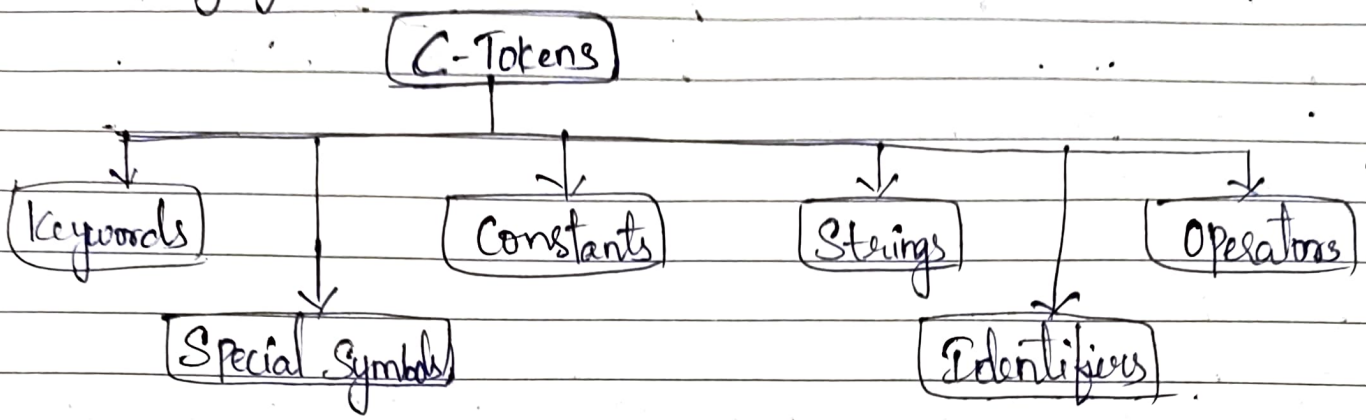
C Language



Keywords :-

Keywords are defined those Variables which have special meanings and are predefined in the C libraries

Ex: For, if, else, main etc

Constants :-

Constants (or) literals are like Variables, but the difference is that, the value of the constants are fixed.

Once declared, they can not be changed.

Syntax:

Const data type Variable_name (or) Const datatype *Variable_name

Strings:

Strings are defined as a collection of characters ~~Such~~ defined in form of any array and end with null ~~character~~ character which describes the end of the String to the compiler.

Syntax:

Char String_name[length_of_string];

"Array of characters are call as strings"

Identifiers:

Identifiers are defined as the names that we declare in a program in order to name a value, Variable, function, array and etc..

Ex: int x=10;

Operators:

- Athematic Operators
  - Used to perform Mathematical Calculations such as add, Subtract
    +, -, *, /, % → Remainder.

- Increment / Decrement Operators
  Increment / Decrement operators are used to inc/Dec the value of the Variable by a specific (one) number.
  ++, --

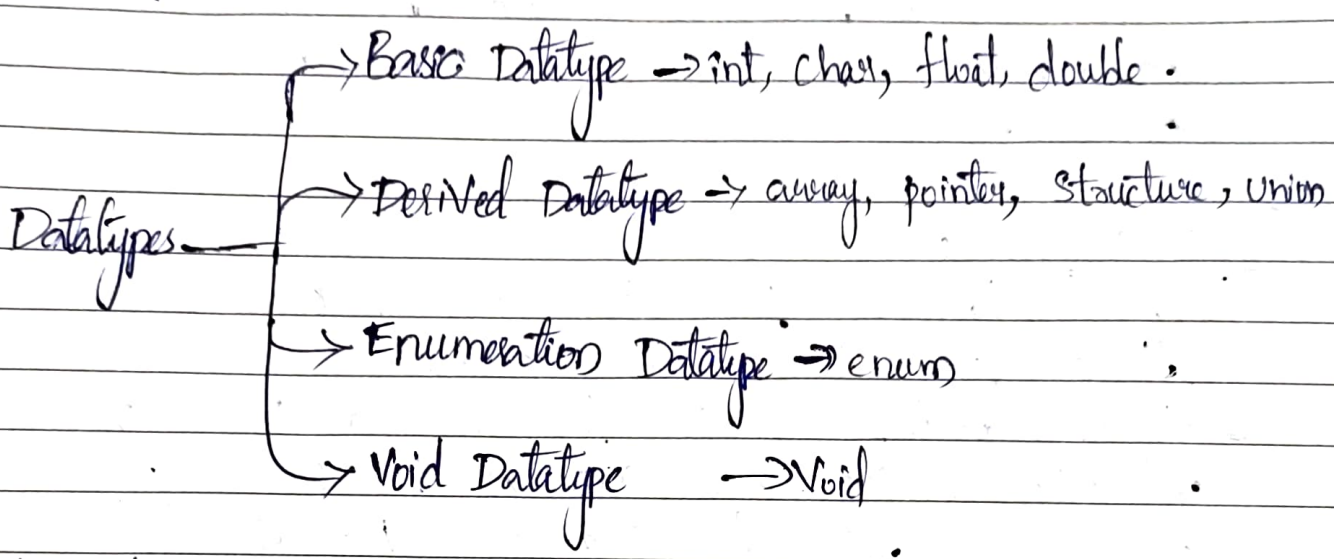
- Assignment Operators
  - Used to assign the values for the Variables
    =, ==, *=, +=, -=, /=, %=

