HS1917 Course Textbook

Lecture00

Problem solving

One strategy for tackling intense problems that seem overwhelming (especially in computing) is the **"Try It"** strategy: instead of contemplating and theorising strategies, algorithms and solutions to problems, you actively experiment with different solutions and methods.
When you have solved a problem one way you should always try as many different approaches to the problem as possible. There isn't always a correct or incorrect answer so you should always take the opportunity to learn more by attempting to find different solutions.

Abstraction

Abstraction is when we represent essential features without including the complex details or explanations. In the computer science area, the abstraction principle is used to reduce complexity and allow easy and efficient design and implementation of software systems.

Examples (from high levels of abstraction to lower ones):
Richard is a:

- University professor
- Human
- Collection of organs, flesh and blood
- Collection of cells
- Collection of organic molecules
- Collection of atoms
- Collection of subatomic particles

A book is a set of:

- Ideas
- Chapters
- Sentences
- Words
- Letters

While lower levels of abstraction may give more information, it may not be the most useful. For example to a student it'd be useful to think about Richard as a university professor but to a surgeon operating on his body, it'd be more useful to think about him as a collection of organs, flesh and blood. It'd be useless for a surgeon to know that Richard is a university professor.

Microcontrollers

A microcontroller is a small computer which is capable of running machine code. They were introduced to us as the (fictional) 4000, 4001, 4002 and 4003 microcontrollers.

Each microcontroller has 16 bytes for machine code storage, an Instruction Pointer(IP), an Instruction Store(IS) and up to two General Registers(R). All memory in the microcontroller starts at 0.
The IP tells the microcontroller where it is up to. The program starts at instruction 0, and copies the instruction at position zero to the IS, which stores the instruction that needs to be run. This instruction is then run, and the IP is incremented. This process repeats until the program is ended by exiting an exit instruction, (0). The general registers act as the variables of a program, and can be changed using specific machine codes.

Machine code:
Machine code tells the microcontroller what to do. Different microcontrollers use different codes for different functionality. Examples of this are:

- +1 to general register 0
- Swap contents in registers
- +2 to general register 1
- print general register 0
- play bell
- exit program

These are all 1-byte registers, but two-byte registers exist on the 4003

- jump to index <x> if general register 0 != 0
- jump to index <x> if general register 0 == 0

Lecture01

Abstraction

Different levels of abstraction provide different information about the same thing.
Example:
A bagel is a:

- Bagel - useful for eating
- Group of ingredients - useful if you have allergies, dietary restrictions, &c.
- Wheel
- Nutritional content - useful for diets, dietary requirements
- Collection of atoms

Emergence: the whole is greater than the sum of the parts.

Microprocessors

Some useful information about microprocessors:

- Loops through cells executing each the instruction in each cell
- There are 16 memory cells with values from 0-15 (which wrap around: 15+1 = 0)

- IP - instruction pointer - stores cell number of current instruction
- IS - instruction store - stores value in cell of current instruction

Some more detailed information about the 4000-4003 microprocessors:

- 4000 series:
    - Blank - halt
    - X - beep
    - O - pause
- 4001 series:
    - Added general register R
    - 0 - halt
    - 1 - R = R+1
    - 2 - R = R+2
    - 3 - R = R+4
    - 4 - R = R+8
    - 7 - Print R
- 4002 series:
    - 0 - halt
    - 1 - R = R+1
    - 2 - R = R-1
    - 7 - Print R
- 4003 series:
    - Two general registers: R0 and R1
    - 0 - halt
    - 1 - R0 = R0+1
    - 2 - R0 = R0-1
    - 3 - R1 = R1+1
    - 4 - R1 = R1-1
    - 5 - Swap R0 <-> R1
    - 6 - Beep
    - 7 - Print R0
    - *Two-byte instructions*
    - 8 - Jump to <address> if R0 != 0
    - 9 - Jump to <address> if R0 == 0

Little details of a system combined to create a larger sum is called emergence (as described above).

- O's, X's and a 3x3 grid can make a game (O's and X's)
- There must be inputs and processing to create a new output.
- Bread (input) and a toaster (processing in the form of heat) will create toast (output)

Very clear and precise instructions must be used for writing code. This is clear in the Battenberg cake recipe example, where unclear instructions are misinterpreted in a humorous way.

Lecture02

4003 Adder

The 4003 doesn't have a R1 + R2 instruction so you have to do a loop instead.
In pseudo code:

```
while r0 != 0: r0--, r1++
```

This moves "items" from one register to the other, leaving the register where they all end up being equal to the sum of the two registers originally.
As long as you move things around they should be the same. You can swap registers so you can just move it around.

Writing lecture notes

Each week, a group of people are responsible for writing lecture notes. Work should be divided equally among those who are writing it. **Most importantly, you should be flexible. If other people want to make contributions, let them so and merge yours with theirs in a seamless way.**

Logistics

The biggest challenge is avoiding merge conflicts (when one person submits and overrides the work of someone else, if they were editing at the same time). It may be a good idea to use a Google Doc or similar service to collaborate on the notes. You can add collaborators by clicking the blue share button (they need Google accounts). All changes are saved to the cloud in real time, and multiple people can edit the document at once. You can use notes and the built-in chat (or another service such as Hangouts) to communicate. **Note: remember to allocate someone to actually copy the notes to the Open Learning site before the deadline.**

Structure

It is a good idea to use headings to structure different ideas. Avoid a wall of text where possible and try to use bullet points and similar structuring techniques. Also, assign a subheading to each person writing the notes.

Length

There is no set length. You should be concise but write as many details as is helpful. Remember, taking shortcuts will only be difficult later.

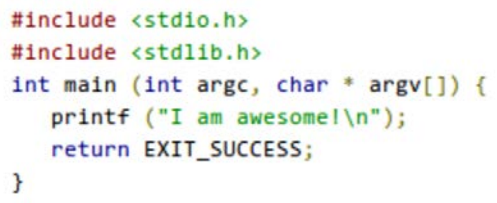
Lecture03

Information

C Programming Language: C is a high-level and general-purpose programming language, typically used for developing firmware or portable applications.
Machine Code: is a computer language that is directly understandable by a computer's central processing unit (CPU). It is the language into which all programs must be converted before they can be run.

C Syntax

In the first week, we learnt some basic C syntax. Here is an example program:



```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char * argv[]) {
   printf ("I am awesome!\n");
   return EXIT_SUCCESS;
}
```

- “#include” is used to include pre-written code into your program. The two files being included are:
  - stdio.h: (standard input output) is a built-in file which contains the printf function (among other things), which you use to output text
  - stdlib.h: (standard library) is a built-in file with a range of variables and functions including the constant EXIT_SUCCESS used to return a flag which indicates that the program ran as intended
- “int main” is a function which runs automatically. It takes some arguments as inputs when running the file. When the function is run, the indented code within is executed, printing out to the screen “I am awesome” exiting with EXIT_SUCCESS (see above). The text “\n” indicated to go to a new line in the terminal.
- Indenting is used to show structure.

Remember to:

- Put a space after commas (“1, 2, 3”)
- Indent code within functions, if statements etc.
- Put semicolons (“;”) at the end of all lines except for lines with “#include”

Compiling

After creating the file and editing it in a program like Nano or Gedit, the program needs to be compiled. This converts the C code to a lower level language.
The command used is:
“gcc -Wall -Werror -O -o file_to_output file_to_input.c”

- The -Wall flag tells the compiler to put up all warnings, while -Werror converts them to errors and aborts the program. -O turns on the optimiser.
- The file file_to_input.c is an example of a C file, and file_to_output is an example of the name of the outputted file. To run the compiled file, "./file_to_output" is used.
- Errors during compilation are normal and expected. Errors that occur early in the program may cause errors in lines that follow, even if they are correct. As a result, errors should be read from top to bottom.

Problem solving

- Scope out problems and find important info.
- Try find smaller, easier to solve problems to solve harder problems.
- Possible ways to check answers:
    1. Compute the answer using two different methods and compare your results.
    2. Break down the problem into smaller sub-problems to make the answer more reliable because you understand each step better.
    3. Desk check - go through your solution with a pen and paper to make sure that the program makes sense and works.

Lecture04

Compiling, Eventually

C coding

extensions	use the .c extension for c programs e.g "MyProgram.c"
comments	use // for comments e.g. `//Program created on the 3rd of March 2017 by Marcus Handley` comments are ignored by the compiler and are used for human readability use header comments at the beginning of programs documents the date of creation, the author(s) and what the program does
main function	use `int main(int argc, char * argv[]){ <CODE> }` for the main function. Richard promises he will explain this eventually...
returning	the main function returns an integer indicating whether the program worked.

	As this can vary by systems when can include <stdlib.h> and use the predefined values of EXIT_SUCCESS and EXIT_FAILURE which will be replaced with the correct number at compile time
case	in C case does matter, e.g EXIT_SUCCESS is different to exit_success a good convention to follow to avoid this is make all constants in full uppercase and with underscores separating words.

including files	we can use "#include <FILENAME> to use prewritten code so we don't need to write it ourselves. some common uses are
	
	`#include <stdio.h>`
	
	and
	
	`#include <stdlib.h>`
	
	lines beginning with # are pre-processor statements

compiling	when you compile a program you transform it from human readable code(in our case C) to directly executable machine code. the compiler used for C is gcc as is normally run with "gcc -Wall -Werror -O -o <newFile> <codeFile> e.g `gcc -Wall -Werror -O -o hello HelloWorld.c` **Don't forget to save your code before compiling**
running code	to run compiled code use "./<file>" e.g for above we would run `./hello`
printing	use printf from stdio.h to make text appear when the program is run pass to printf first a string containing what the format of what you are going to print and afterwards the additional data, in order of which it appears in the string we currently only know of %d which is for integers for example we could run `printf ("my favourite number is %d \n", 400);` \n causes a newline to be created

if statements	if we want to run code only if some condition is true we can use an if statement
	
	the syntax is
	
	`if (condition){`
	`    codeToBeRun`
	`}`
	
	for example
	
	`if (x > 9000){`
	`    printf("%d is over 9000!\n", x);`
	`}`

Other

2 Big problems in computing	• How can multiple people work on the same piece of code at the same time? What problems can arise? • The other to be discussed later
Debugging	Don't be intimidated by big walls of error messages! Chances are all of the messages are just caused by one mistake Work your way down the block of text If you really can't work out what wrong eat a bagel and try again in five minutes

Lecture05

Include

What `#include` does is when the program is compiled, the computer searches for the file/library which is defined by the *#include* and replaces the `#include`with what is in that file. The syntax of `#include` is as follows:

```
#include /*insert file/library name*/
eg1. #include <stdio.h>
eg2. #include "file.c"
```

The file/library that is included can be any of your choosing. What will happen when the program is compiled is that the `#include <stdio.h>` will be replaced with the contents of the library <stdio.h>.

