VIDHIT
TECHNOLOGIES

BUILD.MARKET.SECURE.SCALE

Campus to Corporate

Your tested road for placement success

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

About Speaker

- Ranjit Wagh
- **M. Tech**: Software Systems, BITS Pilani
- ~14 years of experience
- **Automotive** and **Semiconductor** industries
- **System Software Designer** at KPIT. (Worked with EdgeQ, NXP, Xilinx, Visteon, etc)
- **Core Competencies**: C, ARM SoC bringups, Linux Device Driver Development, Bootloaders, Android BSP, Firmware, etc

Navigating the AI Frontier: Cognitive Impacts of LLM Use

Exploring the intricate relationship between AI tools and human cognition. This presentation delves into a recent **MIT study**, examining how Large Language Models shape our thought processes, creativity, and learning.

Study Design: Unpacking the Experiment

An MIT Media Lab experiment, June 2025.

Participants:

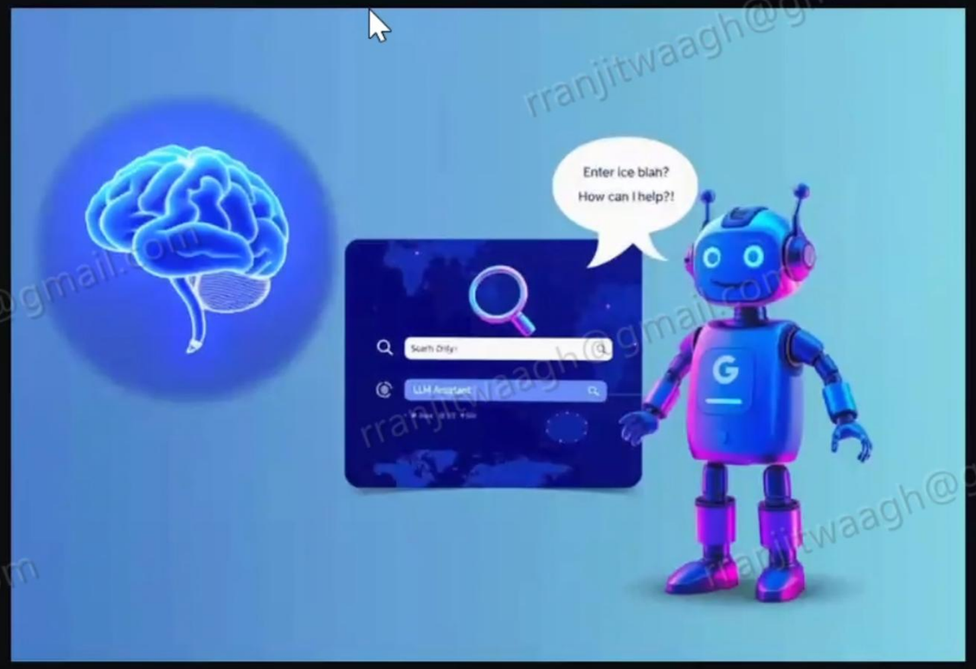
54 individuals, divided into three groups.

Groupings:

Brain-only, Search Engine, LLM Assistant.

Task:

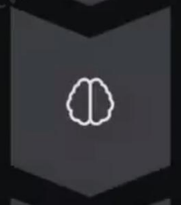
Essay writing across three sessions.



Crossover Session: A Unique Insight

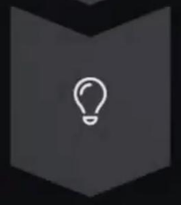
The Shift:

LLM users switched to Brain-only.



The Reversal:

Brain-only users tried LLM tools.



Purpose:

Observe persistent cognitive effects.

This critical fourth session enabled direct comparison of cognitive states with and without AI tools, revealing long-term impacts.

Measuring Outcomes: Quantitative Analysis

Researchers employed advanced techniques to quantify neural activity and essay quality:

Neural Connectivity:

EEG measured brain networks.

Essay Originality:

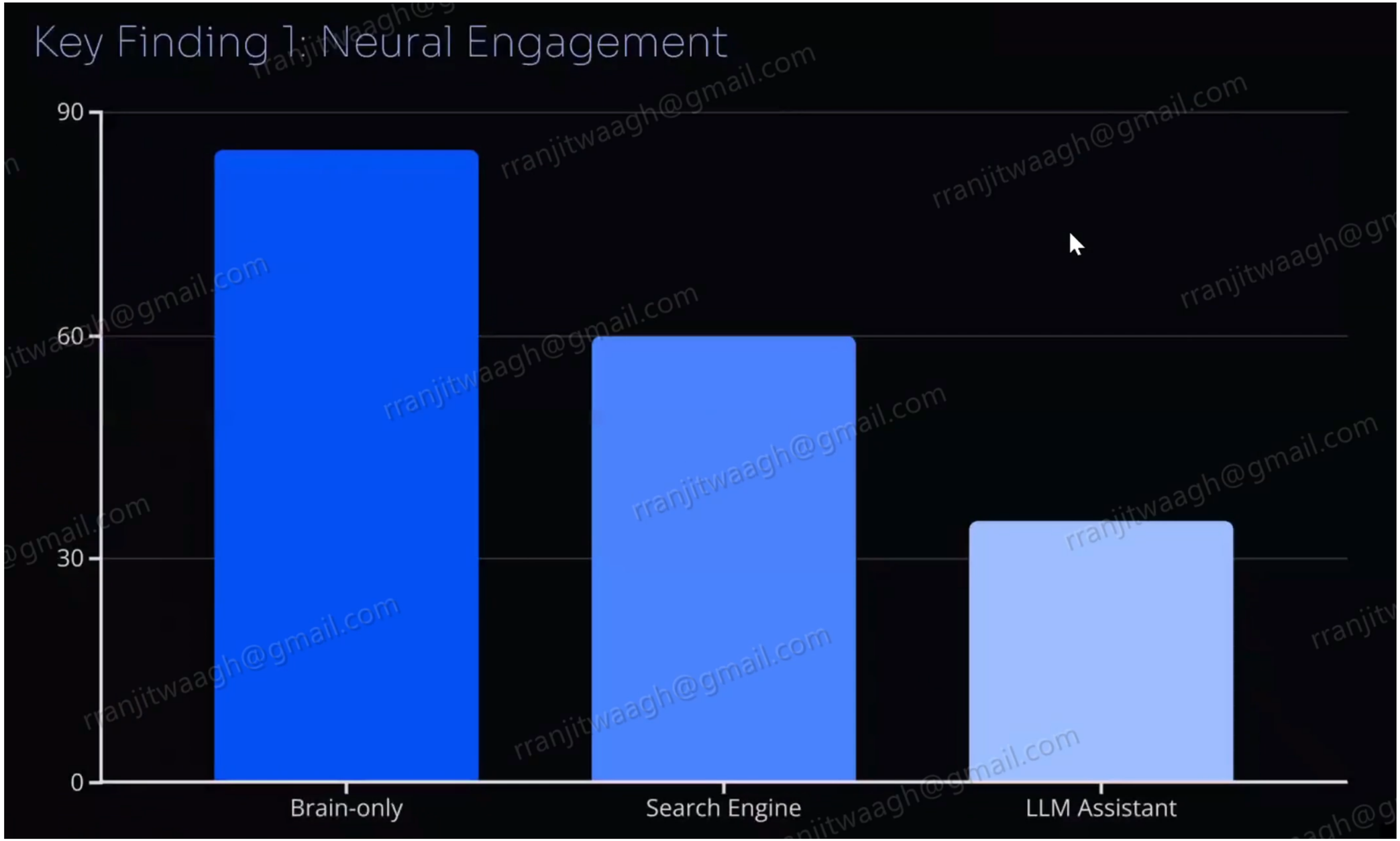
NLP and human/AI scoring.

Memory Recall:

Assessed after writing tasks.

Sense of Ownership:

Self-reported by participants.



Key Finding 2: Behavioral Outcomes

LLM-Assisted Essays:

- Less original content.
- Poorer memory recall.
- Reduced sense of ownership.

Persistent Effects:

- Impacts remained post-LLM use.
- Even after reverting to tool-free writing.

These outcomes highlight a potential "cognitive debt" from heavy AI reliance.

Implications: Mitigating Cognitive Debt

Critical Thinking:

Reduced by over-reliance on AI.

Creativity:

Potential for diminished innovation.

Long-Term Learning:

Risk of shallower knowledge retention.

The study suggests a need for balanced, reflective LLM use in educational settings. Encouraging active engagement, not passive consumption, is key.

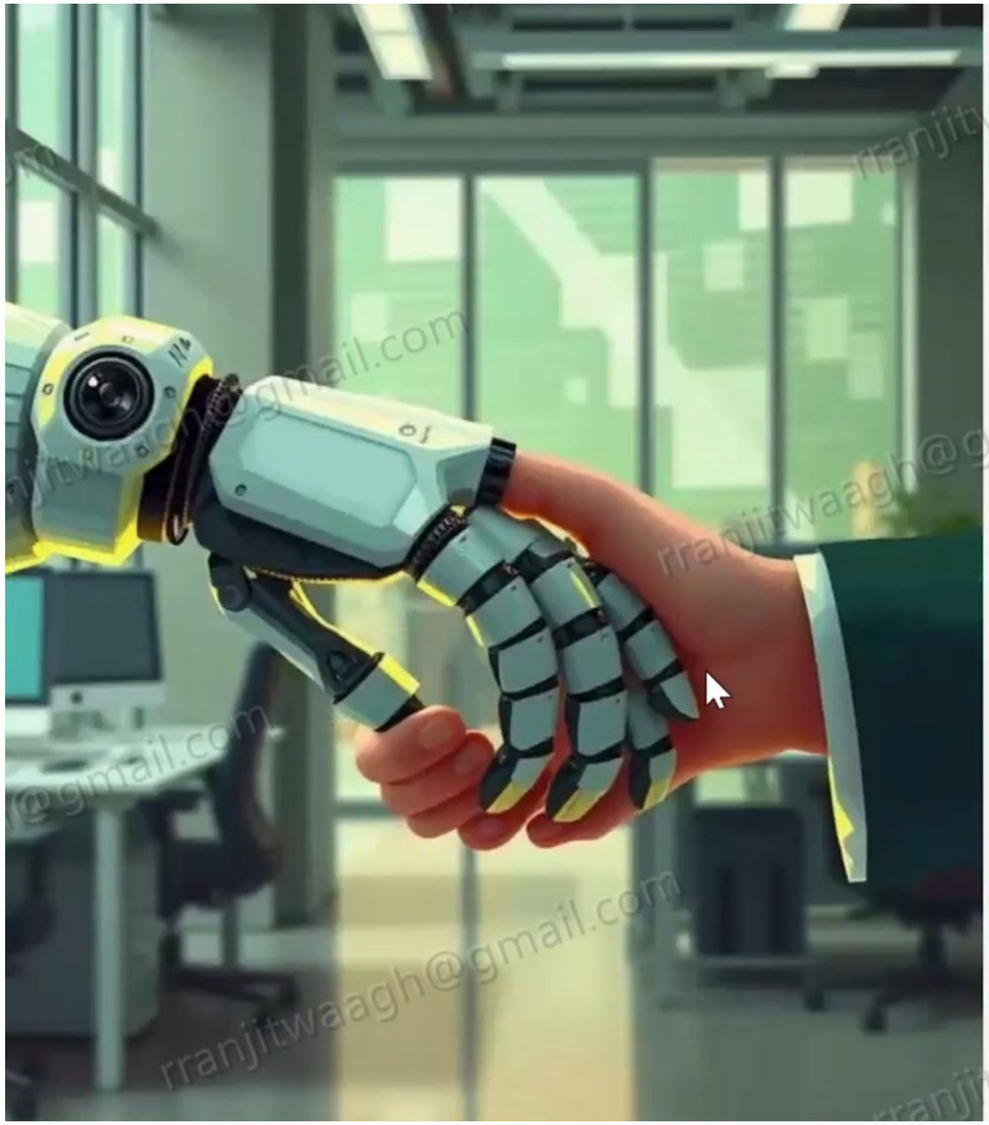
Recommendations: Balanced AI Integration

Promote active learning, critical evaluation of AI outputs, and the development of core cognitive skills.