- Relational Operators
  - Operator that tests (or) defines relation b/w two entities (or) Variables.
    >, <, =, !=, =>, <=

- Logical Operators:
  - Used to perform logical operations on the given expressions.
    Logical AND(&&), OR(||), NOT(!)

- BitWise Operators → Used to perform bit Operations.
  &, |, ~, ^, <<, >>

* Data Types



* Variables:

↳ Variable is defined as the reserved memory space which stores a value of a definite datatype.
The value of the variable is not constant, instead it allows changes.

Types of Variables

- local Variables
- Global Variables
- Static Variables
- automatic Variables
- External Variables.

① Local Variable:

↳ Any variable that is declared inside a code block (or) a function and has the scope confined to that particular block of code (or) function

Ex:

```
Void Ed() { int local_val = 10; }.
```

② Global Variable:
↳ Any Variable that is declared outside a code block (or) function and has the scope across the entire program.

③ Static Variable:
↳ Any Variable that is declared using keyword Static is called Static Variable.
- Static Variables retain the declared value through out the entire execution of the program and will not be changed b/w multiple function calls.

④ Automatic Variable:
↳ It can be declared by using the keyword auto.
By default, all the variables declared in C are in automatic Variables.

```
Void main()
{ int local var = 10; // automatic default
  auto int auto = 20; // Automatic Variable.
```

⑤ External Variable:
↳ These Variables are declared by using the "extern" keyword.
- we can share a Variable in multiple C Source files by using an External Variable.

Ex: extern int external var = 10; // external Variable.

* Pre Processor Directives

High level language (User language)

Pre-Processor

Pure High level language

Compiler

Relocatable Code

Assembler

Assembly level code

Loader/Linker

pure Machine level language.

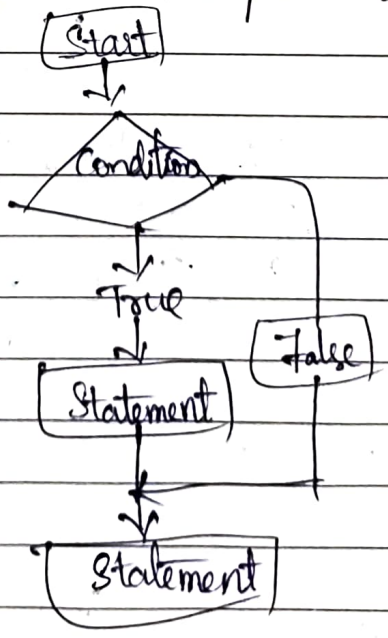
* Control Statements:

↳ It enables us to specify the flow of program control

① If Statement

↳ If statement is a conditional statement that, if proved true, perform an operation (or) display information

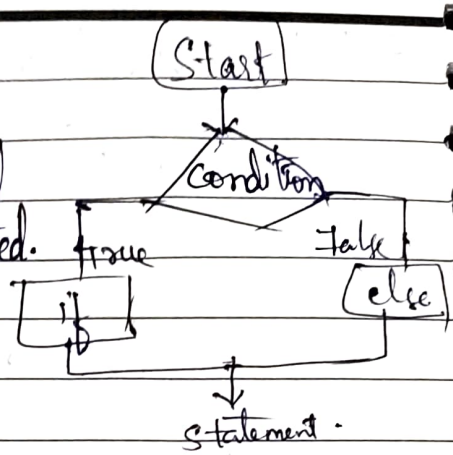
```
if
{
// statement
}
```



② If-else Statement

↳ if it is a conditional statement
if true the → if block will be executed
if false → else block will be executed.

```
if (Condition)
{
 // Statement
}
else
{
 // Statement
}
```

