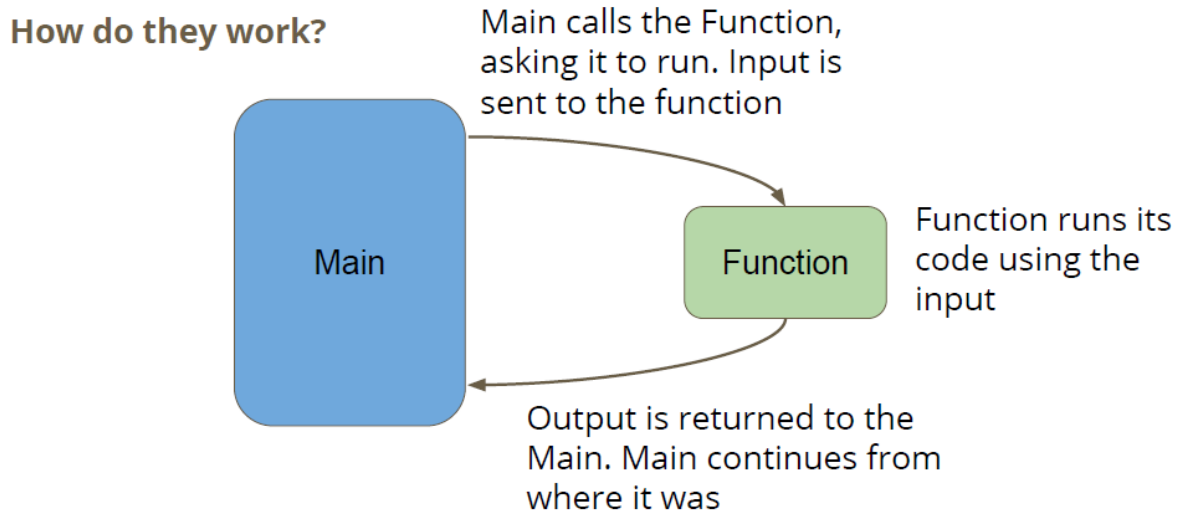
- A mini exam you run yourself
- The detailed rules are in the test itself
- Releases on Thursday and you will have one week to complete it
- Use it as a way to test your progress so far
- Great practice for coding with time pressure and limited resources (exams or job interviews)

Code review

- Have other coders look cover your code and have an active discussion about the code.
- Similar to proof-reading a document, but with fresh eyes
- It allows us to get feedback on how easy it is to understand our code and we hear about other people's ideas on solving the same problem
- 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
- Conducting a code review
    - We are judging the code, NOT the coder!
    - Discuss:
        - Whether it is easy or hard to understand the code
        - What are the different ways the code can solve the problem?
        - Any little issues we can help solve?

Functions

- `main` is a function
- `printf` and `scanf` are also functions
- A function is 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



Function syntax

- 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

- `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?

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

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 will be used without a value in a void function
- It can put stuff on the screen

```
// 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 add (int a, int b) {
   int total = a + b;
   printf("The total is %d", total);
}
```

Week 3

Lecture 6:

- ✓ ***Fundamentals of what a computer is***
- ✓ ***How we use memory in C***
- ✓ ***Arrays – using multiple variables at once***

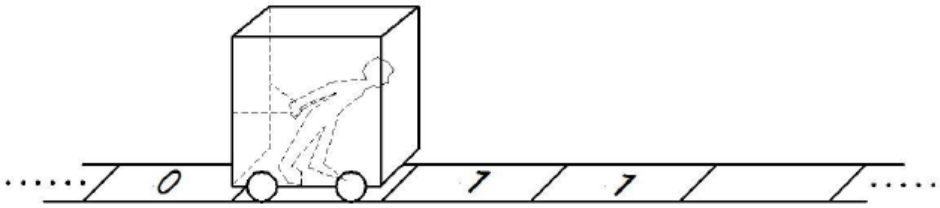
What is a computer?

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

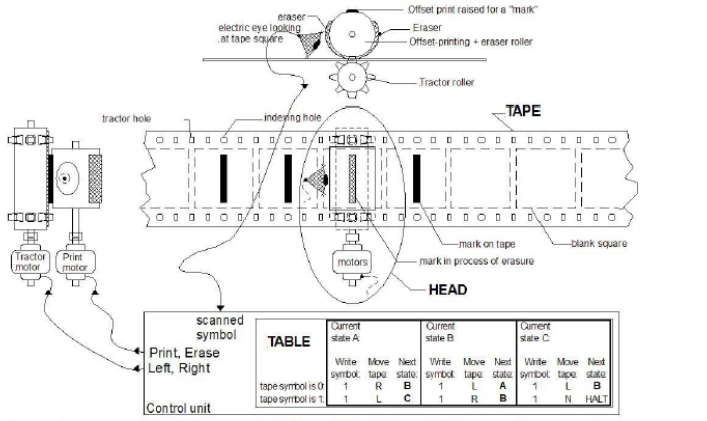
The Turing Machine

- Originally a theoretical idea of computation
  - There is a tape that can be infinitely long
  - There is a "head" that can read or write to this tape
  - The "head" can move to any part of the tape
  - There is a "state" in which the machine remembers its current status
  - Each state has a set of instructions on what to do

- Simply:



- Or a in a more complex perspective:



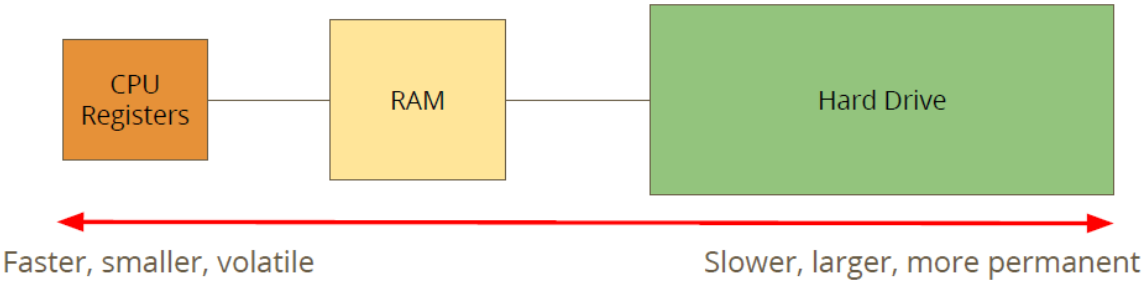
A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on another "read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the TABLE.

The Processor (CPU)

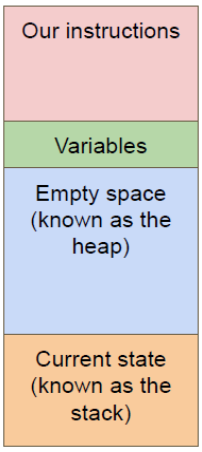
- Maintains a "state", which works based on a current set of instruction
- Can read and write from/to memory
- In our C Programming:
  - State – where are we up to in the code right now
  - Instructions – compiled from our lines of code
  - Reading/Writing – variables

Memory

- There are all forms of data storage on a computer. All are used to store information



- How C uses memory
    - C source code files are stored on the hard drive.
    - Dcc compiles our source into another file, the executable program.
    - 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
- What happens in memory when we run a program?
    - So the OS gives us a chunk of memory
    - Our program copies its instructions there
    - Some space is 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



Why we need Arrays

- We need a collection of variables together of the same type
- For example, we need to record everyone's marks at the end of the term. Everyone has a different mark and we have a large collection of integers…

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

- We what to test whether the student got a fail, pass, credit, distinction, etc. Then we would have to do this for every student! Ridiculous!

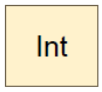
```
int main (void) {
      int marksStudent1;
      int marksStudent2;
      int marksStudent3;
      int marksStudent4;
      // etc
      if (marksStudent1 >= 50) {
            // pass
      }
      if (marksStudent2 >= 50) {
            // pass
      }
            // etc
```

- There is no way to loop the integers!
- Having to rewrite the same code is annoying and hard to read or edit
- So we should use ARRAYS!!!

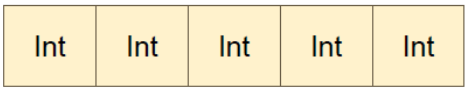
Arrays

- We can list our integers as a collection or as a group
- We can loop through and access each individual element

- A variable is a small amount of memory
- An array is a larger amount of memory and can hold multiple variables
- All elements 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

Array syntax

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

- `int` – the type of 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 and accessing them

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

- Indexes always start at 0
- Trying to access an index outside of an array will cause errors (it's a method of HACKING).
- Element 2 of arrayOfMarks is 44 and element 6 is 62.

- Accessing the 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;
```

User input/output 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 %d", &arrayOfMarks);
```

A basic program using Arrays

- 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

```c
// A Score Tracker for players in a game
// Also a demo of arrays and functions

// Marc Chee (cs1511@cse.unsw.edu.au) June 2020

#include <stdio.h>

#define NUM_PLAYERS 4

void print_scores(int score_array[], int length);
int winning_player(int score_array[], int length);

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

    // directly assign scores to players
    scores[0] = 12;
    scores[1] = 76;
    scores[2] = 34;
    scores[3] = 56;

    print_scores(scores, NUM_PLAYERS);

    int winner_index = winning_player(scores, NUM_PLAYERS);
    printf("Player %d is currently winning the game.\n", winner_index + 1);
}

// Prints out the players and their scores, one per line
void print_scores(int score_array[], int length) {
    int i = 0;
    while (i < length) { // have visited i elements
        printf("Player %d's score is %d.\n", i + 1, score_array[i]);
        i++;
    } // i == length
}

// Returns the index of the player with the highest score
// Assumes scores are positive
int winning_player(int score_array[], int length) {
    // set the highest to a very low number
    int highest = -1;
    // set the highest index to an index that's not in the array
    int highest_index = -1;

    // loop through the scores array
    int i = 0;
    while (i < length) {
        // for each element, check if it's higher than the highest (we've
seen so far)
        if (score_array[i] > highest) {
            // if it's higher, replace the highest
            // keep track of the index of the highest score
            highest = score_array[i];
            highest_index = i;
        }

        i++;
    } // i == length
    return highest_index;
}
```

Week 4

Lecture 7:

- ✓ ***Functions and Libraries***
- ✓ ***Arrays and 2D Arrays***

Recap of Functions

- Functions are code outside of `main` that we can USE and REUSE
- It 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

```
// 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;
}
```

C Libraries

- Lately, we've used `stdio.h` several times
- C has other standard libraries we can make use of
- The simple C reference in the Weekly Tests has some information
- Examples:
  - `math.h`
  - `stdlib.h`
- Look through the references (including `man` manuals in linux)

```
// 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;
}
```

- Note:
  - In the assignment we'll be using `math.h` and `stdio.h`
  - `math.h` are all doubles