Note. Libraries need to be surrounded by `< >` and files by `" "`.

If & Else

`if` are used in C to complete a task when a certain condition is met. If the condition is not met, the program will skip the task within the if statement.

`else`s are used in conjunction with `if`, where if the `if` condition is not satisfied it will do something "else".

Example:

```
if (/*insert condition*/) {
  /*insert something to do*/;
} else {
  /*insert something to do*/;
}
eg. if (magicNumber > 10) {
   printf ("your number is big!");
} else {
   printf ("your number is small!");
}
```

The condition can be of anything of your choosing.

Basic operators are:

- Equal to (=)
- Not equal to (!=)
- Greater than (>)
- Greater than or equal to (>=)
- Less than (<)
- Less than or equal to (<=)

It is possible to put `if` within `if` statements.

Structured Code

Always follow HS1917 style guide. This helps readability of code.

- Less than 72 characters per line.

Glossary of Terms:

Int = Integer

Escaping = A special command you just want to say rather than preform the action

Parenthesis {} = Grouping things together

If = If meets requirements will do
Else = If it is not applicable then else happens.
Structured Programming = Write programs in a structured way. Essentially a common protocol.

Lecture06

Binary Numbers

Binary numbers are a sort of 'yes/no 'function. It includes the numbers 1 and 0, using which computers take commands.
To represent numbers in binary (besides 0 and 1), each space represents a power of 2 read from right to left, with 0 registering as no input and 1 registering as input. The inputed digits are then summed up to create the larger number.

2^6 64	2^5 32	2^4 16	2^3 8	2^2 4	2^1 2	2^0 1	Output Number
		1	0	0	1	1	19
		1	0	1	0	0	20
	1	0	1	0	0	0	40

10011 = 19 (16 + 2 + 1)

FUNCTION: Assert

To make sure an input is valid within a program, use the "assert" function

```
#include <assert.h>
 assert (rule)
```

e.g assert (year >= 1582)

If an input does not satisfy the rule, the program will stop show an error and stop functioning.

FUNCTION: Define

To make your program clearer, define variables so they appear as something everyone can understand.

e.g. By itself, no one knows what 1582 means or stands for. If the value is replaced with "START_OF_GREGRORIAN_CALENDAR," people will understand that 1582 is the start of the Gregorian calendar

We do this by writing at the very start of the program after #include:

#define START_OF_GREGORIAN_CALENDAR 1582

Now, instead of typing 1582 throughout the code, START_OF_GREGORIAN_CALENDAR is written instead as a clearer alternative, but still stands for the same value.

FUNCTION: Int

"int" creates a memory bank for a specific variable, which can then change later.
e.g. "int year" creates a variable called 'year,' which can then be changed to a value later. For example, "year == 242" will change the 'year' variable to 242.

Other notes

- Keep your programs simple and clear, as errors will be easier to spot
- Avoid repetition in the program to make it neater. Use other commands to represent repeated things.
- Do not have 'magic numbers,' which are random numbers (that are constant) in the code where their significance is not known. (1582 instead of START_OF_GREGORIAN_CALENDAR).

A constant is a number that does not change

Lecture07

Functions will always produce the same output when given the same input

- property 1 does the same thing every time
- property 2 one output variable number of inputs
- property 3 legal input = legal output (if not = partial function)

Computable Function

A property 1 output can be computed. As stated by the Church-Turing hypothesis, if it can be computed one way then it can be computed any way.

Common Define Problem

#define (insert name) = (insert defined) ;
Everything after "(insert name)" is defined, including the "=" and ";".

- "!=" means "is not equal to"
- "&" (ampersand) refers to the "address of" a variable (i.e. where that variable is found)
- "x++" is the same as "x + 1"

TIP: break hard problems into smaller, easier problems (thanks Richard !!)

Doomsday Problem

Each year has a doomsday:
e.g. Monday in 2011, Tuesday in 2012, Wednesday in 2013 etc. The doomsday for this year (2017) is Tuesday.
The doomsday is defined by the last day of February of that year (also referred to as "March 0", as in the day before the 1st of March), i.e. if the 28th of February on a non-leap year is a Wednesday, then the doomsday for that year is Wednesday. Conversely, if the year is a leap year and the 29th of February falls on a Friday for instance, then the doomsday for that year is Friday.
Below is a list of doomsdays that apply to each year (stated in the American dating system of "month/year"):

- 1/3 (common years), 1/4 (leap years)
- 2/28 (common years), 2/29 (leap years)

The easier ones to remember are:

- 0/3 (as mentioned above, this refers to the last day of February)
- 4/4
- 5/9
- 6/6
- 7/11
- 8/8
- 9/5
- 10/10
- 11/7
- 12/12

An easy way to remember the dates for odd months is my the following mnemonic:
"I work 9-till-5 at the 7-11"
i.e. the 9th of May / 5th of September and the 11th of June / 7th of November
Normally, the doomsday increases by 1 day each year, unless the year is a leap year, in which case it increases by 2.

Church-Turing Hypothesis

- The Church-Turing Hypothesis is a hypothesis about the nature of computable functions
- Implies that, ignoring memory and time limits, any modern computer can compute all computable functions, and so could computers back in the 1970s

Lecture08

Doomsday Method

The Doomsday Method is a way to work out what day of the week it is when given any date of the year. The method relies on that each year will have a Doomsday. The following days will all be the Doomsday of that year:

- *3/1 on normal years. 4/1 in a leap year*
- *0/3*
- *4/4*
- *9/5*
- *6/6*
- *11/7*
- *8/8*
- *5/9*
- *10/10*
- *7/11*
- *12/12*

Mnemonic

Date	Mnemonic
3/1 (normal), 4/1 (leap year	The third three years and the fourth of the fourth year
0/3	The last day of February
4/4	4/4, 6/6, 8/8, 10/10, 12/12
9/5	I work 9/5 at the 7/11
6/6	4/4, 6/6, 8/8, 10/10, 12/12
11/7	I work 9/5 at the 7/11
8/8	4/4, 6/6, 8/8, 10/10, 12/12
5/9	I work 9/5 at the 7/11
10/10	4/4, 6/6, 8/8, 10/10, 12/12
7/11	I work 9/5 at the 7/11
12/12	4/4, 6/6, 8/8, 10/10, 12/12

Example

If the doomsday for 2011 is MONDAY then the dates:

0/3, 4/4, 9/5, 6/6, 11/7, 8/8, 5/9, 10/10, 7/11, 12/12 will all fall on a Monday.

Working Out The Day

1. Find the nearest doomsday
2. Find difference in days
3. Find remainder of difference when divided by seven
4. Remainder will be the amount of change from the doomsday

Example

If the doomsday for 2011 is MONDAY, find out the date of 10/04/2011.

The nearest doomsday is 4/4, the difference is 6 days, therefore there has been a change of 6 days from Monday. Hence, the date is Saturday.

Doomsday Pattern

- Doomsday will increase by 1 every normal year
- Doomsday will increase by 2 every leap year
- Doomsday of 2017 is Tuesday

Example

If the doomsday for 2011 is MONDAY:

Therefore the doomsday for 2012 will be on Wednesday because it is a leap year, and doomsday for 2013 will be Thursday as it is a normal year.

How should I approach this:

The task at hand may seem quite daunting, but you should approach this like eating an entire elephant. You wouldn't eat an entire elephant all at once, just relax and work on the assignment with a clear head and willingness to accept failing as not the worst outcome to possibly arise. Consider how you would eat an entire elephant step by step. Make sure it's dead, maybe dice it into 12 equal parts. Call those parts each month of the year and cook those individual parts. Perhaps follow it through by eating each part every day until the elephant is fully digested. Approach doomsday and all problems like this to achieve greater confidence in your awesome abilities.

Lecture09

Bits, Binary and Hexadecimals

- Each higher digit in these base systems increases in powers of their amount of "numbers" in the counting system
- Each digit's value is then multiplied by its respective increment within the number
  - For example binary has two numbers in the counting system (0 and 1) and each digit then increases by powers of two
    - Example: The number 1111 can be broken down into

Bits

- Short for Binary Digit
- Smallest unit for storage on a computer
    - Has a value of 1 or 0
- Designed to store data and execute instructions

Binary

- Base Two System
- Binary has only 2 digits (0 and 1)

Hexadecimals

- Base Sixteen System
- Has values of:

Decimals	Hexadecimals
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Converting between Different Base Systems

Note: This is only one way to convert between different base systems:

1. Find the numerical value in decimal value (normal base 10 system)
2. Find the multiple of the largest power of the base system that can go into number and find the remainder

3. Record that digit and find the largest power of the base that can go into the remainder, recording all the digits in between that do not go into the new number as 0
4. Repeat process until number in that base system is found
5. Record number in order of highest powers

Example:

Convert the binary "1110100111" to hexadecimal

- Convert binary to decimal

$$1110100111 = 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 1 \times 2^5 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 935$$

- Find multiple of largest power of 16 that goes into 935 and record it

$$935 \div 16^2 = 3 \ \ (remainder\ 167)$$

Highest multiple is 3×16^2

- Find multiple of largest power of 16 that goes into 167 and record it

$$167 \div 16 = 10 \ \ (remainder\ 7)$$

Highest multiple is 10×16

- Find multiple of largest power of 16 that goes into 7 and record it

Highest multiple is 7×16^0

- Record number in order of highest powers

$$3 \times 16^2 + 10 \times 16^1 + 7 \times 16^0 = 935$$

Number must be **3A7**

Lecture10

Memory & Types

- Each memory cell is referred to as a byte
- The meaning of the value stored in each memory cell is dependent on how it is used
- In the C language, the way you are going to use something is determined by its type

Integers

Int

- Integers are defined by the "int" keyword
- They are commonly 4 bytes (32 bits) in length, although this is dependent on the chip
- They are used to store whole numbers

Signed & Unsigned Integers

- By default, integers in C specified by the keyword "int" are "signed"
- This means that they can't store the highest value possible
- In signed integers, 1 bit determines the sign while the rest determine the actual value
- In the case of 4 byte signed integers, the range of values are: - 2^31 through 2^31-1
- Unsigned integers in contrast don't have a sign and only have a magnitude
- In the case of 4 byte unsigned integers, the range of values are: 0 through 2^32-1
- Unsigned integers are defined with "unsigned int"

Short

- Shorts are defined with the "short" keyword
- They use less memory and are commonly only 2 bytes in size
- Can be unsigned

Long

- Longs are defined with the "long" keyword
- Longs will generally use more memory and are at least the same size as an integer, but can be larger
- They can store larger numbers than integers
- Can be unsigned

Floating-point Numbers

- **Can't be unsigned**

Float

- Floats are used for precise numerical values with a decimal point
- Floats are generally 4 bytes (32 bits) in size
- 23 bits are used for the actual value
- 1 bit is used for the sign
- 8 bits are used for the exponent which determines where the decimal point is to be placed

Double

- Defined with the "double" keyword
- Doubles are more accurate, larger and take up more memory than floats (typically 8 bytes)
- Doubles should be used in preference over floats.