Integrate AI as a tool for augmentation, not replacement.

The Evolving Role of Developers in the AI Era

Explore how Artificial Intelligence is reshaping the landscape for emerging talent in software development and why programmers remain indispensable.



AI's Impact on Hiring

Automation Trends

Many companies are leveraging AI to automate repetitive coding tasks, leading to a perceived reduction in the need for developers for basic coding.

Efficiency Gains

AI tools can significantly boost productivity, allowing senior engineers to focus on complex problem-solving rather than routine coding.

The Core Debate: AI vs. Developers

The Question Arises

With AI's growing capabilities in code generation, a critical question emerges: do companies still need programmers for foundational development tasks?

Beyond Code Generation

While AI can write code, it lacks the contextual understanding, critical thinking, and creative problem-solving skills inherent in human developers.

Thomas Dohmke's Perspective: Adaptability

> "AI can never replace programmers. In fact, fresh talent is more adaptable in the changing world." - Thomas Dohmke, GitHub CEO

This powerful statement from a key industry leader highlights the unique value programmers bring to the evolving tech landscape.

Why Programmers Are Irreplaceable

Innovation & Fresh Perspectives

Programmers bring diverse backgrounds and innovative ideas that can challenge existing paradigms and drive creativity.

Problem-Solving & Learning Agility

Programmers are inherently adaptable, eager to learn new technologies, and quick to pivot in response to evolving project needs.

Shifting Roles and Skill Sets

1

From Coder to Integrator

Programmers will increasingly focus on integrating AI-generated code into larger systems and ensuring its quality and alignment with project goals.

2

Focus on Debugging & Optimization

Their roles will involve refining AI-generated outputs, debugging complex issues, and optimizing code for performance and scalability.

3

Ethical AI & Compliance

Understanding the ethical implications of AI and ensuring compliance with regulations will become a crucial skill for all developers.

Conclusion: A Collaborative Future

The future of software development is not about AI replacing human talent, but about a powerful collaboration between them.

Key Takeaway

Programmers, with their adaptability and fresh perspectives, are essential for innovation and growth in an AI-driven world.

Next Steps

Invest in continuous learning and embrace new skill sets to navigate the evolving landscape effectively.

Did you know Rust, the world's most loved programming language, started because of a broken elevator?

Back in 2006, Mozilla programmer Graydon Hoare came home tired from work.

The elevator in his building was broken yet again. So he had to climb 21 flights of stairs to get to his apartment.

He was probably pretty annoyed about it. But then he found out something that made it even worse: the elevator wasn't actually mechanically broken, it was rather a software bug which was causing it to malfunction.

That really got to him. If a simple software error could make people climb 21 flights of stairs, what other important things were breaking because of bad code?

So Graydon decided to do something about it.

He started working on a new programming language that would prevent these kinds of errors and fix all those crashes and memory problems that make software unreliable.

His idea was pretty simple: make a language so safe that even beginners could write code that actually works.

Today, Rust is everywhere.

Mozilla uses it. So do Microsoft, Google, and Figma, along with countless other companies.

Developers absolutely love it.

Sometimes the best ideas come from unusual places. Who knows what you might create the next time you're forced to take the stairs!

What Do I Need to Qualify into the race?

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

1. Mastery in Core Programming Language (C / C++ / Java (OOP))	
2. Hands-on with Scripting Language (Bash / Python / Perl / etc)	
3. Data Structures (Hands On)	
4. Awareness of OS concepts	
5. GIT (Hands On)	
6. Code integration & Debugging using SDK (Visual Studio/Eclipse/GCC/etc)	

