C Programming

Download MinGW for win
Right click on mingw for w
Apply change on default
Set environment variable

```
#include<stdio.h>   Preprocessor directive
int main() {        <- entry point
    printf("Hello world!");   end of line.
    return 0;
}
```

case sensitive

gcc Helloworld.c
./a.exe.

Variables:- name of memory location which store some data.
Variable rules
Variable datatypes
Constants :- types :- ① Integer constants 1, 2, 3, 0, -1, -2
② Real constants 1.0, 2.0, 3.14, -2.3
③ Character constants 'a', 'b', 'A', '@', '#'.

Keywords:- Reserved words special meanings.
32.

Comments:- Single line //
Multi line /* */

Output:- printf("Hello world");
printf("Back bhi");

Next line printf("\n");

Cases:-
```
int age = 21;
printf("for integer %d", age);
printf("for real number %f", pi);
printf("for character %c", star);
```

Input:- scanf("%d", &age);

Compilation:- Computer translate C code into machine code.

Hello.c → C Compiler → a.exe

Ch:-2 Instructions:- There are statements in a program

Types:- → Type Declaration Instructions
→ Arithmetic Instructions
→ Control Instructions

Type declaration :- Declare var before using it.

Arithmetic instructions :- + − * /

Single var on the LHS.

int a = b+c+d*e/f;

~~int a,b = 2+c;~~ wrong.

Modulus operator % returns remainder

Type conversion:-

int operate int → int

int operate float → float

float operate float → float

Ref. 1.16

Auto → Implicit Explicit → we can do.

int → float int a = (int) 1.9999.;

Operator precedence:-

* / %

+ −

=

Associativity.

For same precedence

Left to Right

Control Instructions :-
use to determine flow of program

a. Sequence control. → line by line execution first to last
b. Decision control → if-else
c. Loop control → for, while.
d. Case control → switch case

Operators :-

Arithmetic operators :→ +, −, /, *, %
Relational → ==, >, >=, <, <=, !=
Logical → && AND, || OR, ! NOT
Bitwise → & | ^ << >> ~
Assignment → =, +=, −=, *=, /=, %=
Ternary → var = Exp1 (condition) ? Exp2 : Exp3 ;

Operator precedence :-

!
*, /, %
+, −
<, <=, >, >=
==, !=
&&
||
=

Conditional statements :-

Types
if-else
switch

```
if (condition) {
    // true part
} else {
    // false part
}
```

```
switch (1) {
    case 1 : // do something;
             break;
    case 2 : // do something;
             break;
    default : // do something;
}
```

** Case can be in any order
means 1 in first & 2 in second
doesn't matter
** Nested switch are allowed

Loop Control Instructions:-

To repeat some parts of the program with certain conditions

Types: → do while
- while
- for

```
do {
    // do something
} while (condition);
```

```
for (initialisation; condition; updation) {
    // do something
}
```

```
while (condition) {
    // do something
}
```

Special things

	Post	Pre
Increment operator	a++	++a
Decrement operator	a--	--a

- Loop counter can be float or even character.
- Infinite Loop. Never ran.

Break Statement:- exit the loop.

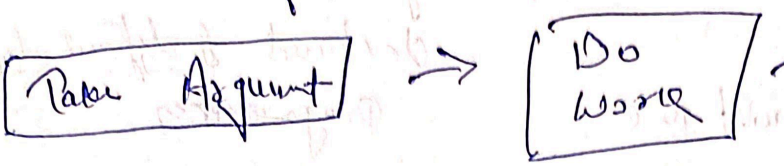
Continue Statement skip to next iteration.

Nested loops:

```
for () {
    for () {

    }
}
```

Functions:

block of code that performs particular task.



* It can be used multiple times
* Increase code reusability.

Syntax:

Function prototype:

```
void printHello();
```

Function Definition:

```
void printHello(){
    printf("Hello");
}
```

Function Call

```
int main(){
    printHello();
    return 0;
}
```

Properties:

* Execution always starts from main.
* A function gets called directly or indirectly from main
* There can be multiple function in a program

4.12

Function Types:-

- library function
  Special functions inbuilt in C
  Scanf(), printf()
- user defined
  declared & defined by Programmer.

