Chapter 1:

* C is a by-product of UNIX systems (Think MobaXterm and CMD).
* Made at Bell laptops
* Relies on compiling source code( ssh command is gcc -Wall -std=c99 -o outputname programBeingComplied
  + Gcc is compiler
  + -Wall gets all warnings
  + -std-c99 sets standard to c99
  + -o is output to a specified name
* Many C based languages (C++,C#,Java,Perl etc.)
* It is a small Low-level language
  + Limited features
  + Good for system programming
  + Access to machine level concepts
  + Op.s closely correspond to computers built in instructions
* C is a permissive language
* Strengths:
  + Efficient: As it is intended for apps. where assembly language is used
  + Portable: Compilers are small and easily written
  + Powerful
  + Flexible and includes a standard library
  + Integration with UNIX
* Weakness:
  + Error prone: Can miss errors when compiling and programming has many pitfalls
  + Difficult at first glance
  + Tough to modify
* Good ways to avoid issues are to avoid pitfalls, use software tools and libraries, use good conventions, and try to avoid complex code

Chapter 2:

* Always include #include <stdio.h> at the top of programs
* All C programs have a name format of name.c
* Most programs usually go under int main(void) where void means nothing is passed in.
  + In this scenario, always include a return 0
  + This means return the value 0 upon termination
* Printf is used to print out a line.
  + Has the form printf(“Line of text, ” %\*, variable name)
    - \* is a temporary replacement, usually is filled with d, c, s, f for integer, character, string, float when wanting to print a variable(below) out
    - \n= new line
* To execute a C program, 3 steps:
  + Preprocessing
  + Compiling
  + Linking: Linker combines object code produced by the compiler with an other code from lib functions
* Above 3 usually automated
* IDE= integrated development environment
* After declaring a procedure header i.e. int main (void), always follow up with a { and close at the end of procedure/function with }
  + Function returns something
* Always end each line with ; with a few exceptions like loops and procedure/function declaring
* You must always have a main, otherwise the program will not run
* To write comments being with /\* and end with \*/
  + In C99 // is also a comment
* Every variable has an associated type and must be declared when creating a variable
  + Int
  + Float-care with rounding
  + Char
  + Uint
* To assign, just do variable name = Thing u want to assign(name is case sensitive)
  + When assigning floats, include an f after the thing you want to assign
  + Names can’t start in numbers no can have any space, for space use \_
  + Variable names can’t be special worsds like continue, float, void etc.
* Not necessary to assign a variable a value when creating it
* When reading an input use scanf
  + Form is scanf(%\*,&variable)
  + Refer to print statement for the \*
  + & acts as a pointer for the specific variable
* When using constants, u can use macro definition
  + Form is #define CONSTANTNAME value
  + No need for semi colon here
* C is case sensitive

Chapter 3

* Printf allows us to format an input
* This is done by inserting at specified parts within the printf call (%\* is that place)
  + Form as mentioned above is printf(string, expr1, expr2);
* Make sure the number of conversion specifications(%8) matches so an error doesn’t occur
* We can also use 2d modification on the printf statement
  + A general form could be %m.px
  + M is the number of characters the output can have as a max. Spaces or a specified character will pad out the output
  + . is a period that is usually linked with p. P is precision. This deals with the number specifiers of:
    - D-integer
    - E-exponential
    - F float/fixed decimal format
    - G float in f format or e format
  + P in d displays min num of digits to display and if omitted p=1
  + P in e and f shows how many digits appear after the dot(6 is standard) and if not displayed, p=0
  + P in g is different, it displays the man number of SIGINIFICANT figures to be displayed. Note: g doesn’t show tailing zeroes and won’t show decimal point if nothing after said point
* Scanf also has formatting that can be done
* Always put the & in scanf to avoid functions
* You can scan in multiple variables in scanf
* It skips over white-space characters by reading them consecutively until it reaches a character

Chapter 4

* You can use various operators within C (Arithmetic, Relational, Logical)
* Additive and Multiplicative are known as Binary due to needing two operands
* There are unary operators which only need one operand(unary plus and minus)
* When using % as an operator., it gives the remainder.
* When int and float are mixed using operators, the result is always float.
* When dividing two integers, the result is truncated and drops fractional part (1/2 will become 0)
* % must be done with integers
* Avoid using 0 on RHS
* There is order precedence incase no brackets are used
  + Order is Unary + and –, then binary multiplication \* / and finally binary + -
  + This is used to understand how the compiler will compile the operation
* Associativity can occur in these cases (left and right can occur)
* Table of operators is consulted to due large number of operators
* To assign a value to a variable, we use a single equal sign
* When assigning one variable to another, the OG value is copied and stored in the desired variable. This means the OG is not changed
* You can chain assignments if needed
  + Just be careful when doing so
* LHS must be variable and nothing else
* You can simplify specific assignments eg i+=9
  + I is variable
  + Note, this can be done with other operators not just +
  + This is the same as i=i + 9
  + This may not always work so be careful
* To increment and decrement a variable, you can use ++ or –
  + They can be put before or after a variable but issues can happen depending where the inc/decrement is placed
  + Postfix causes the inc/dec to occur after statement execution whereas prefix causes it to occur before
* Postfix increment has higher priority than prefix increment which is higher than unary add/mult.