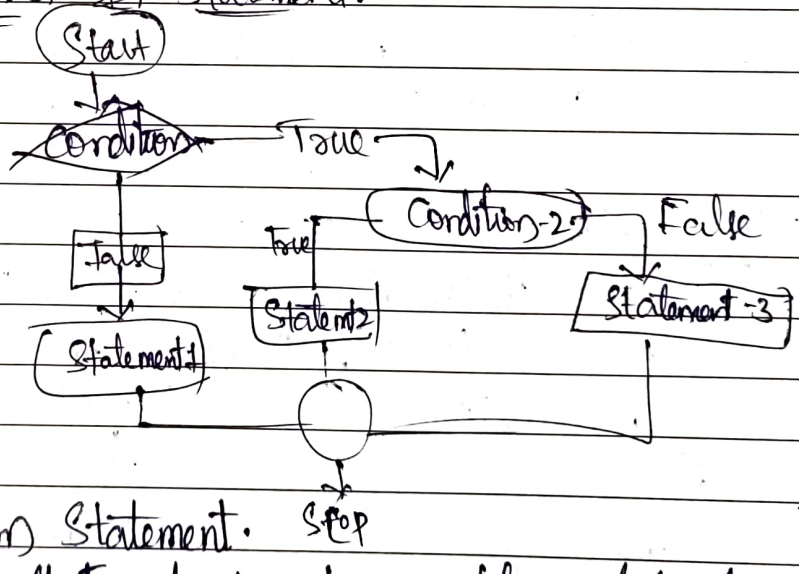


③ If-else ladder Statements:

↳ it is a Conditional Statement
↳ It has multiple ele-if Statement blocks.

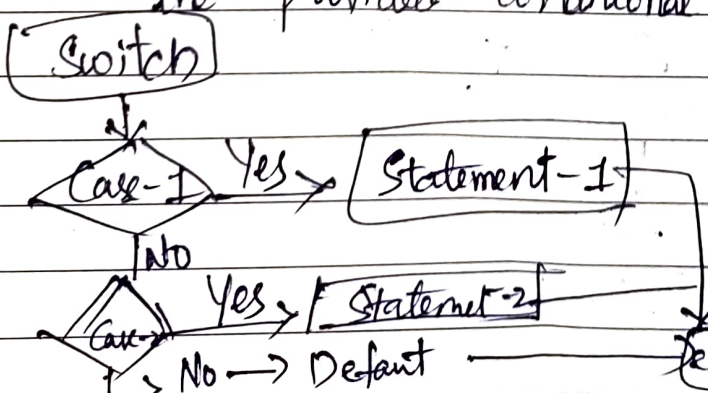
```
if (Condition)
{
 // Statement
}
else-if (Condition)
{
 // Statement
}
:
else
{
 // Statemat
}
```

④ Nested If Statements:



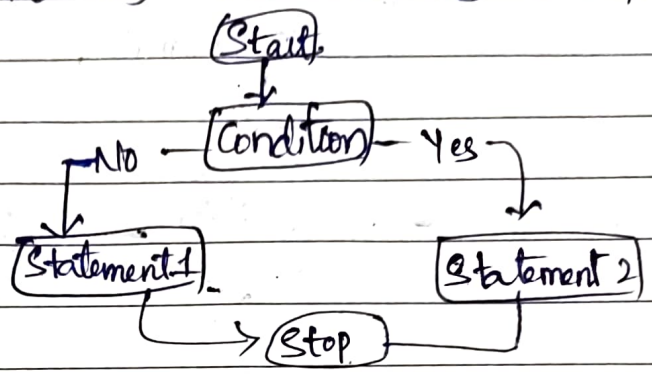
⑤ Switch Statement

↳ It is a Selection Statement.
↳ switch statement that checks for possible match to the provided conditional against the available cases.



* Ternary Control Statement
↳ It is a conditional statement.
The control checks the condition and executes either of the two statements.