Passing arguments:-

function can take values & give some value. → return value

Parameters ← parameter

```
void print (int n) {
    printf("%d", n);
}
print(3);
```

↑ arguments.

Argument V/S Parameter.

- Values that are passed in function call.
- used to send value
- actual parameter.

- Values in function declaration & definition
- used to receive value
- formal parameters.

Note:-

- Function can only return one value at a time.
- Changes to parameters in function don't chage the values in calling function.
  Because a copy of argument is passed to the function.

Recursion:- when a function calls itself, it's called recursion


```
#include <stdio.h>
void printHW (int count);
int main() {
    printHW(5);
    return 0;
}
// recursive function
void printHW (int count) {
    if (count == 0) {
        return;
    }
    printf ("Hello World \n");
    printHW (count-1);
}
```

Recursion Math:-

$f(x) = x^2$

$f(1) = 1^2 = 1$

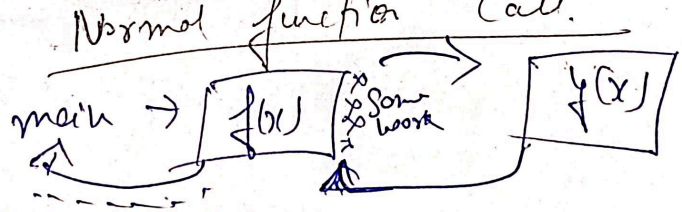
$f(2) = 2^2 = 4$

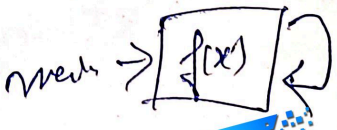
$f(3) = 3^2 = 9$

$f(f(x))$ for $x=2$ $\quad f(x^2) = f(4) = 4^2 = 16$

$f(f(f(x)))$

256 ← 16 ← 4





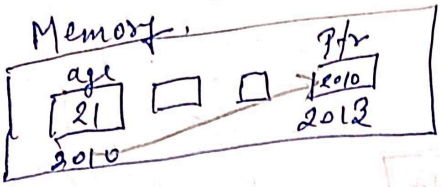
Properties of Recursion:-

* Anything that can be done with iteration, can be done with recursion and vice-versa.
* Recursion can sometimes give the most simple solution.
* Base case is the condition which stops recursion.
* Iteration has infinite loop & Recursion has stack overflow

Code:

```
#include <stdio.h>
int fact (int n);
int main() {
    printf("Factorial is: %d", fact(4));
    return 0;
}
int fact (int n) {
    if (n==0) {
        return 1;
    }
    int factNm1 = fact(n-1);
    int factN = factNm1 * n;
    return factN;
```

Pointers: A variable that stores the memory address of another variable.



Syntax:

```
int age = 22;
int *ptr = &age;
int _age = *ptr;
```

* value at address
& address of

Day 52

Declaring pointers:-

```
int *ptr;
char *ptr;
float *ptr;
```

Format Specifier:-

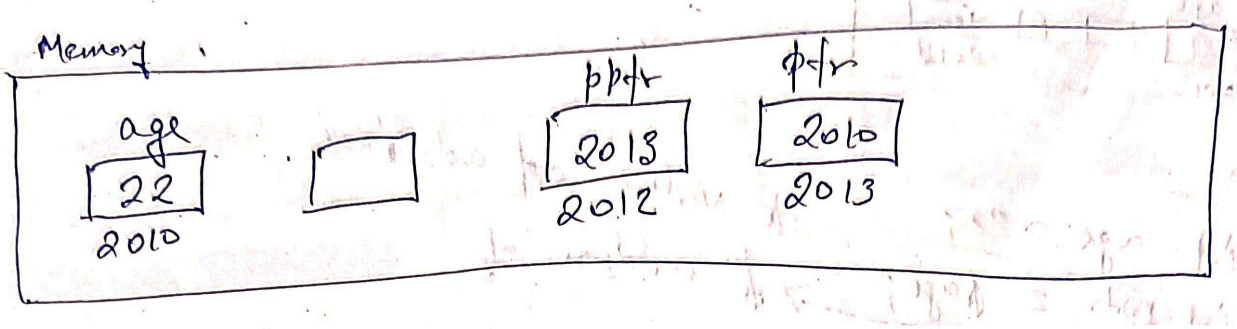
& use for address for.

```
printf("%p", &age);
printf("%p", ptr);
printf("%p", &ptr);

printf("%d", age);
printf("%d", *ptr);
printf("%d", *(&age));
```