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

Agenda

GIT-Hub Setup

C Programming Mental Model

Toolchains & Compilation Process

Revisiting C Basics

GitHub

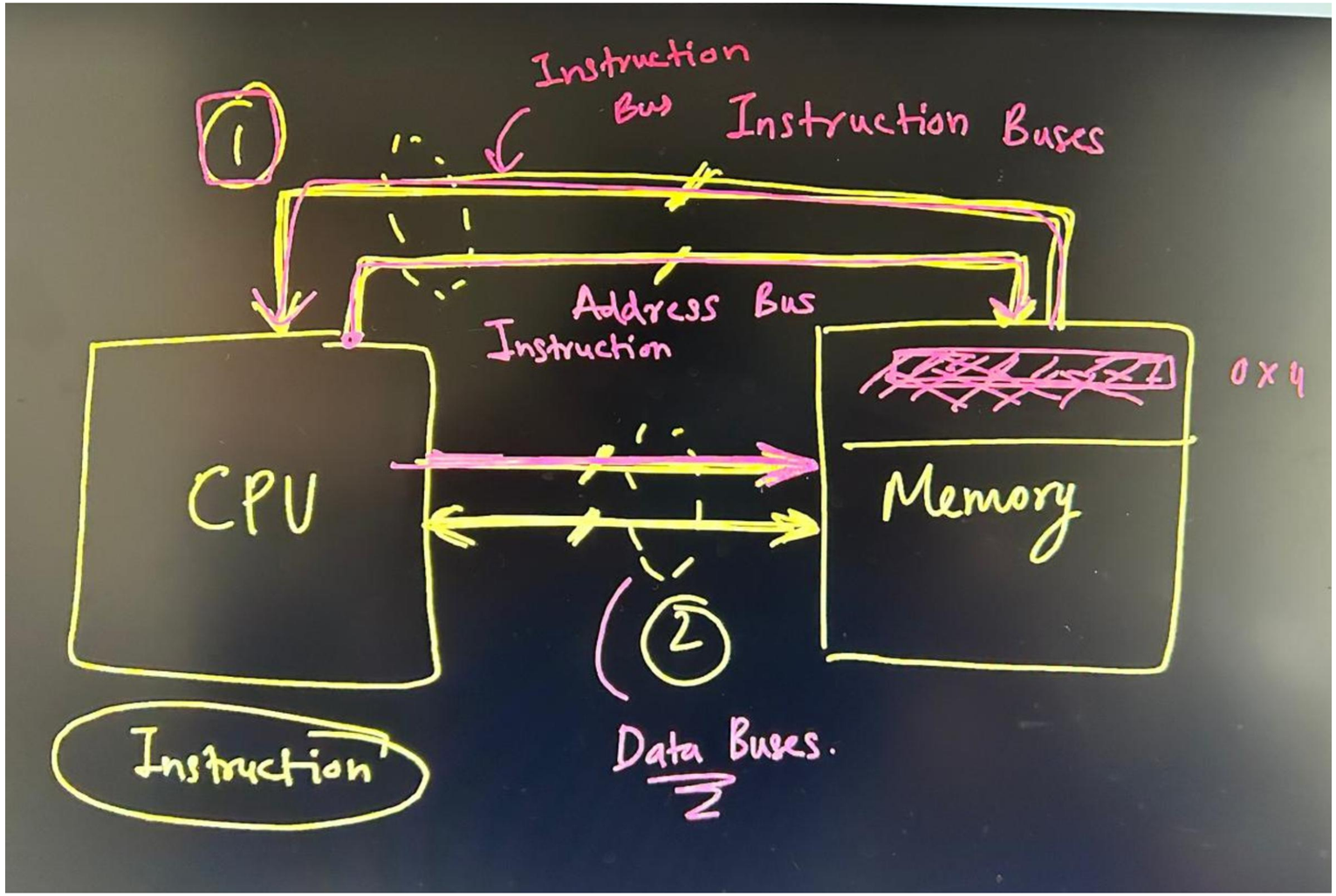
Creating Workspace

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

https://github.com/ranjit27/campus2connect

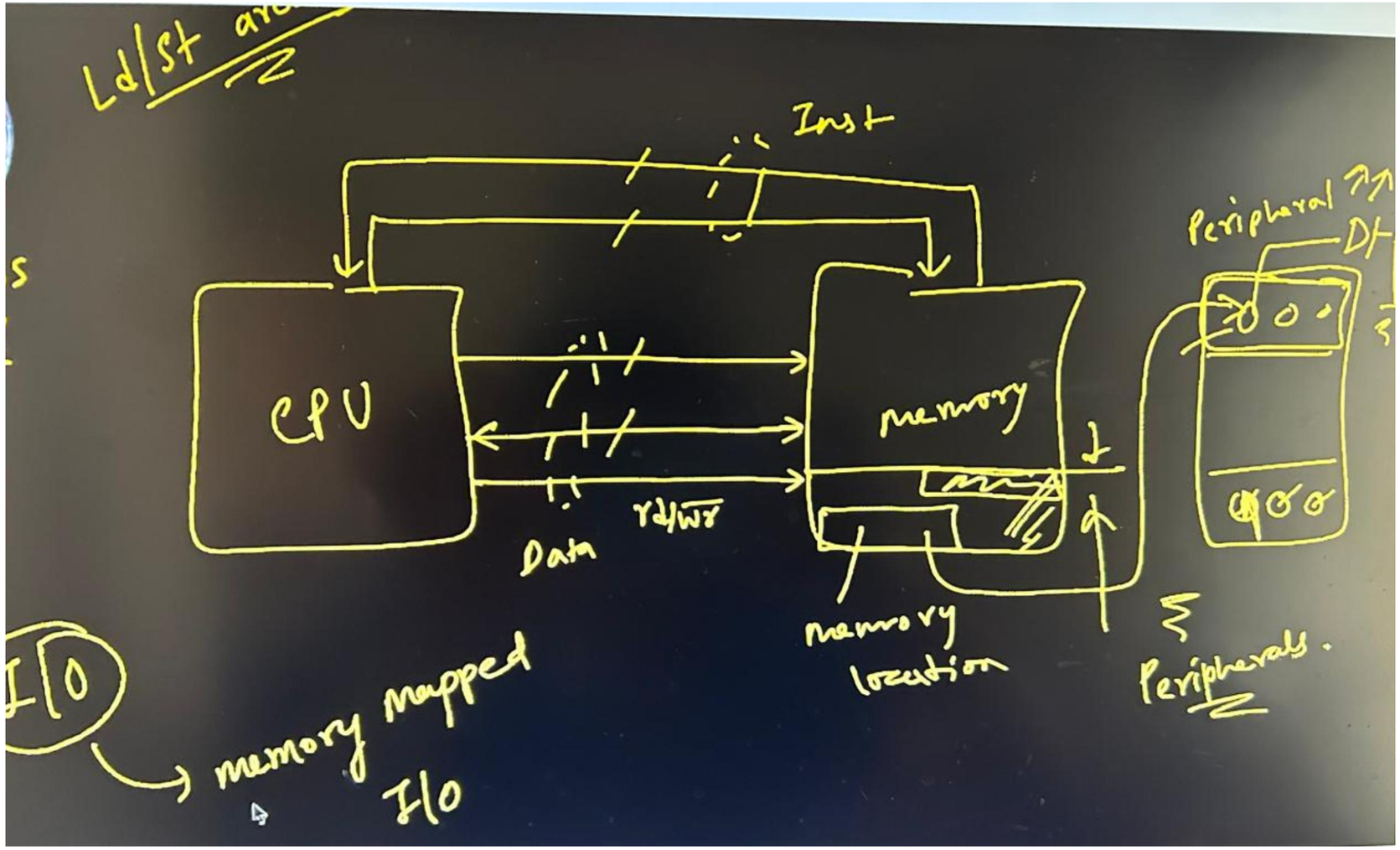
CPR

Mental Model Diagram



CPR

Mental Model Diagram



CPR

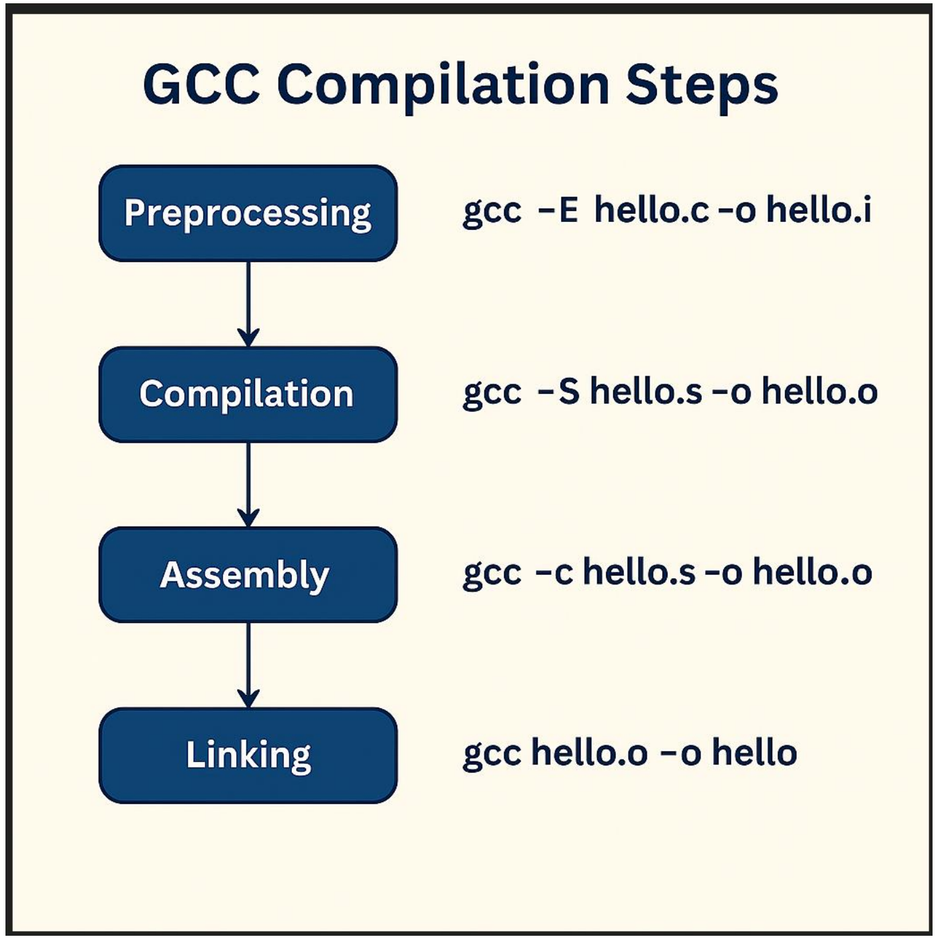
main.c:

```
void main()
{

}
```

What will be the output?

CPR

The Toolchain



CPR

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

Preprocessors

Types of Preprocessor in C

- Macros
- File Inclusion
- Conditional Compilation
- Other directives

CPR

Preprocessors

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

- Macros

Programmers use macros to execute pieces of code inside the macro.
A programmer gives a name to the macro.
And when the given name is encountered by the compiler then the compiler replaces the name with the piece of code which is inside the macro.
The **#define** directive is used to define a macro.

```
#include <stdio.h>

#define LIMIT 4 // defining macro!

int main()

{

        for (int a = 0; a < LIMIT; a++) {

                printf("%d \n",a);

        }

        return 0;

}
```

CPR

Preprocessors

Macros with arguments

We can also pass arguments to the macros. Macros with arguments work similarly as functions.

```
#include <stdio.h>
#define AREA(s) (s * s) // macro with argument
int main()
{
        int s1 = 10, area_of_square;
        area_of_square = AREA(s1);
        printf("Area of square is: %d", area_of_square);
        return 0;
}
```

CPR

Preprocessors

Function like Macros in C

Define macros that work similarly like functions.

```
#include <stdio.h>
#define MIN(a,c) ((a) > (c) ? (c) : (a))
int main(void) {
        printf("Between 30 and 70, minimum number is: %d\n", MIN(30, 70));
        return 0;
}
```

CPR

Preprocessors

File Inclusion

This preprocessor directive tells the compiler to include a file in the program code. A user can include two types of file:-

a. Header or Standard files

These files contain functions like printf(), scanf(). In a single word, it contains predefined functions.
Different header files contain different functions. Like, **string** file contains string handling functions and **stdio** file contains input/output functions.

Syntax:-

```
#include <name_of_the_file>
```

b. user defined files

A programmer can define his/her own header file to divide a complex code into small blocks of code.
Syntax:-

```
#include"filename"
```

CPR

Preprocessors

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

Conditional Compilation in C:

The main purpose of this preprocessor directive is to compile specific parts of the code or skip the compilation of specific parts of the code based upon some conditions.
How to use conditional compilation directives?

With the help of two preprocessing methods, you can implement the conditional compilation directive.
ifdef
endif

Syntax:-

```
#ifdef macro_name
        statement1;
        statement2;
        statement3;
        .
        .
        .
        statementN;
#endif
```

If the macro name is defined, then it will normally execute the block of statements. And if it is not defined, then the compiler will skip the block of statements.
Apart from **#ifdef** and **#endif** directives, there are some other directives such as **#if, #elif, #else** and **#defined** directives.

CPR

Preprocessors

Conditional Compilation in C:

#if, #elif, #else directive in C
You can also use **#else** directive with **#if** directive. If the given expression results in a non zero value, then the codes of conditional are included in the program.

Syntax:-

```
#if expression
        // conditional statements if expression is not zero
#else
        // conditional statements if expression is zero
#endif
```

You can also use nested conditional in your program with the help of **#elif** directive.

Syntax:-

```
#if expression1
        // conditional statements if expression1 is not zero
#elif expression2
        // conditional statements if expression1 is not zero
#else
        // conditional statements if all expressions is zero
#endif
```

CPR

Preprocessors

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

Conditional Compilation in C:

#defined directive in C
Mainly used to check whether a certain macro is defined or not. You can use **#if** directive with **#defined** directive.

Syntax:-
#if defined LENGTH
// body contains code

Uses of C Conditional Compilation:-

- Compile the same source file in different programs.
- Depending on the OS, you can use different code.

CPR

Preprocessors

Other directives in C

There are also two not so used directives available in C.

#undef directive:- Mainly used to undefine an existing macro.

Syntax:-
#undef LIMIT
If you use the above statement, then it will undefine the existing macro LIMIT.

CPR

Compiler / Compilation

2. C Program Compilation

The compiler translates the preprocessed code into assembly language.
It checks for syntax errors and semantic correctness and generates a .s file containing low-level assembly instructions.

Input: Preprocessed file (**.i**)

Output: Assembly code file (**.s**)

CPR

Assembly/Assembler

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

3. Assembly

- The assembler converts assembly code into machine code (binary).
- Produces an object file (.o or .obj).
- **Input:** Assembly file (**.s**)
- **Output:** Object file (**.o**)

CPR

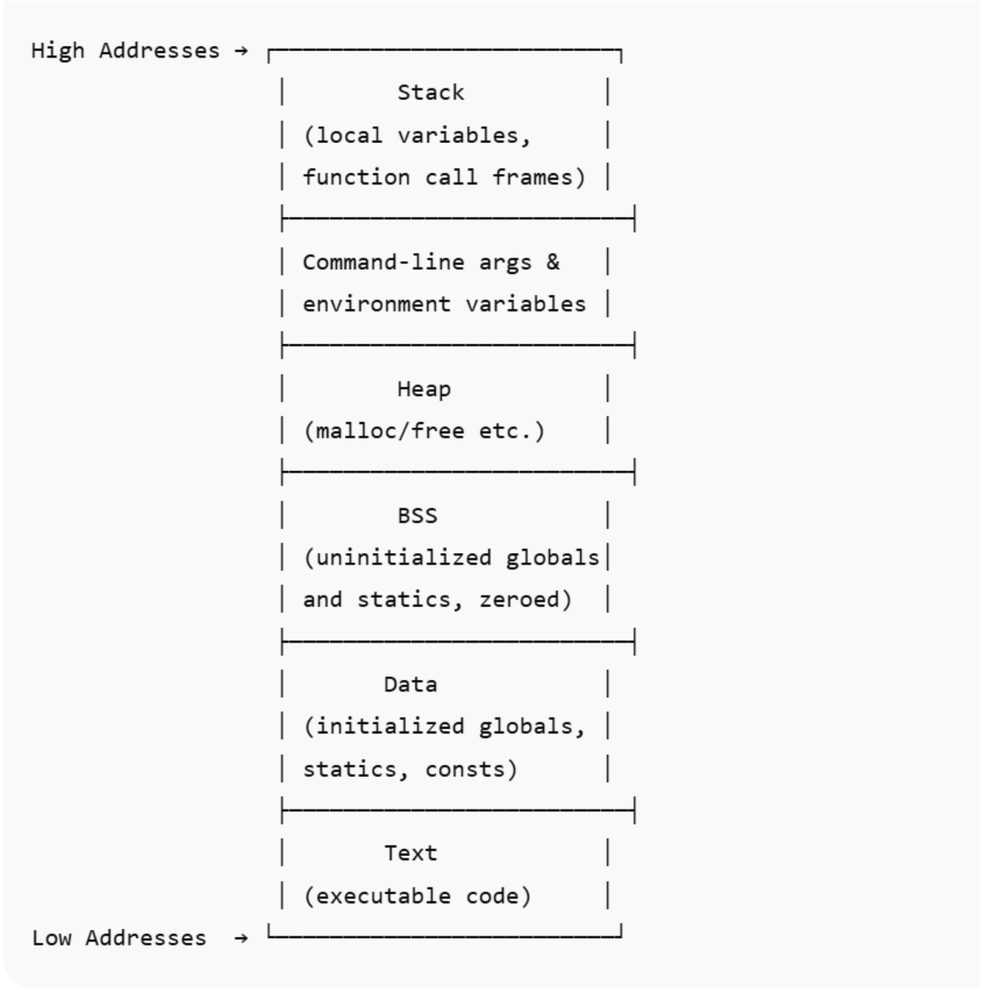
Linker/Linking

4. Linking

- The linker combines all object files and resolves references to external functions (like **printf, scanf**).
- It also links in necessary system libraries, such as the standard C library (**libc**), and produces the final executable.
- **Input:** Object file(s) **(.o)**
- **Output:** Executable file (**.exe**, or just binary)