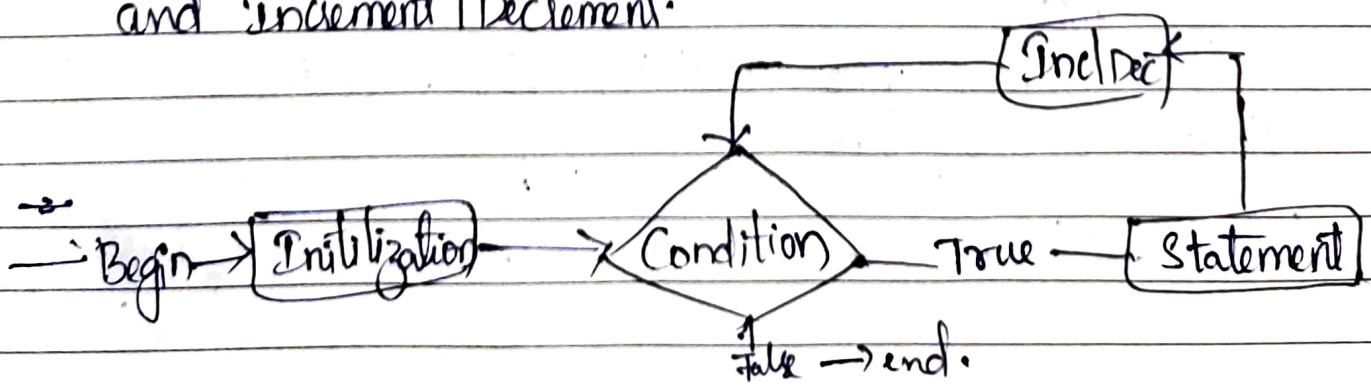
Var = (cond1 > cond2) ? a : b



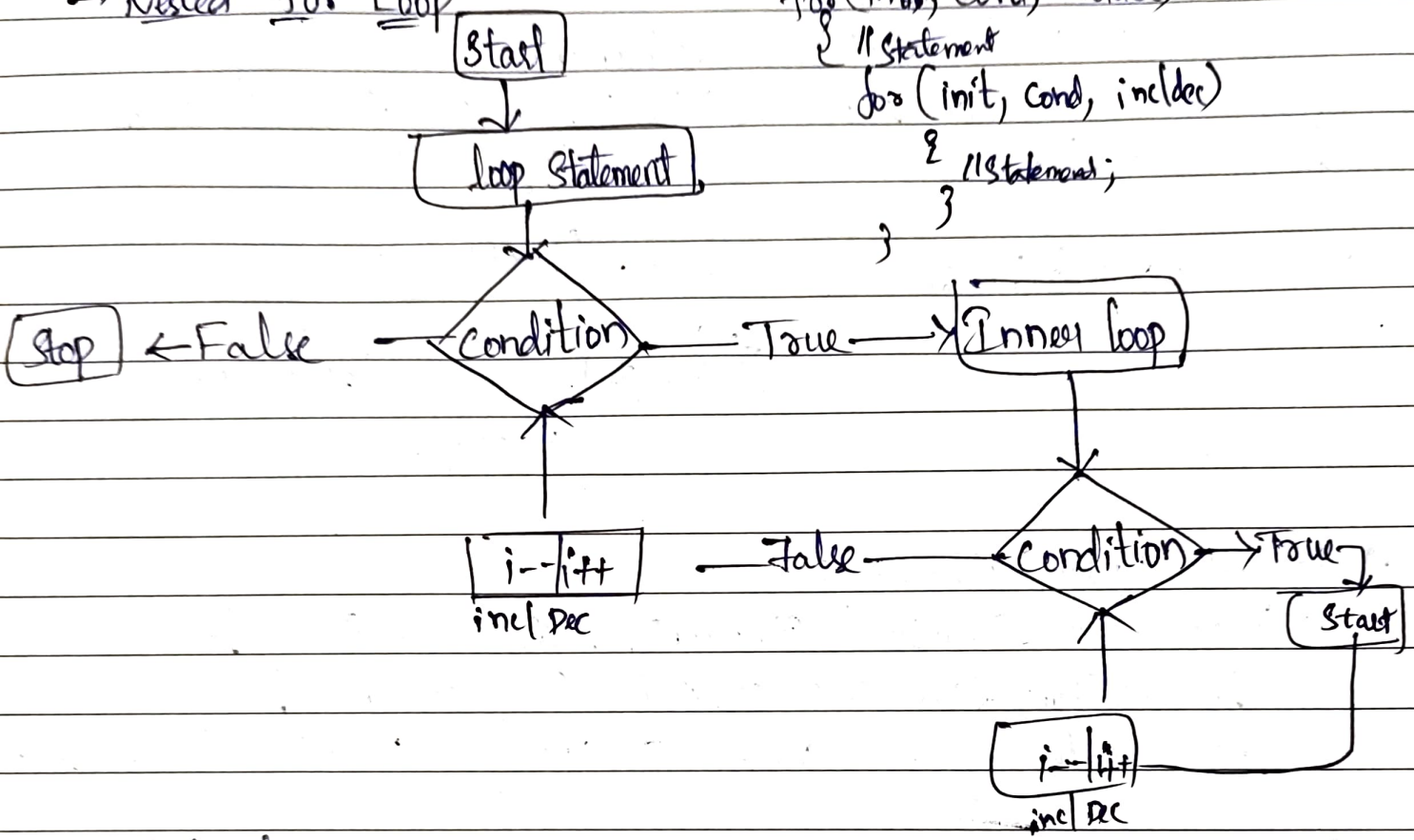
* Break Control Statement:
↳ It is a conditional statement, that is designed to exit the control from the current code segment to the next code segment when a specific condition is satisfied.

* Loops Control Statements:
↳ It is used to perform looping operations until the given condition is true.
Control comes out of the loop statement once condition fails.
3 Types of loops
- do while
- while
- for

→ For:
↳ It is a precise loop which has Initilization, Condition and Increment / Decrement.



