COMP1511 Notes

Lec 1.1

Operating systems (OS) are interfaces between users and the hardware of a computer

- OS execute user programs and make solving problems easier, and also makes the computer system easier to use

Compilers are translators that turns formal human readable C into the actual machine readable program

- Compilers allow use to 'run' programs
- We use dcc

C is the programming language we use for its clarity, speed, flexibility and modern relevance

Terminal Commands

ls

- Lists all files in the current directory

mkdir directoryName

- Makes a new directory called directoryName

cd directoryName

- Changes current directory to directoryName

cd ..

- Moves up one level of directories (one folder level)

pwd

- Tells you where you are in the directory structure at the moment and the path

touch fileName

- Creates a file called filename
- If you put filename.c, vscode will know it's a file in C so the autocomplete suggestions will be for C

code fileName

- Opens fileName for you to code in on vscode

cat fileName

- Shows you what you code you have written in filename

dcc fileName -o Name

- Compiles fileName so that we can write out a file called Name and run it
- Name must not have .c

- You need to compile files again after new changes have been saved for the new saves to appear when you run it

./Name

- Runs the program called Name that is in our current directory

Coding

Basic structure:

```
#include<stdio.h>

int main(void) {
        …;
        …;
        …;
        return 0;
}
```

Every line of code must end with a semicolon (;)

#include<stdio.h>

- Loads the standard input/output library

int main(void) {

- Int is the output (return) type, and stands for integer
- Main is the name of the function
- Void makes it so the function (area appearing under) is blank for you to write from scratch

return 0;

- Returns 0 to the operating system

printf("…\n");

- Prints the "…" string to the standard output (printf = print format)
- \n tells it to continue on a new line after running the program on terminal

//comment

- Allows you to add comments to describe what we want the code to do without impacting the code itself
- Using /* and */ will make everything in between a comment

Lec 1.2

Computer memory can be thought of as a big pile of on-off switches called bits

- Bits are the smallest possible unit in computing, and are choices between two things; a 0 or a 1
- 8 bits are called a byte
- Data is stored as bits in memory to 'remember' things

Variables are artifacts (things) we use to store information, and its value can change

- Think of it as a location in memory we can store a certain number of bits that represent something in
- Variables are created with a specific purpose in mind
- Three data types of variables:
  - int → integer
  - char → a single character
  - double → a floating point number (decimal?)

Variables must be named

- Names are a label and description of the variable, and we can pick them ourselves
- We should use lower case letters in C, noting that C is case sensitive and reserves certain words like "int" and "return"
- Multiple words should be split with an underscore
- Names should be obvious about what they represent
- Type the data type followed by the name

Data type int

- Commonly uses 32 bits (4 bytes), so 2^{32} different possible values only (there is a limit to how many integers can be stored)

Data type char

- A char type stores a single character wrapped in single quotes → 'a'
- Each char is associated with an integer, and can be converted to and from one
- C is case sensitive so 'a' is different from 'A'

Data type double

- Usually 64 bits (double the int count)

Terminal Commands

mv fileName_1 fileName_2

- Moves all content from file 1 to 2

rm fileName

- Removes (deletes) the file

Coding

DataType vName

- Names a variable of DataType vName

vName = Value

- Assigns a Value to variable vName
- Value must be of the same data type as the variabke
- = is the assignment operator

DataType vName = Value

- Names and assigns a value together in one line of code

printf("… %d\n", vName);

- Prints "… Value" provided that the variable has been named and assigned a value
- % is a format specifier that must be followed by some character to let the compiler know what data type to print
- In this case, d in %d means decimal integer
- The \n goes at the end of the string
- For multiple vairables, just put 2 slot indicators (%d) and give both variable names in the order u want them to appear (vName_1, vName_2), and the data type of the slot and the vName must match
- %lf → for long floating number (a double)
- %c → for character

scanf("%d", &vName);

- Reads input from the user in the same format as printf → you reply to it in terminal when you run it
- & tells scanf the address of the variable vName (where it is located) in memory that we want to place the value into → think of & as "scan into" vName
- For char, scanf does not wait for you to enter a char, and will accept nothing (white space) as a char, so put a space before %c → " %c"