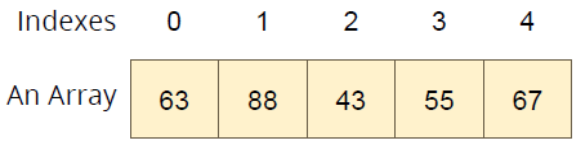
- We can use our typical functions in a more complex manner.
- For example: `scanf()`

```
int num_count = 0;
int numbers[10] = {0};

// scan in 10 numbers and store them in the array
// Note that scanf returns the number of inputs read
// so we're using this as the way to iterate the
// counter. See if you can find the issues this causes!

printf("Please enter 10 numbers for the loop.\n");
while (num_count < 10) {
        num_count += scanf("%d", &numbers[num_count]);
}

// Another option for scanning 10 numbers
// This is a pretty rough way to do this, but it's
// interesting to note that scanf will ignore gaps
// like spaces and newline (\n) in console input
// in between numbers.

printf("Please enter 10 numbers.\n");
scanf( "%d%d%d%d%d%d%d%d%d%d",
        &numbers[0],
        &numbers[1],
        &numbers[2],
        &numbers[3],
        &numbers[4],
        &numbers[5],
        &numbers[6],
        &numbers[7],
        &numbers[8],
        &numbers[9]
);

// If we're looking to only take in values until
// the input ends, we can use EOF (end of file)
// as a way to end a loop of scanf
printf("Please enter up to 10 numbers. Press Ctrl-D if you are finished.\n");
num_count = 0;
int keep_looping = 1;
while (num_count < 10 && keep_looping) {
int result = scanf("%d", &numbers[num_count]);

if (result == EOF) {
        // user has ended input
        keep_looping = 0;
        }
        // using result to increment num_count
        // this means that num_count will only go up if
        // scanf received a valid input.
        num_count += result;
}
```

```
// If we want to, we can even put the result of a function
// into the "question" expression of a loop
// Every time the while loop starts, the "question" function
// will run. The result of the function will determine
// whether the rest of the loop runs
printf("Please enter up to 10 numbers. Press Ctrl-D if you are finished.\n");
num_count = 0;
while (scanf("%d", &numbers[num_count]) != EOF) {
      num_count++;
}

// A loop to print out the contents of the array int
i = 0;
while (i < 10) {
      printf("%d, ", numbers[i]);
      i++;
}
printf("\n");
```

Recap of Arrays

- A collection of variables of the same type
- Declared using a variable type and a size
- Individual variables are accessed using an index



Accessing multiple values at once

- Loops and arrays go perfectly together
- The code below is something you will commonly use:

```
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

- The size of an array is a CONSTANT, it cannot be changed!
- You can use a constant to set the size, making our lives easier if we need to change the size of an array mid-project.

```
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!

```
#define ARRAY_SIZE 4

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

Array inside Arrays

- An Array is a type of variable
- An Array can contain any type of variable
- So… we can put arrays inside arrays, i.e. multi-dimensional arrays.
- THINK of them as a grid, two or more dimensions

Two dimensional Arrays

- A grid.
- The outer array contains arrays
- Each array 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

```
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]);
      }
```

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

```
// A heavily modified version of paint.c for Assignment 1

// This is starter code for the Tourist Program

// This program was written by Marc Chee (marc.chee@unsw.edu.au)
// in June 2019
//

#include <stdio.h>

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

// Helper Function: Print out the map as a 2D grid
void printMap(int map[N_ROWS][N_COLS], int posR, int posC);
```

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

    printMap(map, posR, posC);

    int exitLoop = 0;
    while (!exitLoop) {
        map[posR][posC]++;

        printf("Please enter a direction using the numpad: ");
        int input = 0;
        scanf("%d", &input);

        if (input == 8) { // up
            posR--;
        } else if (input == 2) { // down
            posR++;
        } else if (input == 4) { // left
            posC--;
        } else if (input == 6) { // right
            posC++;
        } else if (input == 0) { // exit
            exitLoop = 1; // Or anything else other than 1,
                          // False is 0, true is anything else
        }

        if (map[posR][posC] > 0) {
            exitLoop = 1;
            printf("I've seen this before... how boring!");
        }
        printMap(map, posR, posC);
    }
    return 0;
}

// Prints the map, by printing the integer value stored in
// each element of the 2-dimensional map array.
// Prints a T instead at the position posR, posC
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");
    }
}
```

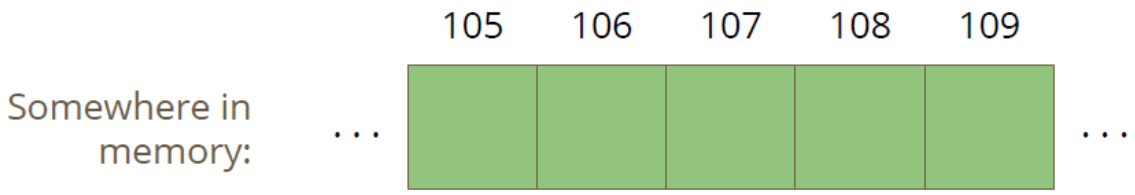
Lecture 8:

- ✓ ***Memory***
- ✓ ***Pointers***

Memory and addressing

- Let's think about how memory works in a computer using this simple analogy
  - Think about a neighbourhood and each house is a piece of memory, which can be a byte or more, depending
  - Every house has a unique address that we can use to find it
- In fact, arrays work a bit like this…
- There is the indexing of arrays to find elements
- You can think of the computer's memory as a big array of bytes

- Every block of memory has an address, which is actually an integer
- If I have the address, it means I can find the variable wherever it is in memory



- The variable is the information in the house
- The 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

- Pointers are variables, so they can change
- Pointers hold memory addresses
- They are created to point at the location of variables

- If a variable was a house, the pointer would be the address of that house
- In C, the pointer is like an integer 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 in a different way 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
// an integer in the location of i
int *i_address = &i;
```

- There are pointers which are `int`s and `double`s

```
// some variables
int number;
double big_number;
// some pointers to particular variables
// * declares a pointer variable
// & finds the address of a variable
int *small_pointer = &number;
double *big_pointer = &big_number;
```

- Therefore, most of the time pointers are 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

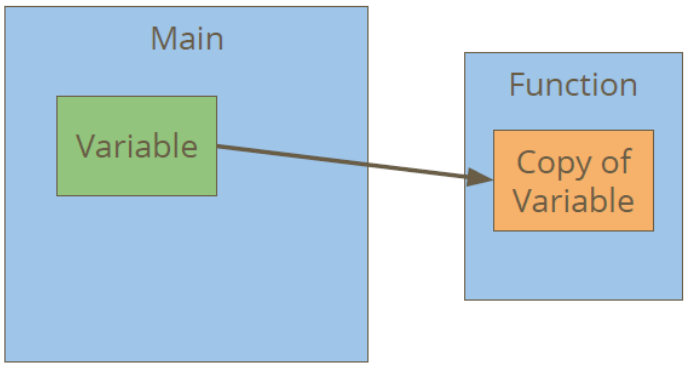
- The name of the pointer (without the `*`) will display the address of the variable using `%p` in `printf`
- The name of the pointer (with `*`) will display the value of the variable the pointer aims at

```
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);
```

- NOTE: This gets confusing, we initialised using *name, now *name is the value of the variable and not the address. Think about that the * when we initialised, is only to initialise and the name = value of the variable

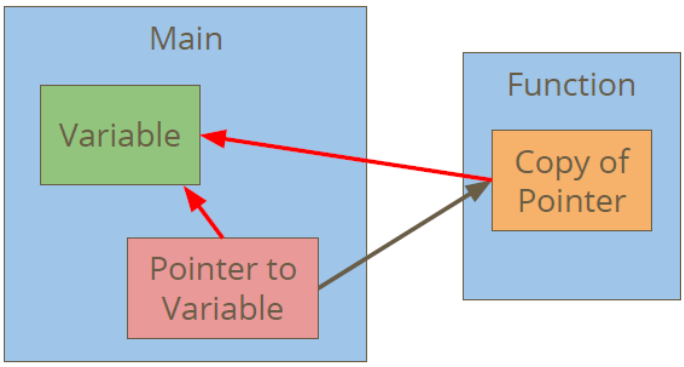
Pointers and Functions

- The advantage of pointers is that instead of passing around the value of the variable, we just pass around the address of the variable
- We can create functions that take pointers as input
- Usually, function inputs are always passed in "by value" which means they're copies, not the same variable



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

- But if I have a copy of the address of a variable, I can still find exactly the variable I'm looking for



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.

- You can access a variable in another function if it is running

- 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;
}
```

There is no RETURN!

- Now we can do more with functions!
- Pointers mean we can give multiple variables to a function and we can 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 is actually 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);
```

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

```
// Jumbler
// A demo program showing the use of
// pointers and functions

// Marc Chee (cs511@cse.unsw.edu.au) June 2020

#include <stdio.h>

#define MAX_INPUT_SIZE 100

int read_inputs(int nums[MAX_INPUT_SIZE]);
void print_nums(int nums[MAX_INPUT_SIZE], int print_quantity);
void swap_nums(int *a, int *b);
void jumble(int nums[MAX_INPUT_SIZE], int num_nums);

int main() {
    int numbers[MAX_INPUT_SIZE] = {0};
    int input_size = read_inputs(numbers);
    jumble(numbers, input_size);
    print_nums(numbers, input_size);
    return 0;
}
```

```
// Reads a number from standard input
// Then reads that many input integers from
// standard input and inserts them in order
// into the array nums.
// Returns the actual number of ints scaned into the array
int read_inputs(int nums[MAX_INPUT_SIZE]) {
    int input_count = 0;
    printf("How many numbers? ");
    scanf("%d", &input_count);

    int i = 0;
    while (i < input_count && i < MAX_INPUT_SIZE) { // have processed i
inputs
        scanf("%d", &nums[i]);
        i++;
    } // i == the actual number of inputs scanned
    return i;
}

// prints out the first print_quantity integers from the
// array nums
void print_nums(int nums[MAX_INPUT_SIZE], int print_quantity) {
    int i = 0;
    while (i < MAX_INPUT_SIZE && i < print_quantity) { // have printed i
numbers
        printf("%d ", nums[i]);
        i++;
    }
    printf("\n");
}