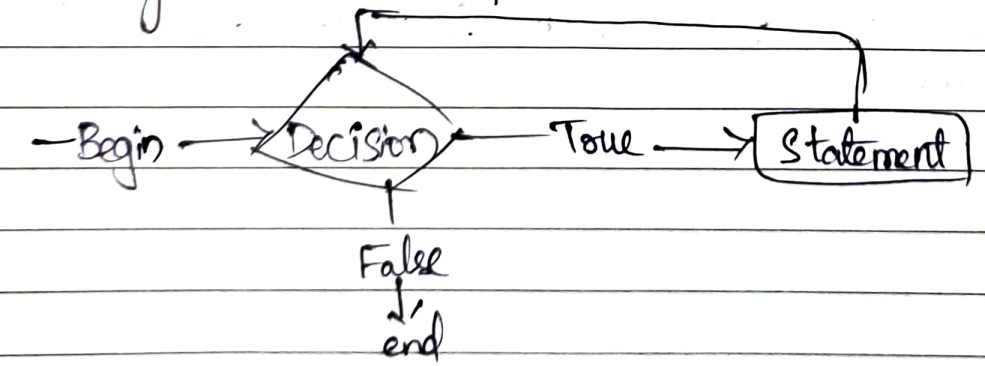
→ Nested for Loop




```
for(init; Cond, incldec)
{ // Statement
  for (init, Cond, incldec)
  { //Statement;
  }
;
```

⇒ while loop

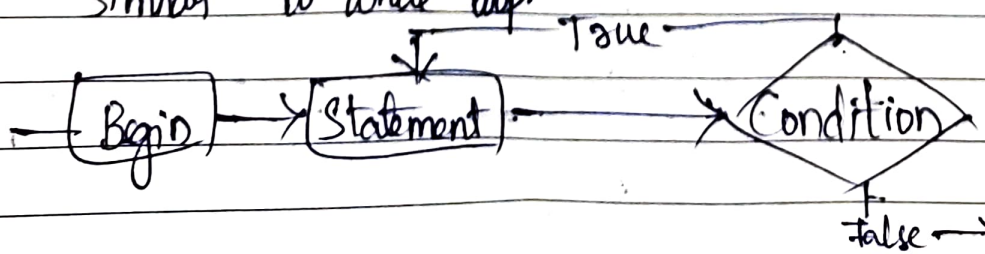
↳ The loop will executes itself repeatedly, until a given Bodean expression (or) a Condition is true.




```
while (condition).
{
//Statement
; ~~Cond++~~ incldec
```

⇒ DO-while loop

↳ It checks the condition after executing the statements. Similar to while loop.




```
do
{ statement;
  incldec
}
while (Cond)
{ }
```

Void Pointer
↳ It is a pointer that has no associated datatype with it.
→ A void pointer can hold address of any type and can be typecast to any type.

Wild Pointer
↳ It is also called as uninitialized pointer.
Because they point to some arbitrary memory location and may cause a program crash.

NULL Pointers
↳ It is a special type of pointer that doesnot point to any memory location.
→ If we assign a NULL value to a pointer, then the pointer is considered as NULL Pointer.

Dangling Pointer
↳ Once had a valid address but no more now pointing to that address.
→ It is a pointer which points to some non-existing memory location

Memory leaks:
↳ The memory leak occurs, when a piece of memory which was previously allocated by the programmer
→ Then it is not deallocated properly by programmer.
→ That memory is no longer in use by the program

* Pointers

↳ A Variable which stores the address of another Variable

→ This Variable can be of type int, char, array, function or other pointer.

→ The size of pointer depends on the architecture.

→ The pointer is declared using * (Asterisk Symbol).

```
int n=10
int *P=&n
```

Advantages of pointer

- Return multiple Values from a function
- Access any memory location in computer's memory.
- Parameter passing
- Register Access
- Dynamic Memory Allocation.

7 Rules of pointer

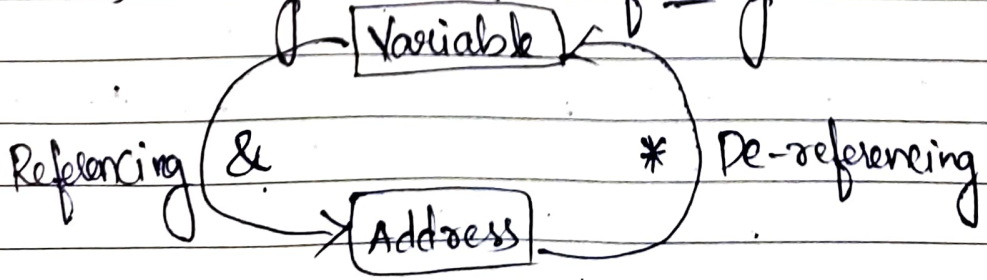
Rule 1: Pointer is an Integer

- Any number we write on a Datatype is an Integer
- Any number we write on a Address bus is an pointer.
  - Address location of Pointer is always an Integer.