Math operations are the same as they are for maple

- + for add
- - for subtract
- * for multiply
- / for divide
- E.g. vName_3 = vName_1 * vName_2 → outputs the product of vName 1 and 2 using their assigned values

#define cName value

- Defines a constant cName to be the value input
- Goes under before int main(void)
- cName always all CAPS

Lec 2.1

Control flow → sometimes we need to make decisions in programs

- We can make our programs branch between sets of instructions using the 'if' statement

In C, true and false are integers

- true → 1
- false → 0

Edge cases are cases where user input is on the 'edge' of a condition for if statements

- i.e for x <= 10, user input of 10 is an edge case

Coded files for lecture:

- control_flow_intro.c → ./cfi
- if_else.c → ./ifelse
- loops.c →

Terminal Commands

Coding

```
if(condition) {
        do_something ();
        do_something_else ();
}
```

- Determines the result of a Boolean (true/false) question, and allows something to be done if something is true
- The <condition> is something that evaluates to true or false
- Everything between {} will run if the condition is true
- If statements can be chained by putting another if statement for a <second condition> immediately after the closing } bracket of the first if statement

Boolean operators

- < → less than
- > → greater than
- <= → less than or equal to
- >= → greater than or equal to
- == → is equal to → == is NOT the same as =, which is assignment
- != → not equal to
- && → and
- || → or

- ! → not

```
} else if (<condition>) {
        do_if_false ();
}
```

- Runs a block of code if <condition> for associated if statement is false
- Must be associated with an if statement → follows immediately after the closing } bracket of an if statement block
- Do not need "if (<condition>)" if you just mean otherwise
- A chain of if, else if, else statements mean if first is true, do this, if first is false and second is true do this, if first and second are false, do this

```
while(<expression>) {
        do_something();
}
```

- C starts at main and executes each line in sequence, and while statements control (loops) that sequence
- 3 types of while loops
    - Counting loops
    - Conditional loops
    - Sentinel loops
- While the <expression> is true, keep doing whatever is between the {brackets}
- When the while statement reaches the end, it jumps back to the start and tests the <expression> again, looping until it is false where it will not run the block
- You can add lines in the while block that impact variables in the <expression>

```
int i = 0;
int num = n
while(i < num) {
        do_something();
        i = i +1;
}
```

- Basic format for a counting loop that does something n times
- The 'i = i+1' impacts the <expression> 'i < num' to make it work as a counter → without this line, it runs an infinite loop → if that happens, press "ctrl + C" to stop running the program

FORMATS FOR CONDITIONAL AND SENTINEL LOOPS LOWK DODGY

- loops.c is a better example of a conditional loop

```
int vName_1 = x;
int vName_2 = y;
int vName_3 = z;

while (vName_2 < vName_3) {
        Do_something();
        vName_2 = vName_2 + vName_1;
```

}

- Basic format for a conditional loop
- Keeps doing something until the desired value for vName_2 is achieved

```
Int vName_1 = a
Int vName_2 = b
Int vName_3 = c
Int vName_4 = d

While(!vName_4) {
        Do_something();
        vName_3 = vName_3 + vName_1;

        if (vName_3 > x) {
                vName_4 = 1;
        }
}
```

- Basic structure of a sentinel loop
- We manually flag when we want to stop looping using the sentinel variable (vName_4)
- Loops until vName_3 > x → the desired value (x) for vName_3 is reached

(rand() % x)

- Generates a random number between 0 and x
- % x → modulo x
- Rand() → generates a random number → needs #include <stdlib.h>

Lec 2.2

Nested loops

- A while loop within a while loop
- Putting a variable from the outer loop in the inner loop affects what the inner loop does each time it runs

Custom data types

- Custom (non-built-in) data types store a group of related data
- Structs are data types we define to store a collection of types → variables that are made of other variables
- Allows a group of variables to have a single identifier while still allowing you to access those variables
- enums are another data type that store a range or set of possible values
- enums provide limitations on possible values (states defined in the enum), which have a nice label
- we can have enums within structs after we define an enum

Coded files for lecture:

- nested_loops.c → nested_loops
- structs.c → structs
- enums.c → enums
- data_type_demo.c → data_type_demo

