Need to know

- Pre-Test Loops – While
- Post-Test Loops – Do-While
- Fixed-Iteration Loops – For
- Break And Continue

- Functions In C
- Function Return Values
- Parameter Passing
  - Pass By Value
  - Pass By Reference
- Variable Scope

- Structure Charts

Iteration (loops): removes bottleneck and saves programmers time. Do I keep repeating or do I stop repeating. Still requires a true false ie if true do statements There are three types:

- **While loop:** a while loop starts with a pretest condition (test before and then executes. Retest condition). Then does iteration statements (which are statements inside while loop). Like iterative if statement. Think of while loop: as if the condition is met true/false do the inside if not skip. (false/outside). Note: Does not need increment updator like update++

Loop Property	**Pre-test (while)**	**Post-test (do-while)**	**Fixed Iteration (for)**
Condition tested	Before loop begins	After each iteration	Before loop begins
Min. No. Iterations	Zero	One	Zero
Special Features	Nil	Nil	Initialisation, Automatic Increment

1. Do you know **how many** times the **loop should repeat?**
   - Or, more accurately, is the number of times it should repeat known immediately before the loop is to begin?
   - If so, then a ***for loop*** should be chosen.
   - This is common where there is a fixed or known amount of data that needs to be processed.
2. Does the loop need to **execute** *at least* ***once***?

- If so, then a ***do-while loop*** is required.
- This is common where there is data or some input from the user etc. that must be received and processed before the decision about whether the loop should repeat can be made.
- An example for this might be a menu program.

//ADD BREAK AND CONTINUE MAYBE?

Break: Used where there is more than one condition ie while then a loop statement

Modular program: programming is with sections
Non-modular programs: are just code in main() function

Top down design:

- One way of making the complexity of a problem more manageable
- TDD means that we approach solving a problem (and writing the code) by starting from the top (the high-level details) and working our way down to the bottom (the low-level details).
- Developing a high-level algorithm containing a number of steps that will solve a problem (general steps not specific) → The taking these steps and decomposing it (work out the steps in more detailed)
- Don't have more than 6 functions just add extra level?

- So, developing a high level algorithm containing a number of steps, which will solve a problem. These should be general steps and will not usually involve the specific calculation needed.

Modularity and top down design:

- Module is a collection of code
- Each refined steps is separated out into distinct module of code. Each module performs each of the refined sub-steps from high level algorithm

Four principles of modular programing (how to divide program):

- An appropriate level of **abstraction** between modules
- Ability to **re-use** the module is maximised
- **High cohesion** within each module
- **Low coupling** between modules

An appropriate level of abstraction:

- Involves hiding details that aren't necessary for us to know while understanding the general terms what is going on

- We just need to know as little information about how it works
- If we write modules: we need to write program so that user doesn't need to know how it works.
- No modules should need to know exactly how other module works. They just need to primarily know the outcome

Code Reuse: Don't re-event the wheel!

- Making solution general enough that it more broadly useful then simply solving the problem you have immediately.
- Define: When writing own module writing in a way that maximises reusability of module. Use module the to solve general case. Making code reuse as general as possible
- Re-use also relates to code libraries which are collection of modules made available to all programmers working with particular language/plateform

High cohesion:
- Each module should be about solving one part of the problems. IE for the sum module don't add the get data module in it. Calculating and printing should not be in the same module (low cohesion)
- Low cohesion: it effects how we can use code-reuse. So if we use module for just a single task but the module has two task then issues arise then we have two extra steps. EG Sum + get data we can't just get use sum module we need both
- Therefore, all the code should be associated with the same logically task

Low-Coupling:
- Minimise interaction between modules: means that each module should be limited ability to interact with data belonging from other data unless necessary. Called scope:
- Reason for low coupling is that if a module can alter the data belonging from another module then this could cause the module to fail and the bug therefore will be hard to find.
- Closely related to abstracting since hiding data associated with module helps abstract the behaviour of the module

//ADD STRUCTURE CHART?

Structure charts:
- Structure charts are an important tool for showing how the different modules in a program are related in a diagram
- Structure charts show each of the modules in the program and which modules call which others
- They are arranged in a hierarchy and always begin with the *main* module at the top.
- The main module calls the first layer of modules, each of which is perform the high-level

steps of the algorithm.
- May then call other modules in larger programs.

Rules for return values:

1. Declare the function with the appropriate return type.
   - That is, if the function returns a value then you must indicate what its data type is.
   - For example:

     ```
     int Sum(int x, int y)

     float Average(int x, int y)
     ```

   - Here the `Sum()` function returns an integer and the `Average()` function returns a float.

2. Return a value of the appropriate type in the function.
   - This means you need a return statement somewhere inside that function which has the values/variable/expression that you want to return inside brackets, for example:

     ```
     return(sum/2.0);
     ```

   - This is the return statement from the Average() function and noticed that the data type of the value returned is a float.
   - Note usually the return statement is the last statement in the function.

3. "Catch" the return value back in the calling function.
   - For the Sum() function in the previous program this involved passing the return value straight to a call to printf():

```
while(condition)
{
  /*  If this condition is met then execute
in here if not skip. Condition is normally
true of false verifying

  */
}

Pseudo code:

While (condition)Do
  Statements
EndWhile

Set Yolo as 10
While (Yolo <= 20) Do
  Print Yolo
  Yolo++
EndWhile
```

Skip down here if condition is not met or no longer met.