* value at that address

Pointer to Pointer:

A variable that stores the memory address of another pointer.



Syntax:

```
int **pptr;
char **pptr;
float **pptr;
```

Pointers in Function Call:-

- Call by value
  - We pass value of variable as argument
- Call by reference
  - We pass address of variable as argument.

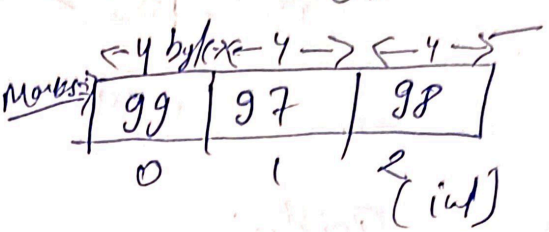
Arrays:-

collection of similar data types stored at contiguous memory locations.

Based indexing

Syntax:-

```
int marks[3];
char name[10];
float price[2];
```

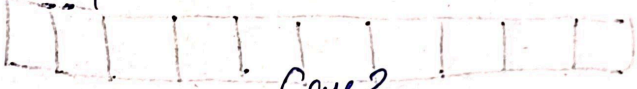


Input & Output:-

```
Scanf("%d", &marks[0]);
Printf("%d", marks[0]);
```

Pointer Arithmetic:-

* Pointer can be increamented & decremented



case 1

```
int age = 21;
int *ptr = &age;
ptr++;   // address increment 4 bytes.
```

case 2

```
char sfar = 'a';
char *ptr = &sfar;
ptr++; // address increment 1 byte.
```

case 2:

```
float price = 20.00;
float *ptr = &price;
ptr++;   // address increment 4 bytes.
```

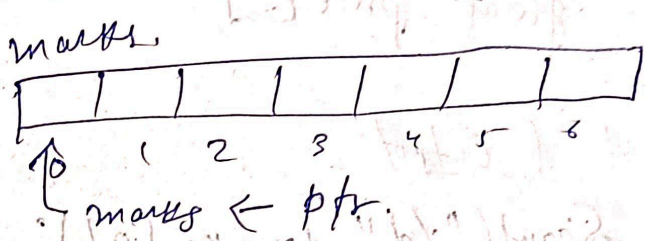
☆ We can also subtract one pointer from another

```
int ptr1 = &age1;
int ptr2 = &age2;
ptr2 - ptr1 ⇒ void.
```

☆ We can also compare 2 pointers.

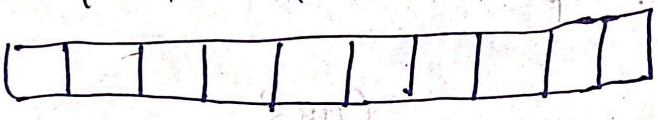
Array is a Pointer:-

```
int *ptr = &arr[0];
   OR          same
int *ptr = arr;
```



Traverse an Array:-

```
int aadhar[10];
int *ptr = &aadhar[0];
```



Arrays as function Argument.

```
void printNumbers(int arr[], int n)
        or
void printNumbers(int *arr, int n)

printNumbers(arr, n);
```

rev code

```
for(int i=0; i<n/2; i++){
    firstval = arr[i];
    secval = arr[n-i-1];
    arr[i] = secval;
    arr[n-i-1] = firstval;
}
```

Multidimensional Arrays.