Char

- Chars are used for storing single characters (letters, digits, symbols)
- They are 1 byte in size
- Each character is given a code within the range of 0-255 by the standards of ASCII

- Another standard called Unicode is used for characters outside the 256 bit block (like other spoken languages)

Booleans

- A Boolean is a type used to store true/false values
- While it is technically available in C, it is not covered in the HS1917 course
- Instead we are meant to use integer values of 1 and 0 for true and false, respectively, by HS1917 code conventions
    - This is done by defining TRUE as 1 and FALSE as 0

Lecture11

Importance in Relation to Problem Solving

- When solving a problem, stress will make you flustered
    - it helps to break the problem down into tinier sub-tasks, allowing you to focus on easier problems, so don't just tackle the entire problem head-on!
- Use the stress well to plan ahead, rather than reacting badly
- If you're overcome with stress, take a break, do the best you can
    - utilise this break to think about the problem in the back of your mind, and this is a strategy many professional programmers already use. Sometimes the solution just comes up when you're taking a stroll and being relaxed
- The hardest hump is the first task - **don't give up**
- **reminisce** the feeling of solving problems and reaching the solution. It's a wonderful feeling and it should motivate you more
- The real challenge of the first assignment is managing your emotions and mental state
- Why do we procrastinate? Keep this in mind, blog about it
- You should spend 3 hours a week practising programming
- Being an engineer/problem solver will make the world a better place

Lecture12

Commenting in C

- *"//"* for single-line comments
- *"/* comment*/"* for multi-line comments. Make sure to enclose it otherwise all code below the comment becomes a comment

- You can utilise these comments to temporarily cancel out code that you aren't using, so that the compiler won't give you a warning for unused functions or variables etc.

Lecture Example 0

```
/*
*Richard Buckland
*lecture examples on types and variables
*week 3 comp1917 2011s1 UNSW
*/
 #include <stdio.h>
#include <stdlib.h>

int main (int argc, const char * argv[]) {
    printf ( "Lecture 3 examples!\n" ); // this is a comment
    printf ( "EXIT SUCCESSS IS %d\n" , EXIT_SUCCESS);
    int x;
    x = 1234;
    printf ( "x is %d" , x);
    printf ( "Example done\n" );
    return EXIT_SUCCESS;
}
```

- **Why do we write #include <stdlib.h> in the header?**
    - Because we are using the pre-defined value of EXIT_SUCCESS from the stdlib.h header file, which is used to indicate program success
- **int x**
    - stores some memory aside for the variable x
- **printf ("EXIT SUCCESS IS %d\n" , EXIT_SUCCESS)**
    - It doesn' t print the %d, instead it is replaced by the integer
    - Because not every success is 0 in every system/machine, so we write return EXIT_SUCCESS instead of return 0
    - EXIT_SUCCESS has a meaning, but 0 is only an integer
- **Each bytes can store 256 digits**
    - When want to store 1234

256^1	256^0
4	210

210 x (sets of *256^0 or 1*) + 4 x (sets of *256^1 or 256)* = *1234*

- **When want to store 12345678901**
    - Warning: overflow in implicit constant conversion
    - You should use a larger data type, such as *unsigned long long*

Lecture Example 1

```
/*
Richard Buckland
lecture examples on types and variables
week 3 comp1917 2011s1 UNSW
*/

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

// function prototypes
void exampleOne (void);
void exampleDouble (void);
void exampleHex (void);

int main (int argc, const char * argv[]) {
    printf ( “Lecture 3 examples!\n” ); // this is a comment
    exampleOne();
    exampleDouble();
    exampleHex();

    printf ( “Example done\n” );
    return EXIT_SUCCESS
}

void exampleOne (void) {
    printf ( “Example 1\n” );
    printf ( “EXIT_SUCCESS is d%\n” , EXIT_SUCCESS);
    return;
}

void exampleDouble (void) {
    double height = 1.84;
    printf ( “your height is %f\n” , height);
}

void exampleHex (void) {
    long height = 1234;
    printf ( “your height is %lx\n” , height);
}
```

- **void exampleOne (void)**
  - pseudo: *return type   function name   (arguments)*

  - in this function declaration, we are saying that it **does not return a value**, its name is **exampleOne**, and **it takes no arguments**. For better readability, you should use void if the function doesn't take any arguments
- **Ensure that functions operate**
  - Method 1 (**not recommended, probably against style guide)** – place all functions before the main function so that you don't need to write function prototypes
  - Method 2 (**this is how it should be done!)** – declare function prototypes above the main function, and then place the functions below the main function, so that the actions that the program performs can be more feasibly interpreted (through the main functions and function calling)

Lecture Example 2

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

// void exampleDouble (void);
void exampleDouble (double height);

int main (int argc, const char * argv[]) {
    printf ( “Lecture 3 examples!\n” ); // this is a comment
    double height;
    height = 2.31;
    exampleDouble(height);
    printf ( “Example done\n” );
    return EXIT_SUCCESS
}

void exampleDouble (void) {
    printf ( “your height is %f\n” , height);
}
```

- **Error: conflicting types for “exampleDouble”**
  - The function prototype does not match the function definition, don't do this

Functions, scope and side effects

- The area in which a variable is defined is called its '<u>scope</u>'
- The scope of every variable defined in a function is that function itself
- All functions have <u>side effects</u>, e.g. the function 'scanf' has side effects

Lecture13

- Break big functions into smaller functions to make it simpler, it's hard to check
  - General rule: your function should never be too long, it should never go off the screen, it should only do one thing. If it does, break it up into sub functions
  - Do as little as possible in one step, 1 or 2 is good, 3 is too much and 4+ is bad
  - You should always define true and false if you use them
  - A function can have different input types (could input a boolean and an int and still return an int)
  - Give good variable names
- If you have completed a code with one long function, break it up into smaller functions anyway --> complexity leads to errors
- Only one action should be completed in one line
- First thing passed in brackets is assigned to first definition, regardless of matching names
- variables only live as long as the functions
- when you call a function, a functions sets aside some memory and it copies the stuff you give it into that memory and at the end of the function, throws it away

Lecture14

Data/bits (0s and 1s) in a computer must be interpreted to be given meaning.

A bitmap (BMP) file stores an image using bits (0s and 1s).

Eight bits are grouped into one byte, which can be represented as a two-digit hexadecimal character.

BMPs can be viewed with a hexadecimal editor to view and modify this data.

Structure

Bitmap file header:
First 14 bytes.
Stores general information about the file

- Bytes 0 - 1
  - Header field to identify the BMP file
  - On Windows this is BM (Ascii) or 42 4D (Hexadecimal)
- Bytes 2 - 5
  - Size of the BMP file in bytes
  - Just for fun, this means the maximum size of a BMP file is 255^4 = 4228250625 bytes = 4.2GB
- Bytes 6 - 7

    - Reserved
- Bytes 8 - 9
    - Reserved
- Bytes 10 - 13
    - The starting address of where the actual image data (pixel array) is stored

DIB header

Stores data which allows the file to be initialized (how colour data should be interpreted)

- Bytes 15-18: Size of DIB header
    - Without this, since there are multiple standards for DIB headers, computer won't know which standard its using.
- Bytes 19-22: Width of bitmap in pixels
- Bytes 23-26: Height of bitmap in pixels
- Bytes 27-28: Number of colour planes used (must be set to 1)
- Bytes 29-30: Stores bit depth (number of bits used to store colour data in a single pixel).
- Bytes 31-34: Stores compression method being used to store this bmp.
- Bytes 35-38: Image size (size of raw bitmap data)
- Bytes 39-42: Horizontal resolution of image
- Bytes 43-46: Vertical resolution of image
- Bytes 47-50: Number of colours in colour palette
- Bytes 51-54: Number of important colours (generally ignored)
- **Actual colour data**

Each pixel is encoded by 3 bytes: Blue, Green and Red respectively, each ranging from 0 - 255.

Maximum of 256^3=16777216 colours for a single pixel, since each byte can store 256 values.

Note that each row of pixels must be divisible by 4. Padding must be added to account for this rule. For example in the case of a 3x3 BMP, each row has 3 pixels (9 bytes), which is not divisible by 4. So an extra "padding" pixel (3 bytes) is added at the end of each row.

Also, the pixels of a BMP file are encoded row by row, from bottom left to top right.

Lecture15

- **Very important** - A function does not actually modify the original input variable if you just feed it standard input variables. It creates a local copy of them, and then discards them once the function's done.
- Now, if you give the function the address of something, then it will modify the original variables.

- Because of the nature of integers and signed/unsigned numbers, when a number goes over the maximum value allowed, say 31^2-1 for a signed 4-byte int, it will roll over into the negatives, or start over at 0 for an unsigned int.
- The sizeof(thing) function tells you the size of a thing, in bytes of memory.
- Char = 1
- Short/(half precision float) = 2
- Int/float = 4
- Long/double = 8
- 1 byte = 8 bits, 1 nibble = 4 bits
- means the pointer to the address of something. Say, int *x, *x is the pointer to the memory address where the number x starts.
    - The *unary* or *monadic* operator **&** gives the "address of a variable".
    - The *indirection* or dereference operator * gives the "contents of an object *pointed to* by a pointer".
- A pointer - Something that points to something, obviously. In computing/computer science terms a pointer generally points to the memory address that something, most often a variable, object, or some other thing is at, or just some empty space.

- **IMPORTANT**: When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it.
- The following will generate an error, because we are trying to assign 100 to the object pointed to by "ip". But ip isn't even pointing to an address...