Chapter 5

* 4 types of statements
  + Selection: if ,switch
  + Iteration: Loops
  + Jump statements: break, continue and goto
  + Compound statements
* Relational operators
  + </<=
  + >/>=
* Equality Ops
  + != means not equal
  + == means equal
* Logical Ops:
  + ! = Negation i.e. 1 becomes 0 and 0 becomes 1
  + &&Logical and
  + || Logical or
* && and || both check the left hand side first and then RHS.
* If expression value can found via LHS alone, then no need for RHS
* If statement can be used to check a specific condition
  + Form is if (expressions) {statement}
  + Be careful about confusing assignment with equality
  + {} is used when controlling multiple statements, no need to include them if there is only a single statement but better safe than sorry
* If checking multiple conditions, you can use an else or else if clause
  + Same form as above
  + Else will perform a statement if It fails the if statement
  + Else if will check another condition if it fails the if statement
  + Multiple else if’s can be used
* You can use nested if’s (ifs inside an if) but make the program easy on the eyes
* C can allow an expression to produce one of 2 values depending on a condition
  + Form is expr1 ? expr2 : expr3
  + Often called ternary op
  + Read as if expr1 then expr2 else expr3 ( expr1 is read and if non-zero its expr2 else it is expr3
* Try and avoid conditional due to making a program harder to read
* To write a bool value in C for c99, write \_C99
* You can use stdbool.h to use Boolean values much more easily
* Instead of nested ifs, you can use switch case
  + Form is : switch (variable) {
  + case constant expression: statement; break;
  + default: statement; break;}
* Include the break otherwise issues may occur
* Break will force the program to exit the loop

Chapter 6

* While statement makes you loop through a specific statement until a condition is not met
  + Form is while (expression) statement
    - If a single statement, follow form above otherwise put in {}
* While (1) causes an infinite loop
  + Use break to exit this loop
* A do statement is similar to a while loop
  + Form is do {statement} while (expression)
* A for loop is used for things that have a known amount of loops or counting
  + Form is : for (expr1;expr2;expr3) { statement}
  + Expr1 usually assigns a counting variable i.e int i=10
  + Expr2 is the condition
  + Expr3 is the way the counting variable changes (increment or decrement)
  + You can rewrite most for loops into while loops and vice versa
  + You can omit some expressions, if one initializes a counting variable before the loop
  + In C99 you can initialize within the loop call and C99 will create a new version of variable used only within loop
  + You can initialize two variables in loops using a comma
* Break can force you to exit the loop
* Continue transfers control to a point just before the loop exits
  + It still remains in the loop
* Goto is allows one to jump to any statement given an identifier
  + Identifier form Is identifier:statement
  + Goto form is goto identifier;
  + Note this can not bypass declaration of a variable length array
* A statement can be null which means it is devoid of symbols except the semicolon

Chapter 7

* For displaying numerals, we have either floats and integers
  + Integers are further split up into unsigned and signed int
  + In a signed int, the leftmost bit is the bit that determines the sign, think 2-s complement. This is the default in C
* Many different types of int
  + Short/long/standard(as in nothing infront) int
  + Unsigned Short/long/standard int
* Can omit int unless necessary
* In C99, can also use long long
* Octal and Hexadecimal are both usable(8 digits and 16 digits respectively)
* If too large for int, compiler will try long int and if that fails it will try unsigned long int
* Ints ending in ll/LL have type long long
* If result cant be stored as an int, overflow will occur due to too little bits
  + Undefined behaviour on signed int
* New notations:
  + %o reads in octal
  + %x reads in base 16
  + %u reads base 10
* s/l/ll In front of modifiers reads it in short/long/long long respectively
* floats have 3 types:
  + float: single-precision floating point, 6 digits
  + double Double-precision floating point 15 digits
  + long double: Extended-precision floating point
* long double barely used
* double gives good precision which is enough for most
* float is the most used
* e int is a way of saying 10 to the power fo an int
  + comes after a float.
  + Stored as double ints
* If putting an f after constant, compiler will store it as a single precision
* New Notations:
  + %e exponential
  + %f float
  + %g either float or exponential
* Char assigned any single character
* Char can be used as ints so therefore it can be signed or unsigned
* Escape sequences used to provide a way to write special characters (think \n)
  + Octal escpae sequence has a \ and three digits in octal
  + Hexadecimal consists of \x and a hex number with no limit but must be unsigned
* Toupper allows one to check if a letter is lower case and returns the upper case for it
  + Must have #include ctype.h
* %c is the standard
* Scanf doesn’t skip whitespace before reading a char so to skip it but a space in front of ^c
* You can check if read char is new line character and skip it
* Getf and putchar are used instead of scanf and printf
  + Getf reads one character and returns it(returns it in int form)
    - Doesn’t skip white spaces
  + Putchar writes a single char
  + Both are faster than scanf and print f
* Implicit conversion: conversions done without programmers involvement
* Explicit conversion: done with programmers involvement
* Implicit is done when:
  + Operands in arithmetic aren’t the same
  + When left sift type != right side type in assignment
  + Argument in function call doesn’t match parameter
  + Type in return doesn’t match functions return type
* Recall when doing ops with a float and int, the int is changed into a floar
* The types of the operands can often be made to match by converting the narrower type to other operand
* Float-🡪double-🡪long double (least narrow to narrower)
* Int 🡪 unsigned int 🡪long int 🡪 unsigned long int
* Avoid unsigned ints as much as possible
* You can use other declared variables to convert something to other type
  + Be careful with floats and ints as ints can drop the fractional part
* Implicit conversion ranking:
  + Long long u/int
  + Long u/int
  + u/Int
  + short u/int
  + char
  + bool
* If neither is a floating type do one of following:
  + If both signed or unsigned, convert to type with higher rank
  + If operands are same rank, changed to unsigned
  + If one can represent all values with signed, convert other operand to that
  + Else convert to signed
* Cast form of (type name) expression
  + Can be used to convert one type to another
  + Used to force conversions we want
* Sizeof(type) determines how much meory is stored for that type
* One can use type defs to set up bools
* Typedef int bool will force compiler to recognize bool in list of types
  + One can use this for money
  + Great for portability
  + Form is: type def type name