- **For loop: Characteristics 4 pretest loop**

  First of all, inner loops reset their counter once you go out of the for loop.
  So:

```
for(digit = 1; digit <= 9; digit++)
            {
```

```
for(test = 1; test <= 9; test++)
{
   printf("%d", test);
   printf("\n");
}
```

*So after each iteration the test counter wil be reset back to 1 while the digit will increase. Ie digit 1 test 1   digit 2, test = 1 still etc

- Remember all conditions are testing true or false. If true it will execute whatever's in it ie-

  ```
  Row = 2 digit = 1
  While(digit <= row)
          Output "yolo"
  ```

  *So the condition is asking: Is 1 Less than or Equal to 2? The answer is true so it will output "yolo"

- Secondly think of for loop as while loop but a cleaner version ie

```
for(row = 1; row <= 9; row++)
    {
            for(digit = 1; digit <= row; digit++)
            {
                    printf("%d ", digit)
```

*Steps in solving/thinking like while loop:

```
 row = 1; digit = 1
While(row <= 9) //So again above is a condition to test. Is 1 (row) less than or equal to 9?? True so  execute below
    While(digit <= row)
            Output digit
            Digit++
Row++
```

Same thing

- Flowchart: Initalisation→Test condition (true/false→execute iteration statements (true) → increment → test condition

```
 for(initialisation; condition; increment)
{

}
```

Can think of it like this→

Initialisation
While(condition true/false)

Iteration statements
Increment

Pseudo code:

```
For var = initial To final Do
    Iteration statement
EndFor
```

Do-while loop: post test loop. Execute iterations one
 Int x = 0;

```
do
{
   /* Loop code in here */
} while(condition); //note it is down there
```

Psuedo code:

```
REPEAT
   Iteration statements
Until ()
```

- Flow chart: Initialisation → Iteration statements → Test condition (true/false) → (true) iteration

Break:

- Creating function. **Define** the function (this is te code contained within module. The behaviour is dictated by code) + **Calling**: Making function execute the code it contains sort of like → Declared/defined function and called function
    - Main function job is to calls the other functions which perform the individual sub steps that make up algorithm.
    - ALWAYS START WITH MAIN FUNCTION. EXECUTES CALLING FUNCTIONS IN ORDER.

- Function declaration: FunctionName()
- Function calling: So in main function → write the function declaration FunctionName()

- Return takes us out of the function to after the calling of the function in main. So next line in main after called function

- Return a value:
    - Calculate something in function but variable scope means function can't get that answer.
    - *Bring data out of the function (Single result calculated) → to whatever function called it*
    - Applies to data/calculation.
        - Get data In
        - Get data in and then out
    - Think of return as formula return(sum/2)
    - Return only one value per call

- **Parameter Variables**: Are declaration of variables inside bracket. Data type must be specified every variable. Used for variables in a function that needs to cross of exchange to other variables
    - **Int Sum(int x)**
- **Local variables**: Are declarations of variables inside a function. No crossing
    - **Int x**
- **Making all variables parameter variables means they need to be passed around =** breaches principle of low coupling which is reducing interactions between other functions
-

Both get data out of function – disallows variable scope

- Get a data in so it can be processed: **Passed by value to get copy of data in the function. Data can only move into function not in. Needed to defeat low coupling**
    - Int sum(int x, int y)
    - Total = sum(a,b)
        - So treat this like printf("%d", a)

        - A value gets copied into x and b value gets copied into y(not same data ie int x is not variable a)
        - Then the return of the function replaces the call. So the sum function is gone
- **Pass by reference**: Two way connection between variables and parameter. *Parameter returns variable that as passed.* Any changes to parameter inside called function effect the original variable back in the calling function. So int sum(int x, int y) → means that *the the variables inside parameter impacts the local variables. Ie sum(Int , Int y)*
- **Return values:** Returns a single value and but how do we get data in to calculate

---

    - **If in sum function gives x = 5 the calling a = 5**
    - **Must have a variable ie calling → sum(a,b, sum)**
    - **Two variables**

- Variable scope: Variable is only visible to functions it is declared it.
    - Sum(int x) not equal to int x in main

- **<u>Think: pass by value returns a single value. Data in + single data out: return(X+y): calculation</u>**
- **<u>Think: pass by reference more than one value. Also used to get multiple data in + out: return; → &a &b : get data</u>**

- **Return values** → Single piece of data is obtained ie—print stament. Return? Properties
    - Pass by value → Calculation return(x+y)
    - Function return type must be indicate what data type
    - Return a value indicated by the data type
    - **<u>Catch the return value back into calling function -> must have a calling function</u>**
        - **<u>Remember: assign result to variable always or use it once</u>**
- Notice: Declare function before the calls

- Pass by reference→ Get data return variable x and variable y

- THINK ABOUT THE FLOW. WHERE DOES DATA FLOW. DOES READ NUM BRING BACK NUMBER TO MAIN? YES? DOES SUM FLOW OUT OR FLOW IN? BOTH (NUMBER READ IN A+b FROM CALL + OUT)

- Remember only put in bracket getnumb(int x) if IT IS PASS BY VALUE. Put void if returning a print function or a value.