// Swap the values of the integers
// that the pointers a and b aim at
void swap_nums(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// use swap_nums repeatedly to change the order
// of the numbers in nums
void jumble(int nums[MAX_INPUT_SIZE], int num_nums) {
    int i = 0;
    while (i < MAX_INPUT_SIZE && i < num_nums) {
        // find another index in the array,
        // does not have a particular reason
        // it's actually a kind of random other index
        int j = (i * 2) % num_nums;
        swap_nums(&nums[i], &nums[j]);
        i++;
    }
}
```

Week 5

Lecture 9:

- ✓ ***Debugging***
- ✓ ***Characters***

Debugging

- Debugging is the process of finding and removing software bugs, it is 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 which we must adhere to
- It is MUCH MORE specific than most human languages

- Slight mistakes in the characters we use, can result in a different behaviour
- Syntax errors are obvious and your compiler will find them
- An error after a syntax error is usually as a result of the syntax error

Logical Errors

- We can write a functional program (with correct syntax), but the code doesn't solve our problem or do what we intended
- Sometimes…
    - we read the problem specification wrongly
    - we forget the initial goal of the program
    - we solve the wrong problem
    - we forget how the program might be used

How do we find bugs?

- Syntax errors
    - Compilers can catch SOME syntactical bugs
    - Code reviews and pair programming
- Logical errors
    - Testing!
    - Code reviews and pair programming

Using our compiler (dcc) to hunt syntax bugs

- ALWAYS start with the first error
- Subsequent errors are usually the result of the first error
- An error is the result of an issue, not necessarily the cause. So at the very least, you will know the line and character of where things went wrong

Hunting for logical errors

- Always TEST your code! Make sure it's doing what it's intended to do.
- Some techniques:
    - Try different input ranges, test 0 and negative numbers
    - Try outputting/printing x and y values to make sure they're working
    - Try outputting loop information so that we can see our structure

Characters

- So far we've only used `int`s and `double`s.
- A new type is `char`
- Characters are 8 bit integers! 2^8 possibilities.
- They represent letters, numbers, spaces, newlines, etc!
- We use them as characters, but they're actually encoded numbers (ASCII)
- We will not be using `char` for individual characters, but in arrays

ASCII

- 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

- `%c` in the `printf` will format the variable as a character
- `%x` in the `printf` will format the variable as the hexadecimal of the character
- Using single quotes to assign a character value

```c
#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;
}
```

Helpful Functions – getchar() and putchar()

- `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);`

```
// 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

Reading multiple characters

```
	// 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>

- `<ctype.h>` is a useful library that works with characters
- For example, go through a bunch of characters detecting if they are uppercase or lowercase
- `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

Lecture 10:

- ✓ ***Strings***
- ✓ ***Structs***

Strings

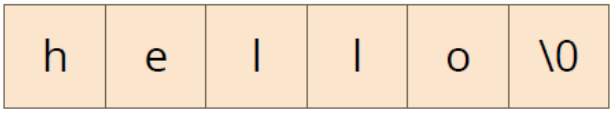
- Multiple characters together is called a string
- Strings in C are arrays of `char` variables containing ASCII code
- Strings are like words (or sentences), while chars are single letters

- Strings have a helping element at the end, a character: `'\0'` called the 'null terminator'.
- It signifies when we are at the end of a string!

- Strings are arrays of type `char` (whereas normal arrays are type `int`).
- This is how we can use them:

```
// 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 strings are created with 6 elements, the last element being the null terminator `'\0'`



Reading and writing strings

- `fgets(array[], length, stream)` is a useful function for reading strings
- It creates an array of type char named `array`, of size `length`
- The stream (where the inputs come from) will be `stdin` in COMP1511. In further courses we'll find out what other streams can exist.
- It could be said, `fgets` is a version of `scanf`

- `fputs(array, stream)` works similarly to `printf`
- It will output the string stored in the array to a stream
- The stream we will be using in COMP1511 is `stdout`

- This code will read a string inputs, of maximum size `MAX_LINE_LENGTH` (which includes the null terminator), then print them. It will repeat this until NULL

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

Helpful functions in the string library

- <string.h> has some 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
  - Syntax: `strncpy(string1, string2)`
  - Overwrite `string1` with `string2`
- `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 . . .

Structs

- 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 of arrays
- They are also able to name their variables
- Structures are like the bento box of variable collections

- Before using a struct, you must declare the type before you can use it!
- Declare what the struct is called, then what the fields (variables) are

```
struct performer {
      char name[MAX_LENGTH];
      char description[MAX_LENGTH];
      int rank;
};
```

Creating a struct variable and accessing its fields

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

```
int main(void) {
      struct performer rm;
      strcpy(rm.name, "Rap Monster");
      strcpy(rm.description, "Leader");
      rm.rank = 1;
      printf("%s's description is: %s.\n", rm.name, rm.description);
}
```

Accessing Structs through pointers

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

```c
struct performer *rapper = &rm;

// knowledge of pointers suggests using this
*rapper.rank = 100;

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

Structs as Variables

- Structs can be treated as variables
- This means that 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

Week 7

Lecture 11:

- ✓ ***Memory***
- ✓ ***Multi-File Projects***

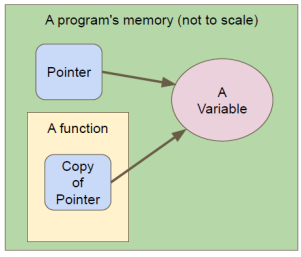
Functions and Memory

- What actually gets passed to a function?
- Everything gets passed "by value"
- Variables are copied by the function and then work with THEIR OWN versions of the variables
- For example, the `doubler` function here does not change the value of number in `main`

```c
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

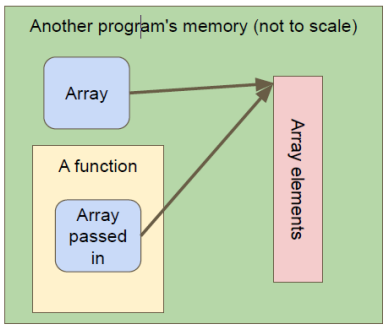
- So we should pass pointers to functions!
- The value of a pointer is a memory address
- So the memory address is copied to a function
- This means **both** pointers, the copy of the pointer in the function and the pointer in main access the same variable.
- For example



```c
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!

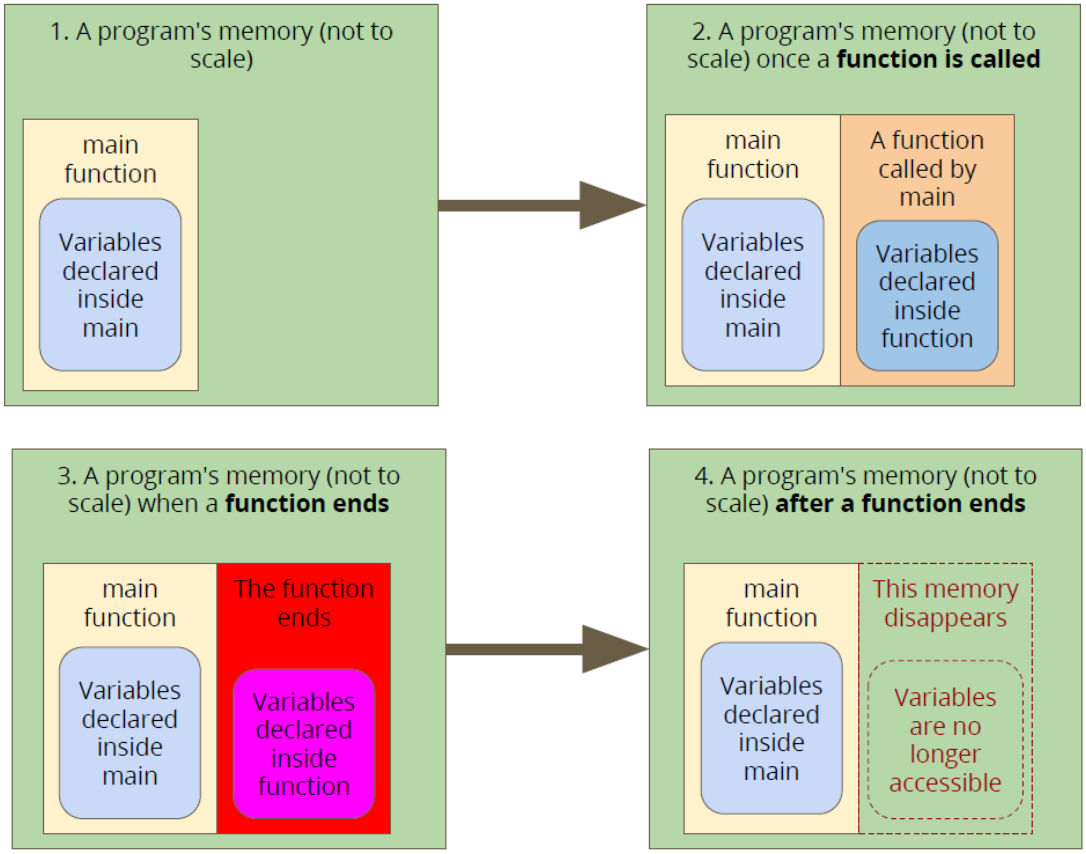
- Arrays and pointers are very similar!
- An array is a variable and it doesn't actually contain all the variables
- When we use the array variable (no []), it's actually the memory address of the start of the elements.
- Arrays and pointers act the same.
- For example




```c
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
}

// 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? It is created then disappears!

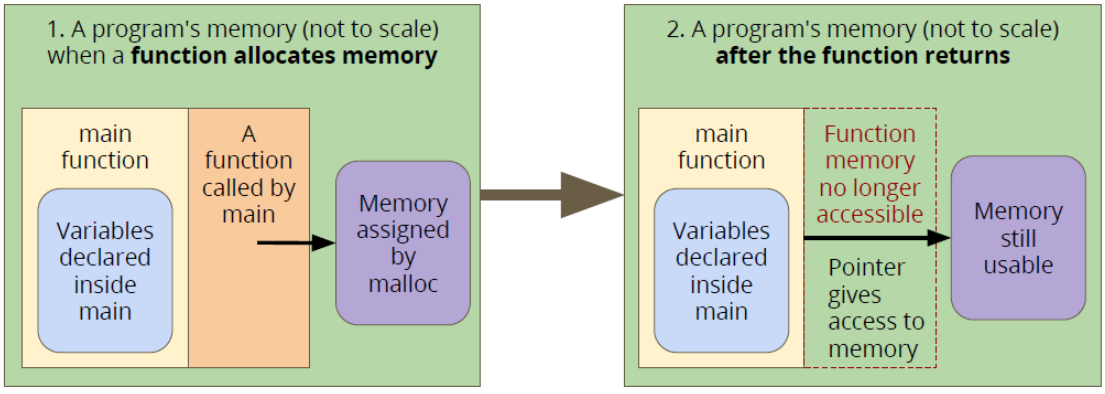


Create something in a function?

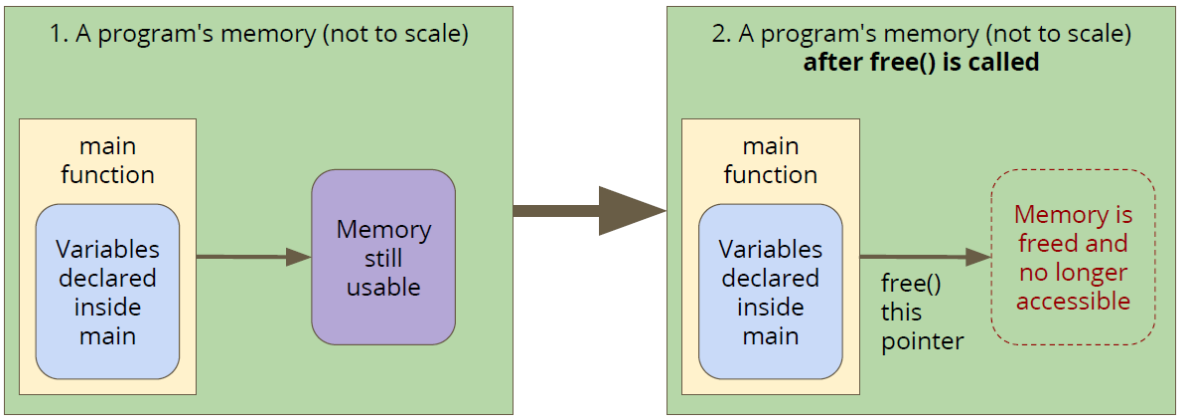
- We often want to create functions that create data
- We can't always pass it back as an output

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
- Therefore, it won't 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



- We can assign a particular amount of memory for use
- The function `sizeof()` allows us to see how many bytes that variable needs
- We can use `sizeof()` to allocate the correct amount of memory
-
- 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!