Chapter 8

* Array is a data structure that have a number of data values of the same type
  + Values are known as elements
* 1D array is simplest as elements arranged one after another
* Declaration form of array is : data type name[num of element];
* Elements are indexed from 0 to (size of array -1)
  + To access an element use a[i] where I is index
  + Treat as variables
  + Note an index can be an expression giving an integer or a variable itself
* To initialize an array, multiple ways
  + You can give all the values, e.g. a[5]={4,2,3,5,6}
  + If number of values in RHS < size of array, remaining elements are filled with 0
  + To make everything 0, a[10]={0}
  + In C99 you can set specific elements in the following way : a[15] ={[2]=29, [9]=7, [14] =48}
    - Known as designates initializers
* To get length of array, use sizeof(a)/restri(a[0]) .(size of array/size of an element)
* Arrays can be multidimensional
  + Form to declare this is datatype arrayname[i][j]….Addin in a pair of square brackets for each dimension
  + 2d array is in form array[i][j]
  + If you use a comma instead of [][], the compiler will only look at the expression after the comma
  + Initialized similar to 1D arrays
  + ALWAYS INCLUDE INNER BRACES
* C stores arrays in row-major order
  + Rows stored one after the other
* Use for loops for multi dimensional arrays, 1 loop for rows and another for columns
* Using const in front of an array makes the array a constant
* You can use a non-constant expression to specify array length (Variable length arrays)

Chapter 9

* All functions feature a return type (type of data function will return) next to the name
* Parameters are variables which are passed into the function which must have a type
* The body is the executable part of the function when it is called
  + Within the body, there will always be a return statement which will return a variable of the return type to the place where function was called
* Calling a function is easy, write the function name alongside a pair of brackets which include the arguments one will pass in.
* Form of function definition is
  + *Return type* Function Name (parameters) {
    - {statements}
    - }
* One must always have a return type
* Parameters separated by commas and if no parameters, write void
* Variables declared within function solely remain in that function
* When one calls a function and it has a return type, you will need to store it into a variable
* Function body must appear before the main or where they are called otherwise compiler will implicitly declare the function
* Another way to avoid call before definition issue is to provide a function declaration which gives a glimpse to compiler before the definition
  + Form: return type function name (parameters);
  + Avoid doing this in C99
* Arguments are passed by value
  + When function is called, arguments is evaluated and assigned to corresponding parameter
* Each parameter acts as a separate variable from the argument
* Sometimes arguments are converted which depend if the compiler has seen the function before
  + Value of each arg is implicitly converted
* Arrays are able to act as arguments
* Be careful with array bounds
* Array elements changed in function will also be changed outside the function
* Length of first dimension in multi dimensional array can be omitted
* Variable length arrays can also be parameters
  + Put length first then array in call
  + If omitting name , use [\*] for array calls
  + Putting static num in [] means the array will be at least length of number, only in first dim
  + Compound literal: Array created on the spot\
    - Form is : (dtype[]){values},
* Return expression;
  + Above is form of a must be included statement In every function
* Complex returns are possible
* Exit allows one to exit a program and pass a program code i.e 0 or EXIT\_SUCCESS/FAILURE
* Exit will terminate the program whereas return will exit the function
* Function is recursive if it will call itself within itself
  + Recursive function must have a statement that will stop it from endlessly looping
  + Used from divide and conquer alg

Chapter 10

* Local variable: A variable declared and only used within a function
  + Following properties:
    - Automatic storage duration: This is the portion of program execution during which storage for the variable exists. This will occur automatically for a local variable and will deallocated once you exit the function. Note, previous value may not remain when function calls again.
    - Block Scope: Scope of a variable is portion of program in which variable is references. A local is visible from declaration till end of function. Note in C99, the scope can be smaller as you can declare the variable anywhere in function
* Static when in front of a local variable causes it to have a static storage duration rather than an auto storage duration(this means it will retain the value throughout execution of program)
  + Stays in the same memory location throughout program execution.
* Parameters have the same properties of the local variables but are initialized when a function is called rather than in the function
* Global/External Variables are variables declared outside of a function
  + Static storage duration: Think of the static keyword
  + File scope: Visible from declaration to end of file
* Problems with external variables can occur(Try and avoid using them If possible)
  + If we change it in program maintenance, we have to check every function in file
  + If an incorrect value is assigned or error occurs, difficult to locate where the error is
  + Hard to re-use in other programs
* One can use compound statements to declare variables
  + If (i>j) {
    - Int temp=I;
    - I=j
    - J=temp;
    - }
* This has an automatic storage and has a block scope (Think of it like a local variable)
* These are useful when we need to use temporary variables
* Same Identifier can have different meanings
* C’s scope allows the programmer to determine which meaning is relevant at what time
* When a declaration inside a block names an identifier that’s already visible, the new declarations temporarily hides the old one and the identifier takes on a new meaning. After the end of a block, the identifier regains old meaning
* Programs include :
  + Preprocessing directives
  + Type definitions
  + Declarations of external variables
  + Function Prototypes
  + Function definitions
* The above can be ordered in any way one desires.

