① → 'C'

"Volatile", Keyword in 'C'.

1) The volatile keyword is intended to prevent the compiler from applying any optimizations on objects that can change in ways that cannot be determined by the compiler.

2) Example:

```
        u_int32_t *p = (u_int32_t *)0x20000004
                       u_int32_t value;
  while (1)
  {
     value = *p;
     if (value) break;
  }
```

memory address ↑

→ If you optimize the code, this value might not be read everytime from memory, as compiler thinks, the value is not changed in the scope (or) loop. but there might be some global variables (or) Interrupt service routines (ISR), (or) processes changing the value.

→ If we use, "volatile" keyword, then the value of the variable will be read from the memory, no matter what ~~may~~ might be the optimisation level.

↓

→ `uint32_t volatile *p = (uint32_t *) 0x20000004`

STORAGE CLASSES IN 'C'

- auto
- extern
- static
- register

~~Storage class~~ ~~identifier~~

storage-class	data-type	identifier;
{auto, extern..}	int	a

→ Datatype, represents size of the variable, ~~how~~ what type of data type is allowed

② → 'c'

→ <u>storage class</u>, represents ① what is the default value
② scope of the variable
③ life time of the variable
④ where it gets memory allocation
⑤ where we can access

→ If, we don't write a storage class, according to the context, area of declaration of variable, it, compiler will write a "default" storage class.

→

<u>Different Scopes:-</u>

1) BLOCK → "{ }" → b/w the braces.
2) METHOD
3) PROGRAM

```
void main()
{
     int a;      → local variable
                 → function scope
                 → holds garbage values
  [illegible]
}
```

PROGRAM SCOPE:-

```
int a = 10;        ──> program scope
                   ──> Go
void main()
{
  int a = 20;
                        ↗ //20
  printf("%d", a);     a = a+10;
}
void check(){
  printf("%d", a);     //10
}
```

AUTOMATIC:-

- If we don't specify any storage class, by default "auto" will be taken

• Keyword	:	Auto
• Memory	:	RAM
• default value	:	Garbage value
• life time & scope	:-	Declaration either inside block (or) main [Local variable] → Can be accessed only within block (or) method

$\textcircled{3} \rightarrow$ 'c'

(auto int a; ⟶ Error → ✗)

```
void   main()
{
    auto  int  a = 10;   → method scope

    { auto  int  a ;   → Block scope
        printf ("a: %d", a);  → Garbage value
    }
    printf ("a : %d", a);  → 10
}
```

REGISTER :-

Keyword : register

Memory : Inside CPU Registers

Default value : Garbage value

- These are also local variables, similar to "Automatic" variables, cannot be declared globally.

- Accessing register variables will be much "faster" than auto variables.

• Variables which we access repeatedly ~~in~~, like loop counters, are used to declare as register variables.

•

STATIC :-

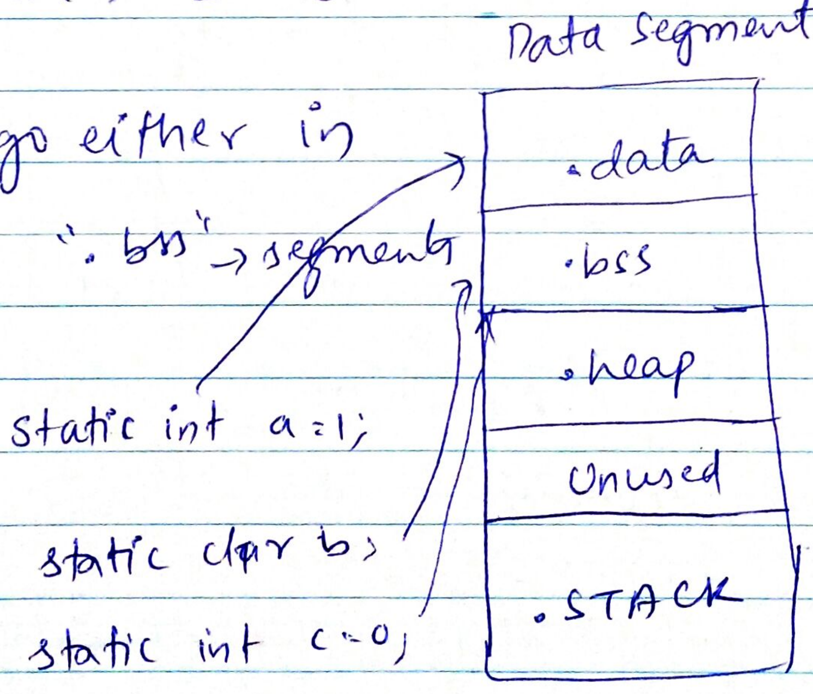
static int i;

Scope : In the same block in which the variable has been declared

life-Time : Until the completion of the program the variable will be alive.

Default-value : "ZERO"

• They can go either in ".data" (or) ".bss" → segments



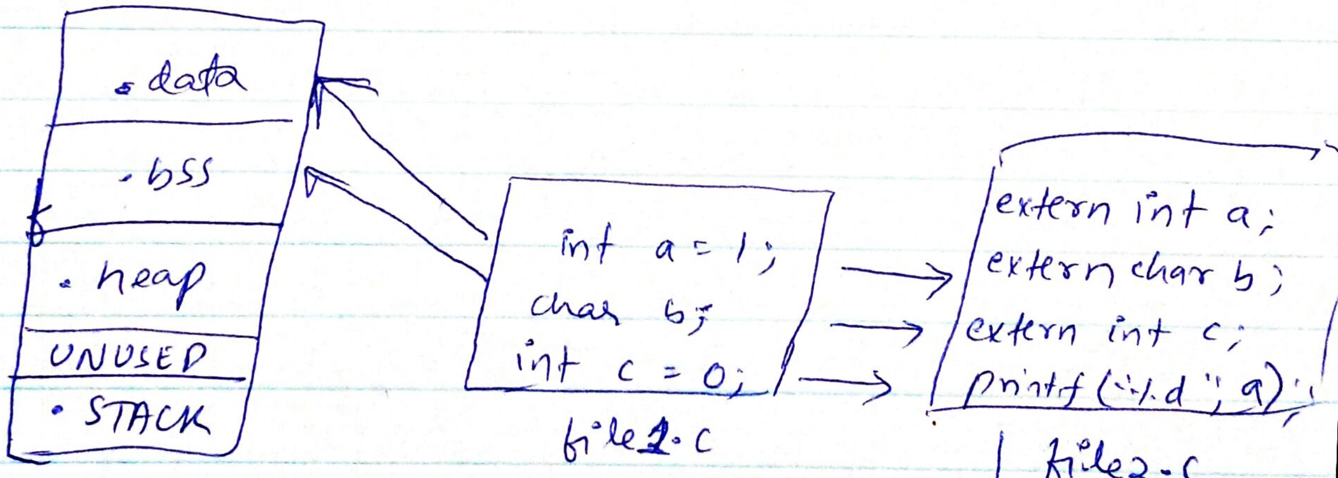
```
static int a=1;
static char b;
static int c=0;
```

(only in the function in which it is declared)

• It has "LOCAL SCOPE" ~~oft~~ ACCESS"
but will be "ALIVE" till the end of the program"

EXTERN :-

- Declares a global reference defined in another file to be visible to the current file
- It is like, "declaration" of ~~a~~ a function, which defined in some header file, so that current file knows that, there exists this function you can use it.



This file got to know about, Variables a, b, c, while using extern.