- For example,

```
int *mallocNumber()

// Use an allocated variable via its pointer then free it
int main(void) {
      int *iPointer = mallocNumber();
      *iPointer += 25;
      free(iPointer);
      return 0;
}

// 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
```

Using memory

- You can use `sizeof()` to figure out how many bytes something needs
- We can malloc arrays and structs 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!
- `dcc -leak-check` can be used to tell you if you have any memory leaks

C Projects with Multiple Files

- For readability and also to separate code by subject
- We've already seen `#include`
- We can `#include` our own files, allowing us to join projects together.
- Sometimes we'll 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 projects

Header Files and C (Implementation) Files

- Two different files for different purposes
- The 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

- Shows you what the code's functions are and all the programmer needs to know on how to use it
- `typedef` (type define) is a way to create OUR OWN types out of another type
- For example, this protects our struct from access and keeps our data safe!
- Function declarations with no definitions
- Comments describe how the functions can be used
- No running code!

Main.c and other Files

- 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

Compiling a Project with Multiple Files

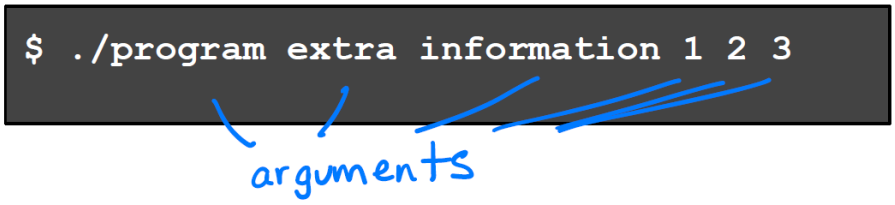
- We must compile all *.c files that we use
- The *.c files will `#include` the necessary *.h files
- Amongst the *.c files there should be exactly one `main()` function
- The compiled program will run from the start of the `main()` function

Lecture 12:

- ✓ ***Command Line Arguments***
- ✓ ***Linked Lists***

Command Line Arguments

- Sometimes we want to give information to our program at the moment we run it!
- The "Command Line" is where we type in commands into the terminal
- Arguments are another word for input parameters



  Note that the first argument here is `./program`
- The extra text we type after the name of our program is passed into our program as strings!

Main functions that accept arguments

- Usually it's `int main(void)`
- But it can be `int main(int argc, char *argv[])`
- The first input, int `argc`, counts how many arguments we will put in (remember including the program name).
- The second input, an array of character pointers. This means each element of the array is the address of another array. Without the `*`, it would just point to one array. Remember arrays already act like pointers. Hence, it is a series of separate words, our arguments!
- For example, this code prints out "Well actually "./program_name" says there's no such thing as "all arguments after the program name".

```
#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

- This means if we input any numbers, they are strings!
- If we write `./program extra information 1 2 3`. How will we process them?
- From the `<stdlib.h>` library, use `strtol()` – "string to long integer"
- Syntax: strtol(argv[i], NULL, 10)
    - First argument is the source of the string
    - Second argument will not be discussed in COMP1511
    - Third argument is that the string is converted to a number of base 10, which is the human counting system. HEX is base 16 and binary is base 2.

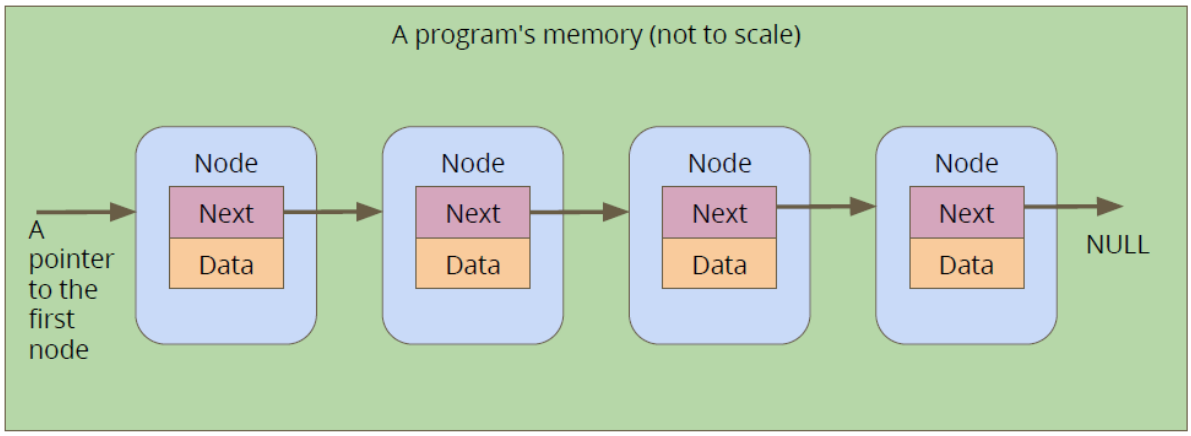
```c
#include <stdio.h>
#include <stdlib.h>

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);
}
```

Nodes - A new kind of struct

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

- This is a node!
- It contains some information.
- It also contains a pointer that aims at another node of the same type.

A Linked List



- A chain of nodes is called a Linked List.
- They can be thought as similar to arrays, but:
    - Not one continuous block of memory, one after the other
    - Items can be shuffled around by changing where the pointers aim
    - Length is not fixed when created

- You can add or remove items from anywhere in the list

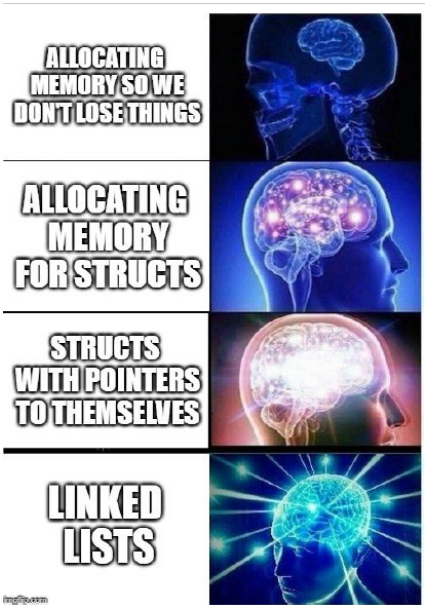
Linked Lists in Code

- What do we need for a simple linked list?
  - A struct for a node
  - A pointer to keep track the start of the list
  - A way to create a node and connect it
- About the code below:
  - This function returns a pointer that can aim at a struct node type.
  - The input it takes in are the info about the struct node, data and a pointer of type struct node named next.
  - So first it creates a pointer that can aim at a struct node type.
  - Where it is pointing right now, it allocates memory of size struct node
  - At where it is pointing right now, it copies the struct node type elements, that is data and next.
  - Then it returns the pointer (the address) to main.

```
// 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;
}
```

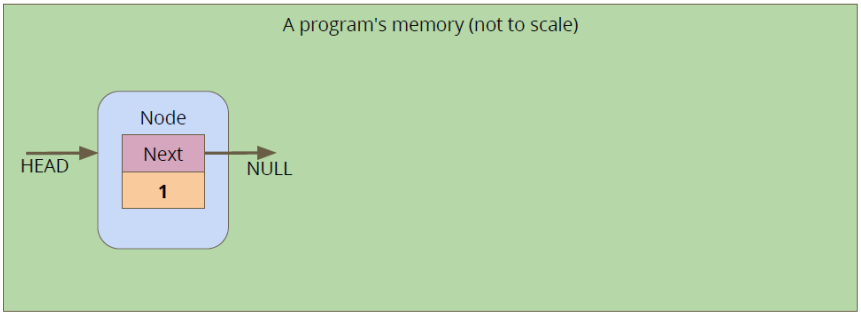
- This code in effect in main:
- What is happening below is:
  - A pointer of type “struct node” is created, it points to an address in memory which contains info about data and struct node *next, which is initially NULL (it doesn’t really point to anything).
  - Then another node is created containing data = 2 and a struct node pointer which aims at the first node since it is head currently. Then the head is now equal to this second node.
  - Rinse and repeat.

