FOUNDATIONS OF ALGORITHMS ~ UNABRIDGED NOTES

BASICS OF C PROGRAMMING

- Put a semi colon *;* after each command if the overall command is not yet done
- We put *int main(int argc, char *argv[]) {* before the block of all of our commands. This is itself a function – we implement all of the block following main(…) in the code
- To **print** we write *printf()*, with \n meaning adding a new line, and \*"* including the quotation mark without ending the quote
- Effectively {} now will represent blocks, with the block of code itself nested inside these braces- we don't need tabs in C

Calculations

- <u>An improper fraction with two integers will still create a new integer (it will round down) – you should thereby represent the initial integers as doubles</u> – You should use variables of the same type under operations
- A calculation with an int and double type will **return a double** (think of doubles as infectious). The *%* operator, however, must be used only with integers
- For indices: pow(2,3) = 2**3
- There is no built-in indices mechanism within C
- To round a number up use ceil(*number*), floor() for round down
- /= and *= are 'divide' and 'multiply by __ and reassign' respectfully

Comments

- */ * comments comments * /*
- / * will begin the comment, * / will end it
- We can comment out a piece of code instead of removing them altogether

Variables

- To initialise variables, it must be written as the structure:
- **data type* *VariableName* = "value"*
- If we want to assign the variable with multiple data inputs (ie a string will have multiple characters and is therefore a collection of inputs) we must put [] directly after the variable name, WITH A NUMBER WITHIN THE BRACKETS, denoting the maximum amount of data inputs within the variable. This is called an **array** (see below):
- We can declare (multiple) variables on a line without associating a value
- *int characterage = 35* vs *char charactername[5] = "john"*
- **int** = integer; **char** = single-character string; **float** = float; **double** = a more accurate float, more space
- we don't have to re-initialise the data type when resetting the variable
- To associate a value with a variable which cannot be altered later in the code we use *const* before introducing the data type and variable name: *const int num = 8;* OR use #define as outlined below
- **static** variables, which are declared by writing *static* before the data type at the declaration line, are only read once- as such, it will be ignored as the code reads over it again, meaning the variable will retain the value with which it was initially initialised, and which may or may not have been altered. Use const instead
- We can use these static variables to, for example, track how many times a function has been called – include the static int *counter* as 1 at the start of the function block. Each time the code runs through, instead of reinitialising to 1, it will increment.
- *#define *vname* *value** will mean you can use the variable name to represent the value – USE THIS FOR ALL CONSTANTS that **will not change.** It should generally be used to introduce **magic numbers**

Arrays

- When we set a variable to an array we use *[]* after the variable name, then represent the different pieces of data around {}, separate them with commas

- We index with [], like python, with 0 being the first element, and we can modify or add any element in an array by just referencing the (new) index

- In situation where we add to arrays, we must first specify the maximum space of the array **with an integer in the square brackets** when first initialising the variable

- **Nested arrays** work the same as nested containers in python. When we introduce a nested array the desired number of array dimensions is the number of square brackets after the variable name. Each square bracket should contain the number of elements desired in each dimension/nest

Pointers

- Each variable has a **memory address**, which is the place where C has stored the value of the variable within the RAM. To access this memory address, we must type: *printf(&p, &*variablename*)* In this instance p represents **'pointer'**
- Pointers are another type of data type – it represents a memory address
- To create a pointer variable from a memory address of an already established variable:
- **data type of established variable* ***** *p*newvariablename* = &*estab.vname**
- **Dereferencing** a pointer variable means we're grabbing the actual value of the memory address: *printf("%d", ******pointervariablename*)*
- & is used to reference a pointer, * to dereference
- To declare a pointer, we **must** have an asterisk preceding the variable name in the declaration. THIS MEANS 'POINTER TO INTEGER TYPE'. (ie int *) Use of the variable name without the asterisk from here will thus return the memory address, with it will be the value. Pointers have their own type, ie int* is of type "pointer to an integer variable". (Asterisk when initialising variable means pointer)
- Functions that need to alter their arguments must receive pointers; the corresponding call of this function must provide addresses of variables of the same type. This is the way we can store the value of a variable through scopes
- *void** allows untyped pointers – pointers which point to no particular type but instead to any memory address.
- **Scope:** variables used within a function are abandoned afterwards, like python. Variables named the same inside and outside a function are unrelated, and ignore each other
- The variable which is a pointer's value is the address in memory. When we want to **dereference,** we use an asterisk in front of the variable – this dereferencing means we are grabbing the **value stored at the pointer's address**. When we want to initialise a pointer, <u>we initialise the variable as the type whose address it will hold</u> with an asterisk before it, then in a later line, we set up *pvariable = &othervariable*, which will grab the address of othervariable
- In this example, *othervariable == *pvariable* and *&othervariable == pvariable*, meanwhile, pvariable without the asterisk is just the address
- * is for dereferencing, & is for referencing (obtaining address)
- We can alter the address of a pointer variable, but this will affect its dereferenced value

F-Strings

- We need to use f-strings all the time when printing any variable
- we use %*letter* in place of the part of the string we want the variable to represent, then include the variable/value after the quotation marks, after a comma: *…", CharacterName)*
- You would have multiple ordered comma-separated values for more than one f-strings
- We can do calculations in the right-hand side to create new float for printed string
- If there are multiple desired inputs into the print statements we could just use *{*variablename*}* instead of *%*letter**
- We can use a field width when printing: *%10d* means print an integer, to at least 10 characters, the unused ones are taken by spaces. *%8.2f* means 8 characters in total, 2 of which after the decimal point. *%-20s* formats a left-justified string with at least 20 characters. (Right justified is the default, given that you specify the width of the line for which it can be right justified.)
- %3d means use **up to** 3 digits
- %.4f means use 4 digits after the decimal point
- %-6d means left justify
- %-6.2f means 6 digits all up, 2 after decimal, left justify
- %d for what is a character will convert it into ascii code
- %c for *k+3* will print the character 3 after k in the ascii code

User Input

- *int *variable*;* define the variable before scanning
- *printf("Enter your age: ");*
- *scanf("%d", &*variable*);* we are storing the inputted age as the variable
- *printf("You are %d years old.", variable);*

TYPE	scanf CONTROL STRING FORMAT DESCRIPTOR	printf OUTPUT LETTER
Integer	%d	%d
Double	%lf (long float)	%f
Float	%f	%f
Exponential	%e	%e