Terminal Commands

Coding

```
struct dName {
        int vName_1
        char vName_2;
        double vName_3;
};
```

- defines a struct and the sub-variables in that struct → no values assigned
- sName becomes a custom variable
- needs semicolon after closing } bracket
- Define structs before int main(void)

struct dName vName

- makes a variable named vName of dName data type
- vName.subvName = x → accesses the sub-variable called subvName within vName, and assigns a value x to the sub-variable
-

enum eName {set of states};

- Creates an enum called eName with a set of states
- This defines the states and the enum
- Need semicolon ; after closing } bracket, and goes before int main(void)
- enum eName vName = state_1;
    - enum eName is the data type, vName is the variable
    - assigns a variable vName to a state state_1 in the set
- states are integers, so we use %d to slot them, and the value of the state is its place in the set counting left to right starting from 0

Lec 3.1

Functions

- printf, scanf and main are examples of functions
- Functions are reusable blocks of code that may have:
  - Input (parameters)
  - Actions (side effects)
  - Output (results)
- Calling on functions execute their body, though input may be necessary
- We can call a function from anywhere in our programs and we can access the result of the function (if any)

Functions terminology:

- Return type → the data type returned by the function
- Result → the actual value returned from a function call
- Parameters → the type and sequence of data to be passed into a function
- Argument → the actual value passed into a function's parameters when called
- Return → the keyword used to end a function and return the result following

Procedures refer to certain types of 'functions'

- Aren't a real thing in C
- Functions that don't return a result are called void functions or procedures
- Procedures do something, but don't have a result and usually have a side-effect

Coded files for lecture:

- functions.c → functions

Terminal Commands

Coding

```
DataType fName ( Parameters ) {
        …
        return Expression;
}
```

- The DataType tells the computer what type of data the result should be
- fName → names the function fName for you to call on later
- Parameters → the sequence and type of input to be passed in
- return Expression → evaluates the expression and returns (calculates) the result → does not automatically print it → return also ends the function
- … → every line of code that will run when the function is called

- If we use an if statement, we need a return within the if block, and a return outside the if block → the return outside the if block acts as an else, but does not need to be formatted as an else statement
- Functions must be declared before they are used, just as variables must be → forward declaration → DataType fName (Parameters); can be placed at the top of the main block to define a function fName for later use

fName (Parameters);

- Calls function fName, giving the inputs within the parameters (correct sequence and data type)
- The () means call
- We call functions within the int main (void) part, but code the function before
- The parameters can be variables of the correct data type defined in int main(void)
- Function calls can act as variables

```
void pName () {
        …
}
```

- This is a function which returns nothing (void) → can be called a procedure

Lec 3.2

Data structures (not structs) are common structures used to store multiples of data

- Multiples of data are usually of the same data type
- Data can scale, allowing a vast range of numbers of elements of data to be stored
- Arrays and linked lists are fundamental data structures

Arrays are collections of data of the same type

- Arrays have a single identifier
- Arrays are a random access data structure → we can access any element in the array at any time
- We can read or modify individual elements in arrays
- Arrays are contiguous data structures → elements are all stuck together, as if they were in a grid
- Each element in an array has an index, starting at 0

Static arrays are arrays that have a set size specified by you

-

Coded files for lecture:

- arrays.c → arrays
-

Terminal Commands

Coding

DataType aName[Num] = {a, b, c, d, e …}

- First half declares an array named aName that stores Num amount of elements (indexed 0 to Num – 1) of DataType data
- Second half enters (initialises) data values a, b, c … into the array in that order
    - An empty set {} will initialise all elements to 0
    - Can only be done when declaring the array, not later

DataType vName = aName[Index]

- Assigns the viariable vName of data type DataType to the element that corresponds with Index in the array aName
- This is how you access elements

aName[Index] = Value

- Changes the value of the element corresponding with Index in array aName to Value

Lec 4.1

Strings are multi-character words

- C does not have a string data type, but we can use arrays to work around this → arrays storing chars can effectively act a string data type