Chapter 11

* Recall: Main memory is divided into bytes(8 bits)
* Byte has a unique address which is occupied by a variable
* A pointer variable declared by type \*p and can be declared alongside other variables
* Pointers only point to that specified type
* \* means indirection and is used to access the thing stored in the address. For address we use the &. &p is address of p and \*P is value in p
* You must always initialize a point when using them
* Assign an address to pointer variable e.g p=&i
* Changing the pointer will also change the variable it points to(aliasing)
* Multiple Pointers can point to the same value
* You can use pointers as parameters for arguments
* When calling a function, use the address and when declaring the function use the indirection
* You can use const key word to prevent the value of the pointer being modified
* You can return pointers although do note you cannot return local variable pointers.
* You can also point to array elements e.g. &a[i]

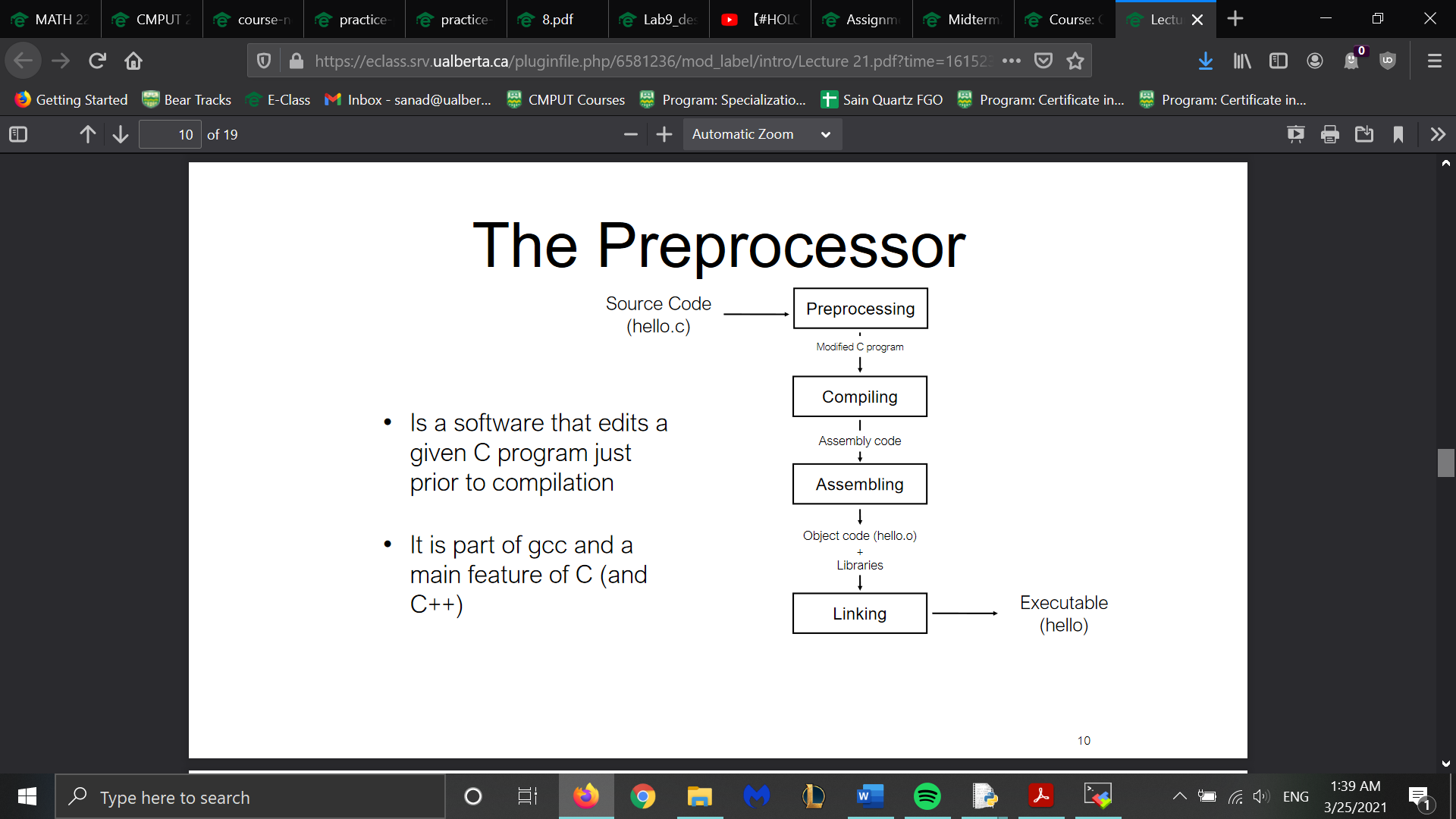
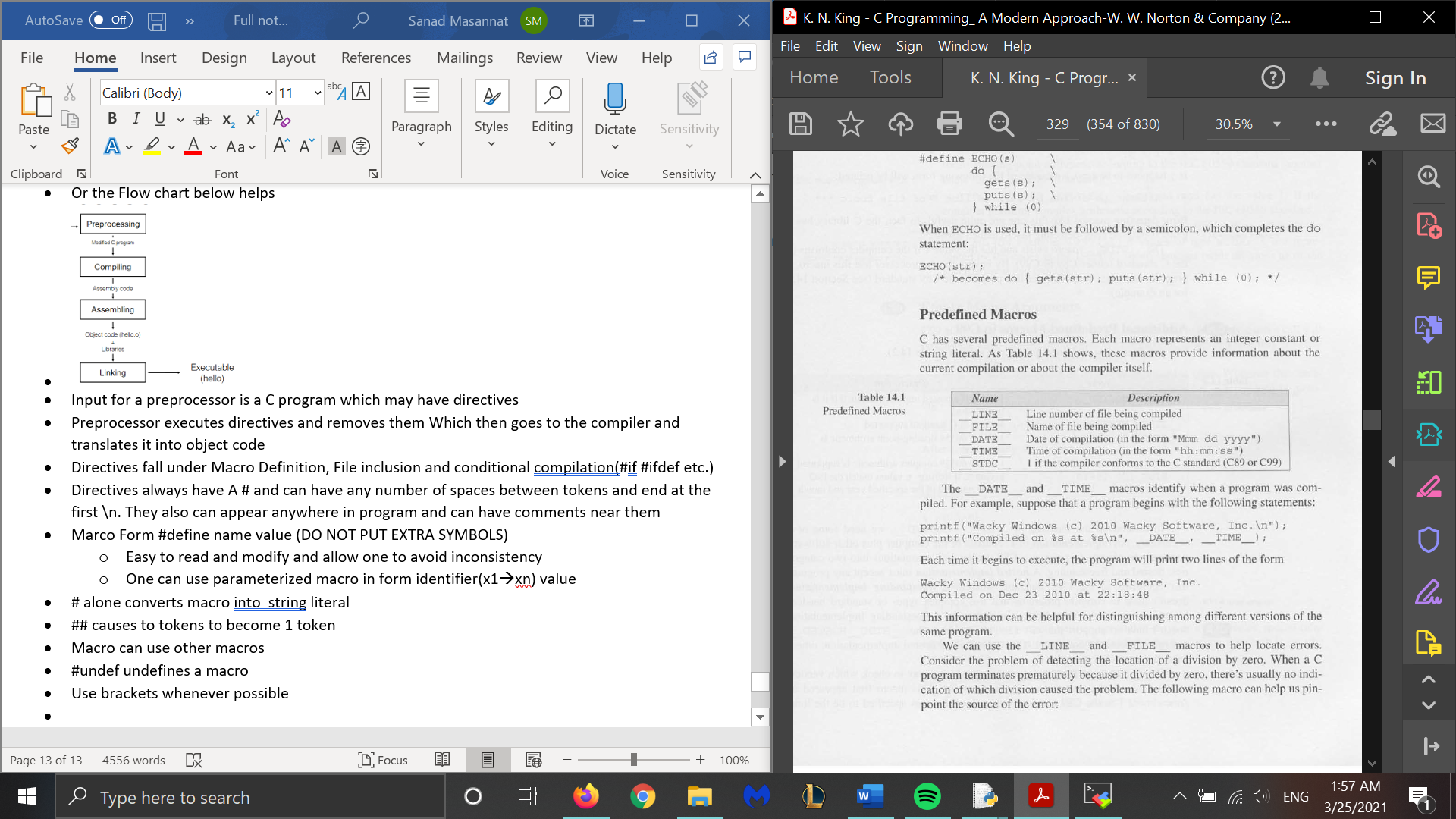
Chapter 12

