FUNDAMENTALS OF PROGRAMMING PRACTICE

Table of Contents

I. INTRODUCTION

This document is a reference to the Syllabus of this course, only focusing on practice of the bellow four main items:

1. Algorithms
2. Flowchart
3. Programming in C
4. Programming in C++

II. ALGORITHMS

An ***Algorithm*** specifies a series of steps that performs a particular computation or task. It is a step by step procedure of solving a problem. It were originally born as part of mathematics from the Arabic writer Muḥammad ibn Mūsā al-Khwārizmī, and now strongly associated with computer science. A good algorithm mean an efficient solution to be developed.

II.1.1. Qualities of a good algorithm

1. Input and output should be well defined with precision.

2. Each steps in an algorithm should be clear and unambiguous.
3. Algorithm should be most effective among many different ways to solve a problem.
4. An algorithm should be human understandable, it shouldn't have computer code. Instead, the algorithm should be written in such a way that, it can be used in similar programming languages.
5. A good algorithm should be programming language independent, ie can be implemented by any software developer using any programming language

II.1.2 Real life Example

Producing a bread at the bakery

To bake a **bread** we have some must be followed steps :

0. Start

1. Preheat the oven;

2. Mix flour, sugar, and other ingredients ;

3. Pour into a baking pan;

4. Heat for some times …

 ……

n-1. Get the bread out of the baking pan

n. Stop

II.1.3. Programming Examples of Algorithm:

II.1.3.1. Calculate the sum of two numbers (A and B) and print the sum to the console.

1. Start

2. Declare number type variables: number one A, number two B and the sum **Sum.**

2. Input A

3. Input B

4. Compute A+B into Sum (Sum ← A+B)

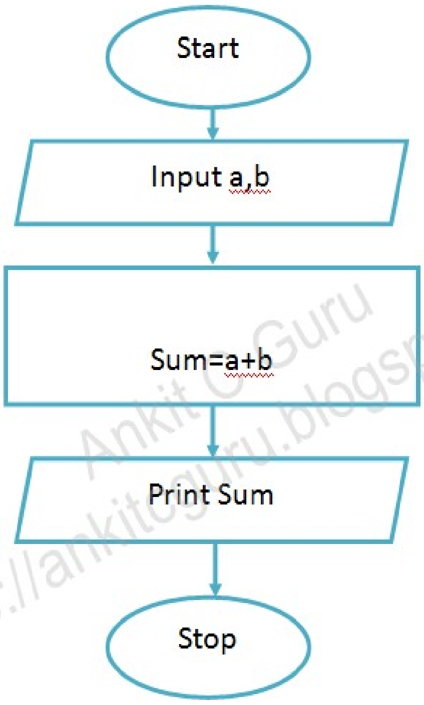
5. Output Sum

6. Stop

Pseudo code:

```
var A,B, Sum;
get A; get B;
Sum=A+B;
print Sum;
```

Flow chart



II.1.3.2. **Find the maximum number in a collection of numbers.**

Problem: Given a collection of n positive numbers, return the largest number in the collection.

Inputs: A collect C of positive numbers. This collection must contain at least one number. (Asking for the largest number in a collection of no numbers is not a meaningful question.)

Outputs: A number x, which will be the largest number in the collection,let call this number **max**.

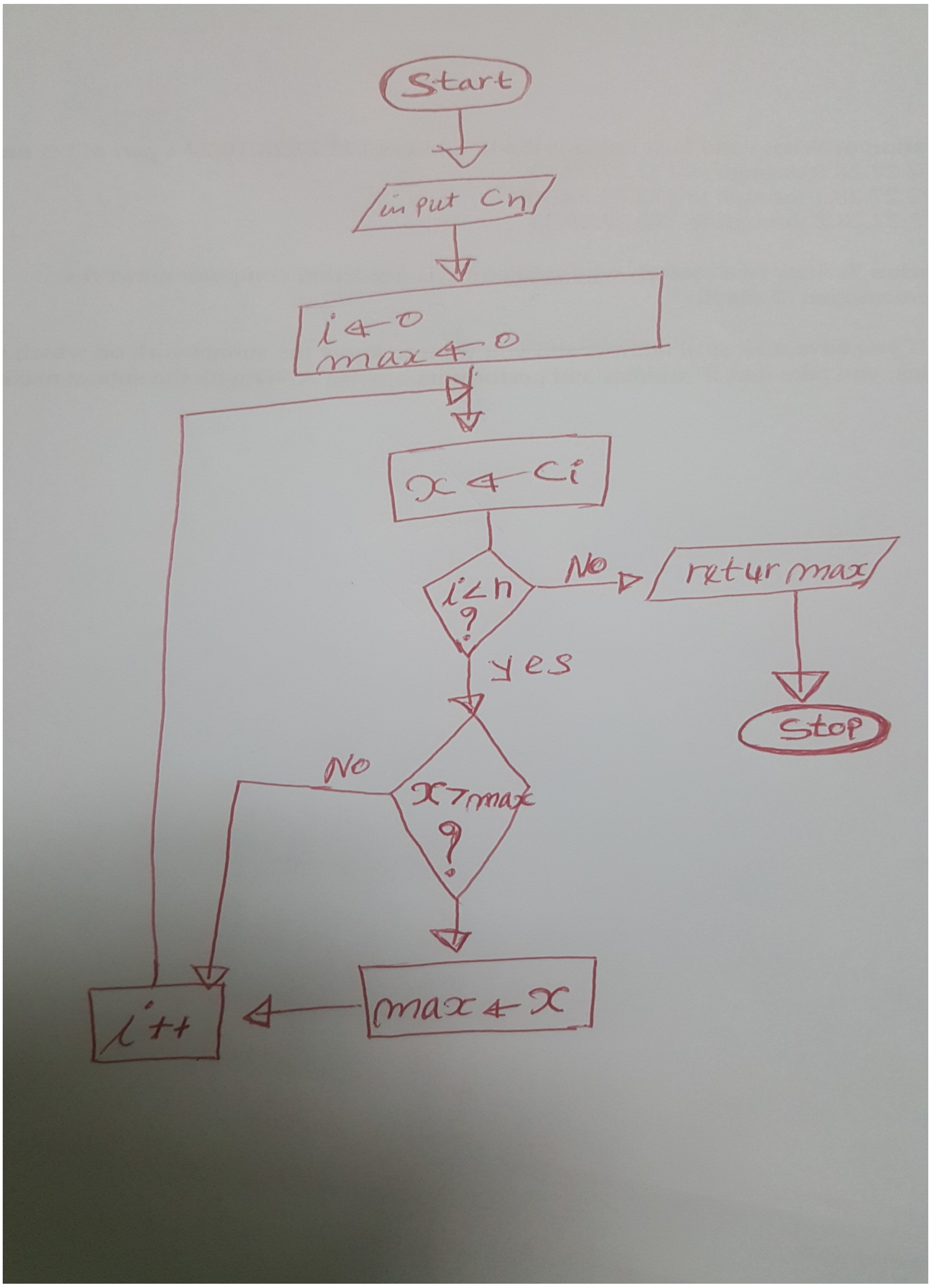
Algorithm:

1. Start
2. Declare variable input of collection c, and the maximum max
3. Receive the collection of Numbers C as Input
4. Set max to 0.
5. Loop through the collection c : For each number x in the collection C, compare it to max.
6. Check condition inside a loop : If x is larger, set max to x.
7. Now max is set to the largest number in the list at the end of the loop.
8. Output the max as the maximum number
9. Stop

Pseudo code:

```
def  c;
def max = 0;
get c;
    for each x in C;
        if x > max:
            max ←  x
end of for loop;
    return max;
```

Flowchart


Start
input Cn
i ← 0
max ← 0
x ← Ci
i<n ?
No
return max
Stop
yes
x>max ?
No
max ← x
i++

II.1.3.3. Calculate the sum and average of all even numbers between 1 and n

Inputs: Numbers from 1 to n

Outputs: The sum and the average

```
Step1: Start
Step 2 :  Declare and initialize variables   sum = 0, i = 1, average, count = 0 and n
Step 3 :  Loop through n elements solving i^th element in order
Step 4:  if  i^th element/ 2 == 0 then go to step 5, else go to on step 6
Step 5: sum = sum + i^thelement, count = count + 1
Step 6:  i^thelement = i^thelement+ 1
Step 7 : if i^thelement <= n then go to on step 4, else go to on step 8
Step 8 : Display "sum"
Step 9: average = sum/ count
Step 10: Display "average"
Step 11: Stop
```

Pseudo code :

II.1.3.**4. Write an algorithm to find the largest among three different numbers.**

```
Step 1: Start
Step 2: Declare variables a,b and c.
Step 3: INPUT variables a,b and c in order.
Step 4: Process a, b and c :
        If a>b
          If a>c
            OUTPUT a is the largest number.
          Else
            OUTPUT c is the largest number.
        Else
           If b>c
              OUTPUT **b** is the largest number.
           Else
              OUTPUT **c** is the greatest number.
Step 5: Stop
```

Pseudo code :

```
def a,b,c;
input a;
input b;
input c;

If a>b
           If a>c
              print" a is the largest number";
           Else
              print" c is the largest number";
        Else
           If b>c
              print" b is the largest number";
           Else
              print" c is the greatest number";
```

II.1.3.5 **Write an algorithm to check whether a number entered by user is prime or not.**

```
Step 1: Start
Step 2: Declare variables n,i,flag.
Step 3: Initialize variables
        flag←1
        i←2
Step 4: Read n from user.
Step 5: Repeat the steps until i<(n/2)
     5.1 If remainder of n÷i equals 0
            flag←0
            Go to step 6
```

```
    5.2 i←i+1
Step 6: If flag=0
          Display n is not prime
        else
          Display n is prime
Step 7: Stop
```

II.1.3.6 **Write an algorithm to find the Fibonacci series till term≤1000.**

```
Step 1: Start
Step 2: Declare variables first_term,second_term and temp.
Step 3: Initialize variables first_term←0 second_term←1
Step 4: Display first_term and second_term
Step 5: Repeat the steps until second_term≤1000
    5.1: temp←second_term
    5.2: second_term←second_term+first term
    5.3: first_term←temp
    5.4: Display second_term
Step 6: Stop
```

Algorithm is not the computer code. Algorithm are just the instructions which gives clear idea to you idea to write the computer code.

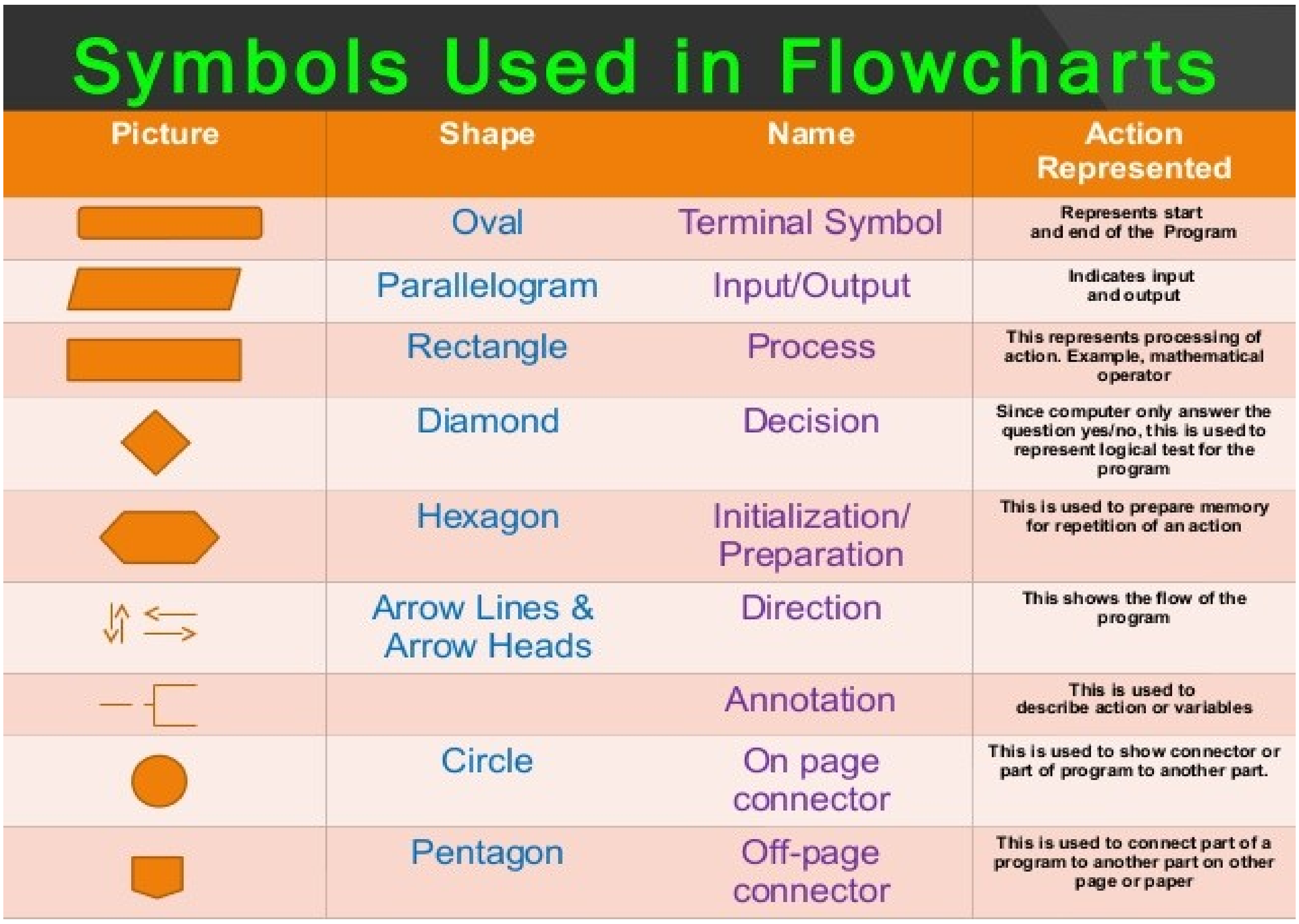
II. 2. FLOW CHAT

II.2.1 Definition:

A Program flowchart is a diagram that uses a set of standard graphic symbols to represent the sequence of coded instructions fed into a computer, enabling it to perform specified logical and arithmetical operations. ... There are seven basic symbols in **program flowchart**,

Symbols :

1. Start or stop:
2. Input:
3. Process:
4. Condition checking /Decision making:
5. Flow of controls / Direction:
6. On page connector:
7. Off page connector:



Symbols Used in Flowcharts

Picture	Shape	Name	Action Represented
▭	Oval	Terminal Symbol	Represents start and end of the Program
▱	Parallelogram	Input/Output	Indicates input and output
▭	Rectangle	Process	This represents processing of action. Example, mathematical operator
◆	Diamond	Decision	Since computer only answer the question yes/no, this is used to represent logical test for the program
⬣	Hexagon	Initialization/ Preparation	This is used to prepare memory for repetition of an action
⇅ ⇆	Arrow Lines & Arrow Heads	Direction	This shows the flow of the program
—[		Annotation	This is used to describe action or variables
●	Circle	On page connector	This is used to show connector or part of program to another part.
⬟	Pentagon	Off-page connector	This is used to connect part of a program to another part on other page or paper

III. C PROGRAMMING

III.1. TOOLS TO BE USED IN THIS PRACTICE COURSE

1. Turbo.C and DEV ++ for windows user
2. GCC open source for Linux users

III.2. PROGRAMMING CULTURE AND ADVICE

Learning to program means learning how to solve problems using code. Conceptually it is not very difficult to write a program that solves a problem that you can solve yourself right ? Yes ! Of course because we already know the

solution to the problem,HuHuHuH! But wait ; how can a computer understand our solution so that it could be used in the future?.
The skills you need to acquire, is nothing other than thinking very precisely about how you solve the problem and breaking it down into steps that are so simple that a computer can execute them. I advice you to first solve a few instances of a problem by hand and think about what you did to find the solution.
For example if the task is sorting lists, then sort some short lists yourself. A reasonable method would be to find the smallest element, write it down and cross it out of the original list and repeat this process until you have sorted the whole list.
Processes Bellow can be used for the list sorting problem :

1. How to find the smallest element.
2. how to write it down
3. how to cross it out, and wrap this in a loop. Then continue this task breakdown process until you're confident you know how to write the necessary program.

Then you have to teach the computer : My God ! Really ? Does Software engineers teach the computer how to solve problems ? Yes they Do !!

To make good progress in your programming task, you need to test your work as early and as thoroughly as possible. Everybody makes mistakes while programming and finding mistakes in programs consumes a very large part of a programmer's work-day. Finding a problem in a small and easy piece of code is much simpler than trying to spot it in a large program. This is why you should try to test each sub task you identified during your task-breakdown by itself. Only after you're confident that each part works as you expect you can attempt to plug them together. Make sure you test the complete program as well, errors can creep in in the way the different parts interact. You should try to automate your tests. The easier it is to test your program, the freer you are in experimenting with changes.
The last important point is *how* you express your thoughts as code. In the same way that you can express the same argument in different ways in a normal

English essay, you can express the same problem-solving method in different ways in your codes.
Understandable codes is key to success.
Remember that you don’t write the program for the computer, you write it for other humans (**maybe a future you!**). Choose variable names that explain things, add comments where these names don’t suffice. Never comment on *what* the code is doing, only write comments that explain *why was the code written.*
Better naming and a better task breakdown make the comments obsolete.
Revise your code just as you would revise an essay. Sketch, write, delete, reformulate, ask others what they think. Ahahahah!! Nowadays **GOOGLE** is our best friend, **GOOGLE IT PLEASE** if you can’t do it on your own!.
Repeat until only the crispest possible expression of your idea remains. Revisit code you’ve written a while ago to see whether you can improve it with things you’ve learned since.
Look at the famous Hello world program.

```
#include <stdio.h>
/*
My first program in c
I wish I could like this languages
*/
int main(){
// Where can I write my Hello world text ?
  printf("Hello, World !\n");
// wawoo!!,  I have written to the console,OK!
   return 0;
}
```

Discuss The Programs
- include <stdio.h>
This " include <stdio.h> " is a preprocessor command.
This command tells compiler to include the contents of stdio.h (standard input and output) file in the program.The stdio.h file contains functions such as scanf() and print() to take input and display output respectively.

- int main(void)

The execution of a C program starts from the main() function. The main function is defined as an entry point of any c or c++ program execution.

-printf()

The printf() is a library function to send formatted output to the screen.
In this program, the printf() displays Hello, World! text on the screen.

- return 0

The **return 0;** statement is the "Exit status" of the program. In simple terms, program ends with this statement.

-{}

the open and closed braces are to mark the start and the end of the main function

- **// and /* */**: Comments

Anything following the "**//**" symbols is a comment and it can not be executed by the compiler, it is part of our program but just for humans not for the computer.
Please use **//** in you programs if you are talking of few thing
or other comments in case you want to be talkative:

/*

Any thing put here is a comment and can not be executed, the compiler will not care of how big was the text in comments!!!.

***/**

Variables

In programming, a variable is a container (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier). Variable names are just the symbolic representation of a memory location. For example:

```
int playerScore = 95;
```

Here, *playerScore* is a variable of int type. Here, the variable is assigned an integer value 95.

The value of a variable can be changed, hence the name variable.

```
char ch = 'a';// some codech = 'l';
```

Literals

A literal is a value (or an identifier) whose value cannot be altered in a program. For example: *1, 2.5, 'c' etc.*

Here, *1,2.5* and *'c'* are literals. Why? Because You cannot assign different values to these terms.

---

1. Integers

An integer is a numeric literal(associated with numbers) without any fractional or exponential part. There are three types of integer literals in C programming:

- decimal (base 10)
- octal (base 8)
- hexadecimal (base 16)

For example:

```
Decimal: 0, -9, 22 etc
Octal: 021, 077, 033 etc
Hexadecimal: 0x7f, 0x2a, 0x521 etc
```

n C programming, octal starts with a *0*, and hexadecimal starts with a *0x*.

2. Floating-point Literals

A floating-point literal is a numeric literal that has either a fractional form or an exponent form. For example:

```
-2.0
0.0000234
-0.22E-5
```

Note: E-5 = 10^{-5}

3. Characters

A character literal is created by enclosing a single character inside single quotation marks. For example: *'a'*, *'m'*, *'F'*, *'2'*, *'}'* etc;

5. String Literals

A string literal is a sequence of characters enclosed in double-quote marks.

For example:

```
"good"                  //string constant
""                 //null string constant
"      "               //string constant of six white space
"x"                    //string constant having a single character.
"Earth is round\n"         //prints string with a newline
```

Rules for naming a variable

1. A variable name can have only letters (both uppercase and lowercase letters), digits and underscore.
2. The first letter of a variable should be either a letter or an underscore.
3. There is no rule on how long a variable name (identifier) can be. However, you may run into problems in some compilers if the variable name is longer than 31 characters.

Note: You should always try to give meaningful names to variables. For example: firstName is a better variable name than fn or fname.

C is a strongly typed language. This means that the variable type cannot be changed once it is declared. For example:

```
int number = 5;   // Good

 variable number = 5.5;       // error

double number;       // error
```

Here, the type of *number* variable is int. You cannot assign a floating-point (decimal) value 5.5 to this variable. Also, you cannot redefine the data type of the variable to double. By the way, to store the decimal values in C, you need to declare its type to either double or float.

Variable's scope

A scope is a region of the program, and the scope of variables refers to the area of the program where the variables can be accessed after their declarations.

In C, every variable is defined in a scope. You can define scope as the section or region of a program where a variable has its existence; moreover, that variable cannot be used or accessed beyond that region, this make sens, yes. In C programming, a variable declared within a function is different from a variable declared outside of a function. The variable can be declared in three places/regions. These are:

Variable Position	Type / named as
Inside a function or a block of code.	local variables
Out of all functions.	Global variables
In the function	Formal parameters

parameters.

III.3. LOCAL VARIABLES

Variables that are declared within the function block and can be used only within the function are called local variables, they are local to that function or block.

III.3.1 Local Scope or Block Scope

A local scope or block is collective program statements put in and declared within a function or block (a specific region enclosed with curly braces) and variables lying inside such blocks are termed as local variables. All these locally scoped statements are written and enclosed within left ({) and right braces (}) curly braces. There's a provision for nested blocks also in C which means there can be a block or a function within another block or function. So it can be said that variable(s) that are declared within a block can be accessed within that specific block and all other inner blocks of that block, but those variables cannot be accessed outside the block.

III.4. Global Variables

Variables that are declared outside of a function block and can be accessed inside the function is called global variables. But global to what ? Uhhhh !!! global to the whole program I guess.

III.4.1 Global Scope

Global variables are defined outside a function or any specific block, in most of the case, on the top of the C program. These variables hold their values all through the end of the program and are accessible within any of the functions defined in your program.
Any function can access variables defined within the global scope, i.e., its availability stays for the entire program after being declared. Ok, they sound grate but they can lead to corruption of data because they are exposed to any sub process, so make global declaration after rethinking your design. Thanks .
Global Variable Initialization
After defining a local variable, the system or the compiler won't be initializing any value to it. You have to initialize it by yourself. It is considered good programming practice to initialize variables before using. Whereas in contrast,

global variables get initialized automatically by the compiler as and when defined. Here's how based on datatype; global variables are defined.

III.5. Constant Types in C

If you want to define a variable whose value cannot be changed, you can constants.

For example,

```
const float PI = 3.14;
```

Notice, we have added keyword **const**.

Here, *PI* is a symbolic constant; its value cannot be changed.

```
const float PI = 3.14;

PI = 2.9; //Error
```

III.5.1. Defining Constants:

In C programming, we can define constants in two ways as shown below:

1. Using ***#define*** pre processor directive
2. Using a ***const*** keyword

Let us now learn about above two ways in details:

1. **Using *#define* preprocessor directive:** This directive is used to declare an alias name for existing variable or any value. We can use this to declare a constant as shown below:

   ```
   #define identifierName value
   ```

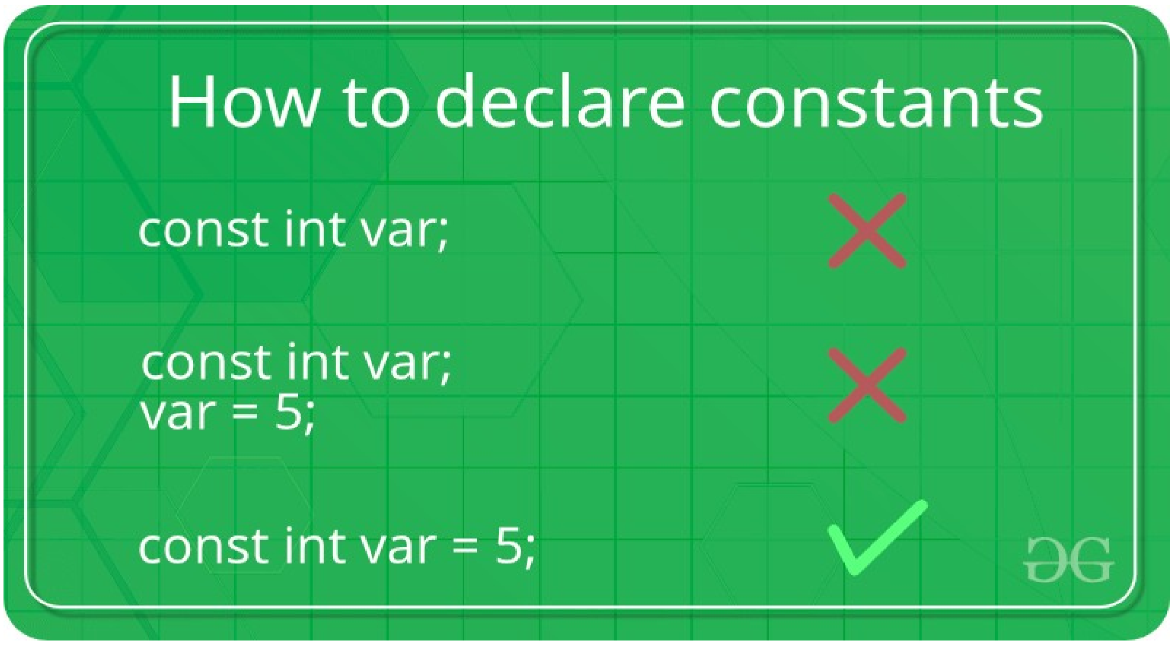
   - **identifierName:** It is the name given to constant.
   - **value:** This refers to any value assigned to identifierName.