```
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;
}
```

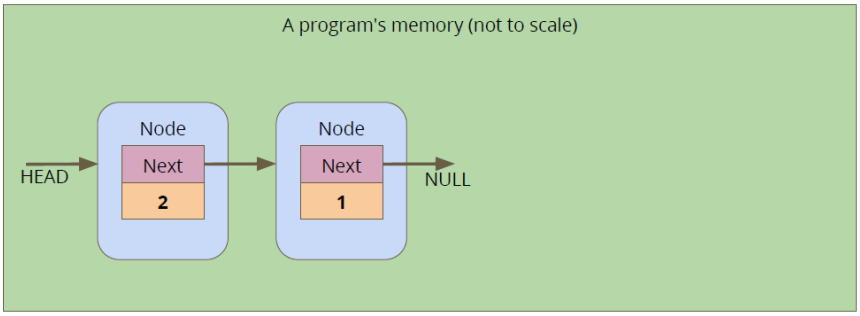


- In a diagram:

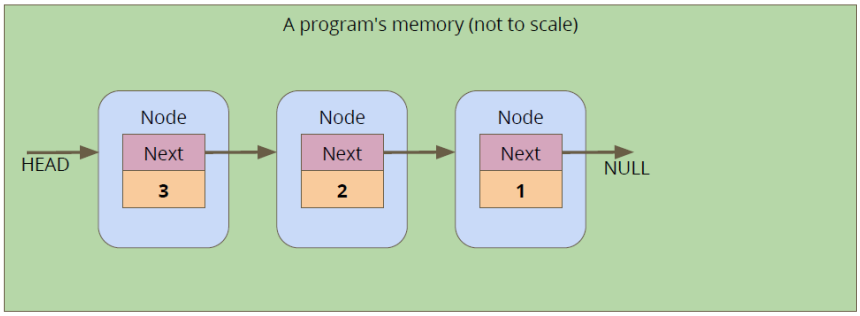
CreateNode makes a node with a NULL next and we point head at it



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



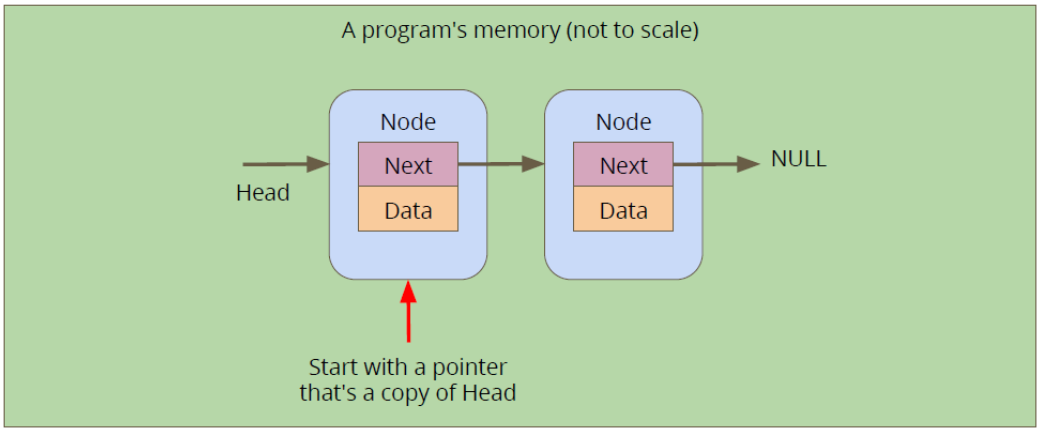
The process continues . . .

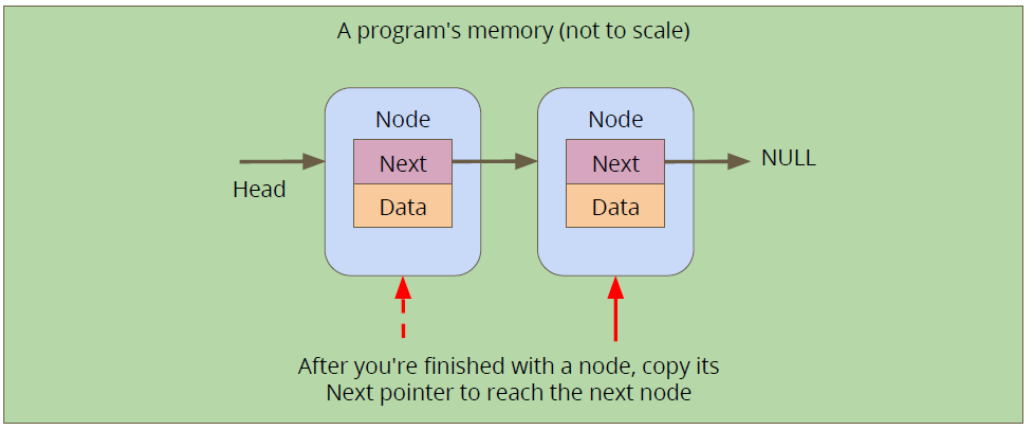


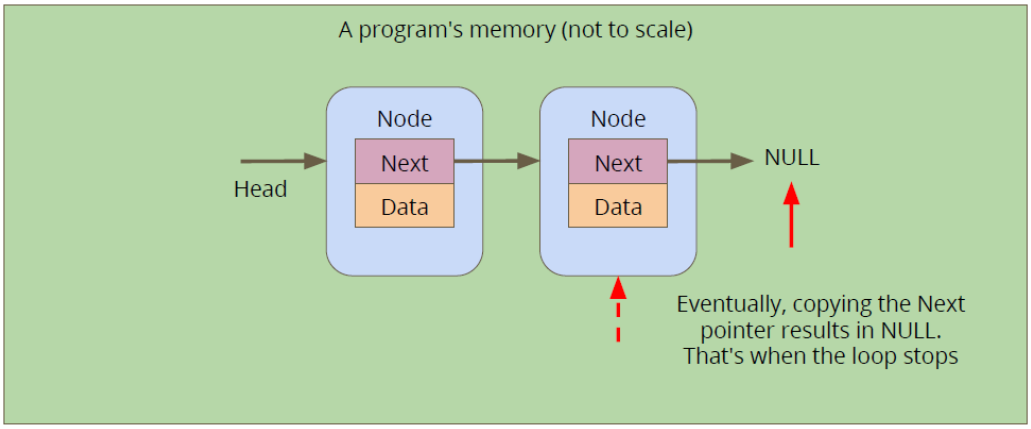
Looping through a Linked List

- We can't loop through a linked list like an array! They don't have indexes :P
- We have to follow them node to node, using the pointers!
- If we reach a NULL node pointer, it means we are at the end of the list
- We input the head pointer first!

```
// 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;
      }
}
```







Week 8

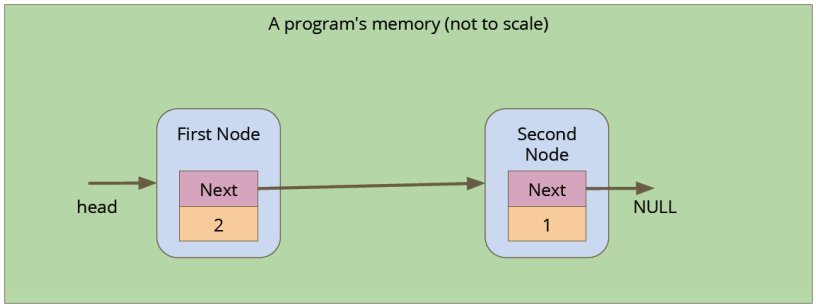
Lecture 13:

- ✓ ***Adding/Inserting Nodes to Linked Lists***

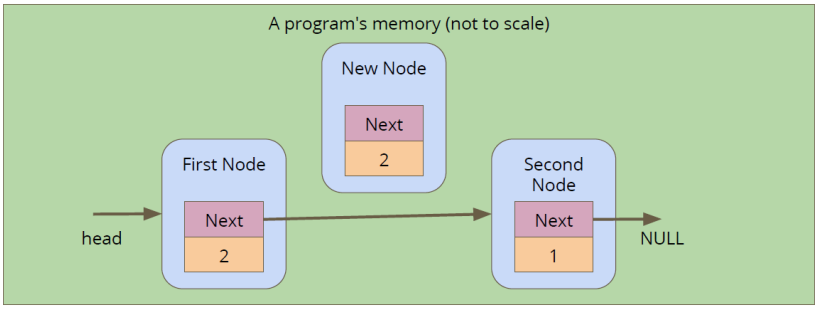
Inserting Nodes into a Linked List

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

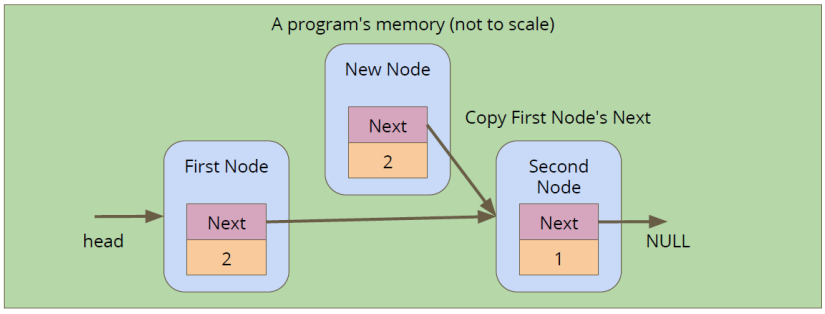
Before we've tried to insert anything



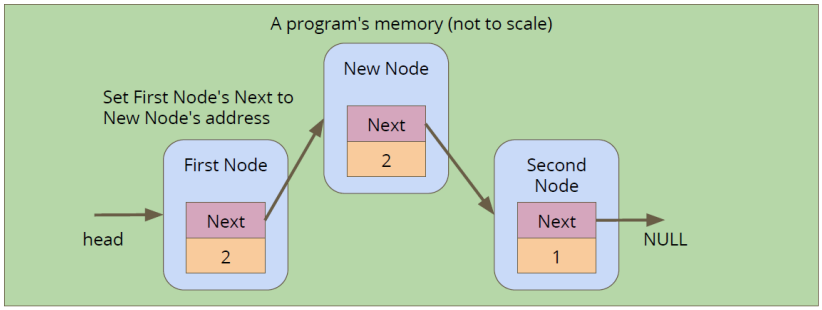
A new node is made, it's not connected to anything yet



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



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



Code for insertion of players

```
// Create and insert a new node into a list after a given insert position.
// The struct player pointer insertPos, is the node before where the new
// node is to be placed.
struct player *insert(struct player* insertPos, char newName[]) {

	// Create a pointer p, which aims at a struct player in memory with
	// the elements in it being, the name: newName and points at NULL
	struct player *p = createPlayer(newName, NULL);

	// If insertPos aiming at NULL, it returns p. If inserPos is not
	// aiming at NULL, that is another node..
	if (insertPos != NULL) {
		// 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 p;
}
```

Inserting Players to create a list

- We can use insertion to have greater control of where players end up in a list

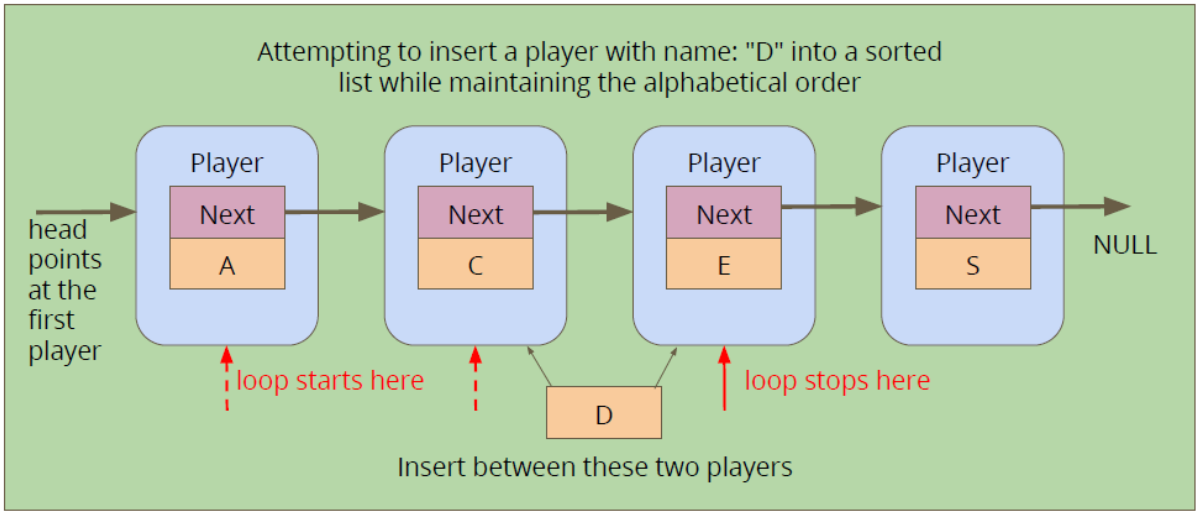
```
int main(void) {