~~Address pointers holds~~

```
i=5
P=5 // warning
P=(int*) 5 ✓
```

Rule 2: Referencing and De-referencing



Rule 3: Pointer means Containing

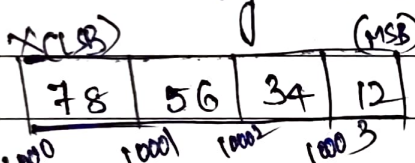
Pointer pointing to a Variable = Pointer Contains the Address of the Variable.

Rule 4: Pointer types

(Endianess) → Byte ordering within a word of System

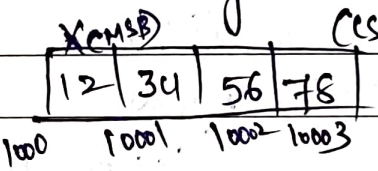
Little Endianess

To store lowest Significant bit in lowest memory address



Big Endian

To store most Significant byte in lowest memory address.



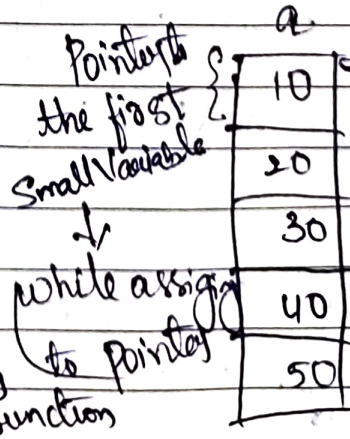
- The type of a pointer represents its ability to perform read or write operations on number of bytes (data) starting from address its pointing to.

- Size of all different type pointers remains Same.
  ex: Sizeof(char*) == Sizeof(long*) any pointer size match dep 64 bit → 8.

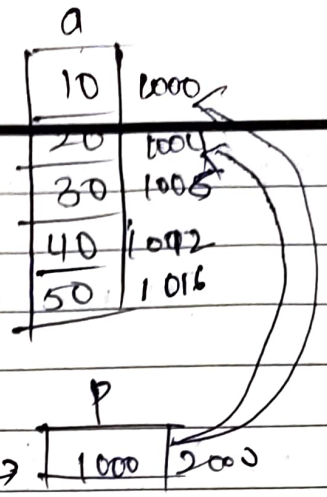
Rule 5: Pointer Arithmetic

Array Interpretation.

```
int main()
{
int a[] = {10, 20, 30, 40, 50};
}
```



int a[] = {10, 20, 30, 40, 50}
int *p = a



p = p+1

p = p+1 * sizeof(datatype)

p = p+1 * sizeof(int)

p = 1000+4

p = 1004

*(p+i) → a[i]

a[i] ↔ *(a+i)

*(p+i) → *(a+i)

p = a

p = p+2;

p = p+2*4 → 1000+8 → 1008

so

p+2 → 1008 → &a[2]

*(p+2) → *(1008) → a[2]

*(p+2) → a[2]

Rule 6: Pointing to nothing

int *p; → Wild pointer

↳ Not initilize

So its an act of initilizating pointer to 0.

(general implementation dependent) at definition

#define NULL ((void*)(0))

Null pointer → If a pointer is initialized with null pointer constant.

↳ De-referencing a NULL pointer is illegal and will lead to System crash.

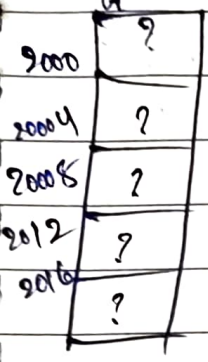
Pointer Value of NULL (or) Zero = NULL Addr = NULL Pointer = pointer to nf

Void Pointer

↳ A pointer to generic data. (which can be used the way we want, Since it doesn't have types attached to it

Rule 7: Static Vs Dynamic Allocation

Static Allocation (Stack)

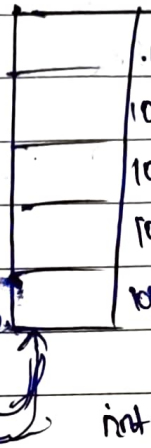


- Named Region
- Managed by Compiler
  - Allocates the need memory internally
  - Done while defining the variable

int a[5]

(memory in stack)

Dynamic Allocation (Heap)



- unnamed Region
- Managed by User
  - Allocate & Deallocates as needed
  - Done at runtime using malloc & free

int *p = malloc(5 * sizeof(int));

→ malloc() → memory is not initialized

Void *malloc(Size_t size)

↳ Allocates memory from heap,

↳ On success it returns the allocated memory else, NULL

→ Calloc()

↳ Clear allocation

Void *calloc(size_t nmemb, Size_t size);

↳ Allocates the memory from heap, the size requested is in bytes (*) number of memory blocks needed.

→ The allocated memory is filled with 0's.

→ Realloc()

↳ Void *realloc(Void *Ptr, Size_t size);

↳ Modifies the size of an already allocated memory either by malloc (or) Calloc.