   Example:

```
include<stdio.h>

#define val 10
#define floatVal 4.5
#define charVal 'G'

int main()
{
        printf("Integer Constant: %d\n",val);
```

```
        printf("Floating point Constant: %.1f\n",floatVal);
        printf("Character Constant: %c\n",charVal);

    return 0;
}
```

2. using a *const* keyword: Using *const* keyword to define constants is as simple as defining variables, the difference is you will have to precede the definition with a *const* keyword.



III.5.2. Constants categoris

Constants are categorized into two basic types, and each of these types has its subtypes/categories. These are:

Primary Constants

1. Numeric Constants
   - Integer Constants
   - Real Constants
2. Character Constants
   - Single Character Constants
   - String Constants
   - Backslash Character Constants

Integer Constant

It's referring to a sequence of digits. Integers are of three types viz:

1. Decimal Integer
2. Octal Integer
3. Hexadecimal Integer

Example:

15, -265, 0, 99818, +25, 045, 0X6

Real constant

The numbers containing fractional parts like 99.25 are called real or floating points constant.

Single Character Constants

It simply contains a single character enclosed within ' and ' (a pair of single quote). It is to be noted that the character '**8**' is not the same as **8**. Character constants have a specific set of integer values known as ASCII values (American Standard Code for Information Interchange).

Example:

'X', '5', ';'

String Constants

These are a sequence of characters enclosed in double quotes, and they may include letters, digits, special characters, and blank spaces. It is again to be noted that "**G**" and '**G**' are different - because "G" represents a string as it is enclosed within a pair of double quotes whereas 'G' represents a single character.

Example:

"Hello!", "2015", "2+1"

Backslash character constant

C supports some character constants having a backslash in front of it. The lists of backslash characters have a specific meaning which is known to the compiler. They are also termed as "Escape Sequence".

For Example:

\t is used to give a tab

\n is used to give a new line

Constants	Meaning
\a	beep sound
\b	backspace
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab
\v	vertical tab
\'	single quote
\"	double quote

\\	backslash
\0	null

Secondary Constant

- Array
- Pointer
- Structure
- Union
- Enumulation

  Enumulation in C

III. 5. 3. Constants declaration demo:

```
int main()
{
// int constant
const int intVal = 10;
// Real constant
const float floatVal = 4.14;

// char constant
const char charVal = 'A';
// string constant
const char stringVal[10] = "ABC";

printf("Integer constant:%d \n", intVal );
printf("Floating point constant: %.2f\n", floatVal );
printf("Character constant: %c\n", charVal );
printf("String constant: %s\n", stringVal);

return 0;
}
```

III.5.4. Enumerations

Enumeration is a user defined datatype in C language. It is used to assign names to the integral constants which makes a program easy to read and maintain. The keyword “enum” is used to declare an enumeration. They are mainly used to assign names to integral constants, the names make a program easy to read and maintain.

Here is the syntax of enum in C language,

```
enum enum_name{const1, const2,  ....... };
```

The **enum** keyword is also used to define the variables of **enum** type. There are two ways to define the variables of **enum** type as follows.

```
enum week{sunday, monday, tuesday, wednesday, thursday, friday, saturday};
enum week day;
```

Here is an example of **enum program demo** in C language,

Example

```
#include<stdio.h>

enum week{Mon=10, Tue, Wed, Thur, Fri=10, Sat=16, Sun};

enum day{Mond, Tues, Wedn, Thurs, Frid=18, Satu=11, Sund};

int main() {

 printf("The value of enum week: %d\t%d\t%d\t%d\t%d\t%d\t%d\n\n",Mon , Tue,
Wed, Thur, Fri, Sat, Sun);

 printf("The default value of enum day: %d\t%d\t%d\t%d\t%d\t%d\t%d",Mond ,
Tues, Wedn, Thurs, Frid, Satu, Sund);

   return 0;
}
```

III.6. DATA TYPES IN C LANGUAGE

Data types specify how we enter data into our programs and what type of data we enter. C language has some predefined set of data types to handle various kinds of data that we can use in our program. These datatypes have different storage capacities.

C language supports 2 different type of data types: ayayayaaa types of type ????

III.6.1 Primary data types

These are fundamental data types in C namely integer(int), floating point numbers(float), character(char) and void.

Primary data types declaration and initialization:

Datatype	Initial Default Value
int	0
char	'\0'
float	0

double	0
void	nothing

III.6.2 Derived data types

Derived data types are nothing but primary data types a little twisted or grouped together like **array**, **structure**, **union** and **pointer**.
As data type determines the type of data a variable can hold; Then I can confirm that : “ If a variable x is declared as type **int**, then it means that x will only hold integer values for its whole life !”. Every variable which is used in the program must be declared as what data-type it is. Now I think I understand what really is a data type, great !!

III.6.2.1 Arrays

An array is a collection of a fixed number of values of a single type. For example: if you want to store 100 integers in a sequence, you can create an array for it. It what ? That data type …….. for example :
int data[100]; uhhh you can’t get it by now.
OK, the above statement states that :”**data**” is a variable capable of storing one hundreds numbers !!! So, it behaves like a collection of a hundred variables of same type int? Yes.
The size and type of arrays cannot be changed after its declaration.
Arrays are of two types:

1. One-dimensional arrays
2. Multidimensional arrays

One-dimensional arrays Declaration, Initialization and Access

type name[**number of elements**];

For example, if we want an array of six integers (or whole numbers), we write:
int numbers[6]; and for a six character array called *letters:*,
char letters[6];
and so on.
You can also initialize as you declare. Just put the initial elements in curly brackets separated by commas as the initial value:

type name[**number of elements**]={**my comma-separated values**}, please not that values in an array are coma separated right ? **Coma Separated** !
For example, if we want to initialize an array with six integers, with 0, 0, 1, 0, 0, 0 a s the initial values then we will easily do it as bellow:
int point[6]={**0,0,1,0,0,0**}; very simple!
Though when the array is initialized as in this case, the array dimension may be omitted, and the array will be automatically sized to hold the initial data,interesting that the size of an array can dynamically be determined at **run time:**
int point**[]**={0,0,1,0,0,0};
This is very useful in that the size of the array can be controlled by simply adding or removing initialized elements from the definition without the need to adjust the **dimension**(size).
If the dimension is specified, but not all elements in the array are initialized, the remaining places will contain a default value of that data type like 0 in the above case of integers. This is very useful, especially when we have very large arrays.
int numbers[2000]={25};
The above example sets the first value of the array of numbers to 25, and the rest to 0.s , Uhhuuhhh it sounds resource wasting.
How to initialize an array?
It's possible to initialize an array during declaration. For example,
float marks[5] = {19, 10, 8, 17, 9};
Arrays in C are indexed starting at 0, as opposed to starting at 1 in real life counting. The first element of the array above is marks[0]. The index to the last value in the array is the array size minus one. In the example above the subscripts run from 0 through 5. C does not guarantee bounds checking on array accesses. T
Array Elements access
You can access elements of an array by indices(**indexes ?**).
Suppose you declared an array *points* as bellow. The first element is point*s[0]*, second element is point*s[1]* and so on until when you reach the 5th element.
The compiler may not complain about the following (though the best compilers do):
char point[6] = { 1, 2, 3, 4, 5, 6 };

//examples of accessing outside the array. A compile error is not always raised
y = point[15]; // the array index out of bound is not thrown but really the error has occurred
y = point[-4]; // the array index out of bound is not thrown but really the error has occurred
y = point[z]; // the array index out of bound is not thrown but really the error has occurred

Note: Your program will run but actually it is erroneous, so unexpected results.

If we want to access a variable stored in an array, for example with the above declaration, the following code will store a 1 in the variable x

```
int x;
x = point[2];
```

During program execution, an out of bounds array access does not always cause a run time error. Your program may happily continue after retrieving a value from point[-1]. To alleviate indexing problems, the **sizeof()** expression is commonly used when coding loops that process arrays.
Many people use a macro that in turn uses **sizeof**() to find the number of elements in an array, a macro variously named "**lengthof**()","**MY_ARRAY_SIZE**()" or "**NUM_ELEM**()","**SIZEOF_STATIC_ARRAY**()",etc.

```
int ix;
short anArray[]= { 3, 567, 9, 12, 15 };

for (ix=0; ix< (sizeof(anArray)/sizeof(short)); ++ix) {
  //Do Some thing With anArray[ix] element;
}
```

Notice in the above example, the size of the array was not explicitly specified but the compiler will know how to size it at 5 because of the five values in the initialization.
Adding an additional value to the list will cause it to be sized to six, and because of the sizeof expression in the for loop, the code automatically adjusts to this change. Good programming practice is to declare a variable *size* , and let it tore the number of elements in the array. Ooh headache   but interesting.

Int size = sizeof(anArray)/sizeof(short)

Few key notes:

- Arrays have 0 as the first index not 1. In this example, *marks[0]*
- If the size of an array is *n*, to access the last element, (n-1) index is used. In this example, ***marks[1999] ,*** **ok n-1 is equal to** ***1999 as n was 2000; I get it right!***
- Suppose the starting address of marks[0] is 2120d. Then, the next address, marks[1], will be 2124d, address of marks[2] will be 2128d and so on. It's because the size of a float is 4 bytes.

C also supports multi dimensional arrays (or, rather, arrays of arrays). The simplest type is a two dimensional array.

Two-dimensional arrays Declaration Initialization and Access

This creates a rectangular array made of rows and columns.

Each row has the same number of columns and each column has the same number of rows. For example to declare a char array with 3 rows and 5 columns we write in C

char two_d[**3**][**5**];

To access/modify a value in this array we need two subscripts:

char ch;

ch = two_d[2][4];// Array access ie. getting the element at rows index and at column indexes.

or

two_d[0][0] = 'x'; // Array initialization ie. putting an element **x** at the row and column indexes.

Similarly, a two-dimensional array can be initialized like this:

```
int two_d[2][3] = {
                    { 5, 2, 1 }, // row one has three columns
                    { 6, 7, 8 } // row two has three columns
                    };
```

The amount of columns must be explicitly stated; however, the compiler will find the appropriate amount of rows based on the initialized list.

There are also weird notations possible:

```
int a[100];
int i = 0;
if (a[i]==i[a])
{
  printf("Hello world!\n");
}
```

a[i] and i[a] refer to the same location. (This is explained later in the next Chapter.)

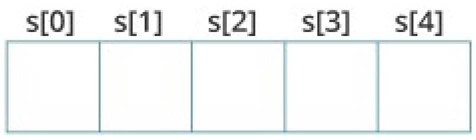
III.7 STRINGS

How to declare a string?

Before you can work with strings, you need to declare them first. Since string is an array of characters. You declare strings in a similar way like you do with character arrays.

Here's how you declare a string s:

char s[5];



5 memory address locations are allocated from the size 5 of the array.

III.7.1. How to initialize strings?

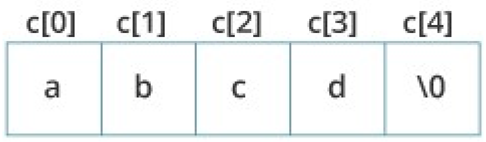
You can initialize strings in a number of ways.

1. char c[] = "abcd";

2. char c[50] = "abcd";

3. char c[] = {'a', 'b', 'c', 'd', '\0'};

4. char c[5] = {'a', 'b', 'c', 'd', '\0'};



c[0]	c[1]	c[2]	c[3]	c[4]
a	b	c	d	\0

All of the above four initialization statements will create a string c (character array) to just hold 5 characters. Great!

III.7.2. Read String from the user

You can use the **scanf**() function from the stdio library to read a string.
The **scanf**() function reads the sequence of characters until it encounters a whitespace (space, newline, tab etc.) and stops there.

Example : scanf() to read a string

```
#include <stdio.h>
int main()
{
   char name[20];
   printf("Enter name: ");
   scanf("%s", name);
   printf("Your name is %s.", name);
   return 0;
}
```

Output:

```
Enter name: Dennis Ritchie
Your name is Dennis. // This output is questionable no?
```

Even though *Dennis Ritchie* was entered in the above program, only "**Dennis**" was stored in the ***name*** string. It's because there was a space after **Dennis**.

Then How to read a line of text as it contains some white spaces?
You can use **gets**() function to read a line of string.
And, you can use **puts**() or **printf** () functions to display the string on the stream.

Example 2: gets() and puts()

```
#include <stdio.h>
int main()
{
    char name[30];
    printf("Enter name: ");
    gets(name);     // read string
    printf("Your Name is: ");
    puts(name);    // display string
    return 0;
}
```

When you run the program, the output will be:

```
Enter name: Kobusinge Peace
Your Name is: Kobusinge Peace
```

III.7.3. Passing Strings to Function

Strings can be passed to a function in a similar way as arrays.

Example 3: Passing string to a Function

```
#include <stdio.h>

void displayString(char str[]);

int main(){
    char str[50];
    printf("Enter string: ");
    gets(str);
    displayString(str);     // Passing string to a function.
    return 0;
}
void displayString(char str[])
{
    printf("The passed String was: ");
    puts(str);
}
```

C has no string handling facilities built in; consequently, strings are defined as arrays of characters; I repeat as **arrays of characters** OK. C allows a character array to be represented as a character string rather than a list of characters, good! With the **null** terminating character automatically added to the end. For example, to store the string "MUKAJISTOS", we would write
char string[10] = "MUKAJISTOS";
or
char string[10] = {'M', 'U', 'K', 'A', 'J', 'I', 'S', 'T', 'O', 'S', '\0'};
In the first example, the string will have a null character (**'\0'**) automatically appended to the end by the compiler; by convention, library functions expect strings to be terminated by a null character (**'\0'**). The latter declaration indicates individual elements, and as such the null terminator (**'\0'**) needs to be added manually.
Strings do not always have to be linked to an explicit variable. As you have seen already, a string of characters can be created directly as an unnamed string that is used directly (as with the **printf** functions.)

To create an extra long string, you will have to split the string into multiple sections, by closing the first section with a quote, and recommencing the string on the next line (also starting and ending in a quote):
char string[58] = "**This is a very, very long " "string that requires two lines.";**
While strings may also span multiple lines by putting the backslash character at the end of the line, this method is **deprecated**.
There is a useful library of string handling routines which you can use by including another header file.

#include <string.h> //new header file

This standard string library will allow various tasks to be performed on strings.

III.7.4. String Manipulations Functions(Important to know)

Function	Work of Function
strlen()	Calculates the length of string
strcpy()	Copies a string to another string
strcat()	Concatenates(joins) two strings
strcmp()	Compares two string
strlwr()	Converts string to lowercase
strupr()	Converts string to uppercase

III.8. ARRAY SORTING

Computers are designed to quickly and merrily accomplish boring tasks, such as sorting an array. In fact, they love doing it so much that “**the sort**” is a basic computer concept upon which many theories and algorithms have been written.

The simplest sort is the ***bubble sort***, which not only is easy to explain and understand but also has a fun name(**Bubble Sort**). It also best shows the basic array-sorting philosophy, which is to **swap** values between two elements. Goog now we all know what sorting is really about!! It is about **swapping** elements: SIMPLE

III.8.1. Bubble Sort

Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.
If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will **swap** both the elements, and then move on to compare the second and the third element, and so on.

If we have total of n elements, then we need to repeat this process for **n-1** times.
It is known as **bubble sort**, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water **bubble** rises up to the water surface.

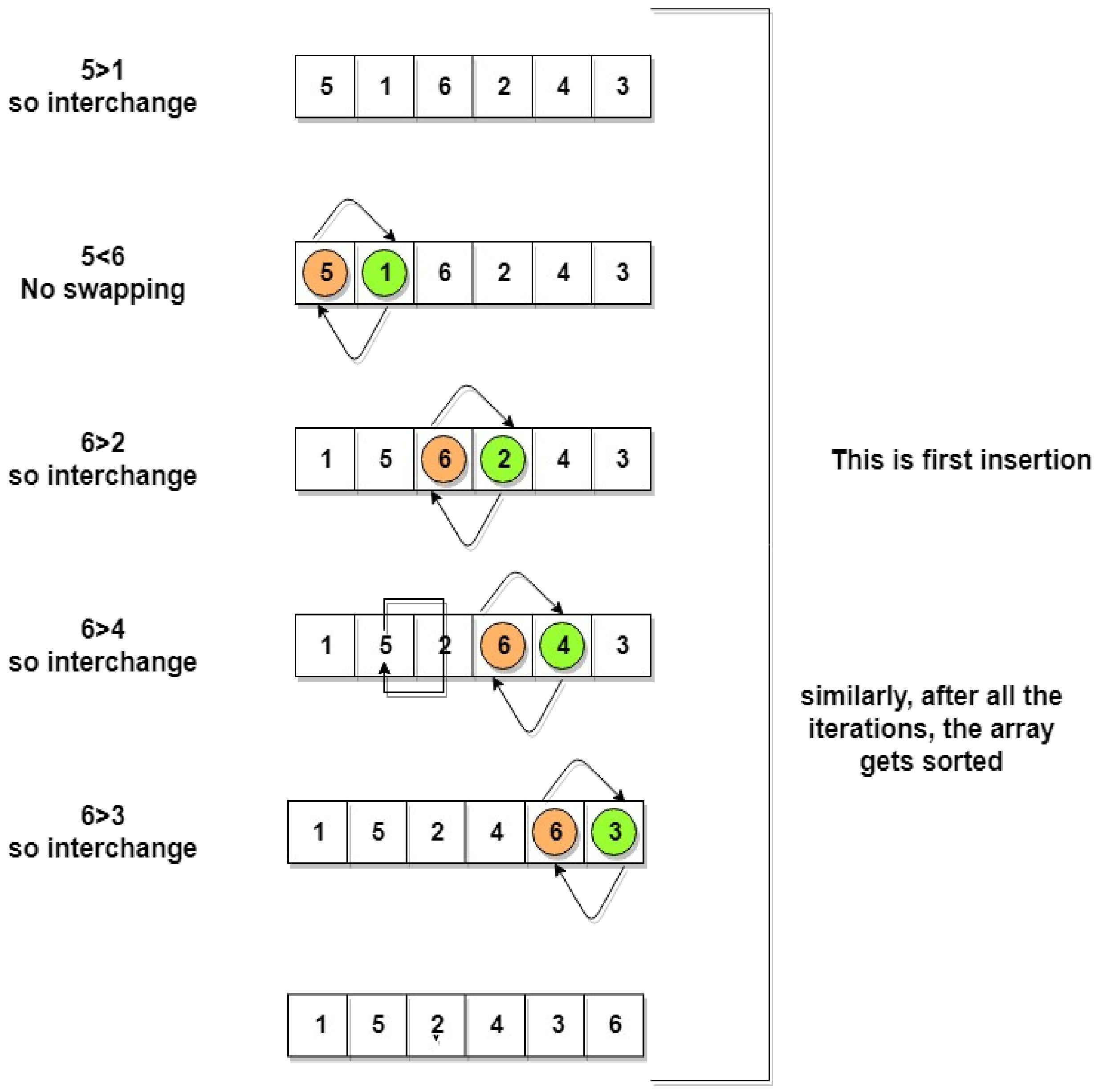
Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

Implementing Bubble Sort Algorithm

Following are the steps involved in bubble sort(for sorting a given array in ascending order):

1. Starting with the first element(index = 0), compare the current element with the next element of the array.
2. If the current element is greater than the next element of the array, swap them.
3. If the current element is less than the next element, move to the next element. **Repeat Step 1**.

Let's consider an array with values {5, 1, 6, 2, 4, 3}
Below, we have a pictorial representation of how bubble sort will sort the given array.



So as we can see in the representation above, after the first iteration, 6 is placed at the last index, which is the correct position for it.
Similarly after the second iteration, 5 will be at the second last index, and so on.

Time to write the code for bubble sort:

// below we have a simple C program for bubble sort

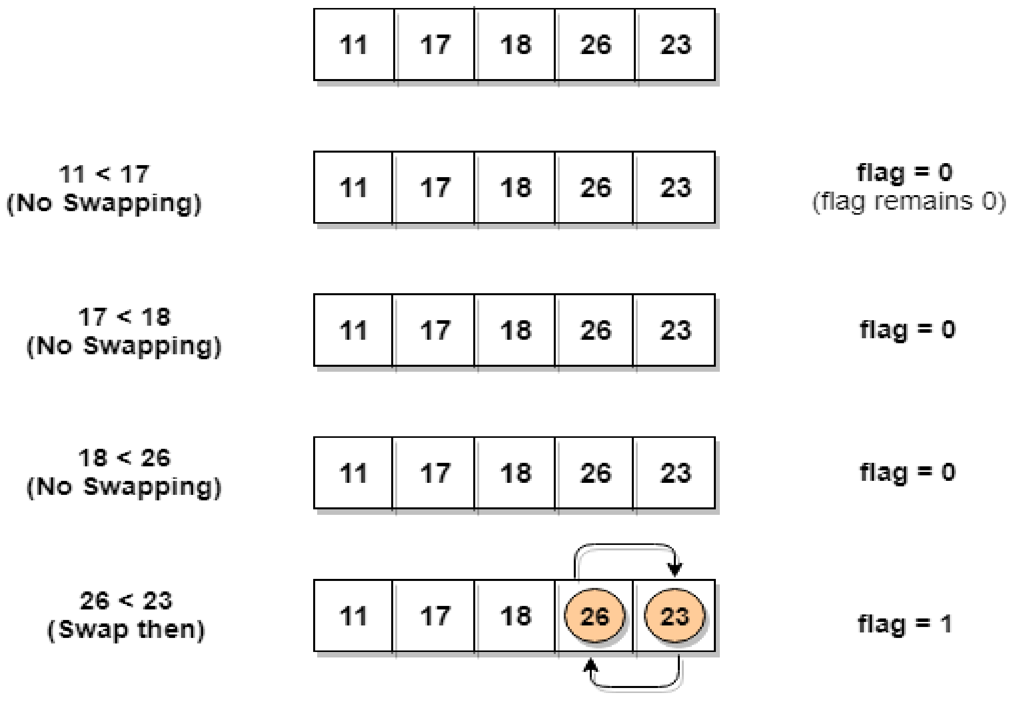
```
#include <stdio.h>
// Defining the function to handle the sorting
void bubbleSort(int arr[], int n)
```

```
{
int array[]={some unsorted elements};
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if( array[j] > array[j+1])
            {
                // swap the elements
                temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
            }
        }
    }
    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", array[i]);
    }
}

int main()
{
    int array[100], i, n, step, temp;
    // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);

    // input elements in the array
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &array[i]);
    }
    // call the function to bubble Sort and give it the array to be sorted
    bubbleSort(array, n);

    return 0;
}
```

Optimized Bubble Sort Algorithm

To optimize our bubble sort algorithm, we can introduce a flag to monitor whether elements are getting swapped inside the inner for loop.
Hence, in the inner for loop, we check whether swapping of elements is taking place or not, every time.

If for a particular iteration, no swapping took place, it means the array has been sorted and we can jump out of the for loop, instead of executing all the iterations.
Let's consider an array with values {11, 17, 18, 26, 23}
Below, we have a pictorial representation of how the optimized bubble sort will sort the given array.



As we can see, in the first iteration, swapping took place, hence we updated our **flag value** to 1, as a result, the execution enters the for loop again. But in the second iteration, no swapping will occur, hence the value of flag will remain 0, and execution will break out of loop.

```
// below we have a simple C program for improved bubble sort
#include <stdio.h>
void bubbleSort(int arr[], int n)
{
 int array[]={some unsorted elements};
    int i, j, temp;
    for(i = 0; i < n; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            // introducing a flag to monitor swapping
            int flag = 0;
            if( array[j] > array[j+1])
            {
                // swap the elements
                temp = array[j];
                array[j] = array[j+1];
                array[j+1] = temp;
                // if swapping happens update flag to 1
                flag = 1;
            }
        }
        // if value of flag is zero after all the iterations of inner loop
        // then break out
        if(!flag)
        {
            break;
        }
    }
    // print the sorted array
    printf("Sorted Array: ");
    for(i = 0; i < n; i++)
    {
        printf("%d  ", arrray[i]);
    }
}

int main()
{
    int array[100], i, n, step, temp;
    // ask user for number of elements to be sorted
    printf("Enter the number of elements to be sorted: ");
    scanf("%d", &n);
    // input elements in the array one by one
    for(i = 0; i < n; i++)
    {
        printf("Enter element no. %d: ", i+1);
        scanf("%d", &array[i]);
    }
    // call the function bubbleSort
    bubbleSort(array, n);

    return 0;
}
```

In the above code, in the function bubbleSort, if for a single complete cycle of j iteration(inner for loop), no swapping takes place, then flag will remain 0 and then we will break out of the for loops, because the array has already been sorted.