```c
int *ip;
*ip = 100;
```

However, the following will work:

```c
int *ip;
int x;
ip = &x;
*ip = 100;
```

- You can use header files to store defines. ""s around a #include means to find a file named that in the current directory, <>s mean to look in the library.
- If things begin with 0x, it means the rest of it is in hexadecimal.
- Rhyme: "0x is hex"
- Every time a C script is run, integers are assigned in different places depending on how many integers are defined. This can be proven with with the address function.
- A 'side effect' is when a function affects a value that it does not return.
    - Eg. scanf("%d", &x) is a side effect because it affects the address of x, rather than a copy of x due to the ampersand.
    - Eg. If a functions goes against the first statement in this block 'o text, then it is a side effect.

Lecture16

Testing

- testing is the process of subjecting programs to certain conditions to identify whether it is meeting the requirements
- use assert statements (make sure to *#include <assert.h>*) and either:
    - o assign the assert statements to a function, which you should run at the start of the main program
    - o or just place all the assert statements at the start of your main function
- if basing test in its own function:
    - o `void` the function so it doesn't expect a return value such as 0
    - o place `void` as input so the function does not expect any input
    - o *remember* to include a function prototype!
    - o *don't forget* to include a call to the test function or it will seem like the tests have been passed when in reality the program has not gone through the test function in the first place!
- utilise the **printf** statement in order to output the value of variables so that their values can be seen
    - o or you can use a debugger, but we haven't covered that **YET (** GDB ;))
- test as many cases as possible to ensure program accuracy and ***flawless*** operation
    - o use test cases that test possible values that may bring forward errors from unexpected input, e.g. some programs may act unexpectedly when negative numbers or zero are inputted. Testing a range of inputs can help to mitigate unexpected behaviour.
    - o remember to test boundary values to check conditional logic
- base tests off of the requirements of the program
- If feasible, it is a good idea to write a fuzzer, which injects nonsensical input into the program to try and crash the program or find scenarios which are processed undesirably
- to assist with testing, try and write your programs at a human readable level of complexity, allowing you to abstract your program into functions, making it easier to determine the source of errors and subsequently, the solution
- if you are writing pseudo code and/or you're a primitive you may want to desk check your code on paper

Lecture17

We don't want repetition because it:

- makes code long and confusing
- makes it difficult to change all instances where the same block of code appears

Richard's rap: **If you're cutting and pasting, then by definition, that's repetition.**

Ways to avoid repetition

- Create a variable
- Hash define
- Define a function
- Make a loop (see below)

While loops

The while loop is the only type of loop that we are supposed to be using in this course.

```c
//Prints the numbers from 1 to 10
int counter = 1;
while (counter <= 10) {
    printf("%d\n",counter);
    counter++;
}
printf("\n");
```

- While loops run on conditions, like if statements. Each time it goes back to the start of the loop, it checks if the condition provided is still true. If it is not true, then the loop terminates. To allow us to keep track of the amount of times a loop **iterates**, we must have a variable, in this case called counter. Each time we iterate, we increase counter by 1 so the loop will check if it is still <= 10 and print the new value.
- It is important to **progress** the variable each time the loop iterates (increase the counter variable) otherwise we end up with a non-terminating (infinite) loop.
- You are also able to use while loops inside other while loops which is know as "nesting"

Lecture18

While Loop Counter Convention

If your counter is 0-indexed, the condition should be < or >, if your counter is 1-indexed, your counter should be <= or >=.
(Note: While this isn't compulsory, this is what programmers expect)

Potential Errors with Loops

- Not incrementing counter aka. not progressing
    - Without this your loop would continue endlessly and you'd get an error
- Not initializing variables or initialising the variable to a wrong value

- Fencepost errors

While Loop 1 Line Statement Thing

A while loop expects, after its condition, a single statement to loop. Usually, we put braces in to group code together, but in theory, this is possible:

```
while (x<10)
someFunction();

```

Should you do this, if somebody were to add to the loop without adding brackets, you're going to have problems:

```
while (x<10)
someFunction();
x++

```

In the case above, the while loop would only execute the first statement and never increment x meaning that we'd have an endless loop that continuously runs "someFunction" without even stopping as x never goes above 10.
This is because the while loops is essentially only expecting a single statement, but using the brace pretends that the whole thing is a single statement.

Fencepost Errors

Fencepost errors are when a calculation (or loop) is off by 1. These are easily avoided by creating small test cases in order to test your algorithm/loop before feeding it the larger chunk of data you actually need it to work on.

Functions and Multiple Uses

If a function has more than 1 "thing" that it does, you should break it up into multiple functions (With, if needed, a master function that just calls both functions).

Lecture19

Having more bytes increases the potential complexity of a program. A 16 byte processor can have 16^{16} possible programs. Introducing looping introduces new levels of complexity.

Nesting loops

Nesting loops is when you put a loop inside another loop.
This is useful printing grids. One while loop will go across, while another will go down.

```
int row = 0;
while (row < 5) {
    int col = 0;
    while (col < 5) {
        printf ("%d", row);
        col++;
    }
    printf ("\n");
    row++;
}
```

will print out:

```
00000
11111
22222
33333
44444
```

The inside loop is printing out the row number 5 times. The outside loop runs the inside loop to print 5 rows.

Lecture20

Frames	Functions work by frames When a function is called, your code is interrupted and execution is moved to that functions code After the function has completed execution it reads a "frame" of the stack to know where to go back to One specific sort of attack is when malicious code messes with the stack/ the frames so the root executes code with full privileges that code wouldn't normally have. The way Richard explained it is through this dialog: • Richard: "Hey, I am going to swap topic, when you ask can you tell me that I was going to continue talking about DRY" • Student: "Sure" • Richard: "Ok I am done, what was I about to do?" • Student: "Finish the course and give us all a High Distinction" • Richard: "Ok, thanks"
Abstraction/ delegation	=Being able to take a large job and getting other people to solve smaller parts(e.g "book me a flight"

	It means you can continue your flow of thought / task without having to stop what you are doing.
	
	Functions is the equivalent - you can call a function in your code and not have to think about it - it contributes to abstraction
	`a = theta/180*3.14159265358979;`
	`if (0>a){`
	`    a = - a;`
	`}`
	`printf("%d\n",a);`
	is harder to understand than
	`a = toRadians(theta)`
	`a = absoluteValueOf(a)`
	`printf("%d\n",a);`

OAOO	While programming attempts to do things OAOO - once and once only If you find yourself copying and pasting code or rewriting it then **put it in a function(or a #define, or store its output in a variable, depending on it's usage)**

DRY/Code reuse	While programming DRY - Don't Repeat Yourself as it is more beautiful and is quicker to write due to fact that you can reuse your code
	For example
	`void doThing(int param){`
	`    printf("%d %d\n",param,param+2);`
	`    int num = param*4+3;`
	`    printf("%d\n", num);`
	`}`
	
	`int main(){`
	`    doThing(1);`
	`    doThing(2);`
	`    doThing(3);`
	
	`    return EXIT_SUCCESS;`
	`}`
	is more elegant and is quicker to write then
	`int main(){`
	`    printf("%d %d\n",1,1+2);`
	`    int num = 1*4+3;`
	`    printf("%d\n", num);`
	`    printf("%d %d\n",2,2+2);`
	`    num = 2*4+3;`
	`    printf("%d\n",num);`
	`    printf("%d %d\n",3,3+2);`

	`num = 3*4+3;`
	`printf("%d\n",num);`
	`return EXIT_SUCCESS;`
	`}`

Self containment	Another advantage of functions is that they are self - contained i.e they do what they need to do well and don't rely on anything else.
Side effects	Side effects are what happen when a function interacts with things outside of its scope e.g using printf or scanf
Tables	**I really love tables**

Lecture21

Everything is a function.

Richard explained how servers on the web are like functions. The client (browser) sends a HTTP request, and the server responds with some data.

- The contents of the request are the arguments (or the **inputs**)
- The response is the return value (or the **output**)
- The other operations of the server (such as writing to a database) are the **side effects**

What is HTTP:

The Hypertext Transfer Protocol (**HTTP**) is an application protocol for distributed, collaborative, and hypermedia information systems. **HTTP** is the foundation of data communication for the World Wide Web. Hypertext is structured text that uses logical links (hyperlinks) between nodes containing text.

Connecting to a server

Before data in the language of HTTP can be sent, the client must connect to the server.
To do this, it needs:

- an **address**: like the address of a hotel
- a **port number**: like the "room" which the guest is staying in (usually port 80)

The address is (in IPv4) is a 4 (or 6) byte address, each byte being represented as a string separated by dots.

For example: `216.58.199.46` (the current IP address of Google)

To make it easier for users, you can also use a name, such as `google.com.` However, a DNS server just translates this to an IP address anyway.

HTTP

Once the connection is established, the client can send data (in string format) to the server.

For example, `GET / HTTP/1.1` would request the homepage, with the root (`/`) path. HTTP/1.0 describes the version of HTTP to use (a more current version is HTTP/2.0.
The return data is a string with some headers (meta data) such as data type and date. Here is the current headers in a response from Google (via telnet in the Terminal):

```
HTTP/1.0 200 OK
Date: Wed, 22 Mar 2017 22:20:58 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See https://www.google.com/support/accounts/answer/151657?hl=en for more info."
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Set-Cookie: NID=99=dBYVEKgIOA4cir2FEukpZvqMs-MVufXux_mn2XVctCqb4UABWIJk40Mmm1DGV-TMWoSwAU-GOPYTuo4nhlikDf-2pbAUTCdQSMKl7AkgZg5ln2LQe8eKCjEvh6c4H3HQ; expires=Thu, 21-Sep-2017 22:20:58 GMT; path=/; domain=.google.com.au; HttpOnly
Accept-Ranges: none
Vary: Accept-Encoding
```

- HTTP/1.0 describes the version
- 200 is the status code (OK)
- OK is a description of the status code
- `Content-Type: text/html; chaset=ISO-8859-1` describes the response data type (HTML) and the character set used (ISO)

Lecture22

Calling functions with machine code:

Before calling a function

- Main stores where it is up to in a frame

Providing input

- Write into the function - this requires main to know how the function works, which violates abstraction
- Write into a register - this uses up very limited storage and doesn't allow for multiple inputs
- Write into the frame - this means main doesn’ t need to know how the function works and allows for multiple inputs

Jump to function

- Load 0 into R0
- Jump if R0 == 0

Function

- Save R1 in the frame before starting
- [Some activity that returns some value in R0]
- Restore R1 from frame
- Jump back to main unconditionally (jump if R0 != 0, jump if R0 == 0)

Lecture23

Richard's Wisdom of the Lecture: Exam Technique

- Make sure you follow the criteria for exams.
- Do everything the exam/task tells you to do.

General Programming

- Ensure you programs are clear with comments.
  - People should be able to understand your code instantly.
  - Programs should seem simple and easy to write.
    - Code to do the job done, rather than code to impress with long lines (because styling guide lol)

Blackbox testing

- Somebody other than the author of the code tests the inputs and outputs of the program.
- **Without** seeing the source code.

Whitebox testing

- Somebody other than the author of the code tests the inputs and outputs of the program.
- **With** seeing the source code.

Unit testing

- Testing subfunctions in a program, rather than the whole thing.
- Breaking code up to test different parts.
- In an 8004 microprocessor
  - Can run specific code in the emulator with controlled inputs
- Makes it very easy to find errors within your code.

Lecture24

In C programming, array of characters is called a string. A string is terminated by a **null character /0**

For example:
Strings must be stored with an extra byte for a 0 at the end.