The null terminator → \0

- Since we don't know when arrays end, we have to keep track of the length ourselves → we can do this by placing the null terminator "\0" at the end of our character arrays
- It acts as a character and occupies the final slot in the array, but cannot be rendered

The <stdio.h> library also has a fput and fget function for printing and scanning (respectively)

- While %s does work for printf, it is ill advised → not allowed for comp1511

Coded files for lecture:

- string_intro.c → string_intro
- strings_demo.c → strings_demo

Terminal Commands

Coding

char sName [] = {'w', 'o', 'r', 'd', '\0'};

- Declares a string in a character array called sName for the string "word"
- We don't need a number in the square brackets [], since the null terminator tells the computer when the string (character array) ends
- BUT we can only do this when we declare the array, so there is another method

char sName [] = "word";

- Declares a string in a character array called sName for the string "word" → does everything the one above does
- This way allows us to assign strings to char arrays easily
- Automatically includes null character when double quotes "" end

fgets (aName, length, stream)

- Scans in a string of the length specified from the terminal into the array aName
- Stream specifies the origin of the string we want to scan in, which is always 'stdin' (standard input) for comp1511
- When we don't know the length of the string we want to scan in, just put the max length → #define MAX_LENGTH value
- Scans in the characters in the string, any newline characters, and the null \0

- We must declare the array aName before using it
- fgets (aName[], MAX_LENGTH, stdin); → what we usually have
- doesn't need & to scan into aName
- We can read until CTRL+D is entered in the terminal by calling fgets in a loop, since fgets stops reading when length-1 or newline characters are read, or when an end of file (EOF or NULL signified by CTRL+D by user or \0) is reached

NOTE: MOST (OR ALL) OF THE BELOW str HEADED THINGS NEED #include<string.h>

fputs (aName, stream);

- Prints the character array aName
- Stream tells it where to print → we always use 'stdout' (standard output) in comp1511

strlen(aName)

- Gives us the length of a string (excluding the \0)
- Can be used to edit what a certain character in the string is → e.g. to remove the newline character in a string so we can have a sentence on one line

strcpy(aName_1, "string")

- Copies the contents of one string to another → assigns aName to "string", and "string can just be another array
- We can't say aName_1 = aName_2, so we use this

strcat(aName_1, aName_2)

- Joins string aName_2 to the end of aName_1

strcmp(aName_1, aName_2)

- Compares two strings
- Used commonly to check if string aName_1 is the same as string aName_2 since we can't do ==
- Returns 0 if the strings are the same, and a rough calculation (sum of the difference of each character slot's ascii value) of how dissimilar the strings are if they aren't
- Can also be used to sort strings

strchr(aName, 'a')

- Finds the first occurrence of a character (in the above case, 'a')

Lec 4.2

We can make arrays of structs to store groups of data under each index

- We can change or input values by using aName[index].subVariable
-

Coded files for lecture:

- 2d_arrays.c → 2d_arrays

Terminal Commands

Coding

DataType aName [ROW] [COL];

- Creates an array with ROW rows and COL columns (both indexed starting at 0)
- aName [i] [j] → accesses the data in the ith row and jth column

Lec 5.1

Command line arguments

- We can provide input via user command (scanf), but if we want to pass input to a program without user input we can modify main to allow for CLI (command line input)
-

Coded files for lecture:

- cli.c → cli
- atoi.c

Terminal Commands

Coding

```
Int main (int argc, char *argv[]) {
        …
        return 0;
}
```

- argc is the number of arguments
- argv stores each string passed into the program (think of it as an array of strings)
- There is always one argument in the program → its name
- ./program fuck you → two arguments → argv[0] is ./program, and argv[1] is fuck, and argv[2] is you → whitespace separates each string CLI
- Allows us to pass data into the program when we first run it
- If we actually want to enter an integer as an integer and not as a part of a string, we need “” marks → need this for actual math operations in program

atoi(const char *str)

- Function that changes a string read integer into an actual integer
- Included in stdlib.h
- A non-integer string just becomes 0
- Also have atol (long), atof (float), and atoll (long long) functions
- Const char*str is one of the cli → you would put argv[i] here for the ith string you want

```
for (int i =0; i < num; i++) {
        …
}
```