III.8.2. Selection Sort

Selection sort is conceptually the most simplest sorting algorithm. This algorithm will first find the **smallest** element in the array and swap it with the element in the **first** position, then it will find the **second smallest** element and swap it with the element in the **second** position, and it will keep on doing

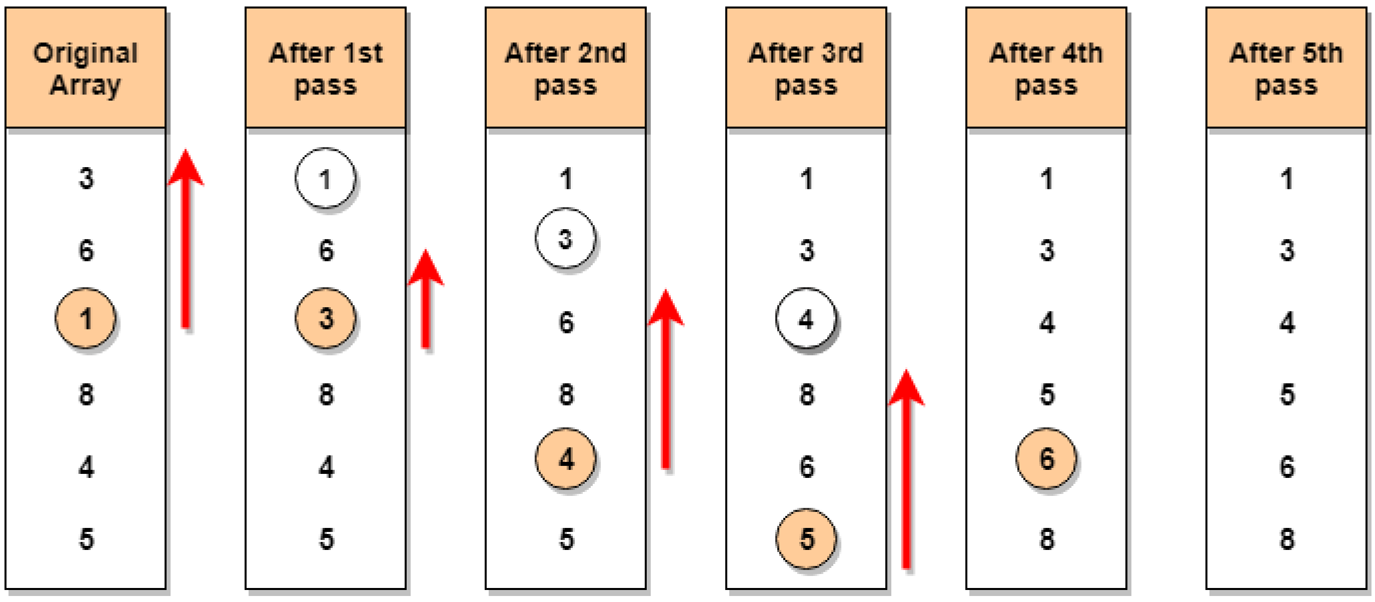
this until the entire array is sorted. **Not that now we are doing ascending order sorting; so, for descending you will just change the direction of the comparison operator**
It is called selection sort because it repeatedly **selects** the next-**smallest /greatest** element and swaps it into the right place.

How Selection Sort Works?

Following are the steps involved in selection sort(for sorting a given array in ascending order):

1. Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
2. We then move on to the second position, and look for smallest element present in the sub array, starting from index 1, till the last index.
3. We replace the element at the **second** position in the original array, or we can say at the first position in the sub array, with the second smallest element.
4. This is repeated, until the array is completely sorted.

Let's consider an array with values {3, 6, 1, 8, 4, 5}
Below, we have a pictorial representation of how selection sort will sort the given array.



In the **first** pass, the smallest element will be 1, so it will be placed at the first position.
Then leaving the first element, **next smallest** element will be searched, from the remaining elements. We will get 3 as the smallest, so it will be then placed at the second position.
Then leaving 1 and 3(because they are at the correct position), we will search for the next smallest element from the rest of the elements and put it at third position and keep doing this until array is sorted.

Finding Smallest Element in a sub array
In selection sort, in the first step, we look for the smallest element in the array and replace it with the element at the first position. This seems doable, isn't it?
Consider that you have an array with following values {3, 6, 1, 8, 4, 5}. Now as per selection sort, we will start from the first element and look for the smallest number in the array, which is 1 and we will find it at the **index** 2. Once the smallest number is found, it is swapped with the element at the first position.
Well, in the next iteration, we will have to look for the second smallest number in the array. How can we find the second smallest number? This one is tricky?
If you look closely, we already have the smallest number/element at the first position, which is the right position for it and we do not have to move it anywhere now. So we can say, that the first element is sorted, but the elements to the right, starting from index 1 are not.
So, we will now look for the smallest element in the subarray, starting from index 1, to the last index.
Confused? Give it time to sink in.
After we have found the second smallest element and replaced it with element on index 1(which is the second position in the array), we will have the first two positions of the array sorted.
Then we will work on the sub array, starting from index 2 now, and again looking for the smallest element in this sub array.

Implementing Selection Sort Algorithm

In the C program below, we have tried to divide the program into small functions, so that it's easier for you to understand which part is doing what.

There are many different ways to implement selection sort algorithm, here is the one that we like:

```
// C program implementing Selection Sort
# include <stdio.h>

// function to swap elements at the given index values
void swap(int array[], int firstIndex, int secondIndex)
{
    int temp;
    temp = array[firstIndex];
    array[firstIndex] = array[secondIndex];
    array[secondIndex] = temp;
}

// function to look for smallest element in the given subarray
int indexOfMinimum(int array[], int startIndex, int n)
{
    int minValue = array[startIndex];
    int minIndex = startIndex;

    for(int i = minIndex + 1; i < n; i++) {
        if(array[i] < minValue)
        {
            minIndex = i;
            minValue = array[i];
        }
    }
    return minIndex;
}

void selectionSort(int array[], int n)
{
    for(int i = 0; i < n; i++)
    {
        int index = indexOfMinimum(array, i, n);
        swap(array, i, index);
    }

}
// the function that prints the content of any size array
void printArray(int array[], int size)
{
    int i;
    for(i = 0; i < size; i++)
    {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main()
{
    int array[] = {46, 52, 21, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(array, n);
    printf("Sorted array: \n");
    printArray(array, n);
    return 0;
}
```

I advise each leaner to rewrite down the above program

Note: Selection sort is an **unstable sort** i.e it might change the occurrence of two similar elements in the list while sorting. But it can also work as a stable sort when it is implemented using **linked list**. Oohh my God yoyoooo !! what are those **Linked lists**? I wish you wait until when you do the course of **data structures and algorithms**, I like C programming almost everything was thought of..

III.8.3. Insertion Sort

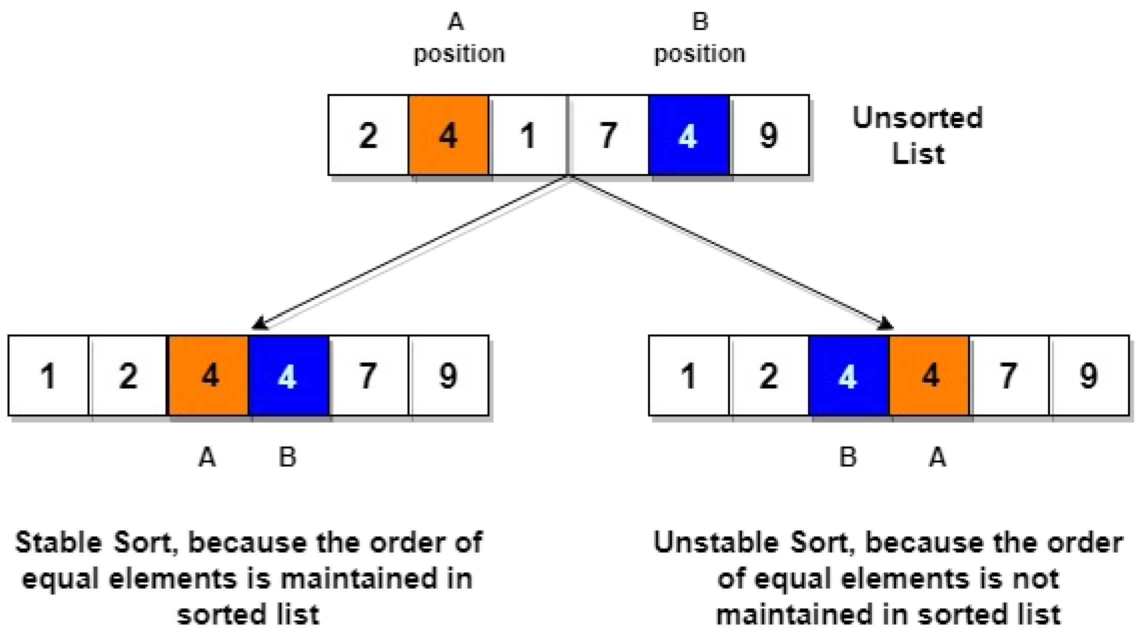
Consider you have 10 cards out of a deck of cards in your hand. And they are sorted, or arranged in the ascending order of their numbers.
If I give you another card, and ask you to **insert** the card in just the right position, so that the cards in your hand are still sorted. What will you do?
Well, let me guess : " you will have to go through each card from the starting or the back and find the right position for the new card, comparing it's value with each card" **Right ?** . Ok, Once you find the right position, you will **insert** the card there,**Voila ! You have done insertion, but is it an insertion sort ?**

Similarly, if more new cards are provided to you, you can easily repeat the same process and insert the new cards and keep the cards sorted too.

This is exactly how **insertion sort** works. It starts from the index 1(**not 0**), and each index starting from index 1 is like a new card, that you have to place at the right position in the sorted sub array on the left.

Following are some of the important **characteristics of Insertion Sort**:

1. It is efficient for smaller data sets, but very inefficient for larger lists.
2. Insertion Sort is adaptive, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
3. It is better than Selection Sort and Bubble Sort algorithms.
4. Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.
5. It is a **stable** sorting technique, as it does not change the relative order of elements which are equal.

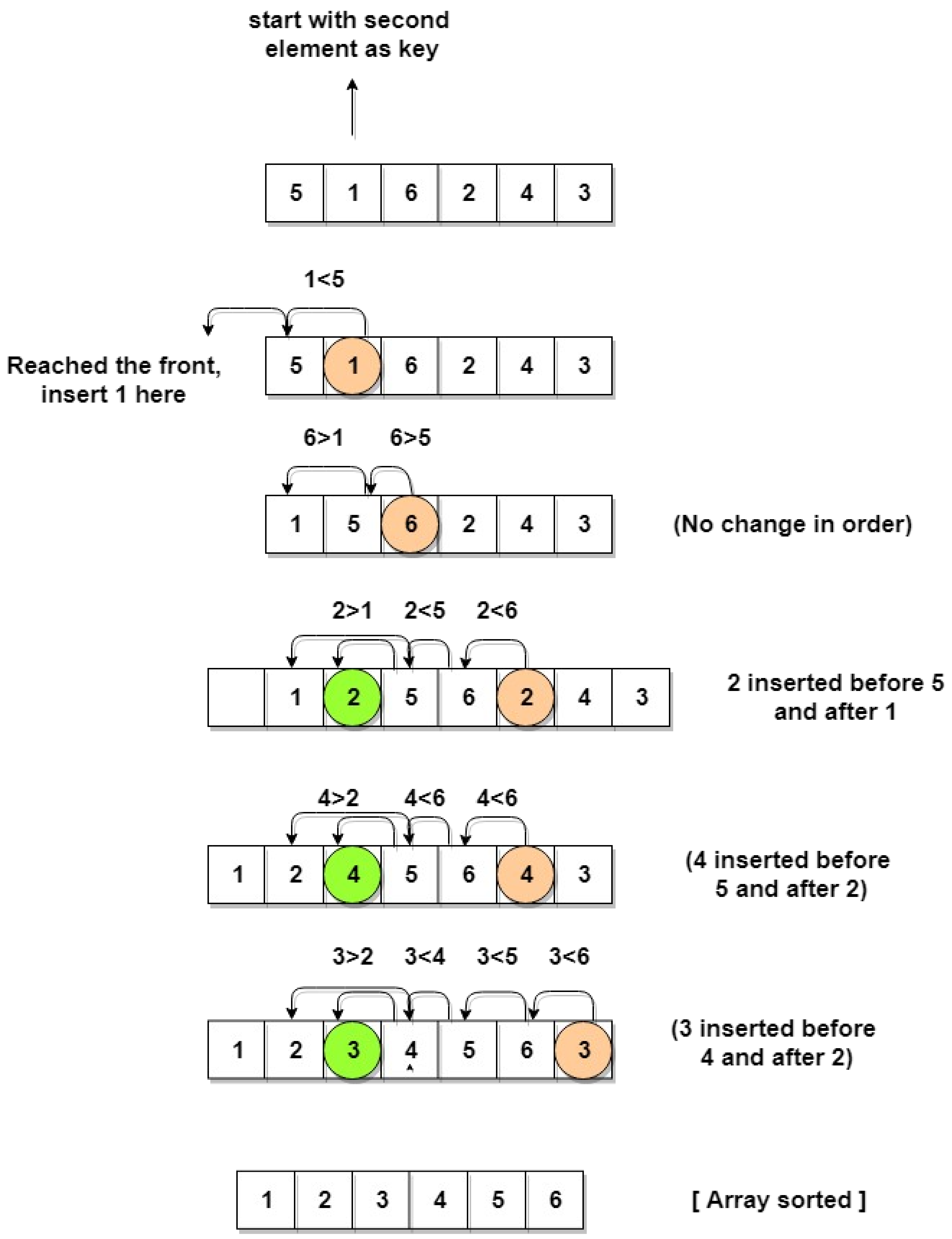

A position
B position
2 4 1 7 4 9
Unsorted List
1 2 4 4 7 9
A B
1 2 4 4 7 9
B A
Stable Sort, because the order of equal elements is maintained in sorted list
Unstable Sort, because the order of equal elements is not maintained in sorted list

How Insertion Sort Works?

Following are the steps involved in insertion sort:

1. We start by making the second element of the given array, i.e. element at index 1, the key. The key element here is the new card that we need to add to our existing sorted set of cards(remember the example with cards above).
2. We compare the key element with the element(s) before it, in this case, element at index 0:
    - If the key element is less than the first element, we insert the key element before the first element.
    - If the key element is greater than the first element, then we insert it after the first element.
3. Then, we make the third element of the array as key and will compare it with elements to it's left and insert it at the right position.
4. And we go on repeating this, until the array is sorted.

Let's consider an array with values {5, 1, 6, 2, 4, 3}
Below, we have a pictorial representation of how bubble sort will sort the given array.



As you can see in the diagram above, after picking a key, we start iterating over the elements to the left of the key.

We continue to move towards left if the elements are greater than the key element and stop when we find the element which is less than the key element.
And, insert the key element after the element which is less than the key element.
Implementing Insertion Sort Algorithm:
Below we have a simple implementation of Insertion sort in C language.

```
#include <stdio.h>
//member functions declaration
void insertionSort(int array[], int length);
void printArray(int array[], int size);

// main function
int main()
{
	int array[5] = {5, 1, 6, 2, 4, 3};
	// calling insertion sort function to sort the array
	insertionSort(array, 6);
	return 0;
}

void insertionSort(int array[], int length)
{
	int i, j, key;
	for (i = 1; i < length; i++)
	{
		j = i;
		while (j > 0 && array[j - 1] > array[j])
		{
			key = array[j];
			array[j] = array[j - 1];
			array[j - 1] = key;
			j--;
		}
```

```
        }
        printf("Sorted Array: ");
        // print the sorted array
        printArray(arrayy, length);
}

// function to print the given array
void printArray(int array[], int size)
{
        int j;
        for (j = 0; j < size; j++)
        {
                printf(“ %d “,array[j]);
        }
        printf(“\n”);s
}
```

Sorted Array: **1 2 3 4 5 6**

Now let's try to understand the above simple insertion sort algorithm.
We took an array with **6** integers. We took a variable key, in which we put each element of the array, during each pass, starting from the **second** element, that is a[1].
Then using the while loop, we iterate, until j becomes equal to **zero** or we find an element which is greater than key, and then we **insert** the key at that position.
We keep on doing this, until j becomes equal to **zero**, or we encounter an element which is smaller than the key, and then we stop. The current key is now at the right position.
We then make the next element as key and then repeat the same process.
In the above array, first we pick **1** as key, we compare it with **5**(element before 1), **1** is smaller than **5**, we insert **1** before **5**. Then we pick **6** as key, and compare it with **5** and **1**, no shifting in position this time. Then **2** becomes the key and is compared with **6** and **5**, and then **2** is inserted after **1**. And this goes on until the complete array gets sorted.

III.9. POINTER VARIABLES

III.9.1. Address in C

Before you get into the concept of pointers, let's first get familiar with address in C.
If you have a variable *var* in your program, &var will give you its address in the memory, where & is commonly called the **reference** operator.

You must have seen this notation while using scanf() function. It was used in the function to store the user inputted value in the address of *var, **I know you have already seen it the magic “&”***.

scanf("%d", &var); // values are actually stored in memory locations of variables

```
#include <stdio.h>
int main()
{
  int var = 5;
 printf("Value: %d\n", var);
  printf("Address: %u", &var);  //Notice, the ampersand(&) before var.
  return 0;
}
```

Output

```
Value: 5
Address: 2686778
```

Note: You may obtain different value of address while using this code.
In above source code, value 5 is stored in the memory location 2686778. *var* is just the name given to that location.

In C, you can create a special variable that stores the address (rather than the value). This variable is called a : “**pointer** variable” or simply a: ”**pointer”** .

III.9.2. How to create a pointer variable?

data_type * pointer_variable_name;

int * p;

Above statement defines, ***p*** as pointer variable of type **int**.

Reference operator (**&**) and dereference operator (***)**

As discussed, & is called reference operator. It gives you the address of a variable.

Likewise, there is another operator that gets you the value from the address, it is called a dereference operator *****.

Below example clearly demonstrates the use of pointers, reference operator and dereference operator.

Note: The * sign when declaring a pointer is not a dereference operator. It is just a similar notation that creates a pointer.

Example: How Pointer Works?

```
#include <stdio.h>
int main()
{
   int * pc, c;

   c = 22;
   printf("Address of c: %u\n", &c);
   printf("Value of c: %d\n\n", c);

   pc = &c;
   printf("Address of pointer pc: %u\n", pc);
   printf("Content of pointer pc: %d\n\n", *pc);

   c = 11;
   printf("Address of pointer pc: %u\n", pc);
   printf("Content of pointer pc: %d\n\n", *pc);
// a pointer can change the valu of a variable
 *pc = 2;
   printf("Address of c: %u\n", &c);
   printf("Value of c: %d\n\n", c);
   return 0;
}
```

Output

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2
```

Explanation of the program

1. int* pc, c;

   Here, a pointer *pc* and a normal variable *c*, both of type int, is created. Since *pc* and *c* are not initialized at first, pointer *pc* points to either no address or a random address. And, variable *c* has an address but contains a random garbage value.
2. c = 22;

   This assigns 22 to the variable *c*, i.e., 22 is stored in the memory location of variable *c*.

   Note that, when printing &c (address of c), we use %u rather than %d since address is usually expressed as an unsigned integer (**always positive)**.

3. pc = &c;

   This assigns the address of variable *c* to the pointer *pc*.
   You see the value of *pc* is same as the address of *c* and the content of *pc* is 22 as well.
4. c = 11;

   This assigns 11 to variable *c*.
   Since, pointer *pc* points to the same address as *c*, value pointed by pointer *pc* is 11 as well. Interesting!.

5. *pc = 2;

   This change the value at the memory location pointed by pointer *pc* to 2.
   Since the address of the pointer *pc* is same as the address of *c*, value of *c* is also changed to 2. This now **more Interesting!** How a pointer can easily modify the value of the variable it point to.

III.9.3. Pointer access and errors

However, initializing pointers unnecessarily could hinder program analysis, thereby hiding bugs.
In any case, once a pointer has been declared, the next logical step is for it to point at something:

```
int a = 5;
int *ptr = NULL;
```

ptr = &a; // address to pointer assignment

This assigns the value of the address of a to ptr. For example, if a is stored at memory location of 0x8130 then the value of ptr will be 0x8130 after the assignment. To dereference the pointer, an asterisk is used again:

```
*ptr = 8;
```

This means take the contents of ptr (which is 0x8130), "locate" that address in memory and set its value to 8. If **a** is later accessed again, its new value will be 8. Which means :”**A pointer can change the value of a variable!**”

This example may be clearer if memory is examined directly. Assume that a is located at address 0x8130 in memory and ptr at 0x8134; also assume this is a 32-bit machine such that an int is 32-bits wide. The following is what would be in memory after the following code snippet is executed:

```
int a = 5;
int *ptr = NULL;
```

Address	Contents
0x8130	0x00000005
0x8134	0x00000000

(The NULL pointer shown here is 0x00000000.) By assigning the address of a to ptr:

```
 ptr = &a;
```

yields the following memory values:

Address	Contents
0x8130	0x00000005
0x8134	0x00008130

Then by dereferencing ptr by coding:

```
 *ptr = 8;
```

the computer will take the contents of ptr (which is 0x8130), 'locate' that address, and assign 8 to that location yielding the following memory:

Address	Contents
0x8130	0x00000008
0x8134	0x00008130

Clearly, accessing a will yield the value of 8 because the previous instruction modified the contents of a by way of the pointer ptr.

Common mistakes when working with pointers

Suppose, you want pointer *pc* to point to the address of *c*. Then,

```
int c, *pc;
// Wrong! pc is address whereas,
// c is not an address.
pc = c;
// Wrong! *pc is the value pointed by address whereas,
// &c is an address.
*pc = &c;
// Correct! pc is an address and,
// &c is also an address.
ptr = &a;
// Correct! * ptr is the value pointed by address and,
// c is also a value (not address).
*pc = c;
```

III.9.4. Strings and Pointers

Similarly like arrays, string names are "decayed" to pointers. Hence, you can use pointer with the same name as string to manipulate elements of the string.

Example : **Strings and Pointers**

```
#include <stdio.h>

int main(void) {
  char name[] = "Harry Potter"; // A string initialization as a character array

  printf("%c", *name);     // Output: H
  printf("%c", *(name+1));   // Output: a
  printf("%c", *(name+7));   // Output: o

  char *namePtr;

  namePtr = name;
  printf("%c", *namePtr);     // Output: H
  printf("%c", *(namePtr+1));   // Output: a
  printf("%c", *(namePtr+7));   // Output: o
}
```

III.9.5. Benefits of using pointers

Below we have listed a few benefits of using pointers:

1. Pointers are more efficient in handling Arrays and Structures.
2. Pointers allow references to function and thereby helps in passing of function as arguments to other functions.
3. They reduce length of the program and its execution time as well.
4. they allows C language to support Dynamic Memory management.

III.10. STRUCTURES

A Structure is a collection of variables (can be of different types) under a single name variable.
For example: You want to store information about a person: his/her name, citizenship number and salary. You can create different variables *name*, *citNo* and *salary* to store these information separately.
What if you need to store information of more than one person? Now, you need to create different variables for each information per person: *name1*, *citNo1*, *salary1*, *name2*, *citNo2*, *salary2* **oooh** agree codes that will serve nothing in long run etc.
A better approach would be to have a **collection of all related information** under a single name Person **structure**, and use it for every person.

III.10.1. How to define a structure?

Keyword **struct** is used for creating a structure.

III.10.2. Syntax of structure

```
struct  [structure_name /tag]{
   data_type member1;
   data_type member2;
   ...
   data_type memeber;
}[variables]; //coma separated variables declaration
```

Here is an example:

```
struct Person{
   char name[50];
   int citNo;
   float salary;
};
```

Here, a derived data type **struct Person** is defined.

III.10.3. Create structure variable

When a structure is defined, it creates a user-defined type. However, no storage or memory is allocated. To allocate memory of a given structure type and work with it, we need to create variables.

Here's how we create structure variables:

```
struct Person
{
   char name[50];
   int citNo;
   float salary;
}person0;

int main()
{
   struct Person person1, person2, persons[20]; // variable declaration
using Person structure
    return 0;
}
```

Another way of creating a structure variable is:

```
struct Person
{
    char name[50];
    int citNo;
    float salary;
} person1, person2, persons[20];
```

In both cases, two variables ***person1***, ***person2***, and an array variable ***persons*** having 20 elements of type **struct Person** are created.

III.10.4. How to Access members of a structure?

There are two types of operators used for accessing members of a structure.

1. Member operator(.)
2. Structure pointer operator(->) (this will be discussed in structure and pointers)

Suppose, you want to access salary of *person2*. Here's how you can do it:

person2.salary **// Very simple**

Example:

```
// Program to add two distances which is in feet and inches
#include <stdio.h>
struct Distance
{
    int feet;
    float inch;
} dist1, dist2, sum;

int main()
{
    printf("1st distance\n");
    printf("Enter feet: ");
    scanf("%d", &dist1.feet);
    printf("Enter inch: ");
    scanf("%f", &dist1.inch);
    printf("2nd distance\n");

    printf("Enter feet: ");
    scanf("%d", &dist2.feet);

    printf("Enter inch: ");
    scanf("%f", &dist2.inch);

    // adding feet
    sum.feet = dist1.feet + dist2.feet;
    // adding inches
    sum.inch = dist1.inch + dist2.inch;

    // changing feet if inch is greater than 12
    while (sum.inch >= 12)
    {
        ++sum.feet;
        sum.inch = sum.inch - 12;
    }

    printf("Sum of distances = %d\'-%.1f\"", sum.feet, sum.inch);
    return 0;
}
```

Output

```
1st distance
Enter feet: 12
Enter inch: 7.9
2nd distance
Enter feet: 2
Enter inch: 9.8
Sum of distances = 15'-5.7"
```

III.10.5. Keyword typedef In C

Keyword **typedef** can be used to simplify syntax of a structure.

This code

```
struct Distance{
    int feet;
    float inch;
};

int main() {
    structure Distance d1, d2;
}
```

is equivalent to

```
typedef struct Distance{
    int feet;
    float inch;
} distances;

int main() {
    distances d1, d2, sum; // declaring variable using typedof
}
```

typedef is a keyword used in C language to assign alternative names to existing data types. Its mostly used with user defined data types, when names of the data types become slightly complicated to use in programs. Following is the general syntax for using **typedef**,

typedef <**existing_name**> <**alias_name**>

Lets take an example and see how typedef actually works.