	// create the list of players
	struct node *head = createPlayer(NULL, "Marc");
	insert(head, "Tom");
	insert(head, "Goku");
	insert(head, "Bulma");
	insert(head, "Master Roshi");
	printPlayers(head);
	return 0;
}
```

Insertion with some conditions

- We could also read the data in a node and decide whether we want to insert it before or after it
- Let's insert our elements into our list based on alphabetical order
- We're going to user `string.h` function, `strcmp()`
-
- Syntax: `strcmp(string1, string2)`
- `Strcmp()` compares two strings character by character, if the first character of two strings are equal, the next character of two strings are compared. They compare ASCII values, not letters! They return
  - 0 if they're equal
  - Negative one if the first has a lower ASCII value than the second
  - positive one if the first has a higher ASCII value than the second
  - Essentially the first - second
- `strcmp("c", "C") = 1`
  Since "c" ASCII is 99 and "C" ASCII is 67, which subtracts to 32. Hence 1.

Where to insert in alphabetical order

- Assuming 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




```
struct player *insertAlphabetical(char newName[], struct player* head) {

	// We make a copy of the head pointer.
	// Then we make a pointer aiming at NULL
	struct player *p = head;
	struct player *previous = NULL;

	// Loop through the list and find the right place for the new name
	// We stop looping when the node is we are looking at has a lower
	// ASCII value than what we are trying to insert OR when we are at
	// the end of the list
	while (p != NULL && strcmp(newName, p->name) > 0) {

		// This previous pointer tails behind the p pointer
		previous = p;

		// p pointer moves to the next node
		p = p->next;
	}

	// We create a new node to insert, this node points at the
	// "previous" pointer
	struct player *insertionPoint = insert(previous, newName);

	// 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;
	}
}
```

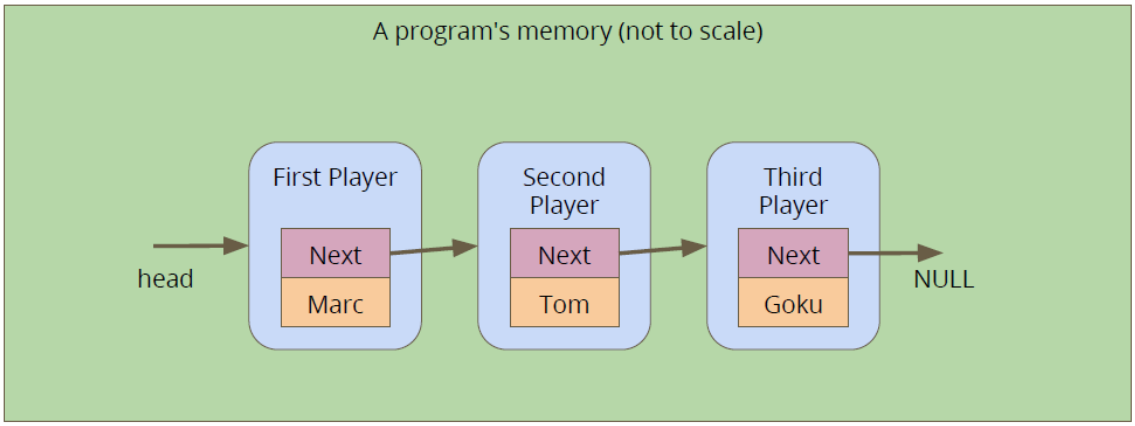
Lecture 14

- ✓ ***Removing Nodes from a Linked List***
- ✓ ***Freeing our allocated memory***

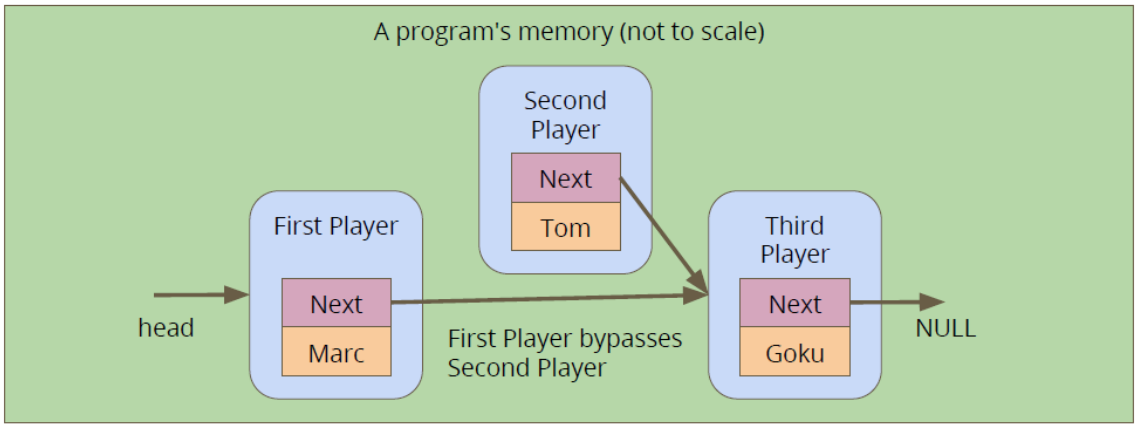
Removing a node – Player example

- We need to look through the list and see if a player name matches with 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.

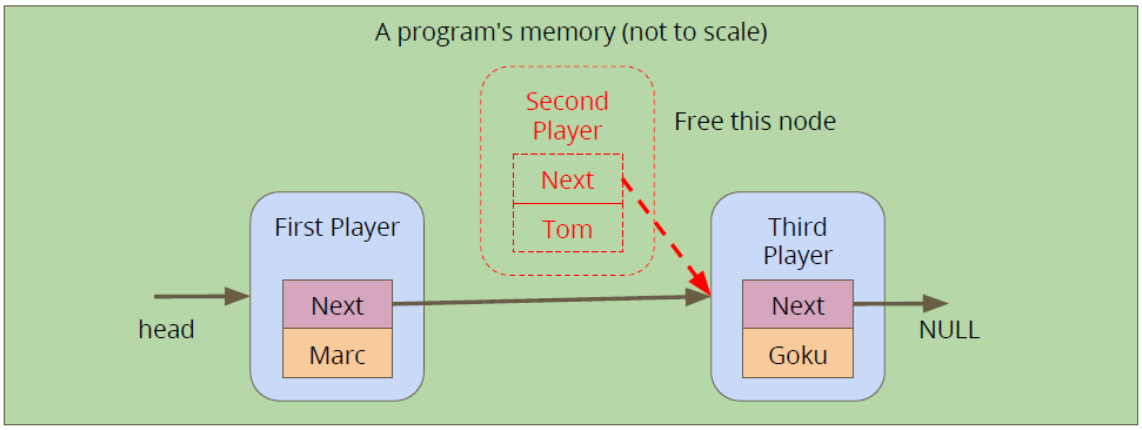
If we want to remove the Second Player



Alter the First Player's **next** to bypass the player node we're removing



Free the memory from the now bypassed player node



Finding the right player

- We are going to loop through the linked list until we find the correct match

```
struct player *removePlayer(char name[], struct player* head) {

	// Create a pointer aiming at the start of the linked list and
	// one at NULL.
	struct player *current = head;
	struct player *previous = NULL;

	// Keep looping until we find the matching name
	// The previous pointer tails behind the current pointer
	while (current != NULL && strcmp(name, current->name) != 0) {
		previous = current;
		current = current->next;
	}

	if (current != NULL) {

		// if current isn't NULL, we found the right player
		if (previous == NULL) {
			// it's the first player
			head = current->next;
		} else { // not the first player
			previous->next = current->next;
		}

		free(current);
	}

	return head;
}
```

NOTE! At `while (current != NULL && strcmp(name, current->name) != 0)`, it will first check if current is not NULL, then for the strcmp. If current is NULL, it will not do the next step. This is important since if it was the other way around, if current was NULL, it would try to first access current->name, leading to an error!

Game code – Players

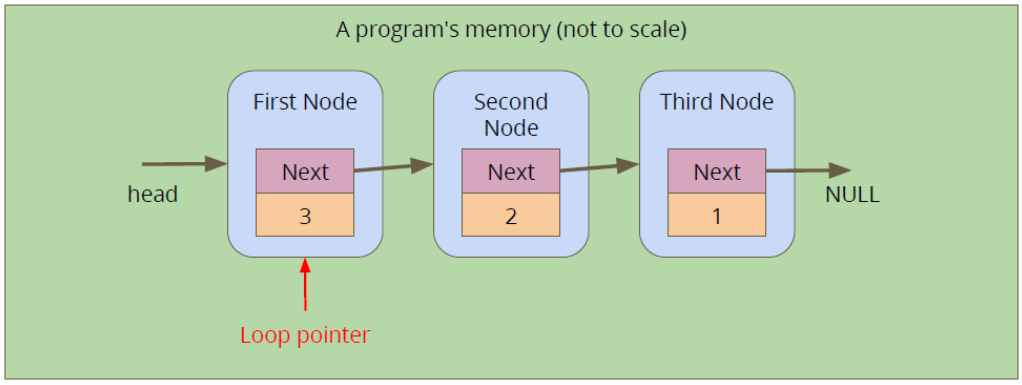
- Once our list is created, we can loop through the game
- We print out the player list (we might want to modify that function!)
- Our use will tell us who was knock out

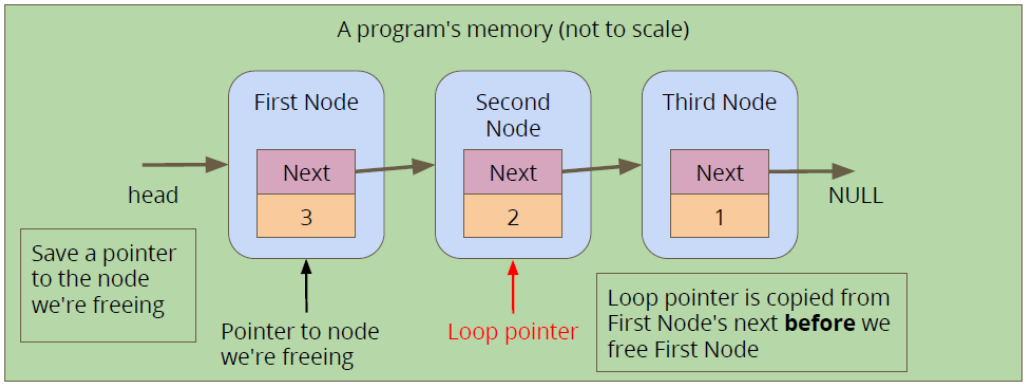
```
// A game loop that runs until only one player is left
while (printPlayers(head) > 1) {
	printf("Who just got knocked out?\n");
	char koName[MAX_NAME_LENGTH];
	fgets(koName, MAX_NAME_LENGTH, stdin);
	koName[strlen(koName) - 1] = '\0';
	head = removePlayer(koName, head);
	printf("----------\n");
}
printf("The winner is: %s\n", head->name);
```

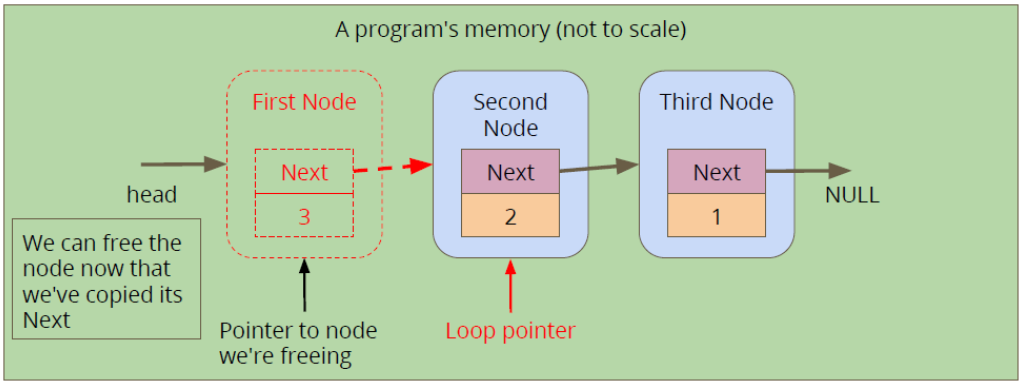
Cleaning up

- Remember, ALL memory allocated (malloc) needs to be freed
- We can run dcc –leak-check to see whether there's leaking memory
- This is a function that frees a whole link list. Just be careful not to free one that we still need the pointer from!