- The counting loop is so common that this shorthand (sort of) exists
- Note that i will not exist outside of the loop

Lec 5.2

Pointers

- Memory can be thought of as a large 2D grid, and each cell of this grid has a unique identifier (an address)
- Storing this address in a variable gives us a pointer, which allows us to access the address instead of the data stored in that address
- Allocating memory is 'expensive' so pointers are useful in that they
- Arrays in C are actually pointers that store data in neighbouring addresses
-

Dereferencing is accessing the value at the address of a pointer

Coded files for lecture:

- pointers_intro.c → pointers_intro
- useful_pointers.c → useful_pointers
- pointers_arrays.c → pointers_arrays

Coding

- %p is the identifier for pointers
- We need an &vName to access the address a variable is stored in → e.g. printf("%d", &vName) will print the address vName is stored in, not the value of vName
    - & queries the address → &vName gets you the address of vName

DataType *vName

- Declares a pointer called vName
- The * means request the memory to store a memory address of a variable of data type DataType, not the storage of of a DataType value itself
    - void *vName creates a variable vName that can store the address of any data type
- E.g. int *pointer creates a variable called pointer that stores the address of integers
- Note that the pointer itself has an address as well
- We can assign the addresses of other variables of DataType to the pointer vName
    - vName = &vName_2 → vName now has the address of vName_2 as its value
- We do not need the & to access vName now that it already stores an address
    - Printf("%p", vName) will print the address of whatever vName is assigned to

*pName

- Dereferences the pointer pName → it will get the value with the address stored in pName

Lec 7.1

The stack is where information about your program execution is stored

- Includes which functions are called, and in what order
- Includes which variables are created, and where
- When a block of code is executed { }, a stack frame is created, and when the block is completed, the stack frame is removed, destroying anything inside it
    - Enough memory to store everything (variables, arrays, etc.) in the frame is allocated to the frame
    - Pointers to variables in a stack will point to unallocated memory when that stack gets removed, and dereferencing it will cause an error
- If we want to create memory with an undetermined size, we can't use stack frames since the program needs to know how much space is needed, so we use the heap

The heap is a large block of memory that sits outside the stack

- In C, the heap is managed entirely by the programmer → nothing is automatically declared or destroyed
- C provides some functions to interact with the heap
- We will need to manually clean up the heap ourselves, else we will cause a memory leak → usually, C automatically frees stack frames after they finish so we don't have to, but this is not the case in the heap

Coded files for lecture:

- dynamic.c -o dynamic
-

Coding

NOTE: malloc, realloc and free require <stdlib.h>

ptr = (cast-type*) malloc(byte-size)

- Returns a pointer named ptr to an allocation of byte-size size memory in the heap
- We can decide how large the allocation is
- (cast-type*) → tells the heap how to read the data in the heap
- We can create a memory allocation in the heap by just using malloc(byte_size), but we would have no pointer to it so its pretty useless
- Malloc is an in built function like printf → cannot use outside of functions
- NOTE: malloc just creates a blob of memory, YOU need to manage data types and shit in the blob

sizeof(dType)

- Returns the size of a data type dType in bytes
- Useful with malloc as you can do things like malloc(sizeof(char) * 50) to allocate 50 characters worth of memory in the heap, which is effectively an array on the heap (NOT a character array, it can just store 50x the size of a character)
- sizeof(sName) will give the size of a struct sName and all its subvariables
- sizeof with malloc and other code is critical to creating dynamic arrays

DataType *ptr = malloc(num_elements * sizeof(DataType));

- This creates an array of data type DataType which can store num_elements of DataType, and stores its address in the heap in a pointer called ptr
- ptr can then be treated as any normal array → ptr[1] returns the element stored in the slot indexed 1
- If a function with this declaration for ptr ends, ptr will not be destroyed as it is stored in the heap

free(ptr)

- This frees up (removes) the memory allocated to the address stored in a pointer ptr → prevents memory leaks
- After ptr is freed, it cannot be accessed

realloc(ptr, byte-size)