2D Arrays.

```
int arr[][] = {{1,2},{3,4}};  //Declaration
```

// Access.

```
arr[0][0]
arr[0][1]
arr[1][0]
arr[1][1]
```

#Ideal

```
int main(){
    int tables[2][10];
    SforTable(tables, 0, 10, 2);
    SforTable(tables, 1, 10, 3);

    for(int i=0; i<10; i++){ //0 to 10
        printf("%d\t", tables[0][i]);
    }
    printf("\n");
    for(int i=0; i<10; i++){ //0 to 10
        printf("%d\t", tables[1][i]);
    }
    printf("\n");
    return 0;
}

void SforTable(int arr[][10], int n, int m, int number){
    for(int i=0; i<m; i++){ //0 to 10
        arr[n][i] = number*(i+1); // 2, 4, 6, 8, 10...
    }
}
```

Strings:

A character array terminated by a '\0' (null char) null character denotes string termination.

Example

```
Char name[] = {'S', 'U', 'R', 'A', 'J', '\0'};
Char class[] = {'A', 'R', 'Y', 'A', '\0'};
```

Initializing Strings:-

```
Char name[] = "SURAJ";
Char class[] = "ARJA";
```

What happens in Memory?

```
Char name[] = {'S', ' ', ' ', ' ', ' ', ' '};
Char name[] = "SURAJ";
```

name

S	U	R	A	J	\0
2000	2001	2002	2003	2004	2005

Code:- main fun

```
Char fname[] = "Suraj";
printStr(fname);
Void printStr(Char arr[]){
  for (int i=0; arr[i] != '\0'; i++){
    printf("%c", arr[i]);
  }
  printf("\n");
}
```

String Format Specifier:-

"%s"

```
Char name[] = "SURAJ";
printf("%s", name);
```

Important

scanf() cannot input multi-word strings with spaces

Here,
gets() & puts() come into picture.

String Functions:

gets(str) -> Dangerous & Outdated.
input a string (even multiword)

puts(str)
output a string

fgets(str, n, file)
Stops when n-1
chars input or new
line is entered.

String using Pointers:-

char *str = "Hello world";
Store string in memory & the
assigned address is stored in the
char pointer str.

str -> array -> ptr

int arr[10];
int *ptr.

char *str = "Hello world"; // can be reinitialized.

char str[] = "Hello world"; // cannot be reinitialized.

Standard Library Functions:- <string.h>

1 strlen(str)
count number of characters excluding '\0'

2. strcpy (newStr, oldStr)
   copies value of old string to new string

3 strcat (firstStr, secStr)
   · concatenates first string with second string.

4. strcmp (firstStr, secStr)
   compares 2 strings & returns a value.

   0 → string equal.
   positive → first > second (ASCII)
   negative → first < second (ASCII

Structures :-

a collection of values of different data types. It is user defined.

Example :-

For a student store the following :-

name (String)
roll no (Integer)
cgpa (float)

Syntax :-

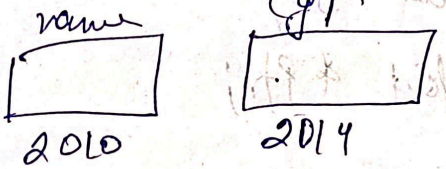
```
struct student {
    char name [100];
    int roll;
    float cgpa;
};

struct student S1;
S1.cgpa = 7.5;
```

Benefits of using Structures :

* Saves us from creating too many variables
* Good data management organization.

Structure in Memory :-

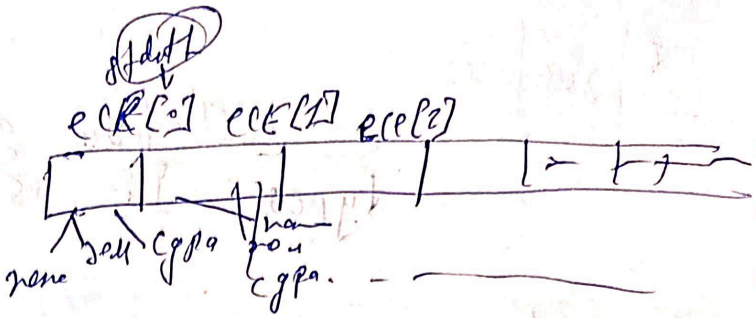
```
struct student {
    char name [100];
    float cgpa;
};
```



Structures are stored in contiguous memory locations.