* 3 pointer arithmetics
  + +/- ints to a pointer
  + – one pointer from another
* If p points to a[0], adding 1 causes it to point to a[1], subtraction works similarly to addition
* Lets say if p points to a[0] and q points to a[9], q-p =9
* You can compare pointers to each other
* You can allow a pointer to point to an element within an array made by a compound literal
* You can use pointers instead of integer values to process an array(remember to use inderction to get the actual value and not the address)
* \*p++ can be used(++ can be replaced by--)
  + \*p++/(\*p)++-value is \*p before increment, then incremented
  + \*++p increments p and the shows the value of \*P
  + ++\*p increments \*p then shows the value of \*p next
* &a[i] is the same as (A+i) when a(array name) is initialized as a pointer
  + You can NOT assign a new value when an array is used as a pointer.
* You can use pointers for multidimensional arrays
  + Form:
  + For (p=&a[0][0]; p<=&a[Rows-1][Cols-1];p++){}
    - This allows us to traverse all of the elements in a 1 dimensional array
    - Can be extended to multiple dimensions
    - Hard readability
* To point to an ith row in 2d array, one can use &a[i][0] or p=a[1]

Chapter 13

* String Literal: Sequence of characters enclosed within double quotes
* When compiler encounters a string literal, sets aside n+1 bytes to store the string plus the \0 character(null character) denoting the end of a string.
  + They can also be empty
* These are stored as an array(char \*)
* We can use subscripts on string literals
* Note: single string literal =/= char as string literal is stored as a pointer whereas a char is stored as an integer(ASCII)
* Recall: String is an array of characters
  + Always declare it to be 1 characters than intended length to store the /0 character
* Chars can initialized upon declaration. Use “”
  + If the string is less than initialized len, the string is the padded with /0
* If no length is given, the compiler will determine a length
* Recall \*p= “hello” is a pointer whereas p[]= “hello” is an array
  + Either works fine for strings but they are not interchangeable
  + In array, the characters can be modified whereas the array points to a string literal
  + P is a name whereas the other p is a variable made to point to other strings
* Declaring a point aint enough to modify a string, you need to make it point to the first element of said string and then it can be used for modification
* To write a string, one can use printf or puts
  + %m.ps prints only p characters in the string and have a field size of m
  + Form for puts is just puts(str)
* Reading strings can be done using scanf or gets
  + Recall one does not need to be & infront of the string argument in scanf as is treated like an array
  + Scanf skips white spaces an stores read in char into string and will stop when a white space or \n / \0 appears. Hence scanf will not read a full line of input.
  + To read a full line, use gets. This acts similar to scanf but it doesn’t stop at a white space nor will it skip it. It will only stop when a new line character is read. It will also store a null character
  + Both can be used but take note that gets will not detect end of a string variable when reading a string in so errors can occur.
  + Getchar() will read in a character at a time and is safer than both gets and scanf
* Recall pointers can be used to deal with accessing arrays
* C has a built in library that allows us to deal with strings
  + #include <string.h>
  + Some things include strcpy(Copies a string to another variable) or strcmp(compares to string and returns 0 if they match).
  + To compare strings, you CANNOT use str1==str2. Use strcmp
  + Strcopy is mainly used to store one string in anther or copy from one string to another. Form is strcpy(str1,str2)
  + StrLen returns the length of the string up to the first /0
  + Strcat concatenates two strings with each other.Form is strcat(str1,str2) (make sure the string is long enough)
* To store strings, one can use a 2-d array to store the strings
* As these can be inefficient, we need a ragged array(2d array which can store different lengths)
  + C cant do this but using an array of pointers helps simulate this

Chapter 14

* Preprocessors controlled by preprocessing directives. These begin with #.
* Recal #define defines a macro and #include makes sure to include a file
* C Program🡪 Preprocessor🡪 Object Code 🡪Compiler 🡪 Object Code
* Or the Flow chart below helps
* 
* Input for a preprocessor is a C program which may have directives
* Preprocessor executes directives and removes them Which then goes to the compiler and translates it into object code
* Directives fall under Macro Definition, File inclusion and conditional compilation(#if #ifdef etc.)
* Directives always have A # and can have any number of spaces between tokens and end at the first \n. They also can appear anywhere in program and can have comments near them
* Marco Form #define name value (DO NOT PUT EXTRA SYMBOLS)
  + Easy to read and modify and allow one to avoid inconsistency
  + One can use parameterized macro in form identifier(x1🡪xn) value
* # alone converts macro into string literal
* ## causes to tokens to become 1 token
* Macro can use other macros
* #undef undefines a macro
* Use brackets whenever possible
* Pre defined macros
* 
* Macro calls can be empty and have Unlimited arguments
* \_\_func\_\_ stores name of current function being executed
* #if and #endif will compile something given that it passes the logic of the #if. #elif #else can also be used here
* #ifdef and #ifndef checks to see if a macro is defined or not defined respectively
* #error message will cause the compiler to print out an error if it is encountered
* #line n “file” allows us to alter line n of the program or file if included
* #pragma allows special behavior to occur in the compiler
  + Expression has from \_Pragma ( string literal)