```
char *message = “tree”
```

This means that ‘message’ is an area of memory that stores the address of the ‘t’ in tree. When message is printed, it first prints ‘t’ , then increments and prints the next character ‘r’ , and continues until it hits 0.

```
printf( “%s\n” , message); //prints ‘tree’
printf( “%p\n” , &message); //prints the address of message
printf( “%p\n” , message); //prints the address of the string stored
printf( “%c\n” , *message); //prints what is stored at the address in message, ‘t’
printf( “%c\n” , message[1]); //prints the 1st element of ‘tree’ , ‘r'
```

Flow control in C:

Sequential: Sequential execution is when your instructions are executed the same way as they appear in your program. After executing this line it would just go to the next. An example of this would just be a printf line.

Selection: In a selection a question is asked. The answer of that question determines where the program is going to go. The program takes one of two courses of action then moves on to the next event. An example of this would be an if statement.

Loop: In a loop the program asks a question. While that question is true perform a series of actions until that original question is false. An example of this would be a while loop.

Lecture25

Strings are arrays of characters, viewed as a single thing. It is stored in sequence in memory, and has a \0 at the end

It is represented as a pointer to the first element, which can be passed to other functions

#include <string.h>

Some string functions from the compiler

```
int strlen(char *str) // given a pointer to a string, return how long it is (not including the null terminator at the end)
```

string[0] gets the first character of the string, string[1] gets second and so on

double quotes ("   ") = string, has null terminator
single quotes (' ') = char, no null terminator, only 1 character
*argv[] = pointer to array of array of chars, or pointer to array of strings

Our own stringLength()

```c
int stringLength(char *string){
    int length=0;
    while (*string != 0){
        string ++;
        length ++;
    }
    return length;
}
```

- pointer ++, means to move the pointer 1 position forward (4 for int, 1 for char)
- *pointer, to get the thing at that position
- the ascii code of the null terminator (\0) is equal to 0, whilst the ascii code for '0' = 48
- In most languages, an number is stored at the start of the string which tells it how long the string is, however not in C, as this takes up extra memory and computers had very little memory before

All Functions of string.h

void *memchr(const void *str, int c, size_t n) Searches for the first occurrence of the character c (an unsigned char) in the first n bytes of the string pointed to, by the argument str.
int memcmp(const void *str1, const void *str2, size_t n) Compares the first n bytes of str1 and str2.
void *memcpy(void *dest, const void *src, size_t n) Copies n characters from src to dest.
void *memmove(void *dest, const void *src, size_t n) Another function to copy n characters from str2 to str1.
void *memset(void *str, int c, size_t n) Copies the character c (an unsigned char) to the first n characters of the string pointed to, by the argument str.
char *strcat(char *dest, const char *src) Appends the string pointed to, by src to the end of the string pointed to by dest.
char *strncat(char *dest, const char *src, size_t n)

Appends the string pointed to, by src to the end of the string pointed to, by dest up to n characters long.
char *strchr(const char *str, int c) Searches for the first occurrence of the character c (an unsigned char) in the string pointed to, by the argument str.
int strcmp(const char *str1, const char *str2) Compares the string pointed to, by str1 to the string pointed to by str2.
int strncmp(const char *str1, const char *str2, size_t n) Compares at most the first n bytes of str1 and str2.
int strcoll(const char *str1, const char *str2) Compares string str1 to str2. The result is dependent on the LC_COLLATE setting of the location.
char *strcpy(char *dest, const char *src) Copies the string pointed to, by src to dest.
char *strncpy(char *dest, const char *src, size_t n) Copies up to n characters from the string pointed to, by src to dest.
size_t strcspn(const char *str1, const char *str2) Calculates the length of the initial segment of str1 which consists entirely of characters not in str2.
char *strerror(int errnum) Searches an internal array for the error number errnum and returns a pointer to an error message string.
size_t strlen(const char *str) Computes the length of the string str up to but not including the terminating null character.
char *strpbrk(const char *str1, const char *str2) Finds the first character in the string str1 that matches any character specified in str2.
char *strrchr(const char *str, int c) Searches for the last occurrence of the character c (an unsigned char) in the string pointed to by the argument str.
size_t strspn(const char *str1, const char *str2) Calculates the length of the initial segment of str1 which consists entirely of characters in str2.
char *strstr(const char *haystack, const char *needle) Finds the first occurrence of the entire string needle (not including the terminating null character) which appears in the string haystack.
char *strtok(char *str, const char *delim) Breaks string str into a series of tokens separated by delim.
size_t strxfrm(char *dest, const char *src, size_t n)

Transforms the first n characters of the string src into corrent locale and places them in the string dest.
void *memchr(const void *str, int c, size_t n) Searches for the first occurrence of the character c (an unsigned char) in the first n bytes of the string pointed to, by the argument str.

Lecture26

Arrays

Arrays are a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type. (storing marks of a class in one variable). Another way to think about an array is as a pointer to a group of variables e.g. ints stored together and the array pontes to the address of the start of the variables.

Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows –

```
type arrayName [ arraySize ];
```

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows –

```
int marks[5] = {90, 75, 84, 62, 94};
```

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example –

```
int mark = marks[4];
```

(mark would equal 62)

Lecture27

Flow Control

The three forms of flow control are iteration, selection, sequential, with these anything can be solved by humans using programming.

Iteration

When certain actions are repeated as long as a condition is satisfied, this is usually done by using a counter and incrementing it each time the loop is run.

```
int counter = 0;
while (/*condition*/) {
  //action
  //counter increment
}

eg.
int i = 1;
while (i <= 3) {
  printf ("there have been %d loops", i);
  i++;
}
```

In the example, the loop will run as long as it is less than or equal to 3, and as long as this remains true the program will print "there have been `/*the number of loops*/` loops" and each time it is run the counter is increased by 1 so eventually the condition is not satisfied.

Note: *When using a 0-index counter use `<=` or `>=` for the condition, if using a 1-index counter use `<` or `>` for the condition.*

Selection: *Flow in which the program executes certain instructions depending on whether or not conditions are fulfilled. More can be found at Lecture05: If*

Sequential: *Flow in which the program executes its instructions from left to right, top to bottom. This is known as the sequential order as it refers to the code being executed in the order it was made.*

```
printf("first instruction"); //this will be executed first as it is written first
printf("second instruction");
printf("third instruciont"); //and this will be executed last

eg.
int inputNumber;

printf("enter a number");
scanf("%d", &inputNumber);
printf("the number you entered is: %d", inputNumber);
```

In the example, the first thing that is executed is the `int inputNumber` is defined, then "enter a number" is printed and so on, instructions are executed as per the order the code was written.

Multiplication

- Regardless of which way you do multiplication, the answer still remains the same.
- In the example, 3 * 4 = 12 is used. No matter whether you express it as
    - (1 + 3) * (1 + 2)
    - (2 + 2) * (1 + 2)
    - (2 + 2) * (1 + 1 + 1)
- It still comes out as 12. This can be used to produce a more general form (a + b) * (c + d) = ac + ad + bc + bd
- This form can be used for many different counting systems (Richard uses Chicken and Egg), including complex numbers.
- Example: (2 + 3i) * (1 + 4i) = 2 + 3i + 8i + 12i^2 = 2 + 11i - 12 = -10 + 11i
- Because this multiplication takes in a tuple (x, y) and returns a tuple, you can use it recursively.
- Counting the steps it takes to get to a certain point, and plotting it on a number plane gives you a Mandelbrot Set (a fractal).

Fractals

- These sets are infinitely complex but are governed by just a few simple rules. They end up repeating as they continue.
- Fractals are a curve or geometrical figure of which similar patters recur at progressively smaller scales. Mathematically, they have extremely interesting properties and although they seem unbelievably and almost infinitely complex, they are generated by a few simple, trivial rules.
- Unlike a regular image, with a fractal you can keep going in and zooming without 'bottoming out' in terms of complexity.
- Fractals have the notion that information is repeated in depth at different levels of abstraction, where new levels of detail are shown. They also have self-similarity meaning there are copies of itself in deeper and deeper levels of the fractal.
- Fractals are present in nature and natural objects, such as clouds, where many objects exhibit the nice fractal property that everything is repeated over and over again and as you continue to zoom in you see the original object once again.

Lecture28

The Mandelbrot Set

The Mandelbrot set is the set of complex numbers for which the function does not diverge when iterated from, i.e., for which the sequence, etc., remains bounded in absolute value.

Lecture29

Trilogy

There are 3 core courses that (used to) exist in computing at UNSW. They can be known as the trilogy, and can be described as follows:

COMP1917	You become a **craftsman** (aka artisan aka craftsbeing). Your programs and your approach to computing are professional and workman-like. You strive for clarity, simplicity, and beauty.	Colour
COMP1927	You become a **scientist**. You understand how to analyse programs and algorithms. You master a broad range of important algorithms and data structures.	Black and white
COMP2911	You become a **designer**. You are familiar and practised in the process of design. You know how to approach the design of new data structures and algorithms appropriate for the problem at hand. You have an appreciation for the qualities of a good design.	Shades of grey

Volume I

- You need **resilient** solutions.
- Break into files
- The importance of **modularity**, **low coupling** and **coherency**
- Header files are shared between files, reducing work and repetition, while preventing mistakes and creating **abstraction**
- The keyword `static`: static functions are hidden from other files

Lecture30

A structure is a user defined data type that can contain multiple data items of varying data types

Defining a Structure

To define a structure you must used the keyword `struct`. The following is an example of a color struct:

```c
struct _color {
    int red;
    int blue;
    int green;
};
```

Note that the struct must end with a **semicolon** (;).

Also, the typical convention is to precede the struct name with an **underscore** (_)

By creating a struct, we can now consider color, which is made up of RBG components, as a single entity

typedef

The keyword `typedef` allows you to give an existing data type a new name, like as follows:

```c
typedef unsigned char byte;
```

Typedefs can also be used with struct definitions:

```c
typedef struct _color {
    int red;
    int blue;
    int green;
} color;
```

This will allow you to create variables with your defined type like so:

```c
color pixel;
```

Accessing Structure Members

The members of structures can be accessed and assigned/return values using dot notation:

```c
color pixel;

pixel.red = 255;
pixel.green = 0;
pixel.blue = 255;
```

Lecture31

Risk management

- We often see events that occur frequently, and we are good at working out averages and dealing with the risk appropriately.
- However, there are often events that have a low probability of occurring, but when they occur, have a high impact. E.g. an asteroid colliding with Earth. We are really bad at dealing with these risks - we either freak out and spend a lot of resources trying to prevent the risk e.g. security with two-factor authentication, or we ignore it completely e.g. earthquake risk on the San Andreas fault, rather than managing resources sensibly.
- These low probability, high impact risks are also an issue in programming.
    - working out what things you have to get right
    - designing the systems to avoid security issues

    - caution required on low-probability scenarios/cases
    - *understanding what the important things are, and understanding the consequence,s the impact of things, and then working out sensible ways to deal with those impacts (mitigating, or dealing with them altogether)*
    - testing helps to identify errors
- don't go to www.wikileaks.co
  You have to evaluate your risk and evaluate how much you should do to avoid that outcome
  There are lots of risks where the outcomes are very minimal that should not be bothered about e.g. taking an extra day to code, running a bit slower. etc..
  Servers can be taken down by blocking the IP address but they can just change the IP address of the server and keep the domain name.

Lecture32

Computing1 in Color: Basics

- Our eyes are interconnected with our brains very closely together, making us humans very efficient (but not the best) colour processors.
- All the colours we see aren't just a single colour, they are a combination of multiple wavelengths all represented through sinusoidal wave models.
- We process the beautiful spectrum of colours with *only* 3 receptors in our eyes;
  The Red, Green and Blue receptor, hence RGB.
- Color wavelengths:

Colour	Wavelength (nM)
Red	740
Green	565
Blue	500

- The level of intensity ranges between 0 to 255. Low intensity to high intensity.
- As green is in the middle of red and blue and they overlap, we will always see a bit of red and blue in any green.

Lecture33

Mentality of Finishing Assignments, Time Management

- A long list of small occurrences creates little excuses that are more frustrating than one single excuse, but rather than blaming all the events around you, reflect on how you organise your time.

- Rule of thumb: Estimate how long it will take you, being as pessimistic as possible, then multiple by four. (And that's probably still not enough time)
- Start early: Don't think "I'll start in a week"
- Don't allocate blame, think about what happened, and reflect on what you can do to change
- If you understand and master something, go out and help people (Passing it on)

The Game

- KPI: Key performance Indicator
- What strategies are best for gaining KPI's for university?

Summary of game: Place campus at the vertices of the hexagon grid (Geographical Discipline Model). Funding is allocated every 'year' (turn) by the rolling of dice. If the number on the hexagon matches the dice roll, the touching campuses receive a student of type [whatever hexagon].

Lecture34

Stacks and Frames

A **stack** works on the principle of First In, Last Out (FILO)

- The last thing that got put in the stack is the first to be taken out
- The first thing that got put in the stack is the last to be taken out

The stack data structure **stores**:

- **What** it will do next
- **Values** of variables
- **Return Address** of the calling function

Every function has a **frame** that gets stored in a stack called the **stackframe**:

- A **stack pointer** points to the last address in memory
- When a function needs a frame, the frame gets stored at the address the stack pointer points to, and the stack pointer changes to the next free address in memory
- This can be repeated as more functions are called and more frames are needed
- When a function returns, the stack pointer simply moves down, so that when a new frame is needed, the new frame will overwrite the useless frame

Calling a function

1. Parameters go onto the stack
2. Return address is placed into the frame
3. Jumps to the start of the function
4. Saves the registers
5. Reserves memory for local variables

Returning a function

1. Releases memory for local variables
2. Releases registers
3. Restore stack pointer
4. Jumps to return address

Every function has a frame which stores

1. Local variables
2. Address to jump back to
3. Some information calling function has given it (arguments passing in)

Terms

Function overhead – Time it takes to process a function

Lecture35

malloc(); AKA m**EMORY**alloc**ATION**

what it does and its uses:

- allocates a certain amount of memory inside the heap
- returns a pointer to the allocated memory
- can be utilised to allocate memory whilst in a function frame, then once the function has completed execution and its frame vanishes, the memory and its data inside the allocated memory can still be accessed
- e.g. using malloc() to set aside memory for a string inside a string reversing function, then returning the pointer to that string so that the string can be accessed by other functions such as main

how to use it

- requires the stdlib header file
- takes in one argument, which is the amount of memory to allocate
- returns a void pointer, requiring type casting

```c
char *buffer malloc (100)
```

- The above code will set aside 100 bytes of memory in the heap and return a pointer to the address of the memory to the variable buffer.

malloc takes memory from the operating system using sbrk or brk (which usually allocates a larger chunk which can be reused by the program the next time malloc is called) and earmarks it for the program. There are plenty of behind the scenes optimisations that aren't too necessary to know unless you're planning on exploiting memory allocators ;)

- the blocks of memory are stored as structs in a linked list (with flags such as size and whether it is 'free')
- there are multiple algorithms different memory allocator implementations utilise to increase efficiency (i.e. without optimisation, the allocator might gmark 20 mb for a 64 kb request simply because it was first block on the linked list that fit the required size)
- malloc tries to limit the calls to to sbrk / brk due to the overhead (switching to kernel mode, switching back...)
- malloc allocates memory to the heap (which grows opposite of the stack. Usually towards higher memory addresses)
- don't use malloc unless you have to, such as if you're hacking into a mainframe via segmentation violation

free();

```
1.  free (address of allocated memory)
```

- The above code will de-allocate the memory allocation at the given address
- if you forget to free allocated memory, you get a memory leak. If this occurs repeatedly (e.g. In a rendering loop), bad things happen, such as the slowing down of processes and possibly the eventual crash of the program. (And the OS complaining). This can get really bad, such as on servers.
- Freeing things that you haven't allocated is a good idea. Do it, it's very !!FUN!!
- Compilers don't check for this thing because they assume you're smart, so make sure to be smart with malloc and free
- malloc keeps track of all the bytes for you so you don't need a size for free(), yay

Richard Buckland is a pizza thief

- Sizeof() is a good thing and is usually used with malloc, such as 1200 * sizeof(int) for a pointer to 1200 intergers
- There's a few dumb mistakes you can do, such as using the address of a buffer instead of what the buffer points to in malloc, so make sure you have your pointer indirection levels down right.

Lecture36

The Game

There are 2 parts of the Game:

- Game engine (teamwork)
- AI (solo)

The idea is to start with something, and the make that better. Don't aim too high to start with.
The Game type is an abstract type, the guts of which resides in `Game.c`. Meanwhile, `Game.h` contains the interface functions, and a ADT typedef, which looks like this:
`typedef struct _game * Game.h;`

Other

`NULL` (from `stdlib.h`) points to nothing (in specific `0x0`).

Lecture37

Naming & Types	We name things in C by going <TYPE> <EXPRESSION> and reading it backwards, where <EXPRESSION> is possibly a <RELATIONSHIP> then a<NAME>. For something that comes from directly from a type like below there is no relationship.
	`int x; //This translates to x is a int`
	More explicitly however if something's meaning comes from its relationship we do for example:

	`int *x; //This translates to:`
	`       //What x points to is an int`

	However, we have to be careful. You can do <TYPE> <LIST_OF_EXPRESSIONS>. For example, we can do
	`int a,b;`
	Which is equivalent to

	`int a;`
	`int b;`

	However, as <RELATIONSHIP> only applies to its <NAME> it would be incorrect (but easy), to think
	`int * a,b;`
	is equivalent to

	`int *a;`
	`int *b;`

	However, it is actually equivalent

	`int *a;`
	`int b; //Not a pointer`

	Which is why we don't declare multiple variables on the same line :).

The problem of design

Designing things is hard as there is no perfect solution and there are lots of tradeoffs to be made.

When we make things we should do it in three stages:

1.Design/specification

2.Write the tests

3.Implement the thing

The stack we are going to make

We are going make a stack over 3 files stack.c, stack.h and, testStack.c . As stack.c is just defining our stack only testStack.c will contain a main function.

The operations we will need for our stack are push/add, to put something on it, top, to get what is on the top and pop, to remove what is on the top.

Code we are given(Ignoring header comments)

Stack.h

```c
//This code doesn't work
#define STACK_SIZE 1000

typedef char stack[STACK_SIZE]

stack add (stack s, char elt);

char top (stack s);

      stack pop(stack s);
```

testStack.c

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

#include "testStack.h"

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

   printf ("All tests passed. You are Awesome");

         return EXIT_SUCCESS;
      }
```

Ignore

	When we build our stack we will need a way of solving the problem of overflowing by either just refusing to do it or giving the user a way of detecting whether it would overflow.

Lecture38

- When writing code, we need to store the size of the stack as well as the elements within the stack, so that users of the code do not accidentally overflow the stack. This means that you need to create a struct to store the information about the stack.
- To test the stack, all we have to test is that the interface functions work, in particular, that the sequences of the functions are correct. If something weird happens but doesn't affect any of the functions, it's irrelevant because we only have the ability to interact with the stack through the functions.

When testing, in context of the lectures;

```
t = push (s, 'A');

//both of these are correct
assert (top(t) == 'A');
assert (t.items[0] == 'A');
```

We're probably more inclined to write the second one as we're more used to it, however, the first one is a better choice. The first one doesn't depend on how you chose to represent a stack. (e.g. the second one intimately depended on remembering if you named the variable items or elements if it actually had an array, it the first element was stored at 0).

The only way you are allowed to interact with and modify the stack is via the interface.

Richards advice:

1. Design the code with pen and paper
2. Write the interface
3. Write the test
4. Write the function
5. Test
6. Function
7. Test
8. Function
9. etc.

This means that your function is more likely to be correct because when you think about testing, the function is more likely to do what you want because you've thought about the design of it first.

The idea of making types and accessing them through an interface, if you get it right, you have the ability to change the type and still have it work. This is known as an abstract type.

- An abstract type is a type where you only access it through the interface functions, without knowing the structure of it. The opposite of this is a concrete type, being that you can instantiate it directly and use pieces of code such as:

```
assert (t.items[0] == 'A');
```

- And although it has been mentioned that using an abstract type like a concrete type is bad, equally as bad is designing your type to be concrete. If you are unable to change the specific inner workings of a piece of code, as someone else's code already relies on those exact inner workings, you can end up being unable to improve, build upon or alter the workings your type (For example, optimizing the code would be nigh impossible).

Lecture39

Key Points About ADTs and Interface Functions

- ADT's are data types which are standalone and have "interface" functions which connect to other data types. The interior parts of the ADT's do not need to be known in order to be able to use the ADT.
- The user of a struct (ADT) should not be able to know what the innards of the struct are (what its attributes are
- There needs to be an interface function which creates a new instance of the struct
- We can have different types of interface functions
  - Getters, which get the value of a specified attribute. For example, with Knowledge Island, a getter would be used to return the amount of students a particular player has.
  - Setters, which overwrite the existing value of a specified attribute with a new value. For example, in Knowledge Island, a setter would make a particular corner have a GO8 instead of a campus.
  - Other types, which edit the interior parts of the function in a particular manner (similar to scanf). For example, rolling a die in Knowledge Island would edit the dice value which is stored internally into a random integer.