```
// 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 = head;

		// move the looping pointer to the next node
		n = n->next;

		// free the current node
		free(remNode);
	}
}
```







Week 9

Lecture 15:

- ✓ ***Abstract Data Types - Queue***

Recap on Multiple File Projects

- We can separate code into multiple files
  - Header file (`*.h`) which has the function declarations
  - Implementation file (`*.c`), where majority of the running code is for the functions
  - Other files can include the header to use its capabilities, such as `main.c`
- Separation protects data and makes functionality easier to read
  - We don't have access to internal information that we don't need
  - We can't change something that is important
  - We have a simple list of functions that we can call

Using Multiple Files

- To link them, a file that `#include`s 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! You will never see `#include "realm.c"`
- All implementation files are compiled and header files are never compiled, they're included
- For example, Assignment 2 – Castle Defence!
  - `realm.h`
    - Contains only defines, typedefs and function declarations
    - Is commented heavily so that it's easy to know how to use it
    - Does not have any `#include`s
  - `realm.c`
    - Contains actual structs
    - Contains implementation of realm.h's functions (once we've written them) so it `#include "realm.h"`
  - `main.c`
    - `#include "realm.h"`
    - Uses functions in `realm.h`
  - `test_realm.c`
    - `#include "realm.h"`
    - Is mutually exclusive with `main.c` because they both have main functions

Abstract Data Types (ADT)

- We can declare types for a specific purpose
  - We can name them
  - We can fix particular ways of interacting with them
  - This can protect our data from being accessed the wrong way
- We can hide the implementation
  - Whoever uses our code doesn't need to show how it was made
  - They only need to know how to use it

Typedef

- We declare a new Type that we are going to use
- **`typedef <original Type> <new Type Name>`**
- This allows us to use a simple name for a possibly complex structure and it HIDES the details of the structure in the `.h` file

```
typedef struct realm *Realm;
```

- We can use Realm as a type without knowing anything else about the struct underlying it
- Essentially: **`typedef <original_name> <alias_name>`**
- So if we have: `typedef struct realm *Realm`
- We making a function instead of writing: `struct realm *new_realm(void) {`
- We can write: `Realm new_realm(void) {`

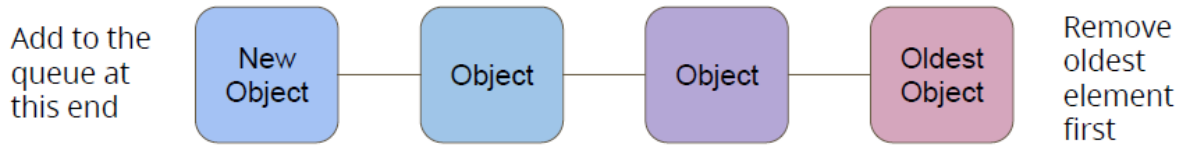
Typedef in a Header file

- We can put this in a header file along with the functions that use it
- This allows someone to see a Type without knowing exactly what it is
- The details go in the implementation 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

Example of an Abstract Data Type – A Queue

- It is just like a queue!
- New things join at the back
- The longest thing there will be the first to leave the queue



Why is a Queue Abstract?

- An array or a linked list is a very specific implementation
- A queue is just an idea of how things should be organised
- There is a structure, but there is no implementation

- We can have a header saying how the queue is used
- The implementation could use an array or linked list to store the objects in the queue, but we wouldn't know!

Building a Queue ADT

- Our user will see a "Queue" rather than an array or linked list
- We will start with a queue of integers
- We will provide access to certain functions:
    - Create a Queue
    - Destroy a Queue
    - Add to the Queue
    - Remove from the Queue
    - Count how many things are in the queue

The Header File for Queue

```
// queue type hides the struct that is is
// implemented as
typedef struct queueInternals *Queue;

// functions to create and destroy queues
Queue queueCreate(void);
void queueFree(Queue q);

// Add and remove items from queues
// Removing the item returns the item for use
void queueAdd(Queue q, int item);
int queueRemove(Queue q);

// Check on the size of the queue
int queueSize(Queue q);
```

- We don't know if the objects of the queue will be stored in a linked list or array! The user doesn't know!!!

What does our Header not provide?

- Standard Queue functions are available
    - We can join the end of the queue
    - We can take from the front of the queue
    - We can't take more than one element
    - We can't loop through the queue
- The power of Abstract Data Types stop us from accessing data incorrectly!

Queue.c

- This is the implementation file
- The C file is like the detail under the "headings" in the header
- Each declaration in the header file is like a title of what is implemented

- Let's use a linked list as the underlying data structure
- It makes sense to use a linked list because we can add to one end and remove from the other
- It also works because we can change the length of the linked list with no issues

The implementation behind a Type definition

- In the implementation file we can create a pair of structs
- queueInternals represents the whole Queue
- queueNode is a single element of the list

```
// Queue internals holds a pointer to the start of a linked list
struct queueInternals {
	struct queueNode *head;
};

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

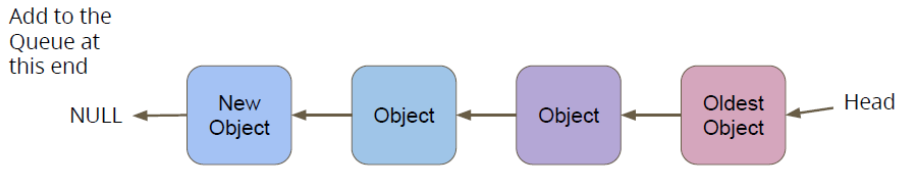
The first function – create a Queue

- We create our Queue empty, so the pointer to the head is NULL

```
// Create an empty queue
Queue queueCreate(void) {
	Queue newQueue = malloc(sizeof(struct queueInternals));
	newQueue->head = NULL;
	return newQueue;
}
```

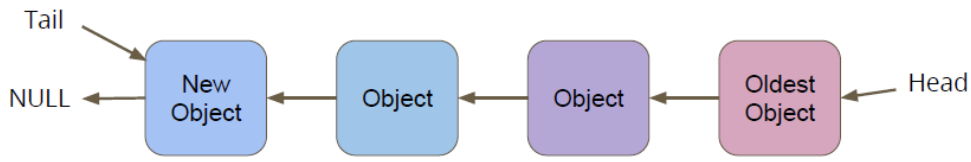
Another function – add an item to the Queue

- As per the requirements, we have to add at the end of the queue, i.e.
- So, we have to find the tail of the queue and add an element there every time

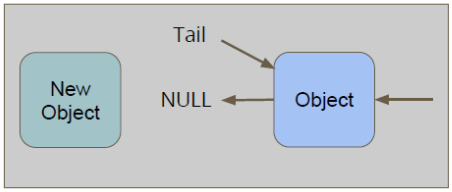


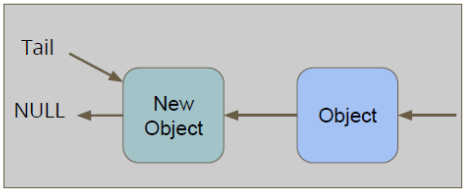
- This can be a little difficult, instead we should make a pointer that aims to the tail of the queue
- We will keep track of the last element in the list using our `queueInternals` struct

```
// Queue internals holds a pointer to the
// start and end of the linked list
struct queueInternals {
	struct queueNode *head;
	struct queueNode *tail;
};
```



- Once we know where the tail is, connect the new object to the current tail
- Then we move the pointer to the new last object
- We no longer need to loop through the whole queue to find the tail

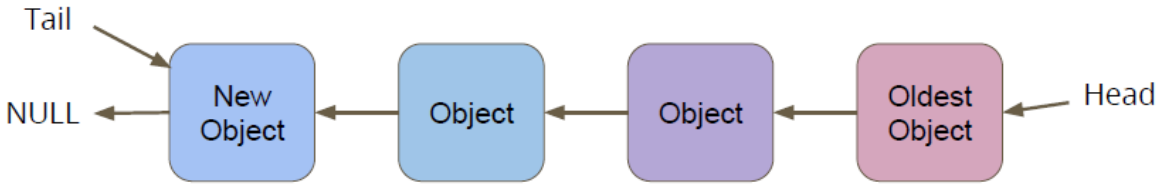


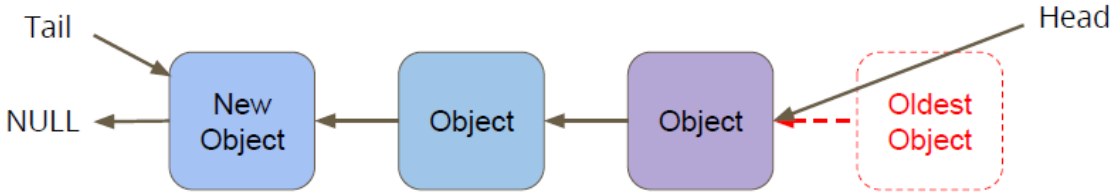



```
void queueAdd(Queue q, int item) {
	struct queueNode *newNode = malloc(sizeof(struct queueNode));
	newNode->data = item;
	newNode->next = NULL;
	if (q->tail == NULL) {
		// Queue is empty
		q->head = newNode;
		q->tail = newNode;
	} else {
		q->tail->next = newNode;
		q->tail = newNode;
	}
}
```

Another function – remove an item from the Queue

- As per requirements, we can only remove the element at the head of the list because it is the oldest






```
// Remove the head from the list and free the memory used
int queueRemove(Queue q) {
	if (q->head == NULL) {
		printf("Attempt to remove an element from an empty queue.\n");
		exit(1);
	}

	// Keep track of the old head
	int returnData = q->head->data;
	struct queueNode *remNode = q->head;

	// move the queue to the new head and free the old
	q->head = q->head->next;
	free(remNode);

	return returnData;
}
```

Testing Code in our Main.c

```
int main(void) {
        printf("Creating the Queue for Ice Cream.\n");
        Queue iceQueue = queueCreate();
        int id = 1;
        printf("Person %d joins the queue!\n", id);
        queueAdd(iceQueue, id);
        id = 2;
        printf("Person %d joins the queue!\n", id);
        queueAdd(iceQueue, id);
        id = 3;
        printf("Person %d joins the queue!\n", id);
        queueAdd(iceQueue, id);

        printf("Person %d just got their ice cream!\n", queueRemove(iceQueue));
        printf("Person %d just got their ice cream!\n", queueRemove(iceQueue));
        printf("Person %d just got their ice cream!\n", queueRemove(iceQueue));
        return 0;
}
```

Other Functionality

- There are some functions in the header we haven't implemented
- Destroying and freeing the Queue
    - We're still at risk of leaking memory because we're only freeing on Removal
- Display the Number of Elements
    - This would be very handy because it would allow us to tell how many elements we can remove before we risk errors
- For next lecture!