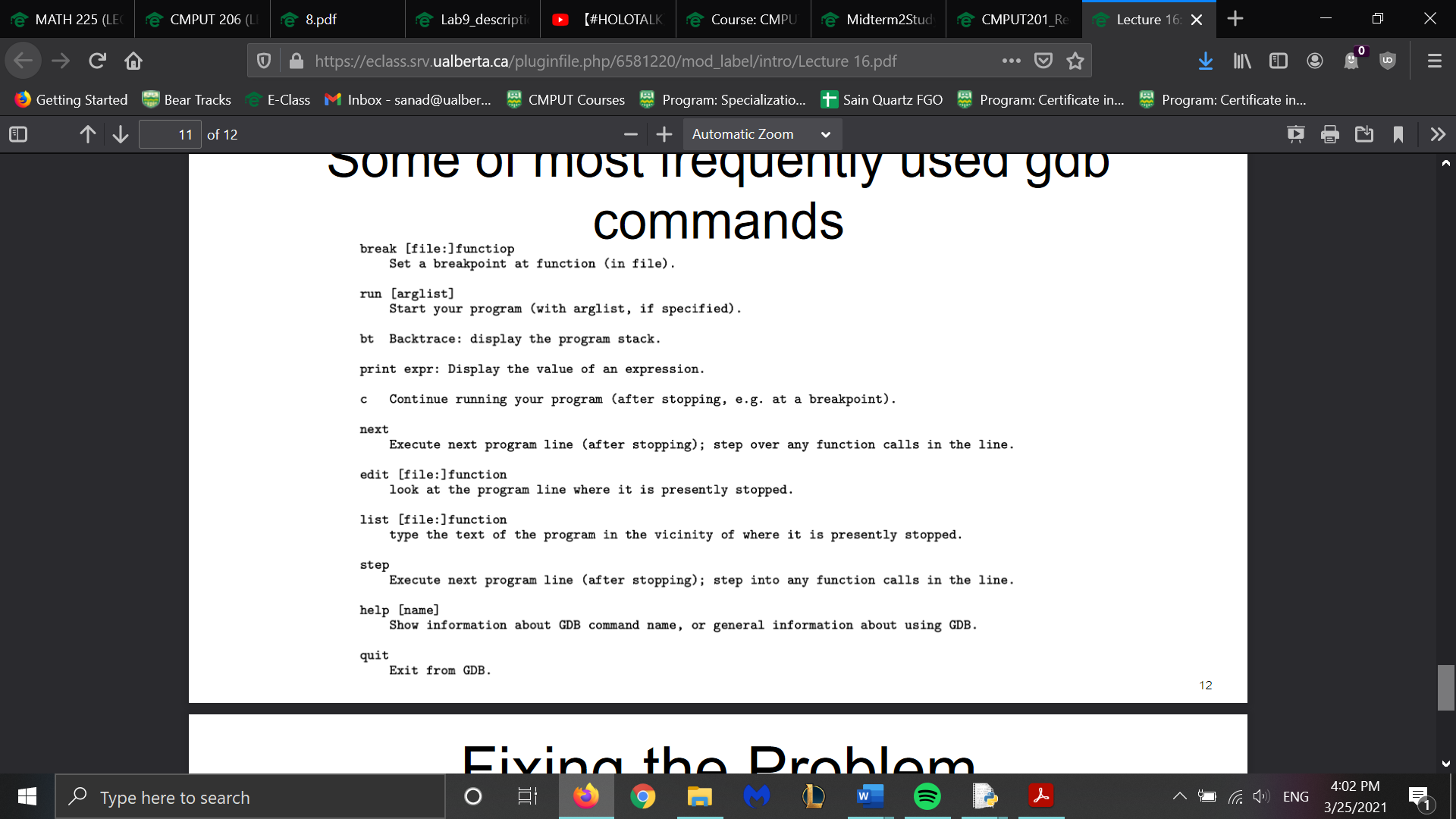
Chapter 15

* Source files: Have an extension of .c
  + Programs can be divided into multiple ones
* Advantages:
  + Clarify program
  + Files can be compiled separately
  + Re-useable
* Header files have the extension .h and are included in source files with #include
  + Form for .h files that you write id #include “file”(This searches current directory)
* You can also include tokens but rarely used
* To share info/macro/type defs in header files to other programs, just include them
* To use a function/variables defined/declared in another file, put the function prototype/definition in a header file and then include it in the specific places
* Header files can also have includes as well
* Protect header files from multiple inclusion by including contents in ifndef
* #error can allow us to check for conditions in header file that shouldn’t be there
* Main function should be in the main function when using multiple files
* You need to compile each source file separately and for each source files are compiled into an .o object file. We can then link object files
* You use makefiles and intermediatory steps and dependencies for this.
  + Make command runs all command and make (…) will only compile part specified in …
* Make sure all names are spelt correctly nd all libraries and files are included
* -D allows us to define a macro outside the program

Chapter 16

* Think of structures being similar to python classes in that the members of a structure do not need to have the same type and have a name to define them
* Form: struct {
  + Type membername
  + } variable name
* These are stored in the order they are declared(separate name space for its members)
* To initialize the variables you can do the following
  + Variable name= { values to be stored in members, separated by commas}
  + Or you can use variable.member = value
* Order doesn’t matter
* Non included members are set to zero
* To access a structure, use variable .member
* They are lvalues
* Once can define a name which represents the type of structure
* Form: struct typename {
  + Type membername
  + };
  + Variables are then defined as struct Typename variablename
* Another way is to use typedef
* Form: typedef struct {
  + Type membername
  + } variable name;
* Passing and returning a structure requires one to make a copy
* You can create a structure without storing it in a variable
* Structures and arrays can be comboed
* Structures can even be used in other structures
* E.g. struct part inventory[100] is an array of part structures that can hold 100 parts
* Initializing array or struct is done the same way as a multidim array
* Unions are like structures but compiler only allocates enough space for largest member
* same from as structure except replace struct with union
  + Only one member is initialized
* Used to save space compared to structs
* Tough to tell when member was last changed
* We embed a union inside a structure which has a field to keep track of what is currently in the union
* Enumerated type is a type whose values are listed by programmer(usually macros)
* Form enum {values} variables;// enum type (values); enum type variables
* C treats enums are ints and enum constants can have same value
* These can be used as tag fields for unions