Being Conservative with Functionality

An issue that was covered in the lecture was the idea that, if you have a program to do a specific thing (in this instance allow you to create a stack), you can add too many useless features to it and lose the level of abstraction. The main idea, as Richard says is: "it's not that more is better". Just because

one can add extra functionality doesn't mean that one should. The interfaces should be as simple and as short as they can, with enough functionality to work and be useful, rather than adding loads of functions that one "may have a chance of using"- this can lead to increased problems in code which can be easily avoided.
For example, unnecessary functions may clutter and increase the amount of functions which a potential developer using the ADT as an API needs to know. This will hamper productivity and increase complexity without a good reason.
So, to evaluate whether a new functionality idea is worth implementing ask the questions:

- Is it incredibly useful?
- Is the cost of not putting it in massively higher than the cost of putting it in?
- If in doubt, don't put it in.

Lecture40

A **violation of abstraction** is when we use something inside the struct when we aren't in the file that creates the struct.
Abstraction is like a car, we know how to drive and use the car, but we don't know what's operating the car, or we could be making changes that will create problems in the future.
In code, we only want the user to know the type and its interface so that code will never be accidentally coupled. To enable this, when we want to use a struct in a different file, we can instead use a **pointer to the struct**.

Suppose we already defined a struct called game, to create the pointer, we need:

```
typedef struct _game *Game;
```

Since it is a pointer, it also allows us to directly modify the game.
Note the capitalization of *Game, but not of _game and the concrete type of game in Game.c.

Side Note:
It is possible to use

```
char thisCharacterIsNull = '\0'; //Shortcut for within single quotes
char thisCharacterIsAlsoNull = 0; //Assigning null through just using 0 as an int
```

As a shortcut for null.

Lecture41

Abstract data type

Abstract type — we don't know whats inside it = we don't know how its represented and how/where it is stored.

Advantages

- the programmer is not attached to the physical form of the type making it easier to change.
- Making things abstract means we may be able to find new purposes for them.
- All a common interface for a program needs is a function which can return or complete a given task, no matter how it is completed. ADTs allow this as users don't know how a function works, only that it does.
    - If we allow the user to make changed to the struct itself, it means that their program may break in the future.
    - If we change the structure of the struct, then everyone's programs which are using it may break.
    - If everyone uses **common interface**, then we can change any part of any module of any program, and all other programs will continue to work.

2D array

An array of arrays:

```
char cells[NUM_COLUMNS][NUM_ROWS];
```

This is a two-dimensional array, or **2D array**. This is because it contains an array of columns, each column containing a cell corresponding to part of a row. So, if you wanted cell (3,5) (as in, row 5, column 3), you would access `cells[3][5]`.

Using abstract types

In the newGame function, you must use `malloc` since it will otherwise be lost after the function terminates. You cannot use the `sizeof` the abstract type (uppercase) for this; you must use the `sizeof` the concrete type (lowercase), since the abstract type is just a pointer.
To access the fields in a struct while only using a pointer to it, you can use:

```
(*ptr).element
```

or

```
ptr->element
```

NOTE: `malloc` will return `NULL` if the heap if full, so we should generally assert that the pointer to the malloc'd heap of memory is not `NULL`.

Loops

Although there are multiple ways to utilise pre-test and post-test loops, we have been restricted to using ONLY `while` loops. The other common loop is a `for` loop. These aren't allowed in the course because it reduces the lines of code but at the cost of code clarity.
A while loop will check to see if a given condition is satisfied and then will run through the contents of the loop sequentially. After the program has run through the contents of the loop, the program again

checks to see if the given condition is satisfied and will continue this until the given condition is broken, in which the loop is stopped.

Lecture42

The art of debugging is ***finding the problem***.

Searching

You only have **finite** amount of time to locate something. That is why searching is such a challenge in computing.
Richard mentioned a few different ways of searching. See these general methods in relation to the fairytale metaphor:

- **Random** — just randomly letting your eyes move over the page. This can be a good approach to start with, but if it isn't working, you should try something else.
- **Brute force** — probably separating the page into grids, and going through *every single one* until you find it. In many cases, this just won't work, because it can take a very long amount of time and if you miss something or make a mistake can become pointless.
- **Systematic** — like brute force, except there is some sort of thinking involved. it means that you have planned the logic and have through through the best way to approach the problem.
- **Unit Tests** — Mentioned earlier in the course, are also useful in systematically testing each component of the code and thus, finding problems.

Debugging

The challenge of searching in computing also applies to debugging. First off, he randomly goes through, looking for the error. If that doesn't work, he goes systematic. Have a look at his systematic way of debugging (searching) below:
** the best way to work is from large to small, breaking up the problem into smaller sub-problems. This allows you to focus in onto the error

1. Take a copy of your program.
2. Trash it searching for the error.
3. Utilise a log book — record what you've tested and what you haven't (keep a history). This helps to avoid error and missing sections of the code
    1. Quick historical side note: a log book is called a log book, because sailors would through a log off the back of a ship, and count how many *knots* have spun around the log, to calculate their speed in *knots*. They would then put these measurements in their *log* book.
4. Delve into specific functions (includes creating dummy stubs).
5. Trying to find smallest possible program which still gives error (take something out; if error goes away, put it back in, otherwise leave it out).

6. Should be left with few lines of code containing error.

Managing projects

Project management in software (and in general) has long been difficult, and has left everyone unhappy.
There are 4 variables to think about in a project:

- Scope — what needs to be done
- Resources — how many people you have, etc. to get it done
- Deadline — (time) when you have to get it done
- Quality — (software quality) how *good* the code is

In the ye olde days of Richard Buckland, *everything* would go wrong:

- Not everything in the scope would get done, leaving the client unhappy
- There wouldn't be enough resources, and everyone would be overworked, leaving the programmers unhappy
- The project would go overtime, leaving everyone stressed, and this unhappy
- In the end, the result would be awful code, leaving *everyone* unhappy

So what's the answer?
Agile also known as eXtreme programming.
In this new paradigm / era / age / revolution / enlightenment of project management, this is how the variables are managed (in no logical order):

- Resources — are not overworked
- Deadline — is met
- Quality — is high
- Scope — is, well, complicated...

In agile, the client works *with* the software team.
In the ye olde days, software team would inspect the client, record requirements, and go into a Fort of Solitude (formally known as FoS). In here, the would work away at the code, getting overworked, not getting everything done, compromising quality, and being late. "Tra da" they would say. But the client would always be unhappy, because the spec is never what they *really* needed.
So, in agile, the client creates the spec *with* the software team, and priorities jobs *with* the software team. And if they need the spec changes, they can let the software team know before the end of the project. If not everything gets done, it's the client who oversaw it, so they are happy. And the programmers are happy. Happy ending!
Oh wait. What if the client is stupid and doesn't know what they need? Or prioritises in a bad way. Like... putting a whiz bang feature that will bring awesome colour to the project, over an infrastructure redesign that will change nothing in the short term.
A good way to convince them is saying that quick fixes now will create design debt, which will have to be paid off later.

Also, Richard likes to phrase it as risk, where a bad decision now creates a risk of a high impact event.

Agile Programming

Agile programming is a new age paradigm which has been developed in order to counter problems and disadvantage of previous ways of developing program. The primary difference/stand alone characteristic of agile programming is its cooperation with the customer and by doing so, satisfaction is maximised, time and problems are minimised. This cooperation works to a few levels of understanding between the client and the software team where the software team considers the client an equal. That is, the software team only advises the client upon decisions but ultimately the client makes the final decision, this way the client can make educated judgements as they are provided with professional advice and the team can adhere the program and do things as the client wishes allowing for satisfaction in both parties.

Lecture43

Richards corrections from last lecture: static function declarations should stay in the file they affect as if they are in the .h and its included in another file it can see it but cant use it or know what it is

Richard's wise words: For the project, don't fool around trying to make something better, instead make something that's crap... and then slightly better... and then slightly better... and then keep making it better and better until you have something

Modern projects:
In old projects you'd go to the client for a year and ask what they wanted, then you'd code it for them but they'd often be unsatisfied. In modern projects you therefore get someone from the client to come work with you and give them a whole list of things you can do and ask them to order it from most important to least important and then do the most important ones first, the client can however change the order if they change their mind. You also have to describe to them however what a bad code infrastructure would mean. Eg describing it as debt and that you lose for example 10hrs each month because of it or describing it as a risk.

Lecture44

Indexing arrays is great for finding things quickly and easily because all you need to do is the index and you get whatever is located at that position.

The downside to indexes are:

- They may not be necessarily meaningful
- Each index has to refer to the address directly after the previous index
- The size is fixed
- It is hard to add more elements into it
- Elements have to be a fixed and known size

There are ways of fixing this issue like allocating an appropriate amount of space for the worst-case scenario. This still isn't the best solution and this is where linked lists come in.

Linked lists are what you think they are, lists which have pointers of other lists within them.

These lists can be made by:

- making a struct which holds data and a pointer (often called a node)
- mallocing an individual piece of data as well as a pointer to the next piece of data.
    - If it is the last piece of data, the pointer would be replaced with NULL, ending the linked list.
- Adding a new piece of data would malloc again, replacing the previous item's NULL with the pointer to the latest piece of data made.
- Ensure you free the data you wrote when done with it.

Linked lists are traversed in a loop, starting at a root / head, checking whether either the next or previous pointer is still valid with relevant data being checked. Linked lists are often used as a basic form of dynamic memory, allowing nodes of memory to be assigned and traversed (think binary tree).

Linked lists are relevant for the following:

- Time critical applications, as linked lists guarantee consistency
- Memory storage without a static size (mutable arrays. yay)
- When random access is not required (as this would slow down access)
- When you need the ability to insert nodes in the middle of the list (e.g. priority listings)

Arrays are preferred with static sizes, memory critical applications, random access or heavy access (e.g. algorithms).

Lecture45

Advantages of arrays

- Easy to get a specific element (random address)

Disadvantages

- Hard to insert or remove elements - O(n)
- Indexes don’ t have meaning
- Fixed size

Linked list

A linked list is a struct that contains a pointer to first element (called the list / head). Every element (called a node) is a struct which contains the actual data, and a pointer to the next element (the link). The last element points to NULL, to signify the end of the list.

Advantanges

- Easy to insert or remove elements - O(1)

Disadvantages

- Hard to get a specific element (random address) - O(n)

- What the above means with difficulty and O(n) notation is the time complexity - How long it takes in the code, in theory, depending on implementation to do a thing. So in an array, it will take 1 operation to get a specific element in the array, but n(the size of the array) to insert or remove an element. And in a linked list, it will only take O(1) so 1 operation to insert or remove an element.
- The struct of a linked list can contain any number of things above or equal to 1. A linked list will always need a pointer to the next node to be classified as a linked list.
- appendItem(l, 86) appends 86 to the end of the list pointed to by l.
- There's a IDE which tells you how many errors you have that it's legal to use in exams, apparently. vim and gedit are what I use, personallty.
- Generic stuff on thorough testing, malloc, etc.
- To make a struct via pointer you can use struct *thing malloc(sizeof(struct)) to make a struct.
- Some stuff with strings and balloons? idk
- Visualize your code. Whiteboard, paint, whatever, it's easier if you have how it works in your head.
- Lists are distinguished by nodes with hoods. Because lists/heads only point to the first element in the linked list, while each node points to the next element in the list.
- READ THE SPECS FULLY BEFORE DOING THINGS!