Array of Structures:-

```
Struct Student ECE[100];
Struct Student COE[100];
Struct Student IT[100];
```



Access

```
IT[0].roll = 200;
IT[0].cgpa = 7.6;
```

Code:-

```
#include <stdio.h>
#include <string.h>
struct student {
    int roll;
    float cgpa;
    char name[100];
};
int main() {
    struct student ece[100];
    ece[0].roll = 1664;
    ece[0].cgpa = 9.2;
    strcpy(ece[0].name, "Suraj");
    printf("name = %s \n", ece[0].name);
    printf("roll = %d\n", ece[0].roll);
```

Initializing Structures

```
struct student s3 = {0};
struct student s2 = {"Arya", 123, 8.2};
struct student s1 = {"Suraj", 98, 7.9};
```

Pointers to Structure:-

```
struct student s1;
struct student *ptr;
ptr = &s1;
```

Arrow operator:-

(*ptr).code ⟷ ptr->code

Passing Structure to function:-

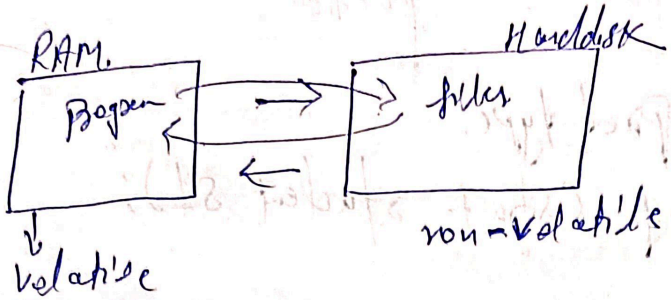
```
// function prototype.
void printInfo (struct student s1);
```

typedef keyword

used to create alias for data types.

```
typedef struct computerEngineeringStudent {
    int roll;
    float cgpa;
    char name [100];
} coe;

coe student1;
```

File Io:



FILE:- container in a storage device to store data

- RAM is volatile
- Contents are lost when program terminates.
- files are used to persist the data

Operation on File:-

create, open, close, Read, write

Type of files:-

Text File
textual data
.txt, .c

Binary File
binary data
.exe, .mp3, .jpg

File Pointer:->

FILE is a (hidden) structure that needs to be created for opening a file.

A file ptr that points to this structure is used to access the file.

File *fptr;

Opening a file:-

FILE *fptr;

fptr = fopen ("filename", mode);

read
write
...

Closing a file

fclose (fptr);

File opening Modes:-

"r"	open to read
"rb"	open to read in binary
"w"	open to write
"wb"	open to write in binary
"a"	open to append

? Check if a file exists before reading from it.

Reading from a file:-

```
char ch;
fscanf (fptr, "%c", &ch);
```

Writing to a file.

```
char ch = 'A';
fprintf (fptr, "%c", ch);
```

Read & write a char.

fgetc (fptr)

fputc ('A', fptr)

EOF (End of file)

fgetc returns EOF to show that the file has ended

Dynamic Memory Allocation.

It is a way to allocate memory to a data structure during the runtime.

We need some functions to allocate & free memory dynamically.

Functions for DMA:

a. malloc() memory allocation
b. calloc() continuous "
c. free()
d. realloc() re-allocation

malloc()

memory allocation.
takes number of bytes to be allocated.
& returns a pointer of type void

```
ptr = (int*) malloc(5 * sizeof(int));
```

calloc()

continuous allocation.
initializes with 0

```
ptr=(int*) calloc(5, sizeof(int));
```

free()

we use it to free memory that is allocated using malloc & calloc

```
free(ptr);
```

realloc()

reallocate (increase or decrease) memory using the same pointer & size.

```
ptr = realloc(ptr, newsize);
```