CPR

Memory Layout of a C Program



Segment Growth Directions

- Heap grows upward (toward higher addresses).
- Stack grows downward (toward lower addresses)
- If they collide, the program runs out of memory

Exploring at Runtime

You can inspect a compiled binary's memory layout using commands like **size** (for .text, .data, .bss) and **readelf -l** or **objdump -h** (for detailed ELF section info)

Why Does This Matter?

- Helps to understand and diagnose issues like **segmentation faults**, **stack overflows**, and **memory leaks**.
- Critical for performance, security (permissions), and systems programming.

Summary

Each region in the memory serves a specific purpose:

- **Text**: code
- **Data/BSS**: global/static variables
- **Heap**: dynamic memory
- **Stack**: function calls and locals
- **Args/Env**: command-line/environmental inputs

CPR

Loader / Loading

What Is a Loader?

- The loader is part of the operating system that takes over once the linker has produced an executable.
- Its job is to prepare that executable for the CPU by loading it into memory and setting everything up to run

CPR

Loader / Loading

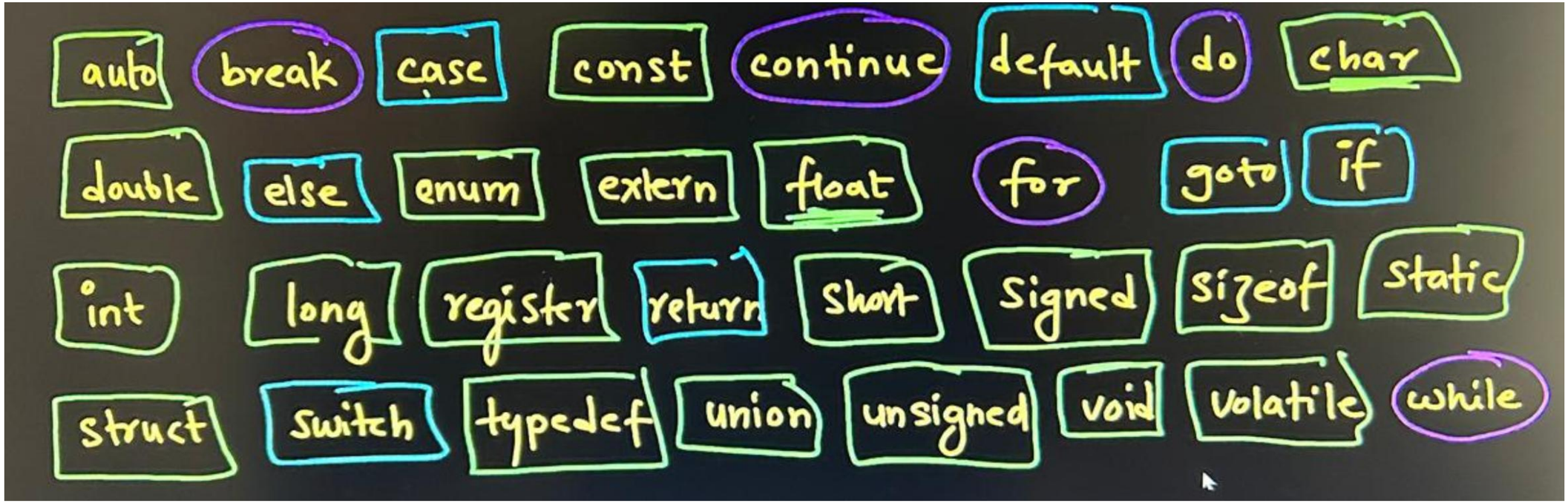
VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

Key Steps in the Loader Process

1. **Reading & Parsing the Executable Header**
   It reads the file's header (like ELF on Linux, PE on Windows) to understand which segments (code, data, etc.) need to be loaded and where.
2. **Allocating Memory**
   The loader allocates memory (in RAM or virtual address space) for each segment—like .text, .data, and .bss—plus the heap and stack.
3. **Loading Code & Data**
   It memory-maps or copies executable contents into the allocated areas.
4. **Relocation**
   If the executable isn't loaded at its preferred base address, the loader adjusts all addresses (patches memory references) using relocation entries.
5. **Symbol Resolution & Dynamic Linking**
   For dynamically linked programs, the loader finds and loads required shared libraries, resolves external symbols (functions, data), and initializes tables like PLT/GOT at runtime.
6. **Initializing Execution Context**
   Sets CPU registers (stack pointer, program counter), environment variables, and other runtime contexts before jumping to the program's entry point.
7. **Control Transfer**
   Finally, it jumps to the program's start address (e.g., _start), transferring control to the runtime—the OS's last step before your main() takes over.