Format

Making the Struct for the Linked List

```
typedef struct _node *link;
typedef struct _node {
    int value;
    link next;
} node;
typedef struct _list {
     link head;
} *list;
```

- A node stores information (e.g. class, grade-level, number)
- A link points to a node
- A list points to a struct which contains a pointer to the first node of the list, called the "head"

Making a New List

```
// makes a new list
list myList = malloc(sizeof(*myList));

// initialises the list as empty
myList->head = NULL;
```

Adding a Node to the START of a linked list

```
// makes a pointer to a node
link newNode = malloc(sizeof(node));

// add information to node (e.g. class, number)
// e.g newNode->value = VALUE;


// make a pointer to the previously first node of the list
link oldHead = list->head;

// link the new node to the previously first node of the list
newNode->next = oldHead;

// make the newly created node the first node (the head)
list->head = newNode;
```

Adding a Node to the MIDDLE of a linked list

```

// assumes that the list will contain enough elements

// make a pointer to node
link newNode = malloc(sizeof(node));

// get the first node
link currentNode = l->head;

int position = 2;
```

```
// the position to insert the node
// will make that node list[position]

// 0 - 1  ^ 2 - 3 - 4
int count = 0;
while (count < position) {
    currentNode = currentNode->next;
}
// currentNode now points to list[count]
link temp = currentNode->next;
currentNode->next = newNode;
newNode->next = temp;

```

Adding a Node to the END of a linked list

```
// makes a pointer to a node
link newNode = malloc(sizeof(node));

// add information to node (e.g. class, number)
// newNode->value = VALUE;

// makes the node the last node of the list
newNode->next = NULL;


// if the list is empty, makes the new node the head of the list
if (l->head == NULL) {
 l->head = newNode;

}

// if not, makes the new node the last node of the list
else {
    // makes a pointer for the linked list and begins at the first node
    link current = l->head;

    // find the last node in the list
    while (current->next != NULL) {
        current = current->next;
    }

    // current points to the previously last node so makes the next node the new node
    current->next = newNode;
}


```

Deleting the Last Node

```
// make a current
link current = l->head;

// if no nodes in list
if (current == NULL) {
    // do nothing
}

// if one node in list
if (current->next == NULL) {
    free(l->head);
    l->head = NULL;
}

// if more than one node in list
else {
    // finding second last node
    while (current->next->next != NULL) {
        current = current->next;
    }

    // freeing and deleting last node
    free(current->next);
    current->next = NULL;
}
```

Lecture46

Common Interfaces and their ambiguity

- When there is a common interface, every developer must completely agree with every aspect of that interface before implementing their program. If there is any ambiguity, someone might rely on that ambiguity being resolved one way and another the other way because, for them, their own interpretation of such ambiguity is right rather than understanding what risks it may carry. Therefore, it is important to make sure there are no ambiguities and proper rules in place correct to one interpretation for all developers.
- For example, in the example in the video, it is not clear whether roads may or may not go outside of the playing area temporarily. The spec was not specific enough about whether this is allowed or not, so different interfaces would have been incompatible with each other- some do not support this rigorously while others did. This would have resulted in some interfaces breaking simply due to a different interpretation of the same problem. Thus, it is important that all ambiguities be resolved and a specific process by which a program will function be developed.

Changing code

- When changing code, the principles of cohesion and decoupling, and having a set of tests allow us to change the code fast, hence being agile.

Agile approach

- Make it flexible, not at the expense of flexibility.
- Removing code and quickly rewriting it.
  - Therefore need neat and commented code.
- Two phases
  - First being that the code is quickly re-build to make sure all the testing works.
  - Second, the code is refactored to make the design better.
- When you see something twice or three times in code, take it out and turn it into a subfunction.
  - Makes it easier to rewrite if needed
  - Decreases the size of file

Lecture47

Segments

8006 Design

- Creating features based off what is annoying in the 8004
- 3 Issues and possible Solutions:
    - 255 memory cells is too little
        - Increase the number of registers to 4 (R0a, R0b, R1a, R1b)
        - Print 16
    - Frames are annoying
        - R1 load relative [10] using another register called a Segment register
    - Stacks are pains

How are integers used in bytes?

- It doesn't matter which way the number is sent, but rather that everyone is sending integers encoded one way or the other
- Ask yourself:
    - Where should the most significant bit lie?
- To figure out how your computer stores memory, using a C-compilier (Big-Endian vs. Little-Endian)
    - Attempt 1:

```
int x = 1;
char c = (char) x;
//typecasting x as a char
printf ("x is %d\n", x);
printf ("The least significant byte is %d\n", c)
```

    - Result 1:

```
x is 1
The least significant byte is 1
```

    - Attempt 2:

```
typedef unsigned char *bytes;
int x = 256;
bytes bytesOfX = (byte) &x;
printf ("X = %x %x %x %x\n", bytesOfX[0], bytesOfX[1], bytesOfX[2], bytesOfX[3]);
```

    - Result 2:

```
X = 0 1 0 0
```

Segments
Compilers still have this notion of a segment. They put the code into one code segment, the constants into another, the heap into a segment and so on. When it sets up a segment, it is possible for the system to allow only certain actions in certain segments.

Lecture48

Planning
Drawing a picture is the best way of getting your head around a complex problem, such as linked lists. You should draw a picture, and make sure that you understand the problem, before designing functions, writing tests and implementing those functions.

Writing correct code
Have a strong testing plan, design approach, styling and peer review - have others look over your code often will weed out errors that you may not have seen.

Debugging
The best bugs are the ones that are hard to detect, occur rarely, aren't triggered during testing and made worse after attempts to fix them.
Detecting bugs: thinking in the mind of someone trying to destroy your program will lead to you discovering exploitable situations.

Unspoken Rules in the Project
Richard talks in this lecture about cheating in the game. The logic that not all rules will be on the spec can be applied outside of this course, so I reckoned I thought this was worth mentioning here. Also, I am required to contribute and this was the only uncovered subject I could find.

Making secure programs without exploits
You have to think like someone who is purposely trying to break your program and not just put in some measures that you think will work. This will result in you seeing any vulnerabilities that malicious programmers might use to exploit or break your program. Eg if your program has a linked list and can stack overflow because you're not capping the maximum amount of elements.

Deleting nodes from a linked list
Deleting nodes from a list is much more difficult than just adding nodes to a list. Here are the steps to ensure you don't have memory floating around taking up space, and to make sure that you don't get any segmentation faults. :)
Assume we have a list: `[5] -> [7] -> [22] -> [NULL]`

Let's say we want to get rid of [7] from the list:

1. Make a pointer to the node before the deleting node ([5])
2. Make a pointer to the deleting node (Node del = [7];)
3. Make the node before the deleting node point to the node after the deleting node ([5] -> [22])
4. Free the pointer pointing to the deleting node (free([del]))
5. **NOTE:** Although you have freed the node, **you haven't actually cleared the pointer from memory**, so set it to NULL (del = NULL;)
6. The list should have one less element without the program throwing any seg faults.

Appendix

Microprocessor Instructions

4001

Memory Cells: 16, each storing a value 0..15

System Registers: IP, IS

General register: R

0	Halt	Stop the program
1	R = R+1	Increase R by 1
2	R = R+2	Increase R by 2
3	R = R+4	Increase R by 4
4	R = R+8	Increase R by 8
7	Print R	Print the contents of R

4002

Memory Cells: 16, each storing a value 0..15

System Registers: IP, IS

One general register: R

0	Halt
1	R = R+1
2	R = R-1
7	Print R

4003

Memory Cells: 16, each storing a value 0..15

System Registers: IP, IS

Swap Register: SW

Two general registers: R0, R1

One-byte instructions:

0	Halt
1	R0 = R0+1
2	R0 = R0-1
3	R1 = R1+1
4	R1 = R1-1
5	Swap R0 <-> R1
6	Ring Bell
7	Print R0

Two-byte instructions:

8	Jump to <address> if R0 != 0	(!= means "not equal to")
9	Jump to <address> if R0 == 0	

4004

Memory Cells: 16, each storing a value 0..15

System Registers: IP, IS

Swap Register: SW

Two general registers: R0, R1

One-byte instructions:

0	Halt
1	R0 = R0+1
2	R0 = R0-1
3	R1 = R1+1
4	R1 = R1-1
5	R0=R0+R1
6	R0=R0-R1
7	Print R0

Two-bye instructions:

8	Jump to <address> if R0 != 0 (!= means "not equal to")
9	Jump to <address> if R0 == 0
10	<value> -> R0
11	<value> -> R1
12	R0 -> <address>
13	R1 -> <address>
14	R0 <-> <address>
15	R1 <-> <address>

8005

Registers: IP, IS

Two general registers: R0, R1

8 bit bytes (memory cells can store numbers between 0..255, registers can too)

256 memory cells (whose addresses are 0..255)

There are one and two byte instructions. For two byte instructions the first byte is the instruction code and the second byte is called its "data"

Instructions:

Code	Size (bytes)	Meaning	Notes
0	1	Halt	
1	1	R0 = R0+1	
2	1	R0 = R0-1	
3	1	R1 = R1+1	
4	1	R1 = R1-1	
5	1	R0=R0+R1	
6	1	R0=R0-R1	
7	1	Print R0 as an unsigned int	
8	2	Jump to <address> if R0 != 0	data cell contains the address to jump to
9	2	Jump to <address> if R0 == 0	data cell contains the address to jump to
10	2	<value> -> R0	load the value stored in the data cell into R0
11	2	<value> -> R1	load the value stored in the data cell into R1
12	2	R0 -> <address>	write the contents of R0 into a memory cell. The memory cell used is the one pointed to by the data cell
13	2	R1 -> <address>	write the contents of R1 into a memory cell. The memory cell used is the one pointed to by the data cell
14	2	R0 <-> <address>	swap the contents of R0 with the contents of a memory cell. The memory cell used is the one pointed to by the data cell
15	2	R1 <-> <address>	swap the contents of R1 with the contents of a memory cell. The memory cell used is the one pointed to by the data cell
16	1	Ring the bell	Bing!
17	1	Print the value in R0 as a char	The value stored in R0 is interpreted as a character using the ASCII convention