- Resizes/reallocates the memory allocated to ptr to an allocation of byte-size size memory
- Either resizes the existing allocation (frees what is no longer needed) at ptr or allocates a new memory allocation in a new location (address) for ptr
    - Allows the size of arrays created in heaps to be changed
    - Allocates a new memory allocation if there isn't room to extend the current allocation (the memory allocated must be contiguous), or if its more efficient for the allocation to go elsewhere
    - Will copy the allocation, move it elsewhere with the new size, and free the old allocation
- Always returns the address of the new allocation, even if it's in the same position (address)
- NEEDS ptr to have been malloc-ed before

Summary:

- malloc() → allocates memory
- free() → deallocates memory
- realloc() → grows/shrinks memory
- all require <stdlib.h>

Lec 7.2

Lec was just a demo, extra notes and little tips

Coded files for lecture:

- lecture_7_2_demo.c -o lect_7_2_demo

Coding

++iName vs iName++

- ++ before adds 1 to integer iName then returns the value of iName
- ++ after gets the value of integer iName then adds 1
- int iName_1 = ++iName_2 == int iName_1 = iName_2++
    - But for int iName_1 = ++iName_2, the value of iName_2 does not change
    - Whereas for int iName_1 = iName_2++ the value of iName_1 and iName_2 are the same after the declaration → iName_2 is incremented by 1

Lec 8.1

Linked lists

- Sort of like arrays in the heap except they DON'T have to be contiguous → each element stored in the array is split up in the heap, and each element points to the next element
- Each element is called a node → nodes are usually structs storing any data we want AND a pointer to the next node
- Useful since they are also dynamically sized but do not need to be realloced to change size → However, to access data in nodes u need to start at the first node and go through each node until you arrive at the one you want (need to follow the pointers to reach the node
- We can also add or remove nodes anywhere in the linked list, and can change the order of the list

Coded files for lecture:

- linked_list_intro.c -o linked_list_intro
-

Coding

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

- Declare your nodes as structs (nodes are structs) with any data you want and a pointer to the next node
- In main or functions, you will malloc each node to a space in the heap with the size of your node, and the first node is typically called "head" → struct node *nName = malloc(sizeof(struct node));
    - Creates a memory allocation the size of a struct node for a node called nName
    - The head node acts as the first node, and also stores the location of the entire linked list since we need to follow the trail to access each node
- Then you will set the pointers to the next node for each node
    - head->next = nName_2;
      nName_2->next = nName_3
      …
      nName_last->next = NULL;
    - Like with dereferenced structs, we use arrows to access subvariables (in this case, next is a pointer of node struct type storing the address of the next node)
    - We assign NULL to the last node since there is no node after it

Lec 8.2

Inserting at the middle of the linked list

- NEED to put the next address as curr->next
-

Coded files for lecture:

- Continued on linked_list_intro.c
    - Added insert_at_index function
    - Added remove_tail function
    - Added remove_at_index function

Coding

Lec 9.1

Coded files for lecture:

- Continued linked_list_intro.c
  - reverse_ll function
  - create_student func
  - print_student func
  - Also changed data from int to struct student, which required changes everywhere

Coding

typedef dType Name

- Allows data of dType to thereafter be referred to as Name
- Useful for structs because then u don't have to type struct car_space carpark[…][…], and could just say Car_space carpark[…][…]
- Title capital the Name
- Typedef struct sName {
  …;
  …;
  …;
  } Name;
  - Creates a struct called sName and names it Name thereafter

```
struct list_node {
        struct node *inner_list_head;
        struct list_node *next;
};
```

- This creates a list of nodes which contain lists themselves → effectively creates a 2D list
- Think of list_nodes as rows → instead of doing row++ after looping through every col with col++, after we traverse the normal nodes with curr = curr->next, we go to the next list_node using curr = curr->next (tho prob change names of curr)

```
struct list_node *curr_ln = head;
while (curr_ln != NULL) {
        struct node *curr_n = curr_ln->inner_list;
        while (curr_n != NULL) {
                …;
                curr_n = curr_n ->next
        }
        curr_ln = curr_ln->next;
}
```

Lec 9.2

BONUS lecture

Coded files for lecture:

-

Coding

Lec week.Lecture_Num

Coded files for lecture:

-

Coding

Lec week.Lecture_Num

Coded files for lecture:

-

Coding