```
typedef unsigned long ulong;
```

The above statement define a term ulong for an unsigned long datatype. Now this ulong identifier can be used to define unsigned long type variables.

```
ulong i, j;
```

Application of typedef

typedef can be used to give a name to user defined data type as well. Lets see its use with structures.

```
typedef struct
{
    type member1;
    type member2;
    type member3;
} type_name;
```

Here **type_name** represents the structure definition associated with it. Now this **type_name** can be used to declare a variable of this structure type.

type_name t1, t2;

Example of Structure definition using **typedef**

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

typedef struct employee
{
    char name[50];
    int salary;
}emp;

void main( )
{
    emp e1;
    printf("\nEnter Employee record:\n");
    printf("\nEmployee name:\t");
    scanf("%s", e1.name);
    printf("\nEnter Employee salary: \t");
    scanf("%d", &e1.salary);
    printf("\nstudent name is %s", e1.name);
    printf("\nroll is %d", e1.salary);
}
```

typedef and Pointers

typedef can be used to give an alias name to pointers also. Here we have a case in which use of **typedef** is beneficial during pointer declaration.

In Pointers * binds to the right and not on the left.

int* x, y;

By this declaration statement, we are actually declaring x as a pointer of type int, whereas y will be declared as a plain int variable. Not clear! I know, but its OK.

typedef int* IntPtr; define an **IntPtr** type that can be used to declare pointers without the use of the magic dereference operator (*****) .

Ckeck bellow :

IntPtr x, y, z;

But if we use **typedef** like we have used in the example above, we can declare any number of pointers in a single statement.

What? x, y and z are **pointers without a * ? Wawoooooooo !!!!!!!!!!!!**

Yes the are!

III.10.6. Nested Structures

You can create structures within a structure in C programming. For example:

```
struct complex
{
 int imag;
 float real;
};

struct number
{
   struct complex comp;
   int integers;
} num1, num2;
```

Suppose, you want to set *imag* of *num2* variable to 11. Here's how you can do it:

num2.comp.imag = 11;

III.10.7. Pointer to structure declaration

```
 struct student *ptr;
```

Allocating memory to the pointer to structure

```
    ptr = (struct student*)malloc(sizeof(struct student));
```

The statement will declare memory for one student's record at run time.

III.10.7.1. Accessing structure members using pointer

The **arrow operator** (**->**) is used to access the members of the structure using **pointer to structure**.

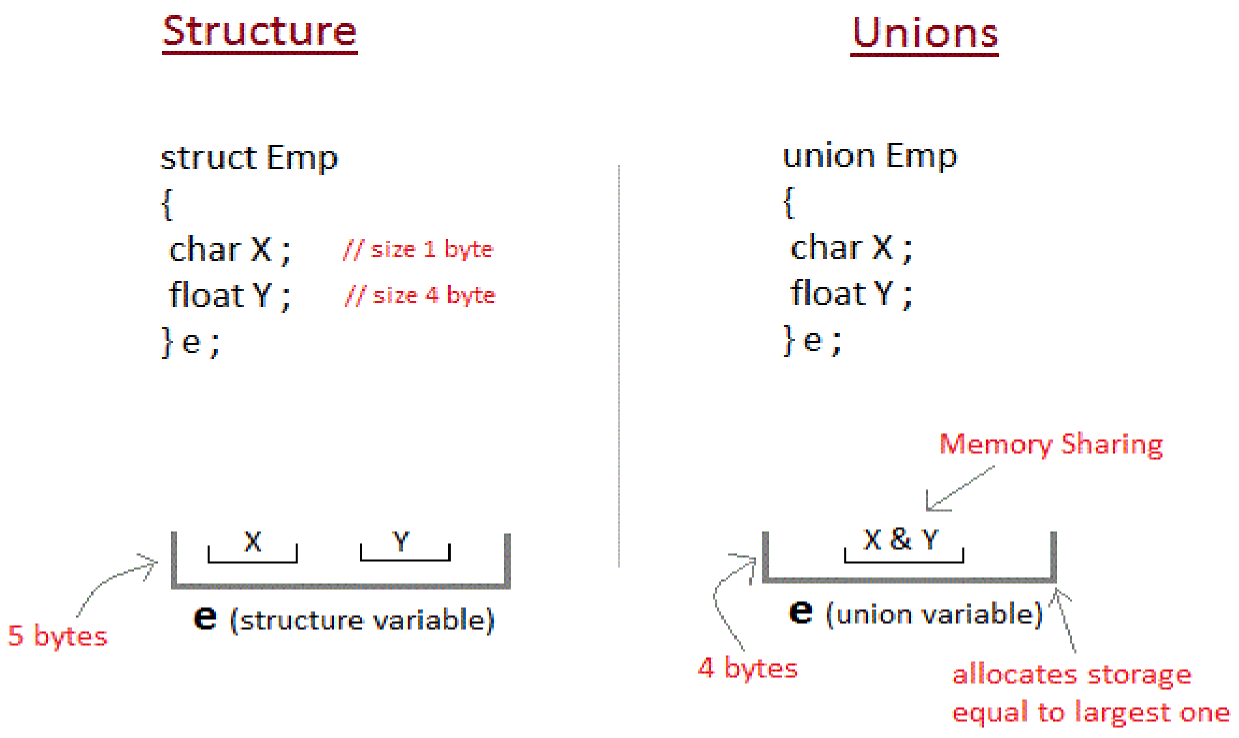
For example, if we want to access member name, and ptr is the **pointer to structure**. The statement to access name will be ptr->name.

C Example program for pointer to structure

```
#include <stdio.h>

//structure definition
struct student{
        char name[50];
        int age;
        int rollno;
};

//main function
int main(){
        //pointer to structure declaration
        struct student *ptr;

        //allocating memory at run time
        ptr = (struct student*)malloc(sizeof(struct student));

        //check memory availability
        if( ptr == NULL){
                printf("Unable to allocate memory!!!\n");
                exit(0);
        }

        //reading values of the structure
        printf("Enter student details...\n");
        printf("Name? ");
        scanf("%[^\n]", ptr->name); //reads string with spaces
        printf("Age? ");
        scanf("%d", &ptr->age);
        printf("Roll number? ");
        scanf("%d", &ptr->rollno);

        //printing the details
        printf("\nEntered details are...\n");
        printf("Name: %s\n", ptr->name);
        printf("Age: %d\n", ptr->age);
        printf("Roll number: %d\n", ptr->rollno);

        //freeing dynamically allocated memory
        free(ptr);

        return 0;
}
```

III.11. UNIONS

Unions are conceptually similar to **structures**. The syntax to declare/define a union is also similar to that of a structure. The only differences is in terms of storage. In **structure** each member has its own storage location, whereas all members of a **union** uses a single shared memory location which is equal to the size of its **largest** data member.
Illustration :



This implies that although a **union** may contain many members of different types, **it cannot handle all the members at the same time**. A **union** is declared using the **union** keyword:

union [union name/tag]

{

  // member variables goes here

}**[union variable]**;

This declares a variable It1 of type union item. This union contains three members each with a different data type. However only one of them can be used at a time. This is due to the fact that only one memory location is allocated for all the union variables, irrespective of their size. The compiler allocates the storage that is large enough to hold the largest variable type in the union.

In the union declared above the member x requires **4 bytes** which is largest among-st the members for a 16-bit machine. Other members of union will share the same memory address.

III.11.1. Accessing a Union Member

Syntax for accessing any union member is similar to accessing structure members,

```
union test
{
    int a;
    float b;
    char c;
}t;

t.a;      //to access members a of union t
t.b;      //to access members b of union t
t.c;      //to access members c of union t
```

Union Sample program

```
#include <stdio.h>

union item
{
    int a;
    float b;
    char ch;
};
int main( )
{
    union item it;
    it.a = 12;
    printf("%d\n", it.a);
    it.b = 20.2;
    printf("%f\n", it.b);
    it.ch = 'z';

    printf("%c\n", it.ch);
```

```
   // Checking for corruption in memory
    printf("%d\n", it.a);
    printf("%f\n", it.b);
    printf("%c\n", it.ch);

    return 0;
}
```

Corrupted output:

```
-26426 20.1999 z
```

As you can see here, the values of a and b get corrupted and only variable c prints the expected result **z**. This is because in union, the memory is shared among different data types. Hence, the only member whose value is currently stored will have the memory.

In the above example, value of the variable c “**Z**”was stored at last, hence the values of other variables were lost.

III.12. FUNCTION IN C

III.12.1. Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts −

return_type function_name(parameter list);

For the above defined function max(), the function declaration is as follows −

int max(int num1, int num2);

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration −

int max(int, int);

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

Types of function:

In C programming language we has two types of functions:

- **Predefined function or Library Function**, "C" provides some function which are created by its own library. that's why these functions are also called library functions . these functions are declared in the header file like- scanf(), printf(), gets(), puts(), ceil(), floor() etc.
- **User defined function** , "C" provides the facility like the other languages to create the function according to the situation by the programmer. user defined functions reduces the complexity of a big program and complexity of code.

III.12.2. Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to **call** that function to perform the defined task.

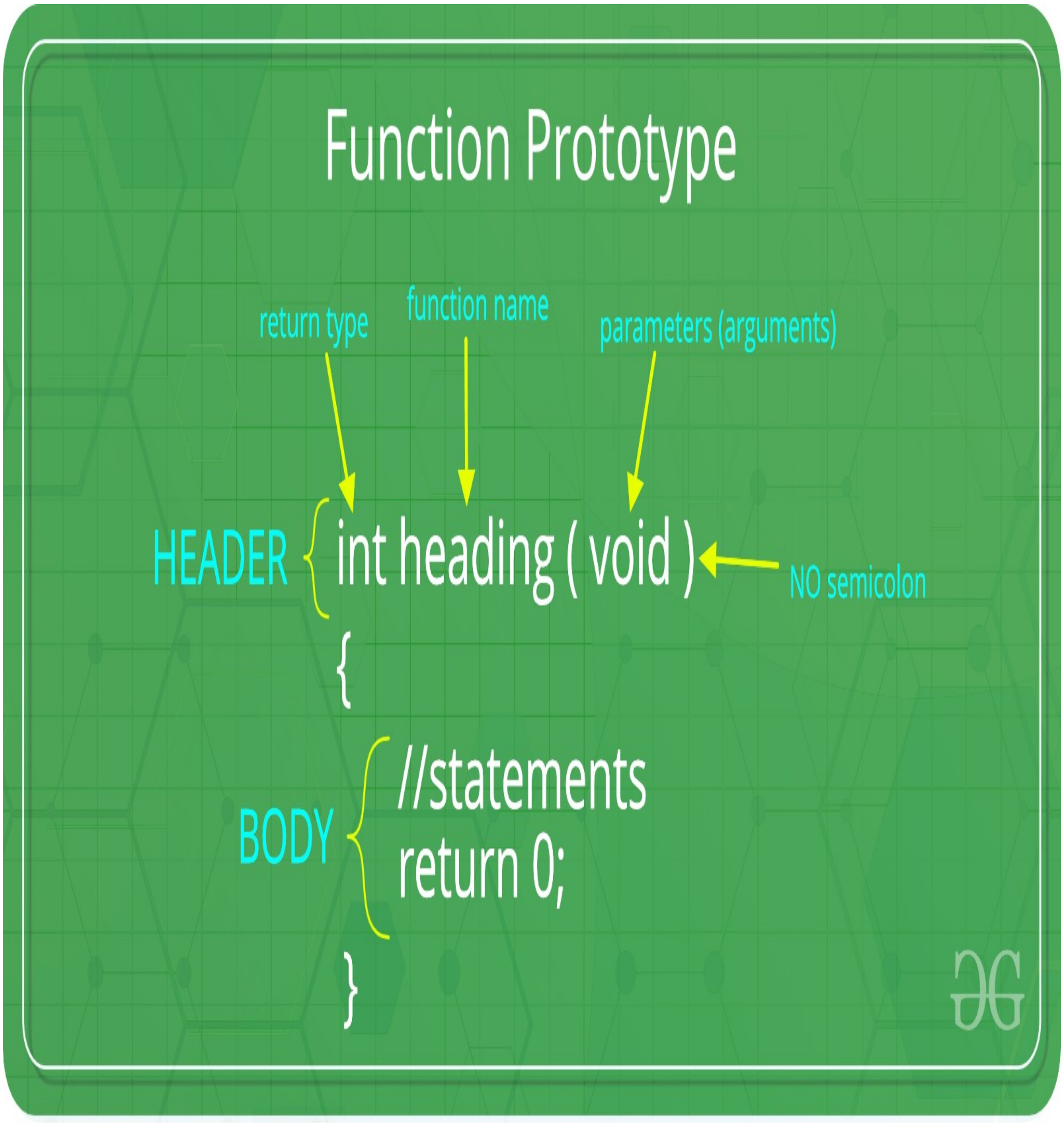
When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. For example −

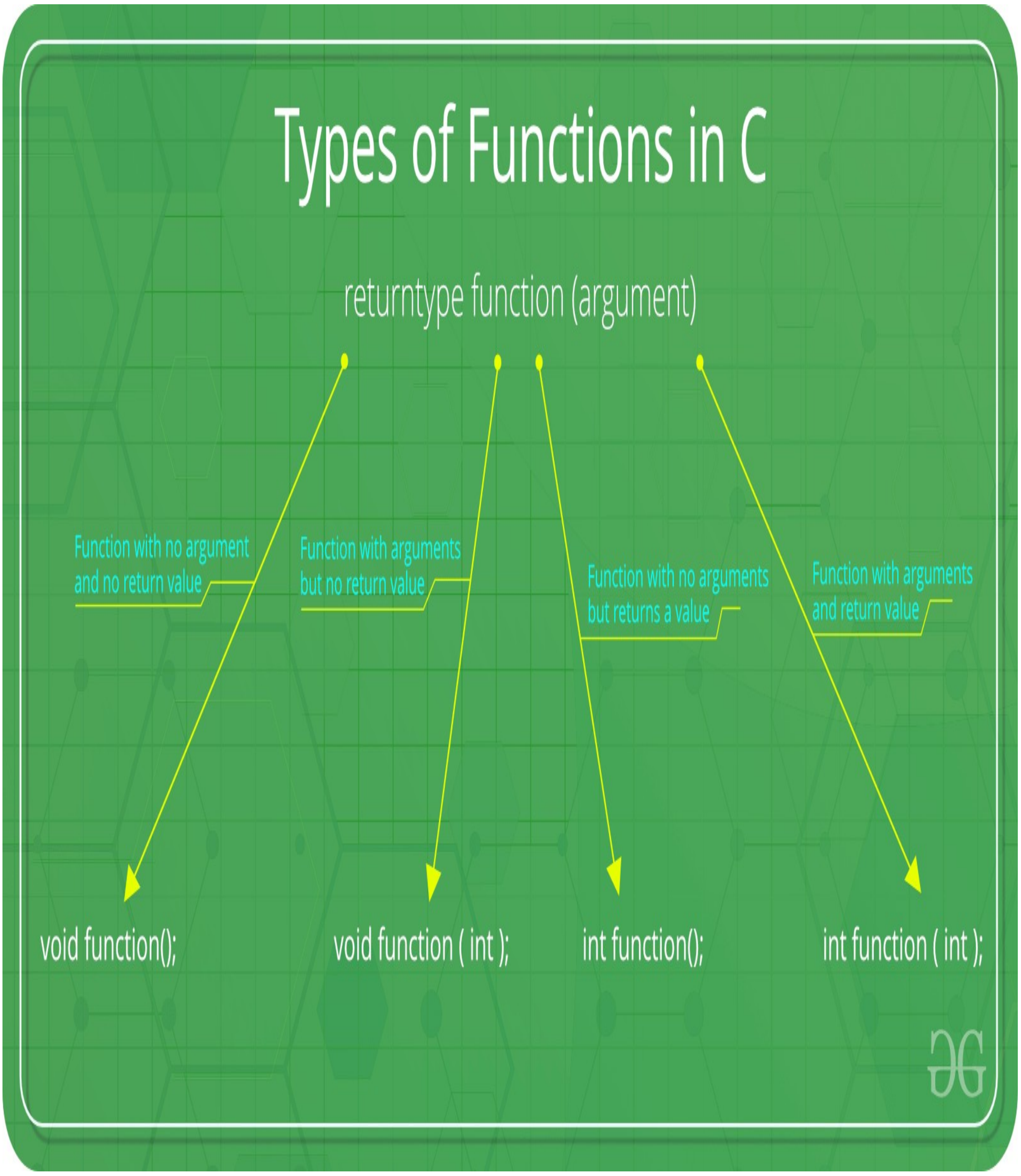
Enjoy functional programming !!!

III.12.2. Function arguments and return values

A function in C can be called either with arguments or without arguments. These function may or may not return values to the calling functions. All C functions can be called either with arguments or without arguments in a C program. Also, they may or may not return any values. Hence the function prototype of a function in C is as below:



There are following categories:


Types of Functions in C
returntype function (argument)
Function with no argument and no return value
Function with arguments but no return value
Function with no arguments but returns a value
Function with arguments and return value
void function();
void function (int);
int function();
int function (int);

Function with no argument and no return value :

When a function has no arguments, it does not receive any data from the calling function. Similarly when it does not return a value, the calling function does not receive any data from the called function.

Syntax :

Function declaration : void function();

Function call : function();

Function definition :

```
void function()
{
  statements;
}
```

C code for function with no arguments and no return value

```
#include <stdio.h>
// Function declaration
void value(void);
void main()
{
value();
}
// Function definition
void value(void)
{
int year = 1, period = 5, amount = 5000, inrate = 0.12;
float sum;
sum = amount;
while (year <= period) {
sum = sum * (1 + inrate);
year = year + 1;
}
printf(" The total amount is %f:", sum);
}
```

Output:

The total amount is 5000.000000

Function with arguments but no return value : When a function has arguments, it receive any data from the calling function but it returns no values.

Syntax :

Function declaration : void function (int);

Function call : function(x);

Function definition:

```
void function( int x )
{
  statements;
}
```

C code for function with argument but no return value

```
#include <stdio.h>

void function(int, int[], char[]);
int main()
{
int a = 20;
int ar[5] = { 10, 20, 30, 40, 50 };
char str[30] = "geeksforgeeks";
myFunction(a, ????? );
return 0;
}

void myFunction(int a, )
{

}
printf("\nvalue of str is %s\n", ?????????????);
}
```

Function with no arguments but returns a value : There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function. A example for this is getchar function it has no parameters but it returns an integer an integer type data that represents a character.

Syntax :

Function declaration : int function();

Function call : function();

Function definition :

```
int function()
{
    statements;
     return x;
 }
```

C code for function with no arguments but have return value

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

int sum();
int main()
{
int num;
num = sum();
printf("\nSum of two given values = %d", num);
return 0;
}

int sum()
{
int a = 50, b = 80, sum;
sum = sqrt(a) + sqrt(b);
return sum;
}
```

Output:

```
Sum of two given values = 16
```

Function with arguments and return value

Syntax :

Function declaration : int function (int);

Function call : function(x);

Function definition:

```
int function( int x )
{
  statements;
  return x;
}
```

C code for function with arguments and with return value

```
#include <stdio.h>
#include <string.h>
int  myFunction(int a);
int main()
{
int result= myFunction(10);

printf("value of is %d\n", result);
}
return 0;
}

int myFunction(int a)
{
int i=0;
a=a+i;
return a;
}
```

Output:

```
value of a is 40
```

III.13. RECURSION

A function that calls itself is known as a **recursive** function. And, this technique is known as **recursion**.

Recursion construction:

```
void recurse()
{
    ... .. ...
    recurse();// it is calling to its self no?
    ... .. ...
}
int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

The recursion continues until some condition is met to prevent it from calling its selves again.

To prevent infinite recursion, if...else statement (or similar approach) can be used where one branch makes the recursive call and other doesn't.

Example: Sum of Natural Numbers Using Recursion

```
#include <stdio.h>
int sum(int n);

int main()
{
    int number, result;
    printf("Enter a positive integer: ");
    scanf("%d", &number);

    result = sum(number);

    printf("sum = %d", result);
    return 0;
}

int sum(int num)
{
    if (num!=0)
        return num + sum(num-1); // sum() function calling to itself
    else
        return num;
}
```

Output

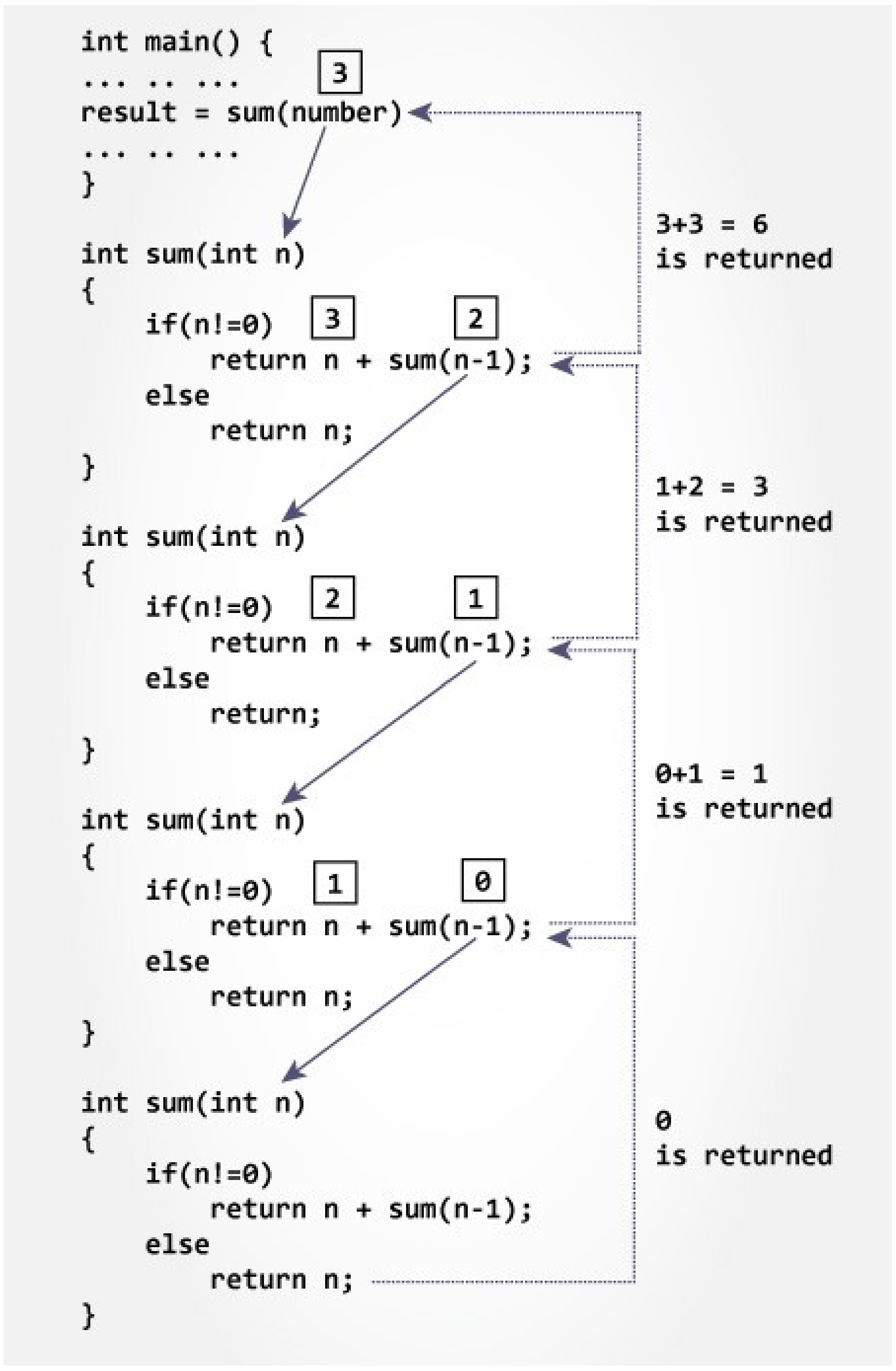
```
Enter a positive integer:3
sum = 6
```

it happened three recursive calls yielding in **3+2+1** ie: **6**

Initially, the sum() is called from the main() function with *number* passed as an argument.