GDB:



Chapter 17

* Most data structs are fixed in size(no. of elements fixed in size/assigned at run time)
* C supports dynamic storage allocation: the ability to allocate storage during run time.
  + Used for strings arrays and structs
* You need to include stdlib.h
* There are 3 memory allocation function:
  + Malloc: Allocates a block of memory but doesn’t initialize it
  + Calloc: Allocates a block of emory
  + Realloc: Resize a previously allocated block of memory
* Malloc is used more often than others and is more efficient than calloc as it doesn’t clear the memory
* Returns a value of type \* (a generic pointer)
* If the function can’t locate a large enough block of memory to allocate, it will return null pointer
  + Represented by NULL
* P==NULL is the same as if(!p) and p!=NULL is the same as (p)
* Malloc has following prototype:
  + Void \*malloc(size\_t size);
* This allocates a block of size bytes and return a pointer to it and size has a type of size\_t
* To allocate space for a string of n chars, we write p= malloc(n+1)
  + P is a char \* variable and the pointer is converted to a char \*
  + One can cast the return type
* INCLUDE SPACE FOR NULL CHAR
* You can use strcpy to initialize the made array
* If Malloc returns a NULL, an error usually occurred
* Make sure to few the space when using malloc as one can run out of error quickly
* To malloc space for an array, same form as string above
  + You will need to use n\*sizeof(data\_type) instead to initialize an array of size n of type data\_type
* Calloc prototype:
  + Void \*calloc(size\_t nmemb, size\_t size)
  + This allocates space for an array with nmemb elements
  + Sets all bits to zero. E.g. a=calloc(n,sizeof(int))
* Realloc has form
  + Void \*realloc(void \*ptr, size\_t size);
  + Ptr must come from a previous call from memory allocation functions
* Various Rules:
  + When expanding a memory block, realloc doesn’t initialize bytes added to block
  + If realloc can’t enlarge memory block as requested it returns a null pointer; old memory block data unchanged
  + If realloc is called with a null pointer, it acts like a malloc
  + If reallocs second parameter is 0, it frees memory block
* Realloc should should expand block but not move it
* Be sure to update all pointers in memory block, since it’s possible that realloc has moved the block elsewhere
* memory allocations obtain memory block from a storage pool known as the heap
* Calling them too often can exhaust heap leading to return a null pointer
* Easy to lose track of memory blocks
* Free allows to release a block of memory that p points to
  + Form is free(ptr)
  + Ptr must be from a memory alloc function
* Free may deallocate pointer but doesn’t change ptr itseld so it no longer points to a valid memory block
* DO NOT ACCESS/MODIFY DEALLOCATED MEMORY
* Linked lists are a chain of nodes that have pointer that point to the next free node
* You need to allocate memory for that node and then add data to said node and then add it to the linked list
* -> allows to access the member of a struct using a pointer
* Recall that if an array had elements of type char\*, a pointer to one of the elements had type char\*\*
* One can use pointers to pointers for linked lists
* One can point to functions