→ free()

Void free(Void *Ptr)

Free the memory which should have allocated with malloc(), Calloc(), realloc

(6+7)

* Functions

↳ Function is basically a set of statements that takes inputs, perform specific task and produces output

Syntax: Return_type function_name (Set of inputs);

Advantages
- → Reusability
- → Abstraction

→ Declaration:

int fun(int, char);

1. Return Type of function	fun
2. Name of function	int
3. Number of parameters	2
4. Type of Parameters	int, char

→ Definition:

↳ Function definition consists of block of code which is capable of performing some specific task.

Parameters and Arguments

↳ is a Variable in the declaration and definition of function

Argument

↳ is the actual Value of the parameter that gets passed to the function.

→ Pointers.

Note: Parameters is also called as Formal Parameters
Argument is also called as Actual parameters.

```
int add(int, int)        → Parameters (or) Formal Parameters
int main()
{
  sum = add(m, n);       → Arguments (or) Actual parameters
}
```

Actual Parameters → The Parameters Passed to a function
Formal Parameters → The Parameters received by a function.

→ Call by Value:

↳ Here Values of actual parameters will be copied to formal parameters and these two different parameters store values in different location

→ Call by Reference:

↳ Here both actual and formal parameters refers to same memory location.
Therefore, any changes made to the formal parameters will get reflected to actual parameters

Here instead of passing values, we pass address.

Types of Functions

- Function without arguments and without return value.
- Function without arguments and with return value
- Function with arguments and without return value.
- Function with arguments and with return value.

* Static Functions

→ By default all functions in C are global.

→ Static functions are restricted to the files where they are declared

* Function pointers

↳ Function pointers are like normal pointers but they have the capability to point to a function

```
int (*ptr)(int,int)
```

Structures

↳ A structure is a user-defined data type that can be used to group elements of different types into a single type.

Syntax

```
Struct structure_name{
{
  /* Members */
};
```

Ex

```
Struct Student
{ int id;
  Char name[10];
  Char address[40];
};
```

→ The memory is not associated with structure until we define a variable of that type.

→ The size of the defined memory would depend on the members put in structure.

Advantages

- Data Encapsulation
- Contiguous memory allocation for all the members.
- Minimize no. of parameters while passing to the functions.

Accessing Members of Structure

↳ we can access members of the structure using dot(.) operator.

* Structure - Function (Pass by Value)

- Pass by Reference (using pointer)
  & → operator.

* Structure Padding

↳ When an object of some structure type is declared then some contigous block of memory will be allocated to structure members.

```
Struct abc {
    char a; } 4 bytes
    char b; }
    int c; → 4 bytes
} var;
```

sizeof(var) → 8

```
Struct abc {
    Char a;
    int b;
    char c;
} var;
```

sizeof(var) → 12

→ For 32-bit architecture → 4-bytes at a cycle.
For 64-bit architecture → 8-bytes at a cycle.

* Structure packing

↳ Because of structure padding, size of structure becomes more than the size of the actual structure.
Due to this some memory will get wasted.

→ We can avoid the wastage of memory by simply writing
#Pragma pack(1) → Fetches 1 byte for cycle

* To no of cycles we use ⇒ pragma
#Pragma pack(4) → Fetches 4 bytes per cycle (32-bytes)
#pragma pack(8) → Fetches 8 bytes per cycle (64-bytes)

* To Save memory we use "packing".
* To Reduce no. of cycles we use "Padding"

* Structure - Padding

↳ It is a process of adding few extra bytes (based on structure definition) in b/w the members

→ The Compiler does this for data access efficiency by aligning the memory allocation on the address which ranges small power of 2, and this process is generally called data alignment.

Bit fields

↳ Bit Fields is a Feature that allows us to break the memory (integral types) in Bits!

↳ Points to be kept in mind.

- The Smallest Size of a bit field is a byte, which can be broken into 8 individual bits max.
- The largest Size would be based on length modifier.
- The default member type is Signed integer.

* Unions

↳ Unions is a user defined datatype but unlike structures, union members share same memory location

→ The memory allocation is based on the biggest member available in union.

→ Used to save space.

Syntax

```
union union_name
{
  /* Members */
};
```

→ If we make changes in one member then it will be reflected to other member as well.

* Typedef

↳ Type Definition, a feature to give a new name / alias to an existing type.

Key points:

- Can be applied on the only datatypes.
- Recommended to provide meaningful name.

* Enum
  ↳ Enumerated Data Type
- Enhances code readability.
- Better alternative to Symbolic constants and avoid magic constants.

Keypoints
↳ Useful to name an integral constants
→ Every member has a unique value unless and until we assign
- The default value of the first member starts with 0
- The consecutive members gets a value from previous member increment by 1.