Suppose, the value of ***num*** is **3** initially. During next function call, **2** is passed to the sum() function. This process continues until ***num*** is equal to **0**.

When ***num*** is equal to **0**, the if condition fails and the else part is executed returning the sum of integers to the main() function.



```
int main() {
... .. ...
result = sum(number)
... .. ...
}

int sum(int n)
{
    if(n!=0)
        return n + sum(n-1);
    else
        return n;
}

int sum(int n)
{
    if(n!=0)
        return n + sum(n-1);
    else
        return;
}

int sum(int n)
{
    if(n!=0)
        return n + sum(n-1);
    else
        return n;
}

int sum(int n)
{
    if(n!=0)
        return n + sum(n-1);
    else
        return n;
}
```

Advantages and Disadvantages of Recursion

Recursion makes program elegant and more readable. However, if performance is vital then, use **loops** instead as recursion is usually much slower.

Note that, every recursion can be modeled into a loop.

III.13.1. Recursion Vs Iteration?

Need performance, use loops, however, code might look ugly and hard to read sometimes. Need more elegant and readable code, use **recursion**, however, you are sacrificing some performance. Now you know both its up to you to let us know how a good programming analyst you are.

C Program to Print the Factorial of a given Number

This is a C Program to print the factorial of a given number.

Problem Description:

This C Program prints the factorial of a given number.

Problem Solution:

A factorial is product of all the numbers from 1 to n, where n is the user specified number. This program find the product of all the number from 1 to the user specified number.

Program/Source Code

```
#include <stdio.h>
void main()
{
    int i, fact = 1, num;
    printf("Enter the number \n");
    scanf("%d", &num);
    if (num <= 0)
        fact = 1;
    else
    {
        for (i = 1; i <= num; i++)
        {
            fact = fact * i;
        }
    }   printf("Factorial of %d = %5d\n", num, fact);
}
```

Factorial Program Explanation

In this C program, we are reading the integer number using ‘num’ integer variable. A factorial is a product of all the numbers from 1 to n, where n is the user specified number.

If condition statement is used to check the value of ‘num’ variable is less than or equal to 0. If the condition is true then it will execute the statement and assign the value of ‘fact’ variable as one. Otherwise, if the condition is false then it will execute the else statement. Using for loop multiply all the numbers from 1 to n and display the factorial of a given number as output.

Factorial of a Number Using Recursion

```
#include <stdio.h>
long int multiplyNumbers(int n);

int main()
{
    int n;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    printf("Factorial of %d = %ld", n, multiplyNumbers(n));
    return 0;
}
long int multiplyNumbers(int n)
{
    if (n >= 1)
        return n*multiplyNumbers(n-1);
    else
        return 1;
}
```

III.14. PATTERNS PROGRAMMING

This the art of writing programs that generate various patterns of numbers and stars or of any symbols. Such programs involve usage of nested for loops (a for loop inside a for loop). A pattern of numbers, star or characters is a way of arranging these in some logical manner or they may form a sequence. Some of these patterns are triangles which have special importance in mathematics. Some patterns are symmetrical while others are not.

Pattern : 1

```
 * * * * *
 * * * * *
 * * * * *
 * * * * *
 * * * * *_
```

Program : 1

```
#include <stdio.h>
//#include <conio.h>
void main() {
 int i,j;
 //clrscr();
for (i=0; i<5; i++) {
for (j=0; j<5; j++) {
 printf(" * ");
 }
printf("\n");   }
 //getch();
}
```

Pattern : 2

```
 *
 * *
 * * *
 * * * *
 * * * * *_
```

Program : 2

```
#include <stdio.h>

void main() {
int i,j;

for (i=0; i<5; i++) {
 for (j=0; j<=i; j++) {
printf(" * ");
}
printf("\n");
}
}
```

Pattern : 3

```
        *
      * *
    * * *
  * * * *
 * * * * *_
```

Program : 3

```
#include <stdio.h>

void main() {
int i,j,k;
   for (i=1; i<=5; i++) {
          for (j=5; j>=i; j--) {
          printf(" ");
          }
for (k=1; k<=i; k++) {
```

```
         printf("*");
}
printf("\n");
}

}
```

Pattern : 4

```
 * * * * *
   * * * *
     * * *
       * *
         *_
```

Program : 4

```
#include <stdio.h>
void main() {
int i,j,k,samp=1;

for (i=5; i>=1; i--) {
         for (k=samp; k>=0; k--) {
         printf(" ");      // only 1 space
         }
         for (j=i; j>=1; j--) {
          printf("*");
         }
         samp = samp + 1;
         printf("\n");
 }

}
```

Pattern : 5

```
 * * * * *
 * * * *
 * * *
 * *
 *_
```

Program : 5

```
#include <stdio.h>
void main() {
int i,j;
for (i=5; i>=1; i--) {
        for (j=1; j<=i; j++) {
        printf(" * ");
        }
        printf("\n");
        }

}
```

Pattern : 6

```
    *
   * *
  * * *
 * * * *
* * * * *_
```

Program : 6

```
#include <stdio.h>

void main() {
int i,j,k,t=0;
for (i=1; i<=5; i++) {
        for (k=t; k<5; k++) {
        printf(" ");
}
for (j=0; j< i; j++) {
        printf(" * ");
        t = t + 1;
}
printf("\n");
```

```
 }

}
```

Pattern : 7

```
        *
      * *
    * * *
  * * * *
* * * * *
  * * * *
    * * *
      * *
        *_
```

Program : 7

```
#include <stdio.h>
void main() {
int i,j,k,samp=1;

 for (i=1; i<=5; i++) {
        for (k=samp; k<=5; k++) {
        printf(" ");
        }
        for (j=0; j< i; j++) {
        printf("*");
        }
 samp = samp + 1;
 printf("\n");
}
 samp = 1;
 for (i=4; i>=1; i--) {
        for (k=samp; k>=0; k--) {
               printf(" ");
        }
 for (j=i; j>=1; j--) {
```

```
          printf("*");
}
 samp = samp + 1;
 printf("\n");
 }
}
```

Pattern : 8

```
Enter number of rows: 5
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15_
```

Program : 8

```
#include <stdio.h>
void main() {
int rw, c, no=1 ,len;

printf("Enter number of rows: ");
 scanf("%d," &len);
for (rw=1; rw<=len; rw++) {
          printf("\n");
          for (c=1; c<=rw; c++) {
                    printf(" %2d ", no);
                    no++;
          }
}
}
```

Output : 9

```
Enter number of rows: 5

          0
        1 0 1
      2 1 0 1 2
    3 2 1 0 1 2 3
  4 3 2 1 0 1 2 3 4
5 4 3 2 1 0 1 2 3 4 5_
```

Program : 9

```
#include <stdio.h>
void main() {
 int no,i,y,x=35;

 printf("Enter number of rows: ");
 scanf("%d," &no);

 for (y=0;y<=no;y++) {
         goto(x,y+1);
         for (i=0-y; i<=y; i++) {
         printf(" %3d ", abs(i));
         x=x-3;
         }
 }
}
```

Output : 10

```
    1
   2 2
  3 3 3
 4 4 4 4
5 5 5 5 5
```

Program : 10

```
#include <stdio.h>
void main() {
int i, j=5, k, x;
for (i=1;i<=5;i++) {
         for (k=1;k<=j;k++) {
         printf(" ");
        }
        for (x=1;x<=i;x++) {
             printf("%d",i);
             printf(" ");
        }
  printf("\n");
  j=j-1;
}
}
```

III.15. FILES IN C

Why files are needed?

- When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.
- If you have to enter a large number of data, it will take a lot of time to enter them all.
  However, if you have a file containing all the data, you can easily access the contents of the file using few commands in C.
- You can easily move your data from one computer to another without any changes.

III.15.1. Types of Files

When dealing with files, there are two types of files you should know about:

1. **Text files**
2. **Binary files**

1. **Text files**

Text files are the normal **.txt** files that you can easily create using Notepad or any simple text editors.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.
They take minimum effort to maintain, are easily readable, and provide **least security** and takes bigger storage space.
2. **Binary files**
Binary files are mostly the .**bin** files in your computer.
Instead of storing data in plain text, they store it in the binary form (**0**'s and **1**'s).
They can hold higher amount of data, are not readable easily and provides a **better security** than text files.

III.15.2. File Operations

In C, you can perform four major operations on the file, either text or binary:

1. Creating a new file
2. Opening an existing file
3. Closing a file
4. Reading from and writing information to a file

III.15.3. Working with files

When working with files, you need to declare a pointer of type file. This declaration is needed for communication between the file and program.
FILE *fptr; // file pointer declaration

Opening a file - for creation and edit
Opening a file is performed using the library function in the **"stdio.h"** header file: **fopen()**.
The syntax for opening a file in standard I/O is:
ptr = **fopen**("**file to pen**","**mode**")
For Example:
- fopen("**C:\\cprogram\\newprogram.txt**","**w**"); // opening a text file in write mode

-fopen("**D:\\cprogram\\oldprogram.bin**","**rb**"); // opening a binary file in read mode

- Let's suppose the file newprogram.txt doesn't exist in the location C:\cprogram. The first function creates a new file named **newprogram.txt** and opens it for writing as per the mode '**w**'.
  The writing mode allows you to create and edit (**overwrite**) the contents of the file.
- Now let's suppose the second binary file **oldprogram.bin** exists in the locationD:\cprogram. The second function opens the existing file for reading in binary mode '**rb**'.
  The reading mode only allows you to read the file, you cannot write into the file.

III.15.4. Files Opening Modes in Standard I/O

File Mode	**Meaning of Mode**	**During Nonexistence of file**
r	Open for reading.	If the file does not exist, fopen() returns NULL.
rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
w	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
ab	Open for append in binary mode. i.e, Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and	If the file exists, its contents are

File Mode	Meaning of Mode	During Nonexistence of file
	writing in binary mode.	overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exists, it will be created.

III.15.5. Closing a File

The file (both text and binary) should be closed after reading/writing.
Closing a file is performed using library function **fclose**().
fclose(fptr); //fptr is the file pointer associated with file to be closed.

III.15.6. Reading and writing to a text file

For reading and writing to a text file, we use the functions **fprintf**() and **fscanf**().
They are just the file versions of **printf**() and **scanf**(). The only difference is that, **fprint** and **fscanf** expects a pointer to the structure **FILE**.

Example 1: Write to a text file using **fprintf**()

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

int main()
{
   int num;
   FILE *fptr;
   fptr = fopen("C:\\program.txt","w");

   if(fptr == NULL)
   {
      printf("Error!");
      exit(1);
   }

   printf("Enter num: ");
   scanf("%d",&num);

   fprintf(fptr,"%d",num);
   fclose(fptr);

   return 0;
}
```

This program takes a number from user and stores in the file program.txt. After you compile and run this program, you can see a text file program.txt created in C drive of your computer. When you open the file, you can see the integer you entered.

Example 2: Read from a text file using fscanf()

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
   int num;
   FILE *fptr;

   if ((fptr = fopen("C:\\program.txt","r")) == NULL){
       printf("Error! opening file");

       // Program exits if the file pointer returns NULL.
       exit(1);
   }

   fscanf(fptr,"%d", &num);

   printf("Value of n=%d", num);
```

```
  fclose(fptr);

  return 0;
}
```

This program reads the integer present in the program.txt file and prints it onto the screen.

If you successfully created the file from **Example 1**, running this program will get you the integer you entered.

Other functions like **fgetchar**(), **fputc**() etc. can be used in similar way.

III.15.7. Reading and writing to a binary file

Functions **fread**() and **fwrite**() are used for reading from and writing to a file on the disk respectively in case of binary files.

III.15.7.1. Writing to a binary file

To write into a binary file, you need to use the function **fwrite**(). The functions takes four arguments: Address of data to be written in disk, Size of data to be written in disk, number of such type of data and pointer to the file where you want to write.

```
fwrite(address_data,size_of_data,numbers_of_record_data,pointer_to_file);
```

Example 3: Write to a binary file using fwrite()

```
#include <stdio.h>
#include <stdlib.h>
struct threeNum
{
  int n1, n2, n3;
};
int main()
{
  int n;
```

```
   struct threeNum num;
   FILE *fptr;
   if ((fptr = fopen("C:\\program.bin","wb")) == NULL){
       printf("Error! opening file");
       // Program exits if the file pointer returns NULL.
       exit(1);
   }
   for(n = 1; n < 5; ++n)
   {
      num.n1 = n;
      num.n2 = 5*n;
      num.n3 = 5*n + 1;
      fwrite(&num, sizeof(struct threeNum), 1, fptr);
   }
   fclose(fptr);
   return 0;
}
```

In this program, you create a new file **program.bin** in the C drive if we are on windows, but on linux the directory will be any of you choice.

We declare a structure **threeNum** with three numbers - *n1, n2 and n3*, and define it in the main function as num.

Now, inside the for loop, we store the value into the file using **fwrite**().

The first parameter takes the address of *num* and the second parameter takes the size of the structure threeNum.

Since, we're only inserting one instance of *num*, the third parameter is 1. And, the last parameter *fptr points to the file we're storing the data.

Finally, we close the file.

III.15.7.2. Reading from a binary file

The function **fread**() also take 4 arguments similar to fwrite() function as above.

fread(**address_data**,**size_data**,**numbers_data**,**pointer_to_file**);

Example 4: Read from a binary file using fread()

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

struct threeNum
{
   int n1, n2, n3;
};

int main()
{
   int n;
   struct threeNum num;
   FILE *fptr;

   if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
       printf("Error! opening file");

       // Program exits if the file pointer returns NULL.
       exit(1);
   }

   for(n = 1; n < 5; ++n)
   {
      fread(&num, sizeof(struct threeNum), 1, fptr);
      printf("n1: %d\tn2: %d\tn3: %d", num.n1, num.n2, num.n3);
   }
   fclose(fptr);

   return 0;
}
```

In this program, you read the same file program.bin and loop through the records one by one.

In simple terms, you read one threeNum record of threeNum size from the file pointed by **fptr* into the structure *num*.

You'll get the same records you inserted in Example 3.

III.15.8 Finding/Getting data using fseek()

If you have many records inside a file and need to access a record at a specific position, you need to loop through all the records before it to get the record. This will waste a lot of memory and operation time. An easier way to get to the required data can be achieved using **fseek**().
As the name suggests, **fseek**() seeks the cursor to the given record in the file.

III.15.8.1. Syntax of fseek()

fseek(FILE * stream, long int offset, int whence)
The first parameter stream is the pointer to the file. The second parameter is the position of the record to be found, and the third parameter specifies the location where the offset starts.

III.15.8.2. Different whence in fseek

Whence	Meaning
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_END	Starts the offset from the end of the file.
SEEK_CUR	Starts the offset from the current location of the cursor in the file.

Example 5: fseek()

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

struct threeNum
{
  int n1, n2, n3;
};

int main()
{
  int n;
  struct threeNum num;
  FILE *fptr;

  if ((fptr = fopen("C:\\program.bin","rb")) == NULL){
      printf("Error! opening file");

      // Program exits if the file pointer returns NULL.
      exit(1);
  }

  // Moves the cursor to the end of the file
  fseek(fptr, -sizeof(struct threeNum), SEEK_END);

  for(n = 1; n < 5; ++n)
  {
     fread(&num, sizeof(struct threeNum), 1, fptr);
     printf("n1: %d\tn2: %d\tn3: %d\n", num.n1, num.n2, num.n3);
     fseek(fptr, -2*sizeof(struct threeNum), SEEK_CUR);
  }
  fclose(fptr);

  return 0;
}
```

This program will start reading the records from the file program.bin in the reverse order (last to first) and prints it.

III.15.8.3. Examples:

1. Write a C program to read name and marks of n number of students from user and store them in a file.

```
#include <stdio.h>
int main()
{
   char name[50];
   int marks, i, num;
   printf("Enter number of students: ");
   scanf("%d", &num);
   FILE *fptr;
   fptr = (fopen("C:\\student.txt", "w"));
   if(fptr == NULL)
   {
       printf("Error!");
       exit(1);
   }
   for(i = 0; i < num; ++i)
   {
      printf("For student%d\nEnter name: ", i+1);
      scanf("%s", name);

      printf("Enter marks: ");
      scanf("%d", &marks);
      fprintf(fptr,"\nName: %s \nMarks=%d \n", name, marks);
   }
   fclose(fptr);
   return 0;
}
```

2. Write a C program to write all the members of an array of structures to a file using fwrite(). Read the array from the file and display on the screen.

```
#include <stdio.h>
struct student
{
   char name[50];
   int height;
};
int main(){
    struct student stud1[5], stud2[5];
    FILE *fptr;
    int i;

    fptr = fopen("file.txt","wb");
    for(i = 0; i < 5; ++i)
    {
        fflush(stdin);
        printf("Enter name: ");
        gets(stud1[i].name);

        printf("Enter height: ");
        scanf("%d", &stud1[i].height);
    }

    fwrite(stud1, sizeof(stud1), 1, fptr);
    fclose(fptr);

    fptr = fopen("file.txt", "rb");
    fread(stud2, sizeof(stud2), 1, fptr);
    for(i = 0; i < 5; ++i)
    {
        printf("Name: %s\nHeight: %d", stud2[i].name, stud2[i].height);
    }
    fclose(fptr);
}
```

III.16. TYPE CASTING IN

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a long value into a simple integer then you can typecast long to int. You can convert values from one type to another explicitly using the cast operator. There are two types of type casting in c language that are Implicit conversions and Explicit Conversions. In this article, we also learn about the difference between type casting and type conversions.

It is best practice to convert lower data type to higher data type to avoid data loss Data will be truncated when the higher data type is converted to lower. For example, if a float is converted to int, data which is present after the decimal point will be lost.

There are two types of type casting in c language.

	Types of type casting in C Programming
1	Implicit Conversion
2	Explicit Conversion

III.16.1. Implicit conversion

Implicit conversions do not require any operator for converted. They are automatically performed when a value is copied to a compatible type in the program.

Example :

```
#include<stdio.h>
void main(){
int i=20;
double p;
p=i; // implicit conversion
printf("implicit value is %d",p);
}
```

Here, the value of a has been promoted from int to double and we have not had to specify any type-casting operator. This is known as a standard conversion.

III.16. 2. Explicit conversion

Many conversions, especially those that imply a different interpretation of the value, require an **explicit conversion**.

They are not automatically performed when a value is copied to a compatible type in the program.

```
#include<stdio.h>
void main(){
  int i=20;
  short p;
 p = (short) i; // Explicit conversion
 printf("Explicit value is %d",p);
}
```

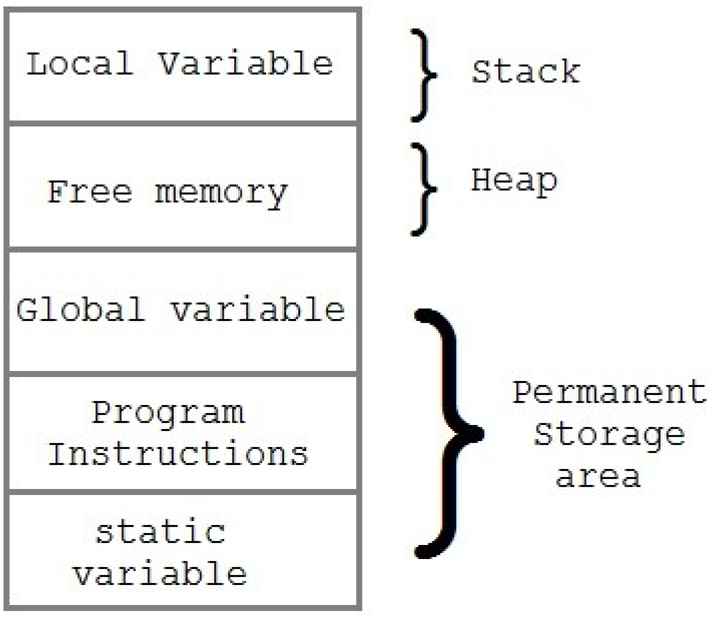
III.17. DYNAMIC MEMORY ALLOCATION IN C

The process of allocating memory at run time is known as **dynamic memory allocation**. Library routines known as **memory management functions** are used for allocating and freeing memory during execution of a program. These functions are defined in **stdlib.h** header file.

Function	**Description**
malloc()	allocates requested size of bytes and returns a void pointer pointing to the first byte of the allocated space
calloc()	allocates space for an array of elements, initialize them to zero and then returns a void pointer to the memory
free	releases previously allocated memory
realloc	modify the size of previously allocated space

Global variables, **static** variables and **program** instructions get their memory in **permanent** storage area whereas **local** variables are stored in a memory area called **Stack**.

The memory space between these two region is known as **Heap** area. This region is used for dynamic memory allocation during execution of the program. The size of heap keep changing.



III.17.1. Allocating block of Memory

malloc() function is used for allocating block of memory at run time. This function reserves a block of memory of the given size and returns a **pointer** of **type void**. This means that we can assign it to any type of pointer using typecasting. If it fails to allocate enough space as specified, it returns a NULL pointer.

Syntax:

void* **malloc**(byte-size)

Time for an Example: **malloc**()

```
int *x;
x = (int*)malloc(50 * sizeof(int));     //memory space allocated to variable x
free(x);    //releases the memory allocated to variable x
```

calloc() is another memory allocation function that is used for allocating memory at run time. calloc() function is normally used for allocating memory to derived data types such as **arrays** and **structures**. If it fails to allocate enough space as specified, it returns a NULL pointer.

Syntax:

void * **calloc**(number of items, element-size)

Time for an Example: calloc()

```
struct employee
{
    char *name;
    int salary;
};
typedef struct employee emp;
emp *e1;
e1 = (emp*)calloc(30,sizeof(emp));  // Type casting
```

realloc() changes memory size that is already allocated dynamically to a variable.