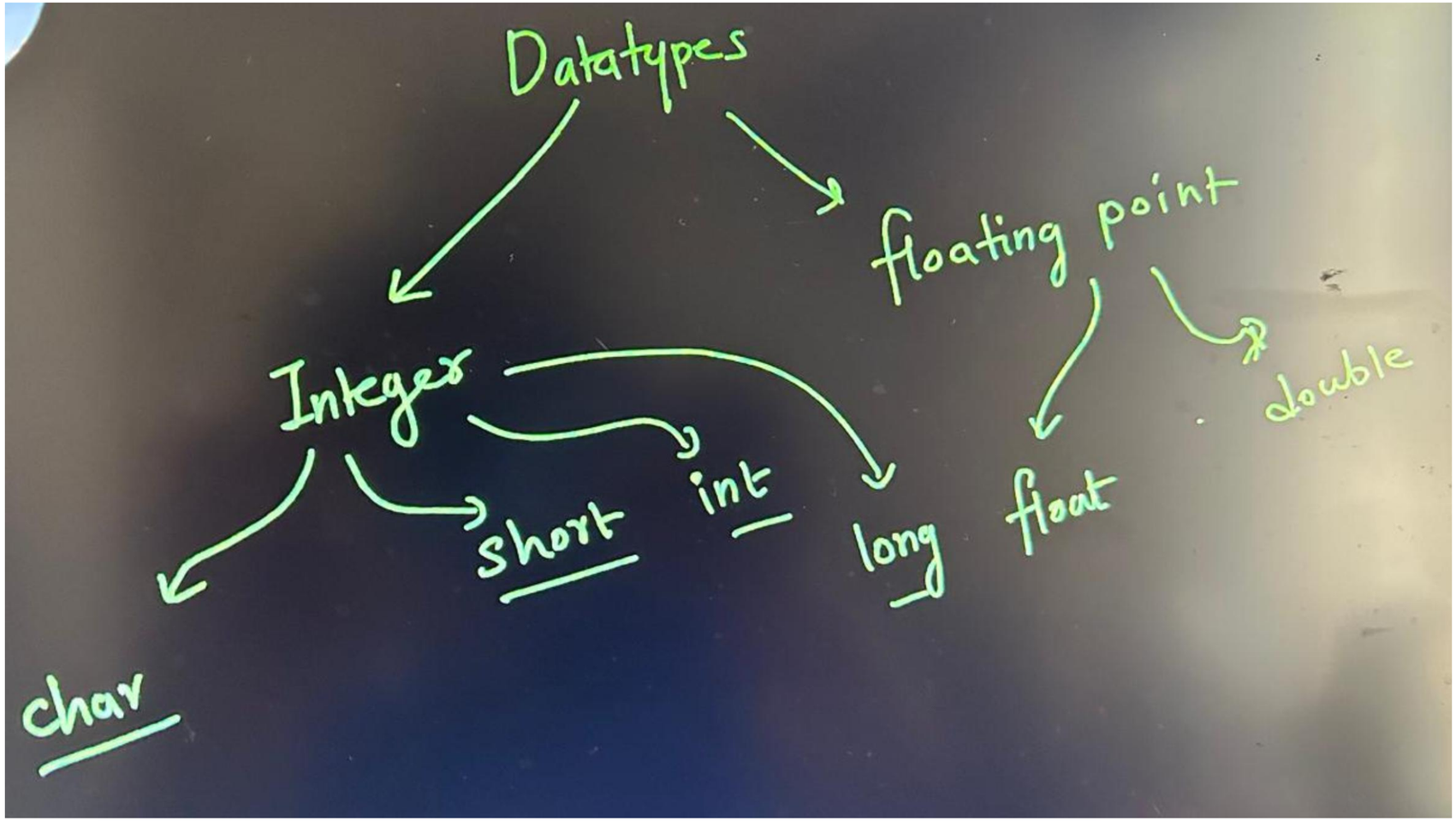
CPR

C Keywords



CPR

C Data Types



CPR

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

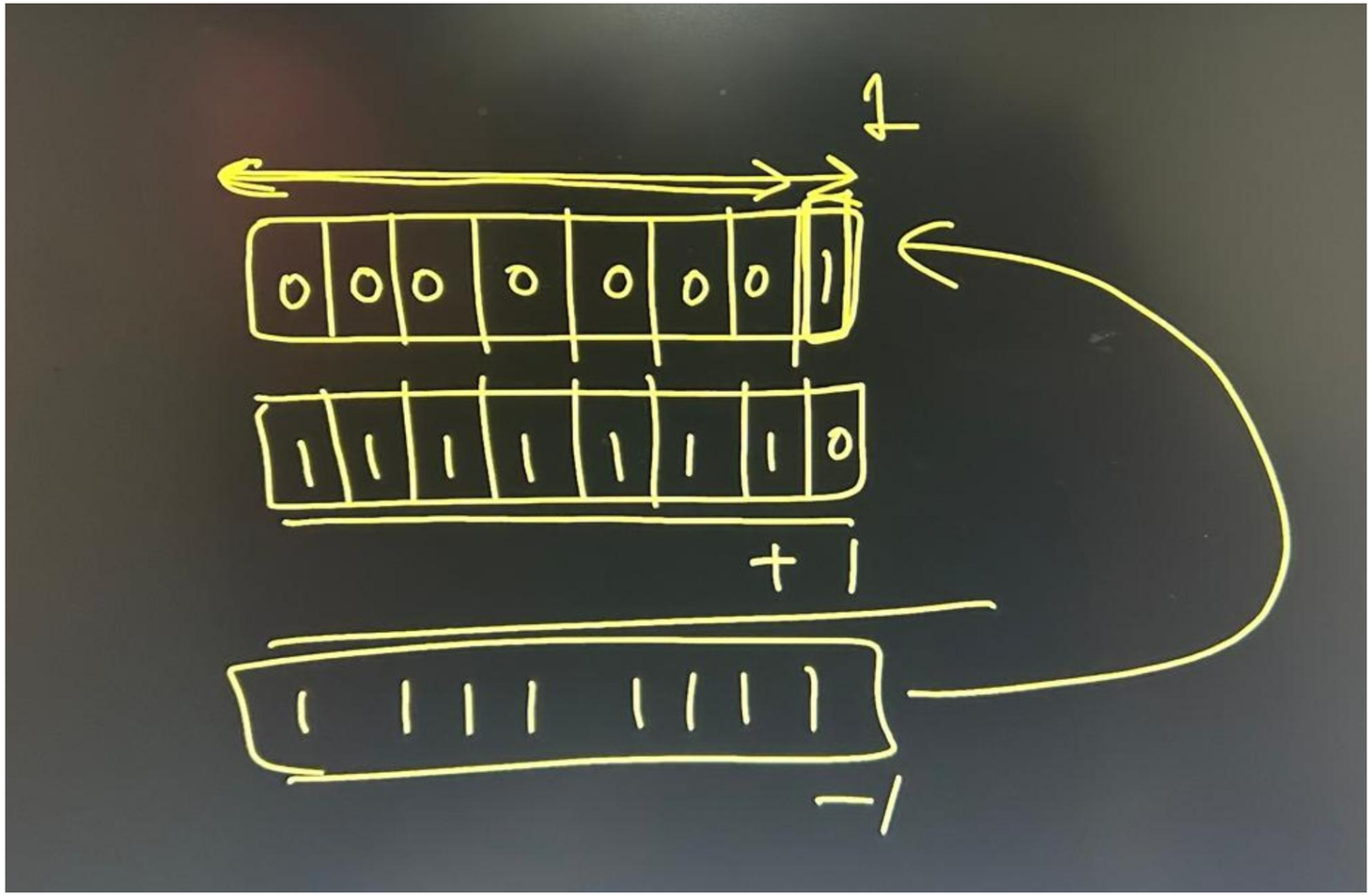
C Data Types Sizes

Program:

```
int main()
{
        int i = 10;
        return 0;
}
```

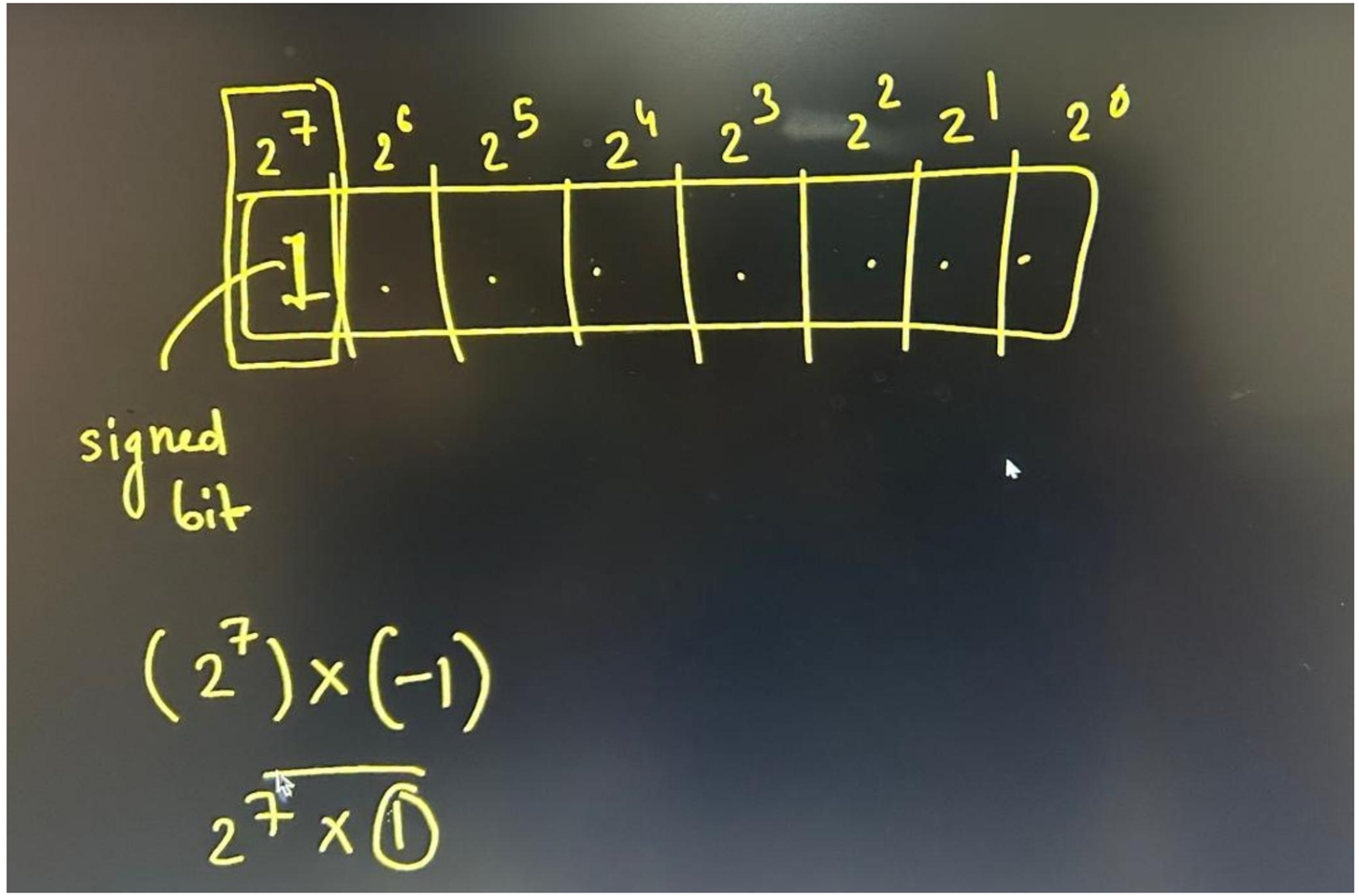
CPR

2’S complement



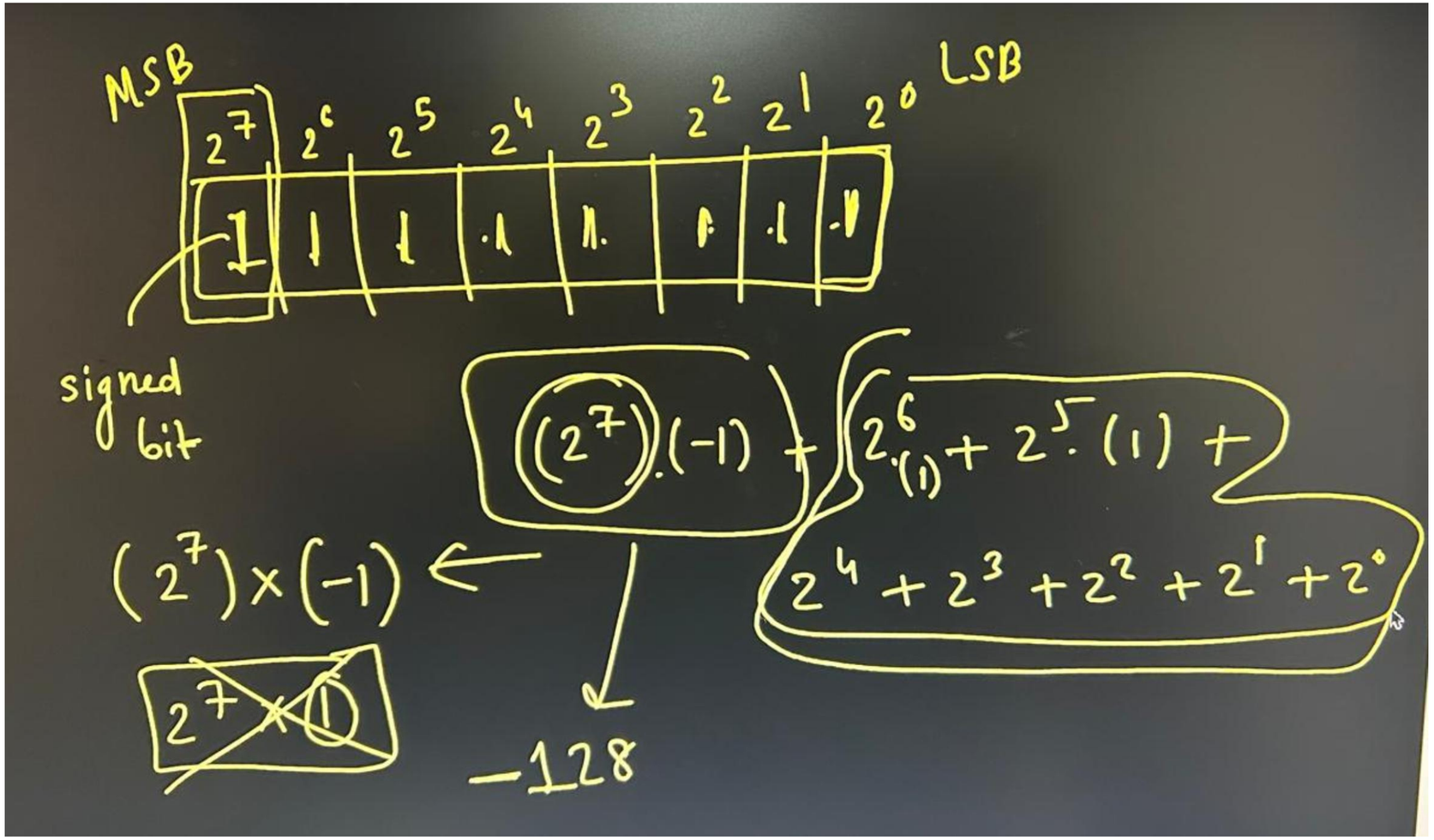
CPR

2’S complement



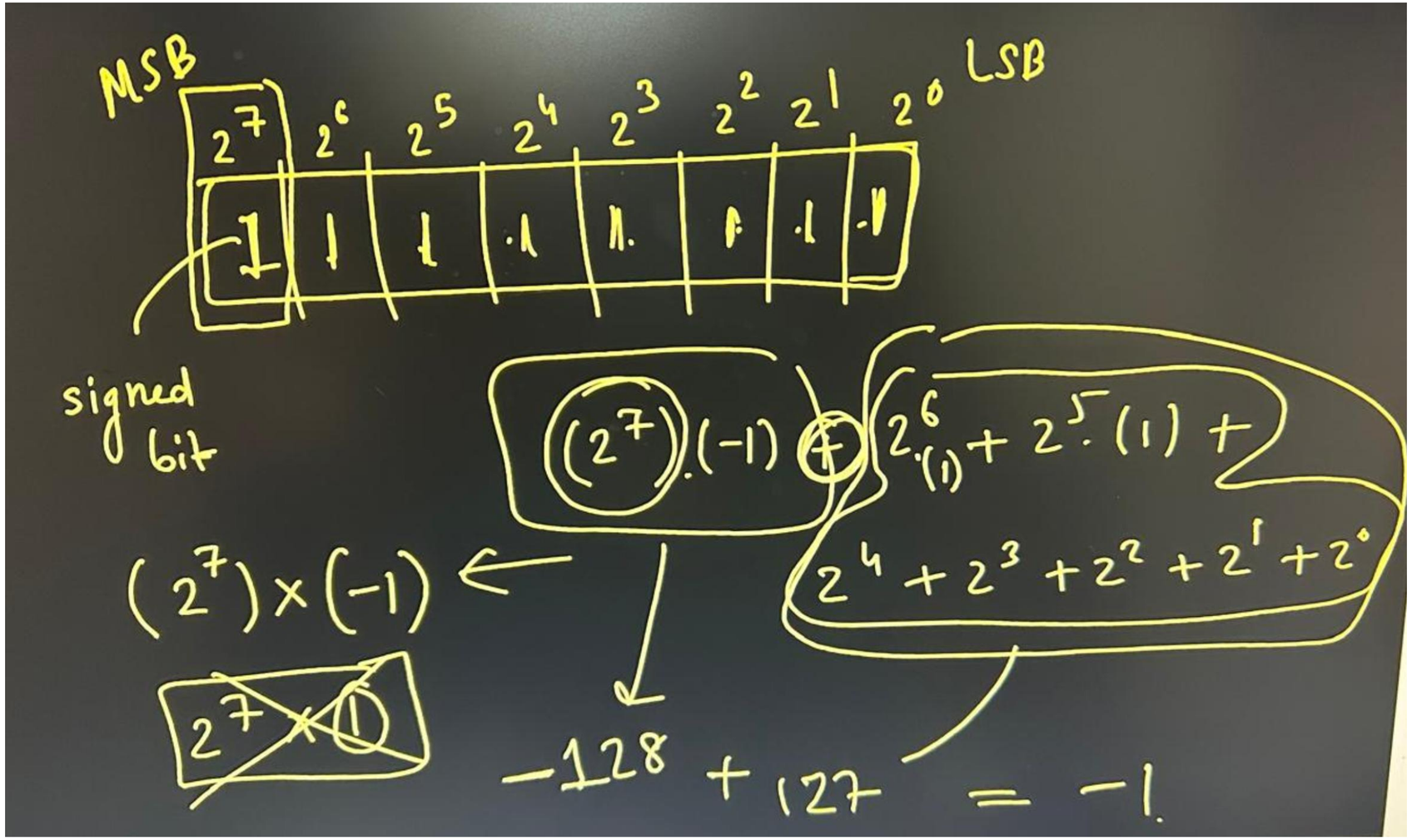
CPR

2’S complement



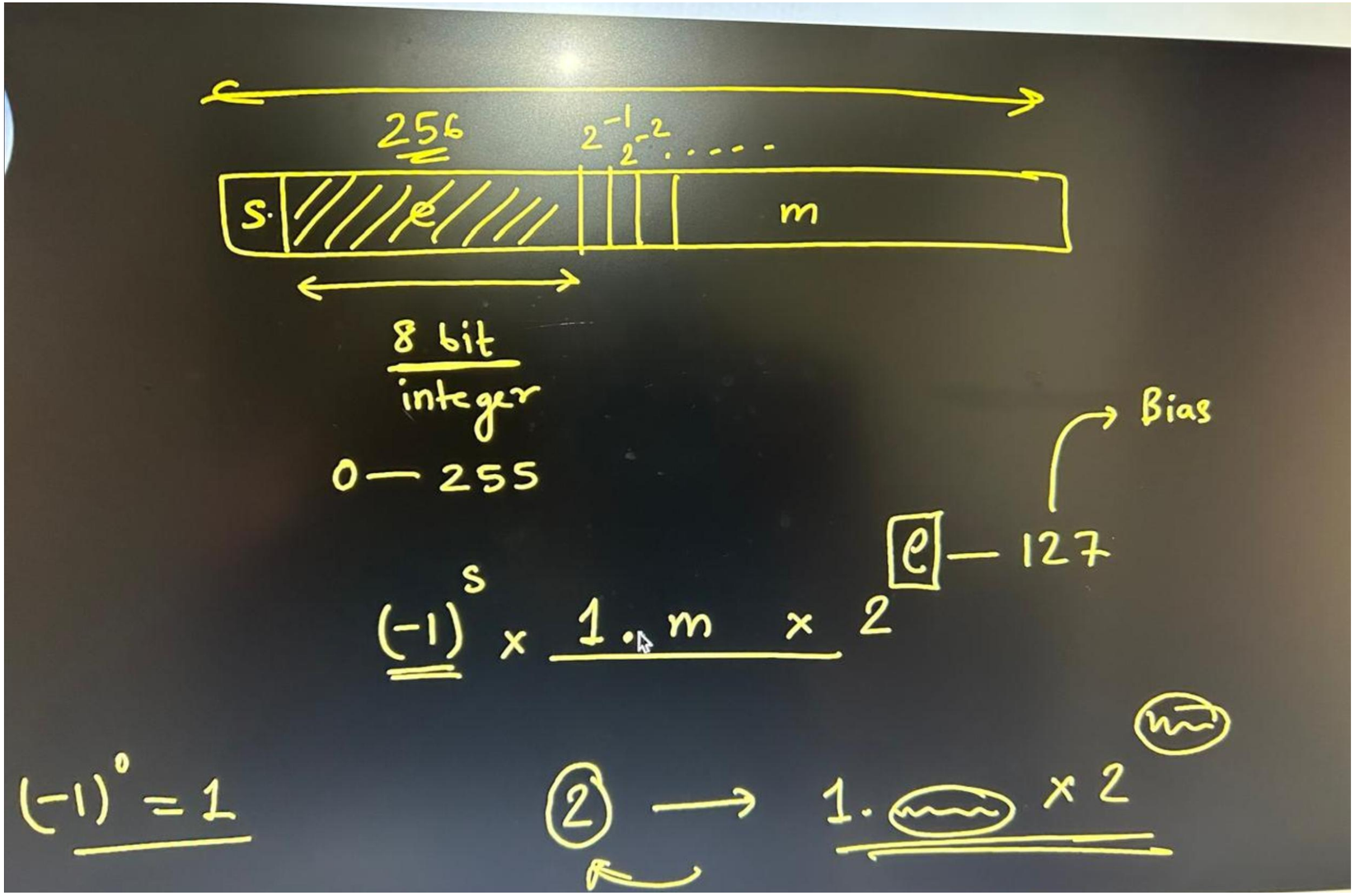
CPR

2’S complement



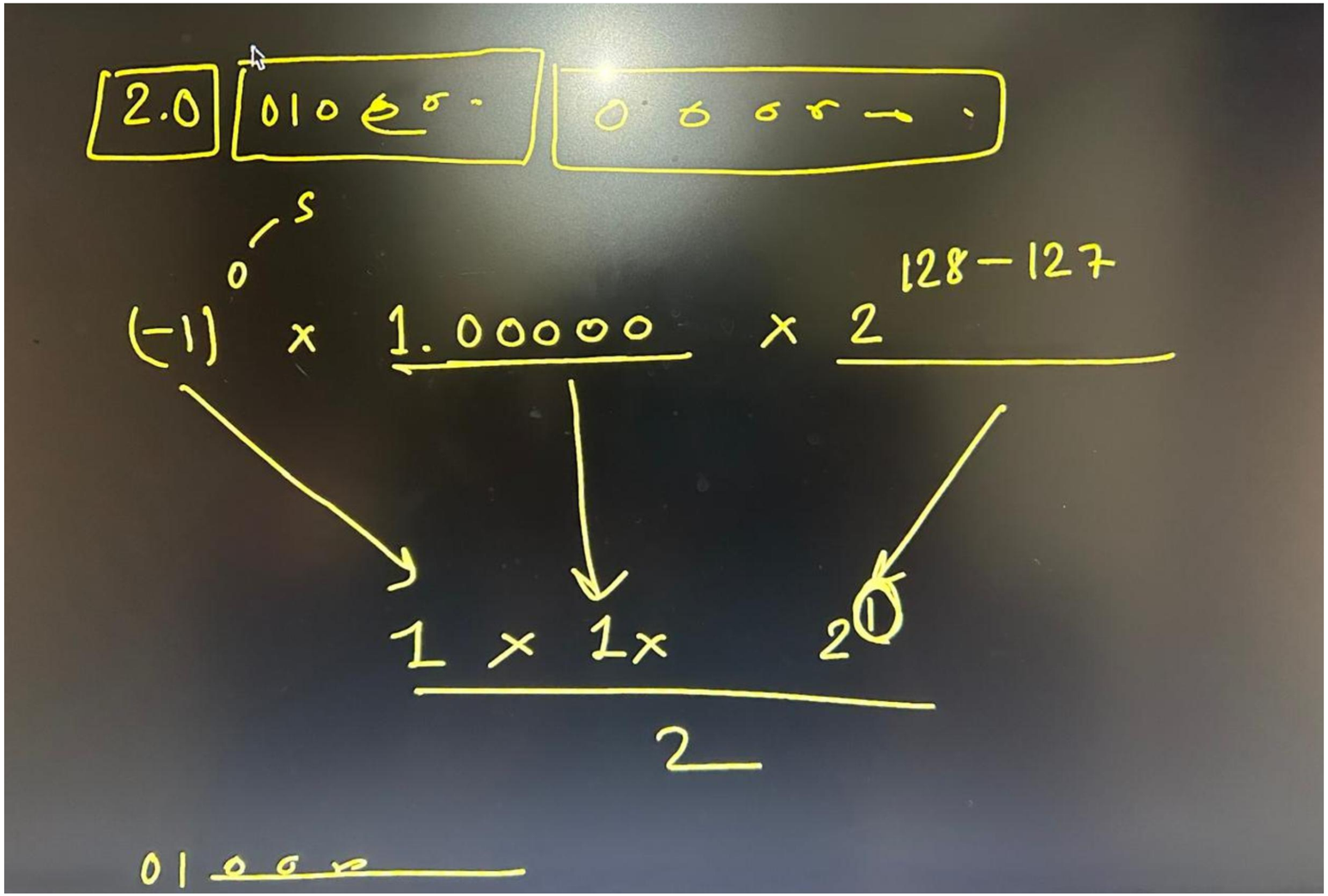
CPR

2’S complement Float



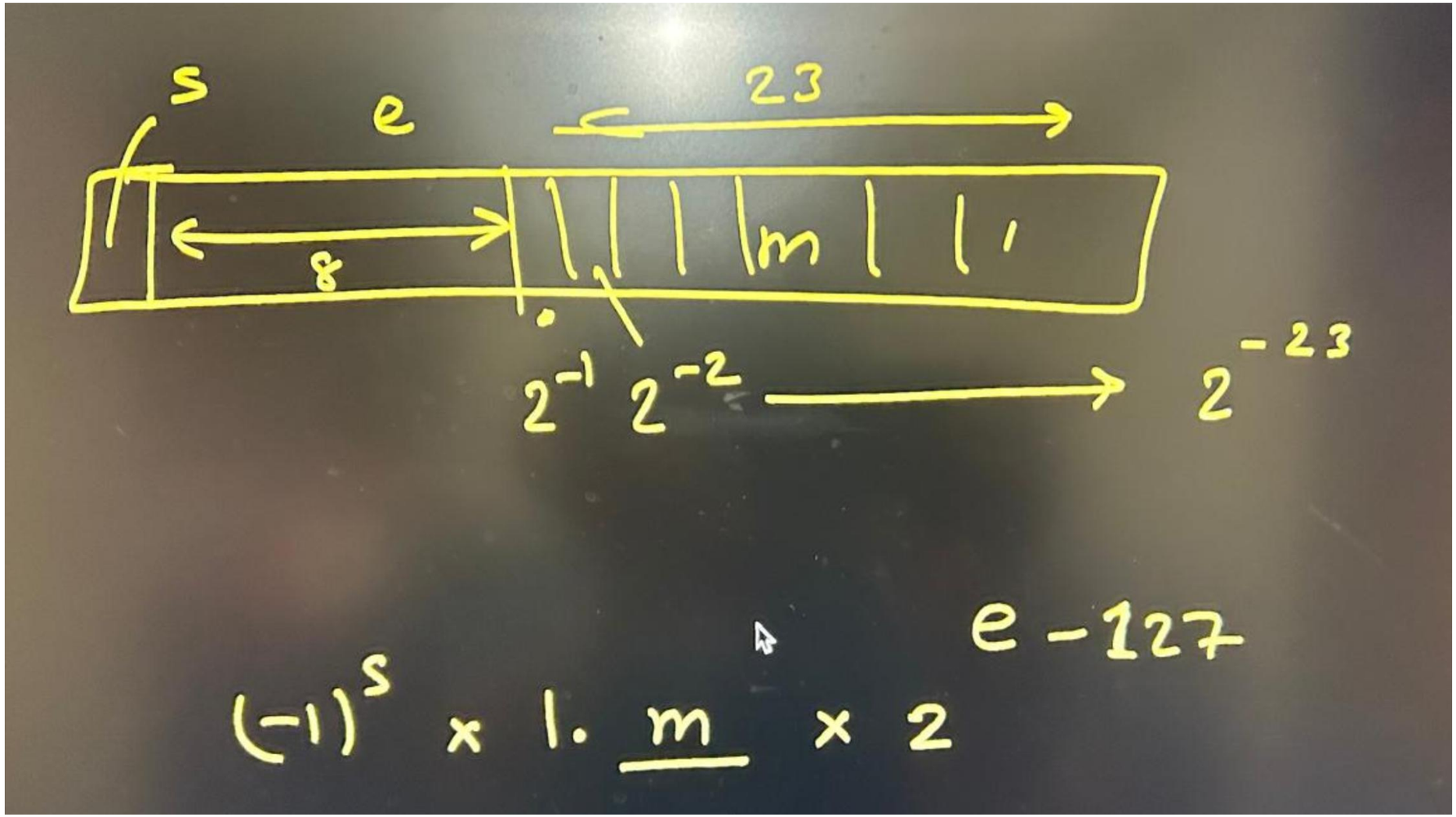
CPR

2’S complement Float



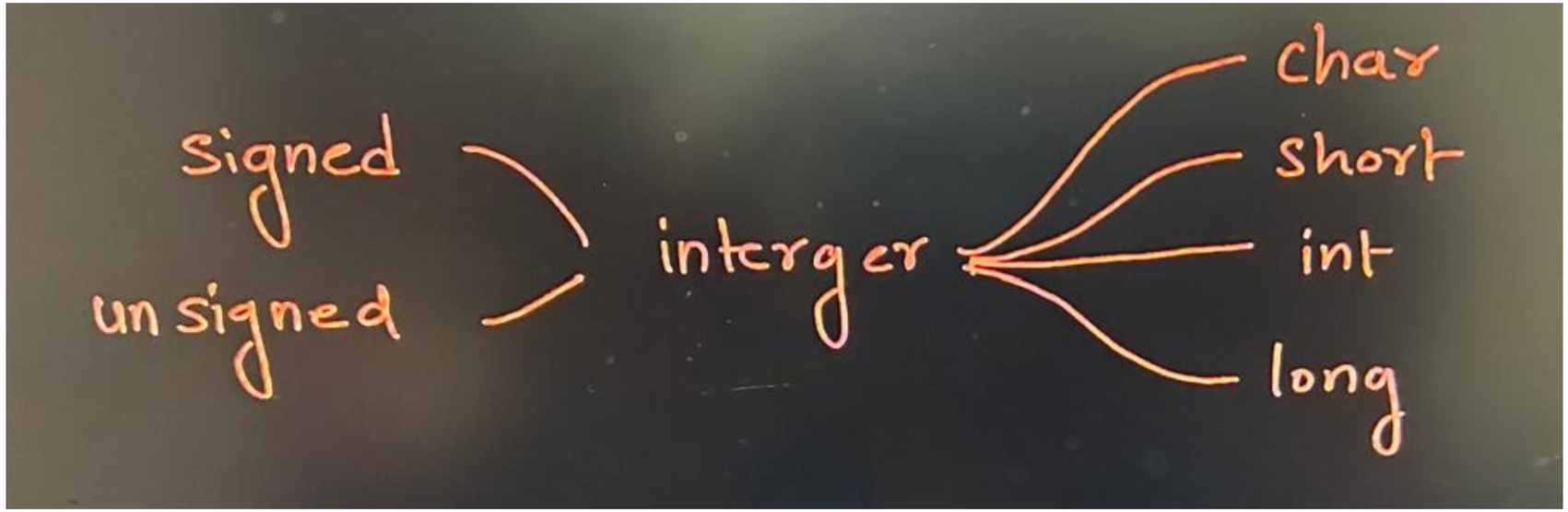
CPR

2’S complement Float



CPR

Signed & Unsigned



CPR

Const

const Keyword (Typed Constants)

const int MAX = 100;