Lecture 16:

- ✓ ***Abstract Data Types – Queue (continued)***
- ✓ ***Another ADT – Stacks***

Another function - queueFree()

- To free all the memory in the linked list that we're using, we should loop through the list and free() each node as we go

```
// Destroy and Free the entire queue
void queueFree(Queue q) {
	while (q->head != NULL) {
		struct queueNode *current = q->head;
		q->head = q->head->next;
		free(current);
	}
	free(q);
}
```

Testing for memory leaks

- Use `dcc <normal compile stuff> --leak-check`
- Remember that all memory allocated with `malloc()` must be freed, including the queue itself!

One last function – the number of items in the Queue

- We could… loop through the entire list until the end and count how many elements there are in the queue

```
// Return the number of items in the queue
int queueSize(Queue q) {

	struct queueNode *iterator = q->head;
	int counter = 0;
	while(iterator != NULL) {
		counter++;
		iterator = iterator->next;
	}
	return counter;
}
```

- But there is a better way! We have a `queueInternals` that can store information, lets store the size!

```
// Queue internals holds a pointer to the start of a linked list
struct queueInternals {
	struct queueNode *head;
	struct queueNode *tail;
	int size;
};
```

So when we add or remove a node, we add or subtract 1 from this variable!
Using: `q->size++` or `q->size--`

Some thoughts on the Queue

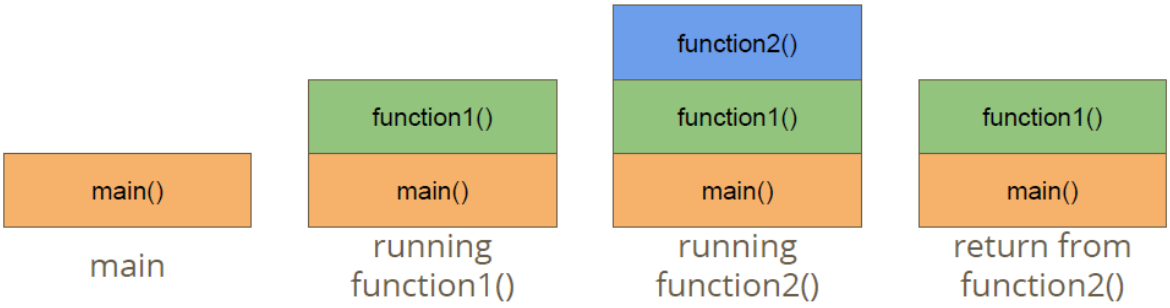
- When we use ADTs, we don't know (or need to know) the implementation. This queue could have been implemented using an array! Challenge!

Stacks – another Abstract Data Type

- 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

Stacks are how functions work!

- The current running code is on top of the stack
- When main() calls function1() – only function1() is accessible
- Then if function1() calls function2() – only function2() is accessible
- Control returns to function1() when function2() returns



Functions a stack needs

- Functionality to put in a header file
  - Create
  - Free
  - Push (add to the top of a stack)
  - Pop (remove from the top of the stack)
  - Top show the top without removing it
  - Size

A stack header looks familiar to Queue

```
// 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 queue
int stackSize(Stack s);
```

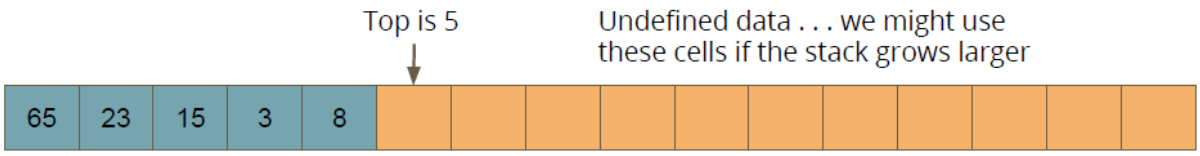
Implementation

- Let's use an array this time!
- We could use a linked list but it we're going to try something different
- Whichever it is, it should be invisible to whoever includes the stack.h file

Array Implementation of a stack

- So, our data will be stored in a large array with a maximum size
- We'll keep track of where the top is with an int

- The top is a particular index and it signifies where our data ends
- It also happens to be exactly the number of elements in the stack!

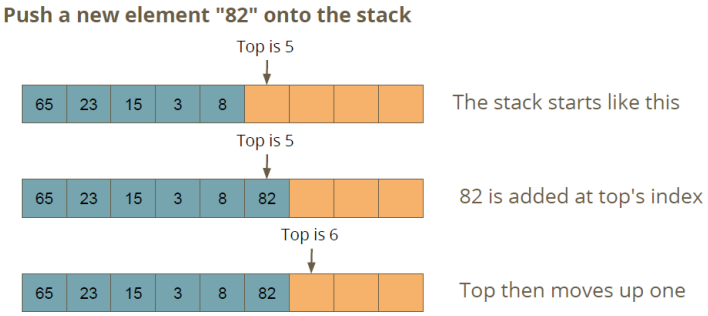


stack.c

```
// Struct representing the stack using an array
struct stackInternals {
	int stackData[MAX_STACK_SIZE];
	int top;
};

// create a new stack
stack stackCreate() {
	stack s = malloc(sizeof(struct stackInternals));
	s->top = 0;
	return s;
}
```

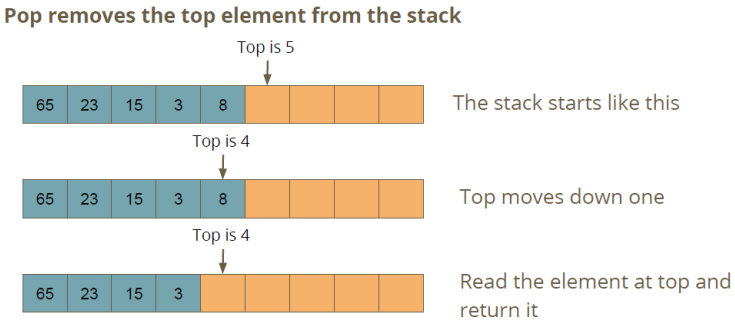
Push and Pop

- Push ADDS an element at the top of the stack
- It will move the top index to the new element




```
// 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++;
}
```

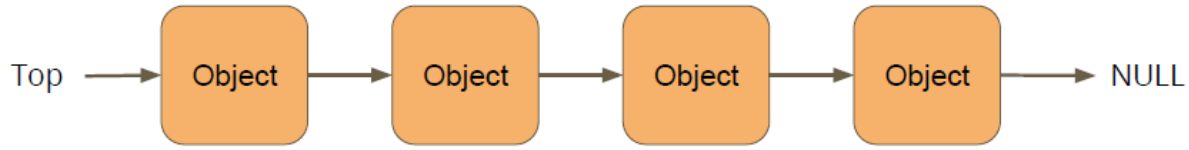
- Pop REMOVES an element from the top of the stack and returns the element at the top of the stack
- It will move the top index down one




```
// 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];
}
```

What if we implemented a stack with a linked list?

- Remember implementation is invisible to the user
- But let's make a stack with a linked list!
- We'll add elements to the list and we'll also remove elements from the list, from the same end!

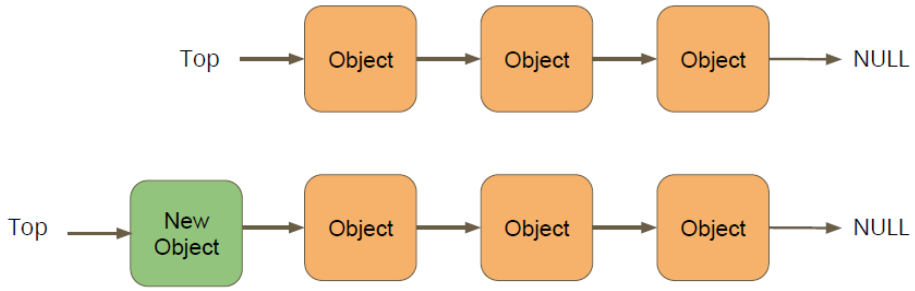


Stack with a Linked List

```
struct stackInternals {
	struct node *top;
};
struct node {
	struct node *next;
	int data;
};
stack stackCreate() {
	stack s = malloc(sizeof(struct stackInternals));
	s->top = NULL;
	return s;
}
```

- Push ADDS an element to the top of the list
- Top will then point at that element

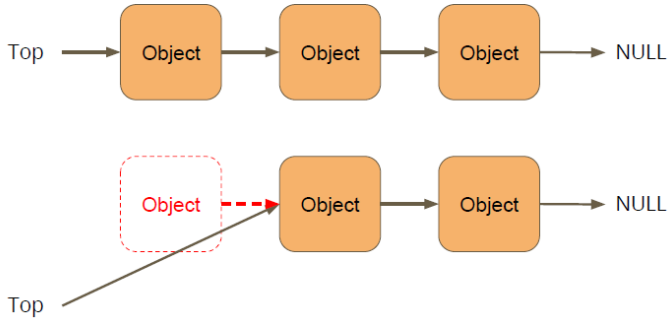
Add a node to the top of the list




```
// Add an element on top of the stack
void stackPush(stack s, int item) {
      struct node *n = malloc(sizeof (struct node));
      if (n == NULL) {
            printf("Cannot allocate memory for a node.\n");
            exit(1);
      }
      n->data = item;
      n->next = s->top;
      s->top = n;
}
```

- Pop removes the top element of the list and returns it
- Top will then point at the next element

Remove the node from the top of the list




```
// Remove the top element from the stack
int stackPop(stack s) {
      if(s->top == NULL) {
            printf("Stack is empty, cannot pop.\n");
            exit(1);
      }
      // keep a pointer to the node so we can free it
      struct node *n = s->top;
      int item = n->data;
      s->top = s->top->next;
      free(n);
      return item;
}
```

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

You’VE COMPLETED COMP1511! 😊