Syntax:

void* **realloc**(pointer, new-size)

Time for an Example: realloc()

```
int *x;
x = (int*)malloc(50 * sizeof(int));
x = (int*)realloc(x,100);   //allocated a new memory to variable x
```

Diffrence between malloc() and calloc()

calloc()	**malloc**()
calloc() initializes the allocated memory with 0 value.	malloc() initializes the allocated memory with garbage values.
Number of arguments is 2	Number of argument is 1
Syntax : (cast_type *)calloc(blocks , size_of_block);	Syntax : (cast_type *)malloc(Size_in_bytes);

Program to represent Dynamic Memory Allocation(using calloc())

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

int main()
{
   int i, n;
   int *element;

   printf("Enter total number of elements: ");
   scanf("%d", &n);

   /*
      returns a void pointer(which is type-casted to int*)
      pointing to the first block of the allocated space
   */
   element = (int*) calloc(n,sizeof(int));
```

```
    /*
        If it fails to allocate enough space as specified,
        it returns a NULL pointer.
    */
    if(element == NULL)
    {
        printf("Error.Not enough space available");
        exit(0);
    }

    for(i = 0; i < n; i++)
    {
        /*
            storing elements from the user
            in the allocated space
        */
       scanf("%d", element+i);
    }
    for(i = 1; i < n; i++)
    {
       if(*element > *(element+i))
       {
            *element = *(element+i);
        }
    }
    printf("Smallest element is %d", *element);
    return 0;
}
```

III.18. SOCKET PROGRAMMING IN C

We now all have a basic knowledge of network such as understanding what is an **IP** address(v4 and v6) right , **TCP**, **UDP** and **HTTP**(s) as network protocols, for sure this is enough for us to look ahead better things that happens in programming: **SOCKET PROGRAMMING** ie: networked applications that runs at OS core level and or at chip level!!!

Before we start our couse, keep in mind that this concept works well on **Linux OS** environment than it will in windows, but its OK for our windows guys we will evolve together, that's cool I guess.

Seeking time usage maximization our leaning will focus on socket programming using **TCP/IP** because it is the most used internet protocol involved in programming, but researchers will go father and do **UDP**. UDP; a right weight data transfer protocol, it is connection-less.

TCP (Transmission control protocol)

A **TCP** is a connection-oriented communication protocal. It is an intermediate layer of the application layer and internet protocol layer in **OSI** and **TCP/IP** models . **TCP** is designed to send the data packets over the network. It ensures that data is delivered to the correct destination.

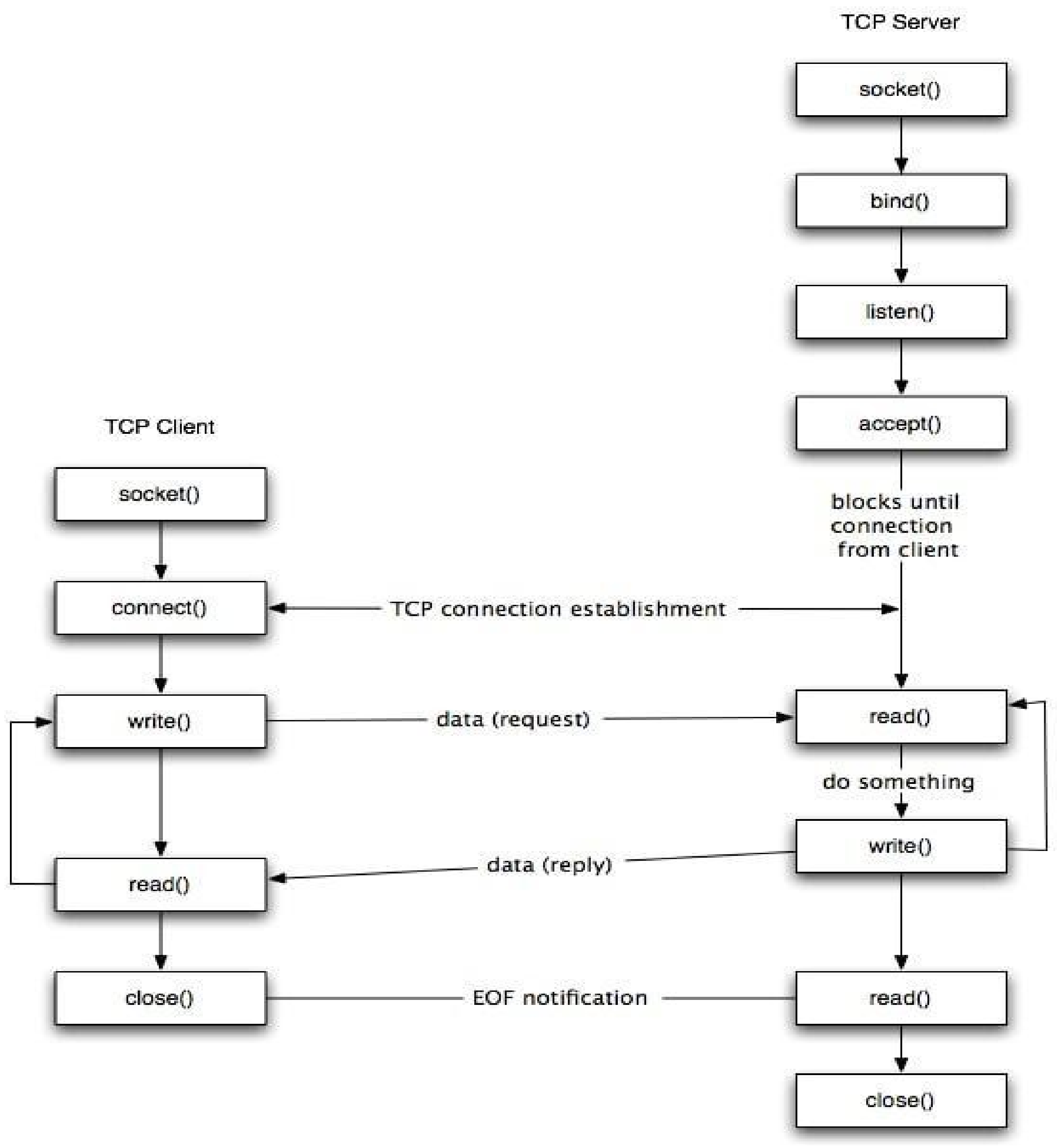
TCP creates a connection between the source and destination node before transmitting the data and keep the connection alive until the communication is active.

In **TCP** before sending the data it breaks the large data into smaller packets and cares the integrity of the data at the time of reassembling at the destination node. Major Internet applications such as the World Wide Web, email, remote administration, and file transfer rely on **TCP**.

TCP also offers the facility of re-transmission, when a **TCP** client sends data to the server, it requires an acknowledgment in return. If an acknowledgment is not received, after a certain amount of time transmitted data will be the lost and **TCP** automatically re-transmits the data.

The communication over the network in **TCP/IP** model takes place in form of a **client-server** architecture. ie, the client begins the communication and establish a connection with a server which was **listening** from a certain **Port**.

Client-server communication Sequential Diagram(Flow chart diagram)



Steps in algorithm to create a client using TCP/IP API

- Create a socket using the **socket**() function in c.
- Initialize the socket address structure as per the server and connect the socket to the address of the server using the **connect()**;
- Receive and send the data using the **recv**() and **send**() functions. Close the connection by calling the **close**() function.

Steps in algorithm to create a server using TCP/IP API

- Create a socket using the **socket()** function in c.
- Initialize the socket address structure and bind the socket to an address using the **bind()** function.
- Listen for connections with the **listen()** function.

- Accept a connection with the **accept()** function system call. This call typically blocks until a client connects to the server.
- Receive and send data by using the **recv()** and **send()** function in c.
- Close the connection by using the **close()** function.

Pseudo code to create a server using TCP/IP API

```
#include<stdio.h>

#include<string.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include<unistd.h>

int SocketCreate(void)
{

int hSocket;
printf("Create the socket\n");

hSocket = socket(AF_INET, SOCK_STREAM, 0);
return hSocket;
}

int BindCreatedSocket(int hSocket)
{
int iRetval=-1;
int ClientPort = 90190;
struct sockaddr_in remote={0};

remote.sin_family = AF_INET; /* Internet address family */
remote.sin_addr.s_addr = htonl(INADDR_ANY); /* Any incoming interface */
remote.sin_port = htons(ClientPort); /* Local port */
iRetval = bind(hSocket,(struct sockaddr *)&remote,sizeof(remote));

 return iRetval;
}

int main(int argc , char *argv[])
{
int socket_desc , sock , clientLen , read_size;
struct sockaddr_in server , client;
char client_message[200]={0};
char message[100] = {0};
const char *pMessage = "hello aticleworld.com";

//Create socket
socket_desc = SocketCreate();
if (socket_desc == -1)
{
printf("Could not create socket");
```

```
return 1;
}
printf("Socket created\n");

//Bind
if( BindCreatedSocket(socket_desc) < 0)
{
//print the error message
perror("bind failed.");
return 1;
}
printf("bind done\n");

//Listen
listen(socket_desc , 3);

//Accept and incoming connection

while(1)
{

printf("Waiting for incoming connections...\n");
clientLen = sizeof(struct sockaddr_in);

//accept connection from an incoming client
sock = accept(socket_desc, (struct sockaddr *)&client, (socklen_t*)&clientLen);
if (sock < 0)
{
perror("accept failed");
return 1;
}
printf("Connection accepted\n");

memset(client_message, '\0', sizeof client_message);
memset(message, '\0', sizeof message);
//Receive a reply from the client
if( recv(sock , client_message , 200 , 0) < 0)
{
printf("recv failed");
break;
}

printf("Client reply : %s\n",client_message);

if(strcmp(pMessage,client_message)==0)
{

strcpy(message,"Hi there !");
}
else
{
 strcpy(message,"Invalid Message !");

}

// Send some data
if( send(sock , message , strlen(message) , 0) < 0)
{
printf("Send failed");
return 1;
}

close(sock);
sleep(1);
}
 return 0;
}
```

Pseudo code to create a client using TCP/IP API

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<sys/socket.h>
#include<arpa/inet.h>
#include<unistd.h>
//Create a Socket for server communication
short SocketCreate(void)
{

 short hSocket;
 printf("Create the socket\n");
 hSocket = socket(AF_INET, SOCK_STREAM, 0);
return hSocket;
}

//try to connect with server
int SocketConnect(int hSocket)
{

        int iRetval=-1;
         int ServerPort = 90190;
         struct sockaddr_in remote={0};

         remote.sin_addr.s_addr = inet_addr("127.0.0.1"); //Local Host
         remote.sin_family = AF_INET;
         remote.sin_port = htons(ServerPort);
        iRetval = connect(hSocket , (struct sockaddr *)&remote , sizeof(struct
        sockaddr_in));
alpha1
        return iRetval;
}

// Send the data to the server and set the timeout of 20 seconds
int SocketSend(int hSocket,char* Rqst,short lenRqst)

{

        int shortRetval = -1;
        struct timeval tv;
        tv.tv_sec = 20;  // 20 Secs Timeout  for the data to be sent
       tv.tv_usec = 0;

  if(setsockopt(hSocket, SOL_SOCKET, SO_SNDTIMEO, (char *)&tv,sizeof(tv)) <
0)
{
    printf("Time Out\n");
   return -1;
 }
  shortRetval = send(hSocket , Rqst , lenRqst , 0);

  return shortRetval;
}

//receive the data from the server
int SocketReceive(int hSocket,char* Rsp,short RvcSize)
{

int shortRetval = -1;
struct timeval tv;
tv.tv_sec = 20; /* 20 Secs Timeout */
tv.tv_usec = 0;

if(setsockopt(hSocket, SOL_SOCKET, SO_RCVTIMEO, (char *)&tv,sizeof(tv)) < 0)
{
printf("Time Out\n");
return -1;

}
shortRetval = recv(hSocket, Rsp , RvcSize , 0);

printf("Response %s\n",Rsp);

return shortRetval;
}


//main driver program
int main()
```

```
{
int hSocket, read_size;
struct sockaddr_in server;
char SendToServer[100] = {0};
char server_reply[200] = {0};

//Create socket
hSocket = SocketCreate();
if(hSocket == -1)
{
 printf("Could not create socket\n");
 return 1;
}

printf("Socket is created\n");

//Connect to remote server
if (SocketConnect(hSocket) < 0)
{
perror("connect failed.\n");
return 1;
}

printf("Sucessfully conected with server\n");

printf("Enter the Message: ");
gets(SendToServer);

//Send data to the server
SocketSend(hSocket , SendToServer , strlen(SendToServer));


//Received the data from the server
read_size = SocketReceive(hSocket , server_reply , 200);

printf("Server Response : %s\n\n",server_reply);

close(hSocket);
shutdown(hSocket,0);
shutdown(hSocket,1);
shutdown(hSocket,2);
 return 0;
}
```

Explanation on socket programming

dddddddddddd

Create a socket

This first thing to do is create a socket. The **socket** function does this.
Here is a code sample :

```
#include<stdio.h>
#include<sys/socket.h>

int main()
{
	int socket_desc;
	socket_desc = socket(AF_INET , SOCK_STREAM , 0);

	if (socket_desc == -1)
	{
		printf("Could not create socket");
	}

	return 0;
}
```

Function socket() creates a socket and returns a descriptor which can be used in other functions. The above code will create a socket with following properties

```
Address Family - AF_INET (this is IP version 4)
Type - SOCK_STREAM (this means connection oriented TCP protocol)
Protocol - 0 [ or IPPROTO_IP This is IP protocol]
```

Next we shall try to connect to some server using this socket.
We can connect to www.google.com

Note

Apart from SOCK_STREAM type of sockets there is another type called SOCK_DGRAM which indicates the **UDP protocol**. This type of socket is non-connection socket. In this tutorial we shall stick to SOCK_STREAM or TCP sockets.

Connect socket to a server

We connect to a remote server on a certain port number. So we need 2 things, **ip address** and **port number** to connect to.

To connect to a remote server we need to do a couple of things. First is to create a sockaddr_in structure with proper values.

```
struct sockaddr_in server;
```

Have a look at the structure

```
// IPv4 AF_INET sockets:
struct sockaddr_in {
    short            sin_family;   // e.g. AF_INET, AF_INET6
    unsigned short   sin_port;     // e.g. htons(3490)
    struct in_addr   sin_addr;     // see struct in_addr, below
    char             sin_zero[8];  // zero this if you want to
};

struct in_addr {
    unsigned long s_addr;          // load with inet_pton()
};

struct sockaddr {
    unsigned short    sa_family;    // address family, AF_xxx
    char              sa_data[14];  // 14 bytes of protocol address
};
```

The sockaddr_in has a member called sin_addr of type in_addr which has a s_addr which is nothing but a long. It contains the IP address in long format.

Function inet_addr is a very handy function to convert an IP address to a long format. This is how you do it :

```
server.sin_addr.s_addr = inet_addr("74.125.235.20");
```

So you need to know the IP address of the remote server you are connecting to. Here we used the ip address of google.com as a sample. A little later on we shall see how to find out the ip address of a given domain name.

The last thing needed is the **connect** function. It needs a socket and a sockaddr structure to connect to. Here is a code sample.

```
#include<stdio.h>
#include<sys/socket.h>
#include<arpa/inet.h>   //inet_addr

int main(int argc , char *argv[])
{
        int socket_desc;
        struct sockaddr_in server;

        //Create socket
        socket_desc = socket(AF_INET , SOCK_STREAM , 0);
        if (socket_desc == -1)
        {
                printf("Could not create socket");
        }

        server.sin_addr.s_addr = inet_addr("74.125.235.20");
        server.sin_family = AF_INET;
        server.sin_port = htons( 80 );

        //Connect to remote server
        if (connect(socket_desc , (struct sockaddr *)&server , sizeof(server)) < 0)
        {
                puts("connect error");
                return 1;
        }

        puts("Connected");
        return 0;
}
```

It cannot be any simpler. It creates a socket and then connects. If you run the program it should show Connected.
Try connecting to a port different from port 80 and you should not be able to connect which indicates that the port is not open for connection.

OK , so we are now connected. Lets do the next thing , sending some data to the remote server.

Connections are present only in tcp sockets

The concept of "connections" apply to S**OCK_STREAM/TCP** type of sockets. Connection means a reliable "stream" of data such that there can be multiple such streams each having communication of its own. Think of this as a pipe which is not interfered by other data.

Other sockets like **UDP , ICMP , ARP** dont have a concept of "connection". These are **non-connection** based communication. Which means you keep sending or receiving packets from anybody and everybody.

Send data over socket

Function send will simply send data. It needs the **socket descriptor** , the data to send and its size.
Here is a very simple example of sending some data to google.com ip :

```
#include<stdio.h>
#include<string.h>      //strlen
#include<sys/socket.h>
#include<arpa/inet.h>   //inet_addr

int main(int argc , char *argv[])
{
	int socket_desc;
	struct sockaddr_in server;
	char *message;
	
	//Create socket
	socket_desc = socket(AF_INET , SOCK_STREAM , 0);
	if (socket_desc == -1)
	{
		printf("Could not create socket");
	}
	
	server.sin_addr.s_addr = inet_addr("74.125.235.20");
	server.sin_family = AF_INET;
	server.sin_port = htons( 80 );

	//Connect to remote server
	if (connect(socket_desc , (struct sockaddr *)&server , sizeof(server)) < 0)
	{
		puts("connect error");
		return 1;
	}
	
	puts("Connected\n");
	
	//Send some data
	message = "GET / HTTP/1.1\r\n\r\n";
	if( send(socket_desc , message , strlen(message) , 0) < 0)
	{
		puts("Send failed");
		return 1;
	}
	puts("Data Send\n");

	return 0;
}
```

In the above example , we first **connect** to an ip address and then send the string message "GET / HTTP/1.1\r\n\r\n" to it.
The message is actually a http command to fetch the mainpage of a website.

Now that we have send some data , its time to receive a reply from the server. So lets do it.

Note

When **sending** data to a socket you are basically **writing** data to that socket. This is similar to writing data to a file. Hence you can also use the **write** function to send data to a socket. Later in this tutorial we shall use **write** function to send data.

Receive data on socket

Function **recv** is used to receive data on a socket. In the following example we shall send the same message as the last example and receive a reply from the server.

```
#include<stdio.h>
#include<string.h>	//strlen
#include<sys/socket.h>
#include<arpa/inet.h>	//inet_addr

int main(int argc , char *argv[])
{
	int socket_desc;
	struct sockaddr_in server;
	char *message , server_reply[2000];

	//Create socket
	socket_desc = socket(AF_INET , SOCK_STREAM , 0);
	if (socket_desc == -1)
	{
		printf("Could not create socket");
	}

	server.sin_addr.s_addr = inet_addr("74.125.235.20");
	server.sin_family = AF_INET;
	server.sin_port = htons( 80 );

	//Connect to remote server
	if (connect(socket_desc , (struct sockaddr *)&server , sizeof(server)) < 0)
	{
		puts("connect error");
		return 1;
	}
	puts("Connected\n");

	//Send some data
	message = "GET / HTTP/1.1\r\n\r\n";
	if( send(socket_desc , message , strlen(message) , 0) < 0)
	{
		puts("Send failed");
		return 1;
	}
	puts("Data Send\n");

	//Receive a reply from the server
	if( recv(socket_desc, server_reply , 2000 , 0) < 0)
	{
		puts("recv failed");
	}
	puts("Reply received\n");
	puts(server_reply);

	return 0;
}
```

Here is the output of the above code :

```
Connected

Data Send

Reply received

HTTP/1.1 302 Found
Location: http://www.google.co.in/
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Set-Cookie:
PREF=ID=0edd21a16f0db219:FF=0:TM=1324644706:LM=1324644706:S=z6hDC9cZfGEowv_o;
expires=Sun, 22-Dec-2013 12:51:46 GMT; path=/; domain=.google.com
Date: Fri, 23 Dec 2011 12:51:46 GMT
Server: gws
Content-Length: 221
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.co.in/">here</A>.
</BODY></HTML>
```

We can see what reply was send by the server. It looks something like Html, well IT IS html. Google.com replied with the content of the page we requested. Quite simple!

Note

When receiving data on a socket , we are basically **reading** the data on the socket. This is similar to reading data from a **file**. So we can also use the **read** function to read data on a socket. For example :

```
read(socket_desc, server_reply , 2000);
```

Now that we have received our reply, its time to close the socket.

Close socket

Function **close** is used to close the socket. Need to include the unistd.h header file for this.

```
close(socket_desc);
```

Thats it.

Your **web browser** also does the same thing when you open **www.google.com**
This kind of socket activity represents a **socket client**. A client is an application that connects to a remote system to **fetch or retrieve data**.

The other kind of socket application is called a **socket server**. A server is a system that uses sockets to receive incoming connections and provide them with data. It is just the opposite of Client. So **www.google.com** is a **server** and your web browser is a **client**. Or more technically www.google.com is a **HTTP Server** and your web browser is an **HTTP client**.

Now its time to do some server tasks using sockets. But before we move ahead there are a few side topics that should be covered just in case you need them, or you have missed my previous explanations.

Get ip address of hostname

When connecting to a remote host , it is necessary to have its IP address. Function **gethostbyname** is used for this purpose.

It takes the domain name as the parameter and returns a structure of type **hostent**. This structure has the **IP** information. It is present in netdb.h. Lets have a look at this structure

```
/* Description of data base entry for a single host.  */
struct hostent
{
  char *h_name;                 /* Official name of host.  */
  char **h_aliases;             /* Alias list.  */
  int h_addrtype;               /* Host address type.  */
  int h_length;                 /* Length of address.  */
  char **h_addr_list;           /* List of addresses from name server.  */
};
```

The **h_addr_list** has the IP addresses. So now lets have some code to use them.

```
#include<stdio.h>; //printf
#include<string.h>; //strcpy
#include<sys/socket.h>;
#include<netdb.h>; //hostent
```

```
#include<arpa/inet.h>;

int main(int argc , char *argv[])
{
	char *hostname = ";www.google.com";;
	char ip[100];
	struct hostent *he;
	struct in_addr **addr_list;
	int i;

	if ( (he = gethostbyname( hostname ) ) == NULL)
	{
		//gethostbyname failed
		herror(";gethostbyname";);
		return 1;
	}

	//Cast the h_addr_list to in_addr , since h_addr_list also has the ip address in long format
only
	addr_list = (struct in_addr **) he->;h_addr_list;

	for(i = 0; addr_list[i] != NULL; i++)
	{
		//Return the first one;
		strcpy(ip , inet_ntoa(*addr_list[i]) );
	}

	printf(";%s resolved to : %s"; , hostname , ip);
	return 0;
}
```

Output of the code would look like :

```
www.google.com resolved to : 74.125.235.20
```

So the above code can be used to find the ip address of any domain name. Then the ip address can be used to make a connection using a socket.

Function inet_ntoa will convert an IP address ifrom **long** format to **dotted** format. This is just the opposite of inet_addr.

So far we have seen some **important structures** that are used. Lets revise them :

1. **sockaddr_in** - Connection information. Used by connect , send , recv etc.
2. **in_addr** - Ip address in long format
3. **sockaddr: T**he super structure for address definition
4. **hostent** - The ip addresses of a hostname. Used by gethostbyname

In the next part we shall look into creating **servers** using socket. **Servers** are the **opposite of clients**, that instead of connecting out to others, they **wait for incoming connections**.

Socket server

OK now onto server things. Socket servers operate in the following manner:

1. Open a socket
2. Bind to a address(and port).

3. Listen for incoming connections.
4. Accept connections
5. Read(**receive**) / write(**Send**)

We have already learnt how to open a socket. So the next thing would be to bind it.

Bind socket to a port

The **bind** function can be used to bind a socket to a particular "address and port" combination. It needs a **sockaddr**_in structure similar to connect function.

```
int socket_desc;
struct sockaddr_in server;

//Create socket
socket_desc = socket(AF_INET , SOCK_STREAM , 0);
if (socket_desc == -1)
{
	printf(";Could not create socket";);
}

//Prepare the sockaddr_in structure
server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = htons( 8888 );

//Bind
if( bind(socket_desc,(struct sockaddr *)&amp;server , sizeof(server)) < 0)
{
	puts(";bind failed";);
}
puts(";bind done";);
```

Now that **binding** is done, its time to make the socket **listen** to connections. We **bind** a socket to a particular IP address and a certain **port number**. By doing this we ensure that all incoming data which is directed towards this port number is received by this **application(our server)**.

This makes it obvious that **you cannot have 2 sockets bound to the same port.**

Listen for incoming connections on the socket

After binding a socket to a port the next thing we need to do is listen for connections. For this we need to put the socket in listening mode. Function listen is used to put the socket in listening mode. Just add the following line after bind.

```
//Listen
listen(socket_desc , 3);// this server can listen to 3 and only three connections ata same time
```

That’s all. Now comes the main part of accepting new connections.