- **Type-safe:** the compiler enforces type rules (e.g., no accidental float-to-int truncations)
- **Scope-limited:** can be local or global, obeying standard C scope rules
- **Allocated in memory:** occupies storage (like regular variables), unless optimized away
- **Readable & debuggable:** you can take its address (&MAX), see it in memory, and it shows up in debugging symbols
- **Enforced immutability:** any assignment to it results in a compile-time error:

```
const int PI = 3.14;

PI = 4; // error: assignment of read-only variable
```

CPR

volatile

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

- A **volatile variable** in C is a special kind of variable marked with the keyword volatile to inform the compiler:
  - **Always fetch from memory** – never cache it in registers.
  - **Never optimize away reads/writes** or reorder accesses, even if the code doesn't show changes nearby

Why Use volatile?

- volatile tells the compiler that the variable could change unexpectedly due to:
  - **Hardware or memory mapped I/O** (e.g., status registers)
  - **Interrupt service routines (ISRs)** modifying it asynchronously
  - **Signal handlers** or even interactions involving setjmp()/longjmp()

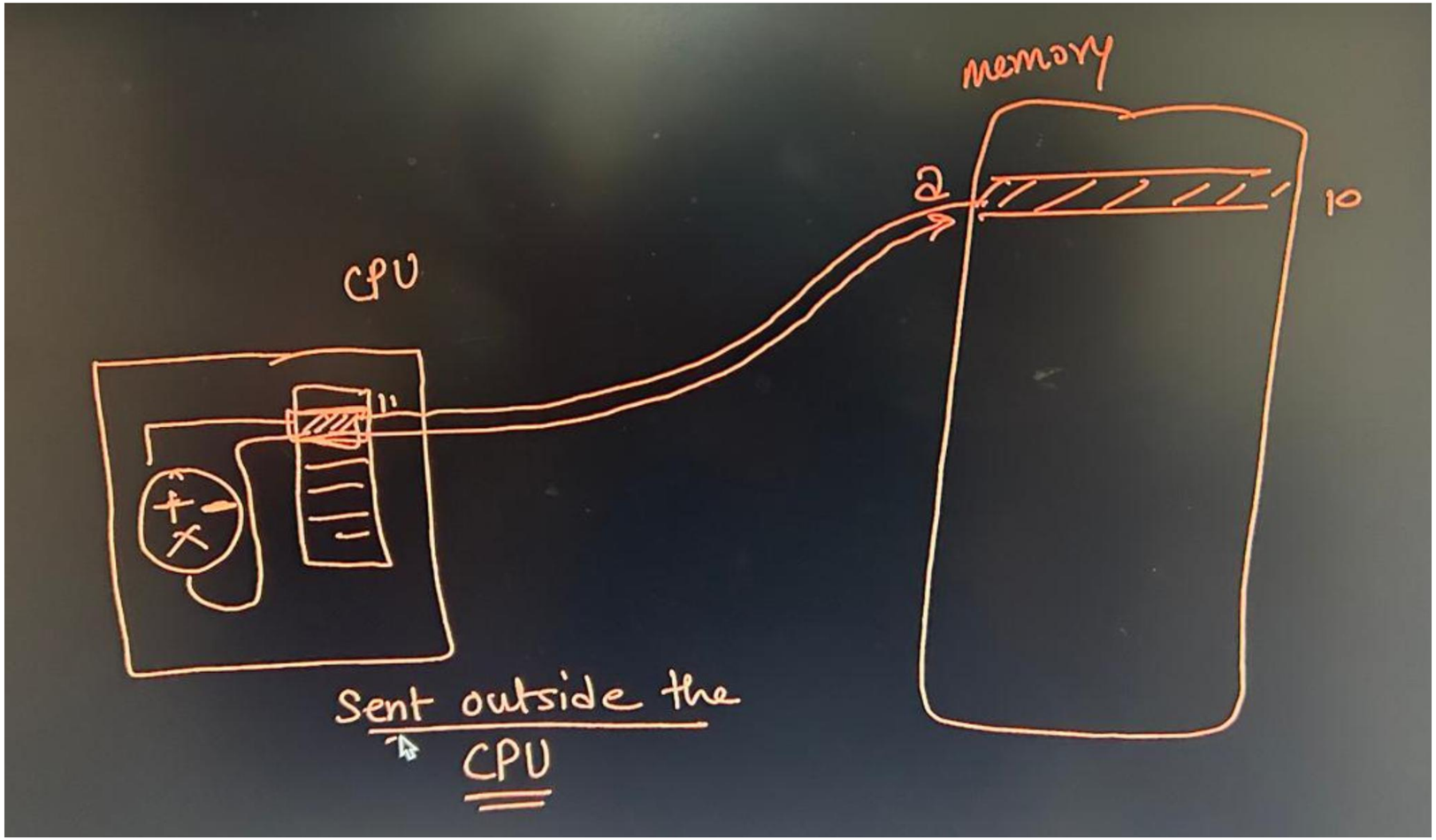
However, volatile is **not** a thread synchronization tool—it doesn't make operations atomic or enforce memory ordering in multithreaded code

```
volatile int flag = 0;

void ISR(void) {
    flag = 1;  // Changed by interrupt
}

int main() {
    while (!flag) {
        ;  // Keep polling – compiler must re-read flag every iteration
    }
    // flag was set — proceed...
}
```