Chapter 18

* Recall that int i tells compilers the meaning of variable i.
* declarations have form
  + specifiers declarators;
* Specifiers describe the properties of the variables or functions being declared and declerators give the names
* Specifiers fall under 3 categories:
  + Storage classes:Auto,static,extern and register
  + Type qualifiers:const,volatile and restrict
  + Type specifiers:Char,void int etc.
  + Specifiers: \*,[] and ()
* Storage classes can be specified for variables and functions
* Properties of variables
  + Storage duration: Auto storage duration. Some have a static duration
  + Scope: Portion of program text in which the variable is referenced
    - Block scope or file scope
  + Linkage: Linkage of a varible determes the extent to which can be shared by different parts of a program.
    - External linkage can be shared by several files in a program
    - Internal limited to a single file
    - No linkage means a single functiom
* Variables declared in a block have auto storage,block scope and no linkage whereas variables outside a block have static storage duration, file scope and external linkage
* Auto storage class is only legal for variables in a block and has no linkage and has block scope
* Static can be used with all variables regardless of where they are declared but it has a different effect on avariable
* Different effect on where variable is
  + Outside of block, variable has internal linkage
  + Inside causes it to have static storage duration
* These retain values indefinitely
* These are initialized once and prior to program
* When using recursion , auto value are changed whereas statics don’t change
* One can return a pointer to a static variable but not an auto variable
* Extern allows several files to share the same variable
* Many declarations but one definition
* Extern is always static definition
* Register tells compiler to store it in a register and not in main memory
* Data can be accessed faster and updated
* These do not have addresses
* These are used for variables accessed and updated frequently
* Only extern and static can be used on functions
* Extern functions are the same as auto variables
* Static functions are useful in that:
  + Faster maintenance
  + Reduced space
* Const is the same as read-only
* Similar to macros but consts have the same rules as variables
* Recall \*=pointer,[]=array,()=functions
* Many combinations of the 3 symbols
* Read declarators from the inside out
* Favor brackets over \*
* = is an initializer
* If types don’t match, C converts it to correct type if possible

Chapter 19

* Collection of services available to parts of the program are called modules
* They have an interface which describes the available services(functions in out case)
* Details of module stored in implementation
* Dividing program has advantages:
  + Abstraction
  + Reusability
  + Maintainability-Most important
* Good modules should have high cohesion and low coupling
  + High cohesion means modules should be closely related
  + Low coupling means that they should be as independent as possible
* Types of Modules:
  + A data pool : collection of related variables and constants. Usually a header files
  + Library: Collection of related functions
  + Abstract object: Collection of functions that operate on a hidden data struct.
  + ADT(Abstract data type): Type whose representation is given
* Info hiding: Deliberately concealing info from clients of a module is known as info hiding
  + 2 advantages:
    - Security:Unable to corrupt program
    - Flexibility: Making changes to internal workings aren’t to difficult
* ADT make it difficult to have more than one instance so to counter this, we create a type
  + Done using structs,unions etc.
* C is has limited support for encapsulating data
  + Incomplete types helps
  + Form is struct t; where t is a variable
* Using multiple ADTs can lead to naming issues

Chapter 20

* Bitwise ops operate on int data at bit level
* There are 6 bitwise ops
* Bitwise shift transforms binary reperesentation of an int by shifting bits left or right
  + Form: >> / <<
  + Shift of n leads to multiply/divide by 2^n depending on the direction you shift
  + Some bits are shifted off and zeros are padded when some bits are shifted off
* <<= allows us to modify original bit and store in the same place
* Remaining ops: Highest to lowest precedence
  + ~: Bitwise complement
  + & bitwise and
  + ^ exclusive or
  + | inclusive or
* Difference between ^ | is that ^ sets bits to zero when both bits are 1 whereas | sets it to 1
* DO NOT CONFUSE &/| FOR &&/||
* We deal with unsigned ints when using bit ops
* Masks are usually used
* We use the above for various reasons:
  + Setting a bit:
  + Clearing a bit
  + Test bits
* Working on consecutive bits(bit-field) is more complicated
  + Bit field ops are modifying a bit-field and retrieving a bit field