Accept connection

The **accept** function is used for this. Here is the code

```
#include<stdio.h>;
#include<sys/socket.h>;
#include<arpa/inet.h>;     //inet_addr

int main(int argc , char *argv[])
{
	int socket_desc , new_socket , c;
	struct sockaddr_in server , client;

	//Create socket
	socket_desc = socket(AF_INET , SOCK_STREAM , 0);
	if (socket_desc == -1)
	{
		printf(";Could not create socket";);
	}

	//Prepare the sockaddr_in structure
	server.sin_family = AF_INET;
	server.sin_addr.s_addr = INADDR_ANY;
	server.sin_port = htons( 8888 );

	//Bind
	if( bind(socket_desc,(struct sockaddr *)&amp;server , sizeof(server)) < 0)
	{
		puts(";bind failed";);
	}
	puts(";bind done";);

	//Listen
	listen(socket_desc , 3);

	//Accept and incoming connection
	puts(";Waiting for incoming connections...";);
	c = sizeof(struct sockaddr_in);

	new_socket = accept(socket_desc, (struct sockaddr *)&amp;client, (socklen_t*)&amp;c);
	if (new_socket<0)
	{
		perror(";accept failed";);
	}

	puts(";Connection accepted";);

	return 0;
}
```

Program output

Run the program. It should show

```
bind done
Waiting for incoming connections...
```

So now this program is waiting for incoming connections on port 8888. Dont close this program , keep it running, ok. It is a server right ? **Servers never stop running !!!!!!**

Now a client can **connect** to our server on the port. We shall use the telnet client for testing this. Open a terminal and type

```
$ telnet localhost 8888  // this mean terminal client is connecting to the server
```

On the terminal you shall get :

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Connection closed by foreign host.
```

And the server output will show

```
bind done
Waiting for incoming connections...
Connection accepted
```

So we can see that the client connected to the server. Try the above process till you get it perfect, till you fell confident of doing it you self, it is not simple but it is doable.

Get the ip address of the connected client

You can get the ip address of client and the port of connection by using the **sockaddr_in** structure passed to accept function. It is very simple :

```
char *client_ip = inet_ntoa(client.sin_addr);
int client_port = ntohs(client.sin_port);
```

We accepted an incoming connection but closed it immediately. This was not very productive. There are lots of things that can be done after an incoming connection is established. After all the connection was established for the purpose of **communication**. So lets reply to the client.

We can simply use the **write** function to **write** something to the **socket** of the **incoming** connection and the **client** should **see** it.

Here is an example :

```
#include<stdio.h>;
#include<string.h>;       //strlen
#include<sys/socket.h>;
#include<arpa/inet.h>;    //inet_addr
#include<unistd.h>;       //write

int main(int argc , char *argv[])
{
	int socket_desc , new_socket , c;
	struct sockaddr_in server , client;
	char *message;

	//Create socket
	socket_desc = socket(AF_INET , SOCK_STREAM , 0);
	if (socket_desc == -1)
	{
		printf(";Could not create socket";);
	}
	
	//Prepare the sockaddr_in structure
	server.sin_family = AF_INET;
	server.sin_addr.s_addr = INADDR_ANY;
	server.sin_port = htons( 8888 );
	
	//Bind
	if( bind(socket_desc,(struct sockaddr *)&amp;server , sizeof(server)) < 0)
	{
		puts(";bind failed";);
		return 1;
	}
	puts(";bind done";);
	
	//Listen
	listen(socket_desc , 3);
	
	//Accept and incoming connection
	puts(";Waiting for incoming connections...";);
	c = sizeof(struct sockaddr_in);
	new_socket = accept(socket_desc, (struct sockaddr *)&amp;client, (socklen_t*)&amp;c);
	if (new_socket<0)
	{
		perror(";accept failed";);
		return 1;
	}
	
	puts(";Connection accepted";);
	
	//Reply to the client
	 message = ";Hello Client , I have received your connection. But I have to go 
now, bye\n";;
	write(new_socket , message , strlen(message));
	
	return 0;
}
```

Run the above code in 1 terminal. And connect to this server using telnet from another terminal and you should see this :

```
$ telnet localhost 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello Client , I have received your connection. But I have to go now, bye
Connection closed by foreign host.
```

So the client(**telnet**) received a reply from server.

We can see that the connection is closed immediately after that simply because the server program ends after accepting and sending reply. A server like www.google.com is always up to accept incoming connections.

It means that a server is supposed to be running all the time. Afterall its a server meant to serve. So we need to keep our server RUNNING non-stop. The simplest way to do this is to put the accept in a loop so that it can receive incoming connections all the time.

Live Server

So a live server will be alive for all time. Lets code this up :

```
#include<stdio.h>;
#include<string.h>;       //strlen
#include<sys/socket.h>;
#include<arpa/inet.h>;    //inet_addr
#include<unistd.h>;      //provide access to the POSIX hence for all UNIX // like operating
system API  to write read and close

int main(int argc , char *argv[])
{
	int socket_desc , new_socket , c;
	struct sockaddr_in server , client;
	char *message;

	//Create socket
	socket_desc = socket(AF_INET , SOCK_STREAM , 0);
	if (socket_desc == -1)
	{
		printf("Could not create socket";);
	}

	//Prepare the sockaddr_in structure
	server.sin_family = AF_INET;
	server.sin_addr.s_addr = INADDR_ANY;
	server.sin_port = htons( 8888 );

	//Bind
	if( bind(socket_desc,(struct sockaddr *)&amp;server , sizeof(server)) < 0)
	{
		puts(";bind failed";);
		return 1;
	}
	puts(";bind done";);

	//Listen
	listen(socket_desc , 3);

	//Accept and incoming connection
	puts(";Waiting for incoming connections...";);
	c = sizeof(struct sockaddr_in);
	while((new_socket = accept(socket_desc, (struct sockaddr *)&amp;client,
(socklen_t*)&amp;c)))
	{
		puts(";Connection accepted";);

		//Reply to the client
		message = ";Hello Client , I have received your connection. But I have to go now,
bye\n";;
		write(new_socket , message , strlen(message));
	}

	if (new_socket<0)
	{
		perror(";accept failed";);
		return 1;
	}

	return 0;
}
```

We haven't done a lot there. Just the **accept** was put in a **loop**.

Hahahah; this lmagic **lopp** will make our server to run **indefinitly**, hence a **live server**

Now run the program in **1** terminal , and open **3** other **terminals**. From each of the 3 **terminal** do a telnet to the server port.

Each of the telnet terminal would show :

```
$ telnet localhost 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello Client , I have received your connection. But I have to go now, bye
```

And the server terminal would show

```
bind done
Waiting for incoming connections...
Connection accepted
Connection accepted
Connection accepted
```

So now the server is running nonstop and the telnet terminals are also connected nonstop. Now close the server program.
All telnet terminals would show "Connection closed by foreign host."
Good so far. But still there is not effective communication between the server and the client.

The server program accepts connections in a loop and just send them a reply, after that it does nothing with them. Also it is not able to handle more than 1 connection at a time. So now its time to handle the connections , and handle multiple connections together.

Handle multiple socket connections with threads

To handle every connection we need a separate handling code to run along with the main server accepting connections.
One way to achieve this is using threads. The main server program accepts a connection and creates a new thread to handle communication for the connection, and then the server goes back to accept more connections.

On Linux threading can be done with the pthread (posix threads) library. It would be good to read some small tutorial about it if you dont know anything about it. However the usage is not very complicated.

We shall now use threads to create handlers for each connection the server accepts. Lets do it pal.

```
#include<stdio.h>;
#include<string.h>;      //strlen
#include<stdlib.h>;      //strlen
#include<sys/socket.h>;
#include<arpa/inet.h>;   //inet_addr
#include<unistd.h>;      //write
```

```
#include<pthread.h>; //for threading , link with lpthread

void *connection_handler(void *);

int main(int argc , char *argv[])
{
	int socket_desc , new_socket , c , *new_sock;
	struct sockaddr_in server , client;
	char *message;

	//Create socket
	socket_desc = socket(AF_INET , SOCK_STREAM , 0);
	if (socket_desc == -1)
	{
		printf(";Could not create socket";);
	}

	//Prepare the sockaddr_in structure
	server.sin_family = AF_INET;
	server.sin_addr.s_addr = INADDR_ANY;
	server.sin_port = htons( 8888 );

	//Bind
	if( bind(socket_desc,(struct sockaddr *)&amp;server , sizeof(server)) < 0)
	{
		puts(";bind failed";);
		return 1;
	}
	puts(";bind done";);

	//Listen
	listen(socket_desc , 3);

	//Accept and incoming connection
	puts(";Waiting for incoming connections...";);
	c = sizeof(struct sockaddr_in);
	while( (new_socket = accept(socket_desc, (struct sockaddr *)&amp;client,
(socklen_t*)&amp;c)) )
	{
		puts(";Connection accepted";);

		//Reply to the client
		message = ";Hello Client , I have received your connection. And now I will assign a
handler for you\n";;
		write(new_socket , message , strlen(message));

		pthread_t sniffer_thread;
		new_sock = malloc(1);
		*new_sock = new_socket;

		if( pthread_create( &amp;sniffer_thread , NULL ,  connection_handler , (void*)
new_sock) < 0)
		{
			perror(";could not create thread";);
			return 1;
		}

		//Now join the thread , so that we dont terminate before the thread
		//pthread_join( sniffer_thread , NULL);
		puts(";Handler assigned";);
	}
	if (new_socket<0)
	{
		perror(";accept failed";);
		return 1;
	}

	return 0;
}

/*
 * This will handle connection for each client
 * */
void *connection_handler(void *socket_desc)
{
	//Get the socket descriptor
	int sock = *(int*)socket_desc;

	char *message;

	//Send some messages to the client
	message = ";Greetings! I am your connection handler\n";;
	write(sock , message , strlen(message));

	message = ";Its my duty to communicate with you";;
	write(sock , message , strlen(message));

	//Free the socket pointer
	free(socket_desc);

	return 0;
}
```

Run the above server and open **3 terminals** like before. Now the server will create a thread for each **client connecting to it**. Intersting and advenced; I will come back to this after completing multhreading in **java** in the coming year, **do not be worried.**

The telnet terminals would show :

```
$ telnet localhost 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello Client , I have received your connection. And now I will assign a handler for you
Hello I am your connection handler
Its my duty to communicate with you
```

This one looks good , but the communication handler is also quite dumb. After the greeting it terminates. It should stay alive and keep communicating with the client.

One way to do this is by making the connection handler wait for some message from a client as long as the client is connected. If the client disconnects , the connection handler ends.

So the connection handler can be rewritten like this :

```
/*
```

```
 * This will handle connection for each client
 * */
void *connection_handler(void *socket_desc)
{
	//Get the socket descriptor
	int sock = *(int*)socket_desc;
	int read_size;
	char *message , client_message[2000];

	//Send some messages to the client
	message = ";Greetings! I am your connection handler\n";;
	write(sock , message , strlen(message));

	message = ";Now type something and i shall repeat what you type \n";;
	write(sock , message , strlen(message));

	//Receive a message from client
	while( (read_size = recv(sock , client_message , 2000 , 0)) >; 0 )
	{
		//Send the message back to client
		write(sock , client_message , strlen(client_message));
	}

	if(read_size == 0)
	{
		puts(";Client disconnected";);
		fflush(stdout);
	}
	else if(read_size == -1)
	{
		perror(";recv failed";);
	}

	//Free the socket pointer
	free(socket_desc);

	return 0;
}
```

The above connection handler takes some input from the client and replies back with the same. Simple! Here is how the telnet output might look

```
$ telnet localhost 8888
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Hello Client , I have received your connection. And now I will assign a handler for you
Greetings! I am your connection handler
Now type something and i shall repeat what you type
Hello
Hello
How are you
How are you
I am fine
I am fine
```

So now we have a server that communicative. That’s useful for now.

Generalization for Windows and Unix:(important **)**

Once connection was established, a proper way to disconnect is by explicitly calling shutdown() function.

 Generally **sendto**() and **recvfrom**() are used to send/receive UDP data grams without explicit connection initiation. This is **UDP** protocal you did in networking course.

The server will mostly use bind() to listen on an opened port but may fail much more likely than listen(), so in our program we have decided to use both of them first bind the listen.
In Windows we will only care of:
1. Use different set of header files - **windows.h** plus **winsock.h**
2. Remember to initialize socket library - **WSAStartup()**
3. Use **send**() and **recv**() rather than **write**() and **read**()
4. Call **closesocket**() instead of **close**()
Of course, in real life examples things get not so simple, but with these hints any hard worker will make it if need be.

III.19. Error Handling in C

C language does not provide any direct support for error handling. However a few methods and variables defined in **error.h** header file can be used to point out error using the return statement in a function. In C language, a function returns -1 or NULL value in case of any error and a global variable **errno** is set with the error code. So the return value can be used to check error while programming.
What is **errno**?
Whenever a function call is made in C language, a variable named errno is associated with it. It is a global variable, which can be used to identify which type of error was encountered while function execution, based on its value. Below we have the list of Error numbers and what does they mean.

Error no	Error value
1	Operation not permitted

2	No such file or directory
3	No such process
4	Interrupted system call
5	I/O error
6	No such device or address
7	Argument list too long
8	Exec format error
9	Bad file number
10	No child processes
11	Try again
12	Out of memory
13	Permission denied

C language uses the following functions to represent error messages associated with **errno**:

- perror(): returns the string passed to it along with the textual representation of the current errno value.
- strerror() is defined in **string.h** library. This method returns a pointer to the string representation of the current errno value.
- Some other error handling functions are customized based on the application, for example in MYSQL, we use :" **mysql_error({connection})**"

Example:

```
#include <stdio.h>
#include <errno.h>
#include <string.h>

int main ()
{
   FILE *fp;

   /*
      If a file, which does not exists, is opened,
      we will get an error
   */
   fp = fopen("IWillReturnError.txt", "r");
```

```
    printf("Value of errno: %d\n ", errno);
    printf("The error message is : %s\n", strerror(errno));
    perror("Message from perror");

    return 0;
}
```

Value of errno: 2 The error message is: No such file or directory Message from **perror**: No such file or directory

Other ways of Error Handling

We can also use **Exit Status** constants in the **exit()** function to inform the calling function about the error. The two constant values available for use are **EXIT_SUCCESS** and **EXIT_FAILURE**. These are nothing but macros defined in the **stdlib.h** header file, they are just numbers like **0** and **1**.

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

extern int errno;

void main()
{
    char *ptr = malloc( 1000000000UL);  //requesting to allocate 1gb memory
space
    if (ptr == NULL)    //if memory not available, it will return null
    {
        puts("malloc failed");
        puts(strerror(errno));
        exit(EXIT_FAILURE);     //exit status failure
    }
    else
    {
        free( ptr);
        exit(EXIT_SUCCESS);     //exit status Success
    }
}
```

Here exit function is used to indicate exit status. Its always a good practice to exit a program with a exit status. **EXIT_SUCCESS** and **EXIT_FAILURE** are two macro used to show exit status. In case of program coming out after a successful operation **EXIT_SUCCESS** is used to show successful exit. It is defined as 0. **EXIT_Failure** is used in case of any failure in the program. It is defined as -1.

Division by Zero

There are some situation where nothing can be done to handle the error. In C language one such situation is division by zero. All you can do is avoid doing this, because if you do so, C language is not able to understand what happened, and gives a run time error.
Best way to avoid this is, to check the value of the divisor before using it in the division operations. You can use if condition, and if it is found to be zero, just display a message and return from the function.

IV. Introduction to data Structures

IV.1. Linked List/Node List

Static arrays that we have seen before are structures whose size is fixed at compile time and therefore cannot be extended or reduced to fit the data set.

A dynamic array can be extended by doubling the size but there is overhead associated with the operation of **copying** old data and **freeing** the memory associated with the old data structure. One potential problem of using arrays for storing data is that arrays require a contiguous block of memory which may not be available, if the requested contiguous block is too large, we will fall in a **core dumped error**.

However the advantages of using arrays are that each element in the array can be accessed very efficiently using an index. However, for applications that can be better managed without using contiguous memory we define a concept called "**linked lists**".

A linked list is a collection of objects linked together by references from one object to another object. By convention these objects are named as **nodes**. So the basic linked list is collection of nodes where each node contains one or more data fields AND a reference to the **next node**. The last node points to a **NULL** reference to indicate the **end of the list. A such linked list is called :"**Singly linked list**"**

The entry point into a linked list is always the first node or **head** of the list. It should be noted that head is NOT a **separate** node, but a reference to the **first** Node in the list. If the list is empty, then the head has the value NULL, ie not head available and obviously no linked list available, **there can not be any linked list without a head node.**

Unlike Arrays, nodes cannot be accessed by an index since memory allocated for each individual node may not be continuous. We must begin from the head of the list and traverse the list sequentially to access the nodes in the list. Insertions of new nodes and deletion of existing nodes are fairly easy to handle and will be discussed in the this lesson. Recall that array insertions or deletions may require adjustment of the array (overhead), but insertions and deletions in linked lists can be performed very efficiently.

IV.2. Types of Linked Lists

There are few different types of linked lists.

A **singly** linked list as described above provides access to the list from the head node. Traversal is allowed only one way and there is no going back.

A **doubly** linked list is a list that has two references, one to the next node and another to previous node. Doubly linked list also starts from head node, but provide access both ways. That is one can traverse forward or backward from any node.

A **multi-linked** list is a more general linked list with multiple links from nodes.

For examples, we can define a Node that has two references, age pointer and a name pointer. With this structure it is possible to maintain a single list,The entry point into a linked list is always the first or head of the list.

IV.3. Implementation of a Linked List

Designing the Node:

Linked list is a collection of linked nodes. A node is a struct with at least a data field and a reference to a node of the same type. A node is called a self-referential object, since it contains a pointer to a variable that refers to a variable of the same type(**actually while defining a note you should notice a recursion in its objects reference**).

Like arrays, **Singly Linked List is a linear data structure**. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are **linked using pointers**.

For example, a **struct Node** that contains an int data field and a pointer to another node can be defined as follows:

struct Node {

int data;

struct Node* next;

}

typedef struct Node node;

node* head = NULL;

Allocating memory for the first node Memory must be allocated for one node and assigned to head as follows.

head = (node*) malloc(sizeof(node)); // verry simple but important

the add data to the first node as follow:

(*head).data = 10;

(*head).next = NULL;

Adding the second node and linking

node* nextnode = malloc(sizeof(node));

(*nextnode).data = 12;

(*nextnode).next = NULL;

// the actual linkage is bellow:

(*head).next = nextnode;

Advantages over arrays

1) Dynamic size

2) Ease of insertion/deletion

Drawbacks:

1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.
2) Extra memory space for a pointer is required with each element of the list.
3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation:

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty ie **non existing !!!!**, then the value of the head is NULL.
Each node in a list consists of at least two parts:
1) **Data**
2) **Pointer** (Or Reference) **to the next node.**

Linked List Traversal
In the previous program, we have created a simple linked list with three nodes. Let us traverse the created list and print the data of each node. For traversal, let us write a general-purpose function printList() that prints any given list.

```
// A simple C program for traversal of a linked list

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

struct Node {
int data;
struct Node* next;
};


int main()
```

```
{
// allocate 3 nodes in the heap memory

struct Node* head = (struct Node*)malloc(sizeof(struct Node));
struct Node* second = (struct Node*)malloc(sizeof(struct Node));
struct Node* third = (struct Node*)malloc(sizeof(struct Node));
struct Node* forth = (struct Node*)malloc(sizeof(struct Node));;



head->data = 1; // assign data in first node
head->next = second; // Link first node with second

second->data = 2; // assign data to second node
second->next = third;

third->data = 3; // assign data to third node
third->next = forth;

forth->data = 4; // assign data to forth node
forth->next = NULL;


// Traversal happens bellow:
// This consist in printing the contents of linked list starting from the
first node

while (head != NULL) {
printf(" %d ", head->data);
head = head->next;
}

return 0;
}
```

Improved Version of the previous programmer

V. C and DBMSes

V.1. About MySQL database

MySQL is a leading open source database management system. It is a multi user, multi threaded database management system. MySQL is especially popular on the web. It is one part of the very popular *LAMP* platform consisting of Linux, Apache, MySQL, and PHP. MySQL currently owned by Oracle. MySQL database is available on most important OS platforms. It runs on BSD Unix, Linux, Windows, or Mac OS. Wikipedia and YouTube use MySQL. These sites manage millions of queries each day. MySQL comes in two versions: MySQL server system and MySQL embedded system.

Configure the mysql:

to configure the mysql libraries you just have to run the bellow command:

```
$ sudo apt-get install libmysqlclient-dev
```

To be able to compile C examples, we need to install the MySQL C development libraries. The above line shows how we can do it on Debian based Linux.

C99

This tutorial uses C99. For GNU C compiler, we need to add the -std=c99 option. For Windows users, the Pelles C IDE is highly recommended. (MSVC does not support C99.)

```
MYSQL *con = mysql_init(NULL);
```

In C99, we can mix declarations with code. In older C programs, we would need to separate this line into two lines.

v.2 First example

Our first example will test one MySQL function call.

```
#include <my_global.h>
#include <mysql.h>

int main(int argc, char **argv)
{
  printf("MySQL client version: %s\n", mysql_get_client_info());

  exit(0);
}
```

The mysql_get_client_info() shows the MySQL client version.

```
#include <my_global.h>
#include <mysql.h>
```

We include necessary header files. The **mysql.h** is the most important header file for MySQL function calls. The **my_global.h** includes some global declarations a functions. Among other things, it includes the standard input/output header file, very nice.

```
printf("MySQL client version: %s\n", mysql_get_client_info());
```

This code line outputs the version of the MySQL client. For this, we use the **mysql_get_client_info()** function call.

```
exit(0);
```

We exit from the script.

```
$ gcc version.c -o version  `mysql_config --cflags --libs`
```

Here is how we compile the code example.

```
$ ./version
MySQL client version: 5.1.67
```

Example output.

V.3 Creating a database

The next code example will create a database. The code example can be divided into these parts:

- Initiation of a connection handle structure
- Creation of a connection
- Execution of a query
- Closing of the connection

```
#include <my_global.h>
#include <mysql.h>

int main(int argc, char **argv)
{
  MYSQL *con = mysql_init(NULL);

  if (con == NULL)
  {
      fprintf(stderr, "%s\n", mysql_error(con));
      exit(1);
  }

  if (mysql_real_connect(con, "localhost", "root", "root_pswd",
          NULL, 0, NULL, 0) == NULL)
  {
      fprintf(stderr, "%s\n", mysql_error(con));
      mysql_close(con);
      exit(1);
  }

  if (mysql_query(con, "CREATE DATABASE testdb"))
  {
      fprintf(stderr, "%s\n", mysql_error(con));
      mysql_close(con);
      exit(1);
  }
  mysql_close(con);
  exit(0);
}
```

The code example connects to the MySQL database system and creates a new database called testdb.

```
MYSQL *con = mysql_init(NULL);
```

The mysql_init() function allocates or initialises a MYSQL object suitable for mysql_real_connect() function. Remember this is C99.

```
if (con == NULL)
{
    fprintf(stderr, "%s\n", mysql_error(con));
    exit(1);
}
```

We check the return value. If the mysql_init() function fails, we print the error message and terminate the application.

```
if (mysql_real_connect(con, "localhost", "root", "root_pswd",
        NULL, 0, NULL, 0) == NULL)
{
    fprintf(stderr, "%s\n", mysql_error(con));
    mysql_close(con);
    exit(1);
}
```

The mysql_real_connect() function establishes a connection to the database. We provide connection handler, host name, user name and password parameters to the function. The other four parameters are the database name, port number, unix socket and finally the client flag. We need superuser priviliges to create a new database.

```
if (mysql_query(con, "CREATE DATABASE testdb"))
{
    fprintf(stderr, "%s\n", mysql_error(con));
    mysql_close(con);
    exit(1);
}
```

The mysql_query() executes the SQL statement. In our case, the statement creates a new database.

```
mysql_close(con);
```

Finally, we close the database connection.

```
$ gcc createdb.c -o createdb -std=c99  `mysql_config --cflags --libs`
```

The second example already utilizes features from C99 standard. Therefore, we need to add the -std=c99 option.

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| testdb             |
+--------------------+
3 rows in set (0.00 sec)
```

This is the proof that the database was created.

V.4. Creating and populating a table

Before we create a new table, we create a user that we will use in the rest of the tutorial.

```
mysql> CREATE USER user12@localhost IDENTIFIED BY '34klq*';
```

We have created a new user user12.

```
mysql> GRANT ALL ON testdb.* to user12@localhost;
```

Here we grant all priviliges to user12 on testdb database.

The next code example will create a table and insert some data into it.

```
#include <my_global.h>
#include <mysql.h>

void finish_with_error(MYSQL *con)
{
  fprintf(stderr, "%s\n", mysql_error(con));
  mysql_close(con);
  exit(1);
}

int main(int argc, char **argv)
{
  MYSQL *con = mysql_init(NULL);

  if (con == NULL)
  {
      fprintf(stderr, "%s\n", mysql_error(con));
      exit(1);
  }

  if (mysql_real_connect(con, "localhost", "user12", "34klq*",
          "testdb", 0, NULL, 0) == NULL)
  {
      finish_with_error(con);
  }

  if (mysql_query(con, "DROP TABLE IF EXISTS Cars")) {
      finish_with_error(con);
  }

  if (mysql_query(con, "CREATE TABLE Cars(Id INT, Name TEXT, Price INT)")) {
      finish_with_error(con);
  }
  if (mysql_query(con, "INSERT INTO Cars VALUES(1,'Audi',52642)")) {
      finish_with_error(con);
  }

  if (mysql_query(con, "INSERT INTO Cars VALUES(2,'Mercedes',57127)")) {
      finish_with_error(con);
  }

  if (mysql_query(con, "INSERT INTO Cars VALUES(3,'Skoda',9000)")) {
      finish_with_error(con);
  }

  if (mysql_query(con, "INSERT INTO Cars VALUES(4,'Volvo',29000)")) {
      finish_with_error(con);
  }

  if (mysql_query(con, "INSERT INTO Cars VALUES(5,'Bentley',350000)")) {
      finish_with_error(con);
  }

  if (mysql_query(con, "INSERT INTO Cars VALUES(6,'Citroen',21000)")) {
      finish_with_error(con);
  }

  if (mysql_query(con, "INSERT INTO Cars VALUES(7,'Hummer',41400)")) {
      finish_with_error(con);
  }

  if (mysql_query(con, "INSERT INTO Cars VALUES(8,'Volkswagen',21600)")) {
      finish_with_error(con);
  }

  mysql_close(con);
  exit(0);
}
```

We don't use any new MySQL function call here. We use mysql_query() function call to both create a table and insert data into it.

```
void finish_with_error(MYSQL *con)
{
  fprintf(stderr, "%s\n", mysql_error(con));
  mysql_close(con);
  exit(1);
}
```

In order to avoid unnecessary repetition, we create a custom finish_with_error() function.

```
if (mysql_real_connect(con, "localhost", "user12", "34klq*",
        "testdb", 0, NULL, 0) == NULL)
{
    finish_with_error(con);
}
```

We connect to testdb database. The user name is user12 and password is 34klq*. The fifth parameter is the database name.

```
if (mysql_query(con, "CREATE TABLE Cars(Id INT, Name TEXT, Price INT)")) {
    finish_with_error(con);
}
```

Here we create a table named Cars. It has three columns.

```
if (mysql_query(con, "INSERT INTO Cars VALUES(1,'Audi',52642)")) {
    finish_with_error(con);
}
```

We insert one row into the Cars table.

```
mysql> USE testdb;
mysql> SHOW TABLES;
+------------------+
| Tables_in_testdb |
+------------------+
| Cars             |
+------------------+
1 row in set (0.00 sec)
```

We show tables in the database.

```
mysql> SELECT * FROM Cars;
+------+------------+--------+
| Id   | Name       | Price  |
+------+------------+--------+
|    1 | Audi       |  52642 |
|    2 | Mercedes   |  57127 |
|    3 | Skoda      |   9000 |
|    4 | Volvo      |  29000 |
|    5 | Bentley    | 350000 |
|    6 | Citroen    |  21000 |
|    7 | Hummer     |  41400 |
|    8 | Volkswagen |  21600 |
+------+------------+--------+
8 rows in set (0.00 sec)
```

We select all data from the table.

V.5. Retrieving data from the database

In the next example, we will retrieve data from a table.

We need to do the following steps:

- Create a connection
- Execute query
- Get the result set
- Fetch all available rows
- Free the result set

```
#include <my_global.h>
#include <mysql.h>

void finish_with_error(MYSQL *con)
{
  fprintf(stderr, "%s\n", mysql_error(con));
  mysql_close(con);
  exit(1);
}

int main(int argc, char **argv)
{
  MYSQL *con = mysql_init(NULL);

  if (con == NULL)
  {
      fprintf(stderr, "mysql_init() failed\n");
      exit(1);
  }

  if (mysql_real_connect(con, "localhost", "user12", "34klq*",
          "testdb", 0, NULL, 0) == NULL)
  {
      finish_with_error(con);
  }

  if (mysql_query(con, "SELECT * FROM Cars"))
  {
      finish_with_error(con);
  }

  MYSQL_RES *result = mysql_store_result(con);

  if (result == NULL)
  {
      finish_with_error(con);
  }

  int num_fields = mysql_num_fields(result);

  MYSQL_ROW row;

  while ((row = mysql_fetch_row(result)))
  {
      for(int i = 0; i < num_fields; i++)
      {
          printf("%s ", row[i] ? row[i] : "NULL");
      }
          printf("\n");
  }

  mysql_free_result(result);
  mysql_close(con);

  exit(0);
}
```

The example prints all columns from the Cars table.