Syntax

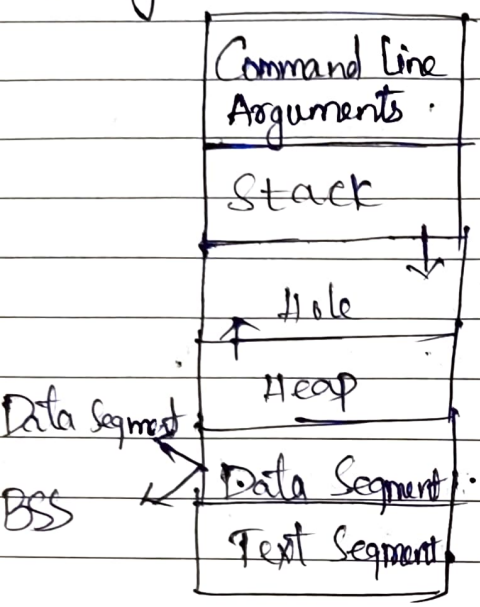
```
enum enum_name
{
  [* Members *]
};
```

→ The size of enum → is same as the size of Integer.

→ Enum names are automatically initialized by the Compiler.

Data Segment: Obtained when our application have global (or) statically modified local variables.
- Data Segment → One for initilized variables
- BSS → One for uninitilized variables.

Storage classes



Text Segment
↳ Also called as Code Segment.

→ Hold the a Section of code part of .o file, which will be loaded into memory.

- This Section is read only and Cannot be modified at runtime

Stack:
- Created when the function is called.
- The place for automatic variables.
- will have dedicated register to keep track of push & pop.
- Stack frame may be
  - Place for local (automatic Variables)
  - Return Address
  - Parameters.

Heap:
↳ Generally a larger block of memory which could be requested dynamically.

→ malloc(), calloc() and realloc() will return an address of a requested block from this Section.

→ will vary dynamically based on the applications running in your System when they free the requested block.

*Strings

↳ String literal (or string constant) is a sequence of characters enclosed within double quotes.

- %s is a placeholder.
- Contiguous sequence of characters
- Stores all the printable ASCII characters.
- Every string ends with a '\0' NULL character.

⇒Storing Strings

↳ String literals are stored as an array of characters.

2 Types of Strings

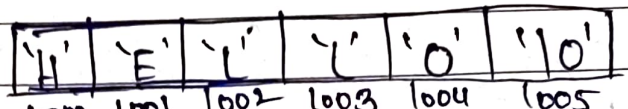
- Modifiable strings:
  - Can be changed at runtime.
  - Would be stored in stack, Heap and Data Segment.
- Constant Strings:
  - Generally known as String literals
  - Get stored in rodata
  - If we want to chang const strings we get Segmentation fault.

*Initilization of Strings

```
Char S1[3] = {'H', 'i', '\0'};
Char S2 [] = {'H', 'i', '\0'};
Char S3 [3] = "Hi";
Char S4 [] = "Hi";
Char *S5 = "Hi";
```

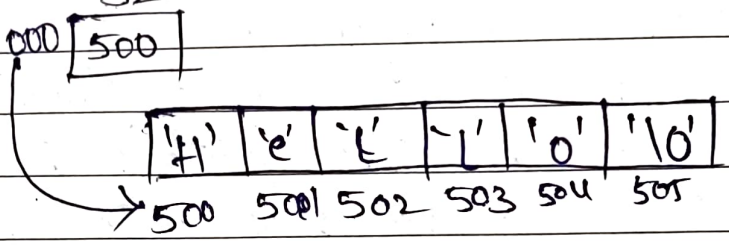
* Memory Allocation → Modifiable

Char S1[] = "Hello";



Constant

Char *S2 = "Hello";



* Scanning a string

Char str[2];

→ then illegal access ↳ stack smashing

→ Scanf("%s", str) → Scans only the first word.
"Hello world"

Printf("%s", str) → Hello.

→ gets(str); // Hello world → it accepts the spaces.
↳ Funtion is depreciated (Not recommended)
Doesnot have control on bytes to be stored.

(Read from a file)

→ fgets(str, 12, stdin);
↳ less than buffer size the new line also stored in buffer.

→ Selective Scan:

Scanf("%[^\n]s", str); ✓
↳ Recomended.

* Passing string to a function

* Juxtaposing
  ↳ Clubbing two strings together into one.

```
Printf("Hello" "world\n");
```

* Standard String Functions

Library → #include<string.h>

① Strcpy → string copy Function

Prototype : char* strcpy(char* destination, const char* source);

Strcpy used to copy a string pointed by source (including NULL character) to the destination (character array)

```
Strcpy(Destination, Source, Sizeof(destination));
```

② Strlen → String length Function.

Prototype : size_t Strlen(const char* str);
  ↳ size_t is an unsigned integer type of at least 16-bits

Strlen() → Used to determine the length of given string.
  ↳ It doesn't count NULL