```
if (mysql_query(con, "SELECT * FROM Cars"))
{
    finish_with_error(con);
}
```

We execute the query that will retrieve all data from the Cars table.

```
MYSQL_RES *result = mysql_store_result(con);
```

We get the result set using the mysql_store_result() function. The MYSQL_RES is a structure for holding a result set.

```
int num_fields = mysql_num_fields(result);
```

We get the number of fields (columns) in the table.

```
MYSQL_ROW row;

while ((row = mysql_fetch_row(result)))
{
    for(int i = 0; i < num_fields; i++)
    {
        printf("%s ", row[i] ? row[i] : "NULL");
    }
        printf("\n");
}
```

We fetch the rows and print them to the screen.

```
mysql_free_result(result);
mysql_close(con);
```

We free the resources.

```
$ ./retrieva_data
1 Audi 52642
2 Mercedes 57127
3 Skoda 9000
4 Volvo 29000
5 Bentley 350000
6 Citroen 21000
7 Hummer 41400
8 Volkswagen 21600
```

Example output.

V.6. Last inserted row id

Sometimes, we need to determine the id of the last inserted row. We can determine the last inserted row id by calling the mysql_insert_id() function. The function only works if we have defined an AUTO_INCREMENT column in the table.

```
#include <my_global.h>
#include <mysql.h>

void finish_with_error(MYSQL *con)
{
  fprintf(stderr, "%s\n", mysql_error(con));
  mysql_close(con);
  exit(1);
}

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

  MYSQL *con = mysql_init(NULL);
  if (con == NULL)
  {
      fprintf(stderr, "mysql_init() failed\n");
      exit(1);
  }

  if (mysql_real_connect(con, "localhost", "user12", "34klq*",
          "testdb", 0, NULL, 0) == NULL)
  {
      finish_with_error(con);
  }

  if (mysql_query(con, "DROP TABLE IF EXISTS Writers"))
  {
      finish_with_error(con);
  }

  char *sql = "CREATE TABLE Writers(Id INT PRIMARY KEY AUTO_INCREMENT, Name TEXT)";

  if (mysql_query(con, sql))
  {
      finish_with_error(con);
  }

  if (mysql_query(con, "INSERT INTO Writers(Name) VALUES('Leo Tolstoy')"))
  {
      finish_with_error(con);
  }

  if (mysql_query(con, "INSERT INTO Writers(Name) VALUES('Jack London')"))
  {
      finish_with_error(con);
  }

  if (mysql_query(con, "INSERT INTO Writers(Name) VALUES('Honore de Balzac')"))
  {
      finish_with_error(con);
  }

  int id = mysql_insert_id(con);

  printf("The last inserted row id is: %d\n", id);

  mysql_close(con);
  exit(0);
}
```

A new table is created. Three rows are inserted into the table. We determine the last inserted row id.

```
char *sql = "CREATE TABLE Writers(Id INT PRIMARY KEY AUTO_INCREMENT, Name TEXT)";
```

The Id column has an AUTO_INCREMENT type.

```
int id = mysql_insert_id(con);
```

The mysql_insert_id() function returns the value generated for an AUTO_INCREMENT column by the previous INSERT or UPDATE statement.

```
$ ./last_row_id
The last inserted row id is: 3
```

Output.

V.7. Column headers

In the next example, we will retrieve data from the table and its column names.

```
#include <my_global.h>
#include <mysql.h>

void finish_with_error(MYSQL *con)
{
  fprintf(stderr, "%s\n", mysql_error(con));
  mysql_close(con);
  exit(1);
}

int main(int argc, char **argv)
{
  MYSQL *con = mysql_init(NULL);

  if (con == NULL)
  {
      fprintf(stderr, "mysql_init() failed\n");
      exit(1);
  }

  if (mysql_real_connect(con, "localhost", "user12", "34klq*",
          "testdb", 0, NULL, 0) == NULL)
  {
      finish_with_error(con);
  }

  if (mysql_query(con, "SELECT * FROM Cars LIMIT 3"))
  {
      finish_with_error(con);
  }

  MYSQL_RES *result = mysql_store_result(con);

  if (result == NULL)
  {
      finish_with_error(con);
  }

  int num_fields = mysql_num_fields(result);

  MYSQL_ROW row;
  MYSQL_FIELD *field;

  while ((row = mysql_fetch_row(result)))
  {
      for(int i = 0; i < num_fields; i++)
      {
          if (i == 0)
          {
             while(field = mysql_fetch_field(result))
             {
                printf("%s ", field->name);
             }

             printf("\n");
```

```
        }

        printf("%s  ", row[i] ? row[i] : "NULL");
      }
  }

  printf("\n");

  mysql_free_result(result);
  mysql_close(con);

  exit(0);
}
```

We print the first three rows from the Cars table. We also include the column headers.

```
MYSQL_FIELD *field;
```

The MYSQL_FIELD structure contains information about a field, such as the field's name, type and size. Field values are not part of this structure; they are contained in the MYSQL_ROW structure.

```
if (i == 0)
{
    while(field = mysql_fetch_field(result))
    {
        printf("%s ", field->name);
    }

    printf("\n");
}
```

The first row contains the column headers. The mysql_fetch_field() call returns a MYSQL_FIELD structure. We get the column header names from this structure.

```
$ ./headers
Id Name Price
1  Audi  52642
2  Mercedes  57127
3  Skoda  9000
```

This is the output of our program.

V.8. Multiple statements

It is possible to execute multiple SQL statements in one query. We must set the CLIENT_MULTI_STATEMENTS flag in the connect method.

```
#include <my_global.h>
#include <mysql.h>

void finish_with_error(MYSQL *con)
{
  fprintf(stderr, "%s\n", mysql_error(con));
  mysql_close(con);
  exit(1);
}
```

```
int main(int argc, char **argv)
{
  int status = 0;

  MYSQL *con = mysql_init(NULL);

  if (con == NULL)
  {
      fprintf(stderr, "mysql_init() failed\n");
      exit(1);
  }

  if (mysql_real_connect(con, "localhost", "user12", "34klq*",
          "testdb", 0, NULL, CLIENT_MULTI_STATEMENTS) == NULL)
  {
      finish_with_error(con);
  }

  if (mysql_query(con, "SELECT Name FROM Cars WHERE Id=2;\
      SELECT Name FROM Cars WHERE Id=3;SELECT Name FROM Cars WHERE Id=6"))
  {
      finish_with_error(con);
  }

  do {
      MYSQL_RES *result = mysql_store_result(con);

      if (result == NULL)
      {
          finish_with_error(con);
      }

      MYSQL_ROW row = mysql_fetch_row(result);

      printf("%s\n", row[0]);

      mysql_free_result(result);

      status = mysql_next_result(con);

      if (status > 0) {
          finish_with_error(con);
      }

  } while(status == 0);

  mysql_close(con);
  exit(0);
}
```

In the example, we execute three SELECT statements in one query.

```
if (mysql_real_connect(con, "localhost", "user12", "34klq*",
        "testdb", 0, NULL, CLIENT_MULTI_STATEMENTS) == NULL)
{
    finish_with_error(con);
}
```

The last option of the mysql_real_connect() method is the client flag. It is used to enable certain features. The CLIENT_MULTI_STATEMENTS enables the execution of multiple statements. This is disabled by default.

```
if (mysql_query(con, "SELECT Name FROM Cars WHERE Id=2;\
    SELECT Name FROM Cars WHERE Id=3;SELECT Name FROM Cars WHERE Id=6"))
{
    finish_with_error(con);
}
```

The query consists of three SELECT statements. They are separated by the semicolon ; character. The backslash character \ is used to separate the string into two lines. It has nothing to do with multiple statements.

```
do {
...
} while(status == 0);
```

The code is placed between the do/while statements. The data retrieval is to be done in multiple cycles. We will retrieve data for each SELECT statement separately.

```
status = mysql_next_result(con);
```

We expect multiple result sets. Therefore, we call the mysql_next_result() function. It reads the next statement result and returns a status to indicate whether more results exist. The function returns 0 if the execution went OK and there are more results. It returns -1, when it is executed OK and there are no more results. Finally, it returns value greater than zero if an error occurred.

```
if (status > 0) {
    finish_with_error(con);
}
```

We check for error.

```
$ ./multiple_statements
Mercedes
Skoda
Citroen
```

Example output.

V.9. Inserting images into MySQL database

Some people prefer to put their images into the database, some prefer to keep them on the file system for their applications. Technical difficulties arise when we work with lots of images. Images are binary data. MySQL database has a special data type to store binary data called BLOB (Binary Large Object).

```
mysql> CREATE TABLE Images(Id INT PRIMARY KEY, Data MEDIUMBLOB);
```

For our examples, we create a new Images table. The image size can be up to 16 MB. It is determined by the MEDIUMBLOB data type.

```
#include <my_global.h>
#include <mysql.h>
#include <string.h>

void finish_with_error(MYSQL *con)
```

```
{
  fprintf(stderr, "%s\n", mysql_error(con));
  mysql_close(con);
  exit(1);
}

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

  FILE *fp = fopen("woman.jpg", "rb");

  if (fp == NULL)
  {
      fprintf(stderr, "cannot open image file\n");
      exit(1);
  }

  fseek(fp, 0, SEEK_END);

  if (ferror(fp)) {

      fprintf(stderr, "fseek() failed\n");
      int r = fclose(fp);

      if (r == EOF) {
          fprintf(stderr, "cannot close file handler\n");
      }

      exit(1);
  }

  int flen = ftell(fp);

  if (flen == -1) {

      perror("error occurred");
      int r = fclose(fp);

      if (r == EOF) {
          fprintf(stderr, "cannot close file handler\n");
      }

      exit(1);
  }

  fseek(fp, 0, SEEK_SET);

  if (ferror(fp)) {

      fprintf(stderr, "fseek() failed\n");
      int r = fclose(fp);

      if (r == EOF) {
          fprintf(stderr, "cannot close file handler\n");
      }

      exit(1);
  }

  char data[flen+1];

  int size = fread(data, 1, flen, fp);
  if (ferror(fp)) {

      fprintf(stderr, "fread() failed\n");
      int r = fclose(fp);

      if (r == EOF) {
          fprintf(stderr, "cannot close file handler\n");
      }

      exit(1);
  }

  int r = fclose(fp);

  if (r == EOF) {
      fprintf(stderr, "cannot close file handler\n");
  }

  MYSQL *con = mysql_init(NULL);

  if (con == NULL)
  {
      fprintf(stderr, "mysql_init() failed\n");
      exit(1);
  }

  if (mysql_real_connect(con, "localhost", "user12", "34klq*",
          "testdb", 0, NULL, 0) == NULL)
  {
      finish_with_error(con);
  }

  char chunk[2*size+1];
  mysql_real_escape_string(con, chunk, data, size);

  char *st = "INSERT INTO Images(Id, Data) VALUES(1, '%s')";
  size_t st_len = strlen(st);

  char query[st_len + 2*size+1];
  int len = snprintf(query, st_len + 2*size+1, st, chunk);

  if (mysql_real_query(con, query, len))
  {
      finish_with_error(con);
  }

  mysql_close(con);
  exit(0);
}
```

In this example, we will insert one image into the Images table.

```
#include <string.h>
```

This include is for the strlen() function.

```
FILE *fp = fopen("woman.jpg", "rb");

if (fp == NULL)
{
    fprintf(stderr, "cannot open image file\n");
    exit(1);
}
```

Here we open the image file. In the current working directory, we should have the woman.jpg file.

```
fseek(fp, 0, SEEK_END);

if (ferror(fp)) {

    fprintf(stderr, "fseek() failed\n");
    int r = fclose(fp);

    if (r == EOF) {
        fprintf(stderr, "cannot close file handler\n");
    }

    exit(1);
}
```

We move the file pointer to the end of the file using the fseek() function. We are going to determine the size of the image. If an error occurs, the error indicator is set. We check the indicator using the fseek() function. In case of an error, we also close the opened file handler.

```
int flen = ftell(fp);

if (flen == -1) {

    perror("error occurred");
    int r = fclose(fp);

    if (r == EOF) {
        fprintf(stderr, "cannot close file handler\n");
    }

    exit(1);
}
```

For binary streams, the ftell() function returns the number of bytes from the beginning of the file, e.g. the size of the image file. In case of an error, the function returns -1 and the errno is set. The perrro() function interprets the value of errno as an error message, and prints it to the standard error output stream.

```
char data[flen+1];
```

In this array, we are going to store the image data.

```
int size = fread(data, 1, flen, fp);
```

We read the data from the file pointer and store it in the data array. The total number of elements successfully read is returned.

```
int r = fclose(fp);

if (r == EOF) {
    fprintf(stderr, "cannot close file handler\n");
}
```

After the data is read, we can close the file handler.

```
char chunk[2*size+1];
mysql_real_escape_string(con, chunk, data, size);
```

The mysql_real_escape_string() function adds an escape character, the backslash, \, before certain potentially dangerous characters in a string passed in to the function. This can help prevent SQL injection attacks. The new buffer must be at least 2*size+1 long.

```
char *st = "INSERT INTO Images(Id, Data) VALUES(1, '%s')";
size_t st_len = strlen(st);
```

Here we start building the SQL statement. We determine the size of the SQL string using the strlen() function.

```
char query[st_len + 2*size+1];
int len = snprintf(query, st_len + 2*size+1, st, chunk);
```

The query must take be long enough to contain the size of the SQL string statement and the size of the image file. Using the snprintf() function, we write the formatted output to query buffer.

```
if (mysql_real_query(con, query, len))
{
    finish_with_error(con);
};
```

We execute the query using the mysql_real_query() function. The mysql_query() cannot be used for statements that contain binary data; we must use the mysql_real_query() instead.

V.10. Selecting images from MySQL database

In the previous example, we have inserted an image into the database. In the following example, we will select the inserted image back from the database.

```
#include <my_global.h>
#include <mysql.h>

void finish_with_error(MYSQL *con)
{
  fprintf(stderr, "%s\n", mysql_error(con));
  mysql_close(con);
  exit(1);
}

int main(int argc, char **argv)
{
  FILE *fp = fopen("woman2.jpg", "wb");

  if (fp == NULL)
  {
      fprintf(stderr, "cannot open image file\n");
      exit(1);
  }

  MYSQL *con = mysql_init(NULL);

  if (con == NULL)
  {
      fprintf(stderr, "mysql_init() failed\n");
      exit(1);
  }

  if (mysql_real_connect(con, "localhost", "user12", "34klq*",
          "testdb", 0, NULL, 0) == NULL)
  {
      finish_with_error(con);
  }

  if (mysql_query(con, "SELECT Data FROM Images WHERE Id=1"))
  {
      finish_with_error(con);
  }

  MYSQL_RES *result = mysql_store_result(con);

  if (result == NULL)
  {
      finish_with_error(con);
  }

  MYSQL_ROW row = mysql_fetch_row(result);
  unsigned long *lengths = mysql_fetch_lengths(result);

  if (lengths == NULL) {
      finish_with_error(con);
  }

  fwrite(row[0], lengths[0], 1, fp);

  if (ferror(fp))
  {
      fprintf(stderr, "fwrite() failed\n");
      mysql_free_result(result);
      mysql_close(con);

      exit(1);
  }

  int r = fclose(fp);

  if (r == EOF) {
      fprintf(stderr, "cannot close file handler\n");
  }

  mysql_free_result(result);
  mysql_close(con);

  exit(0);
}
```

In this example, we will create an image file from the database.

```
FILE *fp = fopen("woman2.jpg", "wb");
```

```
if (fp == NULL)
{
    fprintf(stderr, "cannot open image file\n");
    exit(1);
}
```

We open a new file handler for writing.

```
if (mysql_query(con, "SELECT Data FROM Images WHERE Id=1"))
{
    finish_with_error(con);
}
```

We select the Data column from the Image table with Id 1.

```
MYSQL_ROW row = mysql_fetch_row(result);
```

The row contains raw data.

```
unsigned long *lengths = mysql_fetch_lengths(result);
```

We get the length of the image.

```
fwrite(row[0], lengths[0], 1, fp);

if (ferror(fp))
{
    fprintf(stderr, "fwrite() failed\n");
    mysql_free_result(result);
    mysql_close(con);

    exit(1);
}
```

We write the retrieved data to the disk using the fwrite() function call. We check for the error indicator with the ferror() function.

```
int r = fclose(fp);

if (r == EOF) {
    fprintf(stderr, "cannot close file handler\n");
}
```

After we have written the image data, we close the file handler using the fclose() function.

V. INDEX: Practical Exercises

1. Write a program that prints ‘Hello World’ to the screen.
2. Write a program that print a triangle of stars on the console.
3. Write a program that asks the user for their name and greets them with their name.
4. Modify the previous program such that only the users Alice and Bob are greeted with their names.

5. Write a program that add two numbers and returns the sum to the calling function or method
6. Write a program that repeatedly add two numbers entered by the user up on request and stops when the user decide to stop.
7. Write a program that asks the user for a number n and prints the sum of the numbers 1 to n
8. Modify the previous program such that only multiples of three or five are considered in the sum, e.g. 3, 5, 6, 9, 10, 12, 15 for n=17
9. Write a program that asks the user for a number n and gives them the possibility to choose between computing the sum and computing the product of 1,...,n.
10. Write a program that prints a multiplication table for numbers up to 12
11. Write a program that prints *all* prime numbers.
12. An Armstrong number of three digits is an integer such that the sum of the cubes of its digits is equal to the number itself. For example, 371 is an Armstrong number since 3**3 + 7**3 + 1**3 = 371.

Write a program to find all Armstrong number in the range of 0 and 999.

13. Write a guessing game where the user has to guess a secret number. After every guess the program tells the user whether their number was too large or too small. At the end the number of tries needed should be printed. It counts only as one try if they input the same number multiple times consecutively.
14. Write a program that prints the next 20 leap years.
15. Write a function that returns the largest number in a collection n positive of numbers.
16. Write a simple calculator program that can perform all 4 basic mathematical operation and print the result. Lets the program be able to only accept numbers as user inputs and reject any other data type.
17. Write three functions that compute the sum of the numbers in a list: using a for-loop, a while-loop and recursion.
18. Write a function that computes the list of the first 100 Fibonacci numbers. The first two Fibonacci numbers are 1 and 1. The n+1-st Fibonacci number can be computed by adding the n-th and the n-1-th

Fibonacci number. The first few are therefore 1, 1, 1+1=2, 1+2=3, 2+3=5, 3+5=8.

19. Write a function that takes a number and returns a list of its digits. So for 2342 it should return [2,3,4,2].

20. Write a function that takes a list of strings an prints them, one per line, in a rectangular frame. For example the list ["Hello", "World", "in", "a", "frame"] gets printed as:

```
*********
* Hello *
* World *
* in    *
* a     *
* frame *
*********
```

18 . Write function that translates a text to Pig Latin and back. English is translated to Pig Latin by taking the first letter of every word, moving it to the end of the word and adding ‘ay’. “The quick brown fox” becomes “Hetay uickqay rownbay oxfay”

19. Write a program that outputs all possibilities to put + or - or nothing between the numbers 1,2,…,9 (in this order) such that the result is 100. For example 1 + 2 + 3 - 4 + 5 + 6 + 78 + 9 = 100.

20. Write a program that takes the duration of a year (in fractional days) for an imaginary planet as an input and produces a leap-year rule that minimizes the difference to the planet’s solar year.

1. Write a program in C to read an existing file

3. Write a program in C to write multiple lines in a text file.

4. Write a program in C to read the file and store the lines into an array.

5. Write a program in C to Find the Number of Lines in a Text File.

6. Write a program in C to find the content of the file and number of lines in a Text File.

7. Write a program in C to count a number of words and characters in a file.

8. Write a program in C to delete a specific line from a file.

9. Write a program in C to replace a specific line with another text in a file.

13. Write a program in C to encrypt a text file.

14. Write a program in C to decrypt a previously encrypted file into the original file

15. Write a program in C to remove a file from the disk.

16. Write a C program to read numbers from a file and write even, odd and prime numbers to separate file.

17. Write a C program to compare two files.

18. Write a C program to copy contents from one file to another file.
19. Write a C program to merge two file to third file.

21. Write a C program to remove a word from text file.

22. Write a C program to remove empty lines from a text file.
23. Write a C program to find occurrence of a word in a text file.
24. Write a C program to print source code of same program.
25. Write a C program to convert uppercase to lowercase character and vice versa in a text file.

26. Write a C program to find properties of a file using stat() function.
27. Write a C program to rename a file using rename() function.
28. Write a C program to list all files and sub-directories recursively.

29. C program to read and print an employee's detail using structure.

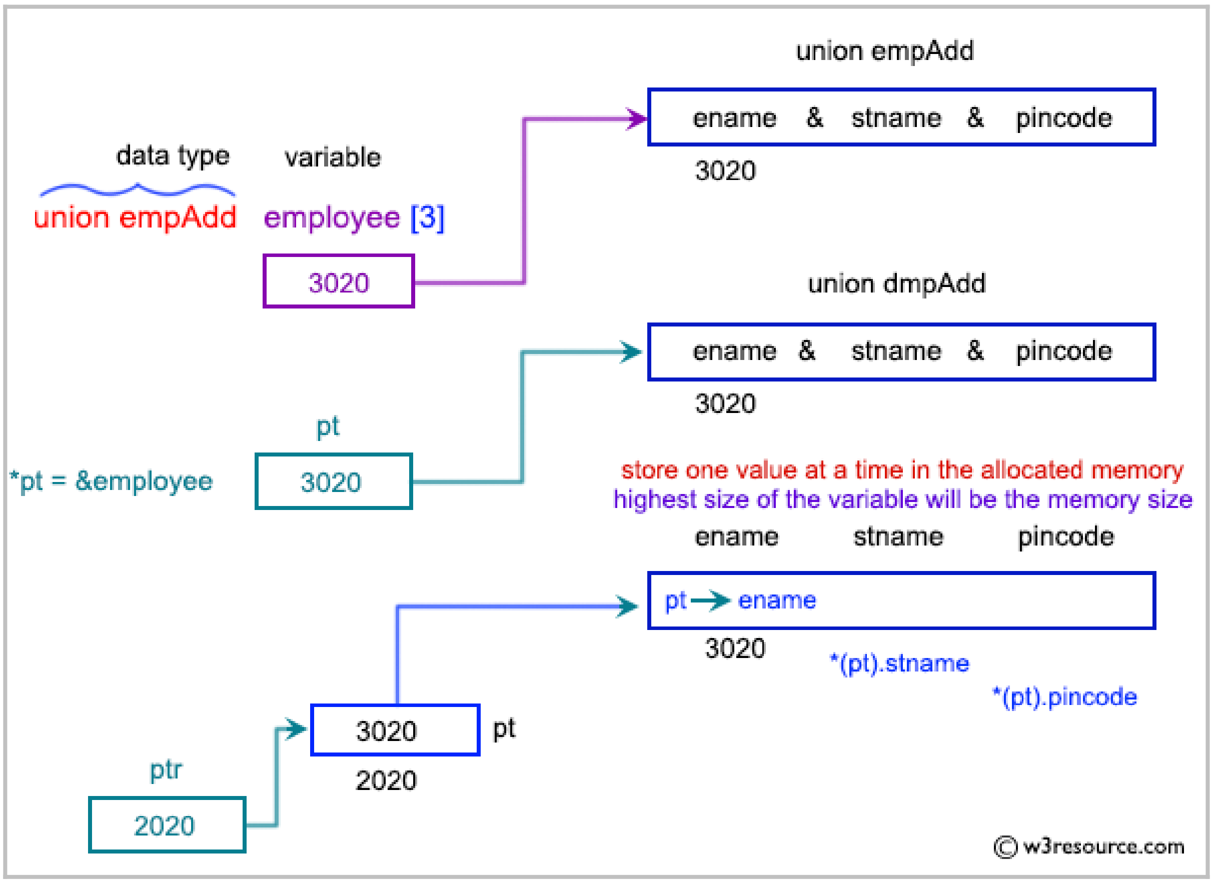
30. C program to add two distances in feet and inches using structure.

31. C program to extract individual bytes from an unsigned int using union.

32. C program for passing structures as function arguments and returning a structure from a function.
33. Write A program that can Calculate any party expenses using C program.

34. Write a program in C to show a pointer to union.

Pictorial Presentation:



REGULAR EXPRESSIONS in C:

https://www.regular-expressions.info/email.html

https://lwn.net/Articles/542629/ : REUSE SOCKET/PORT

By MASENGESHO Donatien

Software development Instructor at

Rwanda Coding Academy