Without volatile, the compiler could optimize away the loop check, making it infinite

CPR

volatile



CPR

void

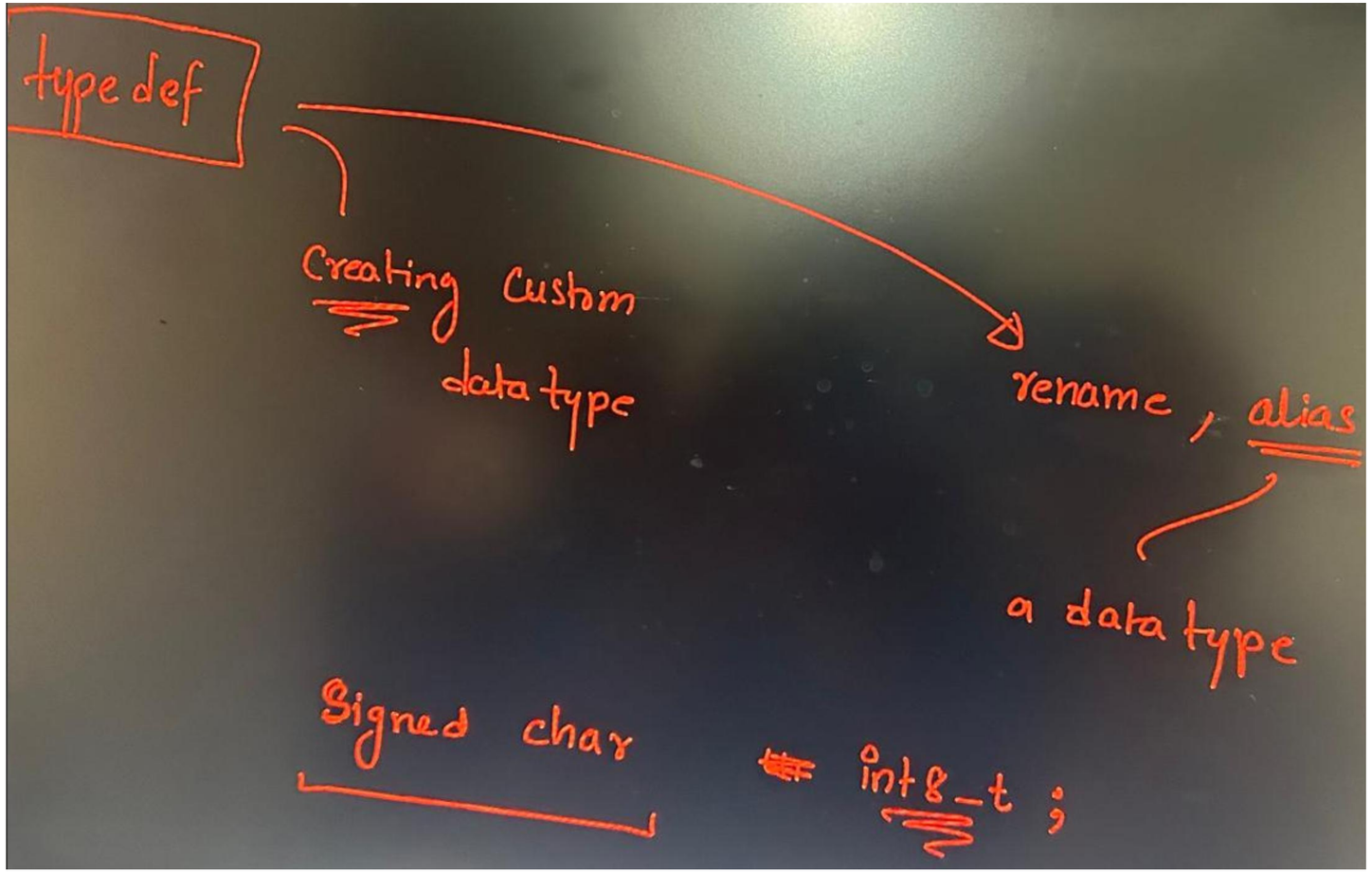
VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

In C, the void keyword represents **“no data”**

Context	Meaning
void f(void)	Function takes no args and returns nothing
void *ptr	Generic pointer to any object type
(void)expr;	Explicit discard of a value or unused param
void by itself	An incomplete type – cannot have variables

CPR

typedef


typedef
Creating Custom
data type
rename, alias
a data type
Signed char
int8_t ;

CPR

sizeof

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

- The sizeof operator in C is a compile-time, **unary operator** that yields the storage size of its operand in **bytes**, returning a value of type size_t (an unsigned integer)

Syntax:

- sizeof(type); // type must be enclosed in parentheses
- sizeof expression; // parentheses optional
- **Returns** the number of bytes needed to store the given type or expression, without evaluating the expression (unless it's a variable-length array)

- **Examples:**

sizeof(char) // always = 1

sizeof(int) // typically 4 on many platforms

sizeof(double) // often 8 bytes

sizeof ptr // size of pointer type, e.g., 8 bytes on 64-bit

sizeof arr // total bytes of array (e.g., 10 ints → 40 bytes)

sizeof expr // size of the expression's resulting type

CPR

Storage classes

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

Specifier	Storage Duration	Scope	Linkage	Init by default
auto	Automatic (stack)	Block	None	Garbage
register	Automatic (register)	Block	None	Garbage
static (local)	Static (program)	Block	None	Zero (if no init)
static (global)	Static (program)	File	Internal (file)	Zero
extern	Static (program)	Global	External (all units)	Zero

CPR

static

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

- #include <stdio.h>

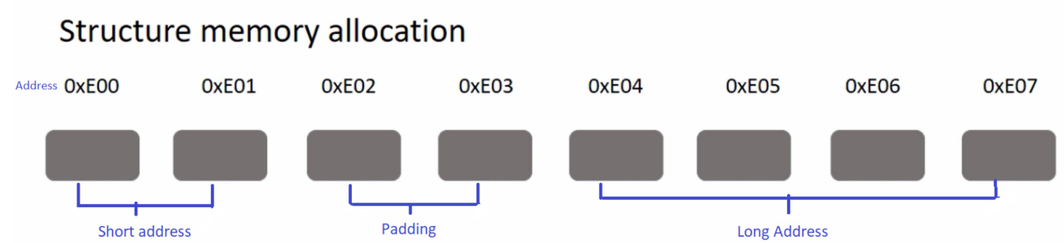
```
Void foo ()
{
        static int a = 4;
        a = a + 1;
        printf("a = %d\n", a);
}

Int main()
{
        foo();      // output ?
        foo();      // output ?
        return(0);
}
```

CPR

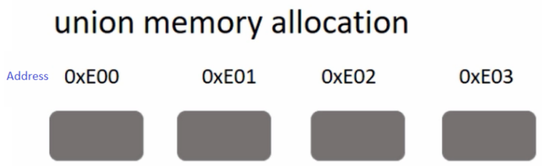
Struct & unions

- **struct:** multiple members, full usage, higher memory.
- **union:** one-member-at-a-time, memory-efficient, useful for specialized low-level use cases.

```
struct address
{
    uint16_t shortAddr;
    uint32_t longAddr;
};
```




```
union address
{
    uint16_t shortAddr;
    uint32_t longAddr;
};
```



CPR

enums

When to Use Enums

- Group related constants clearly.
- Use in switch statements for readability and compiler checks.
- Prefer over macros for integer constants to avoid naming collision and improve maintainability.
- Great for finite states/options (modes, flags) with auto-assignment and logical grouping.

CPR

Branching Statements

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

Statement	Purpose	Use Case
if / else if / else	Conditional branching	Best for ranges or complex logic
switch	Multi-way branching on integer values	Cleaner for many discrete choices
?:	Inline conditional expressions	Compact decisions
break	Exit loop or switch early	Early termination
continue	Skip to next loop iteration	Loop control
goto	Unconditional jump	Rare cleanup/error cases
return	Exit function	End of function or error early exit

CPR

Loops

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

Loop Type	Use When...
for	You know iteration count; want clean control over a counter.
while	Condition-based loops with unknown or dynamic count.
do…while	Body needs to run at least once (e.g. input prompt).

CPR

Variables

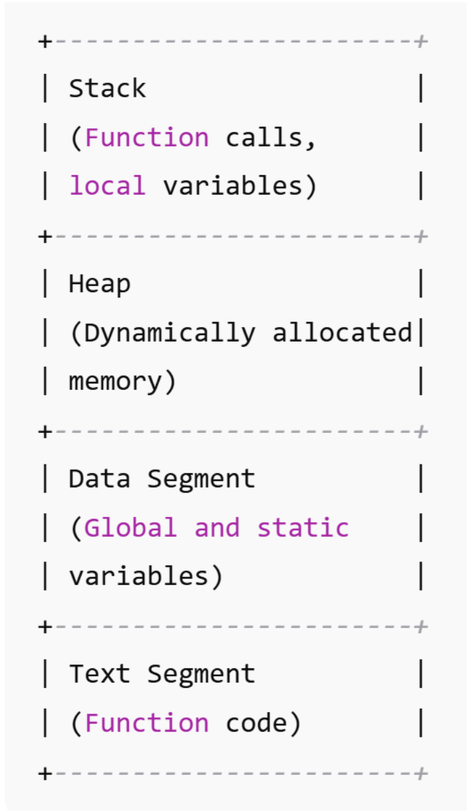
VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

Why Variables Matter

- They give names to memory locations, making code clear and maintainable.
- Types enforce correct operations and memory use.
- Proper scope and lifetime management prevent errors and undefined behavior.
- They underpin everything in programming: expression, control, data structures, I/O, and more

CPR

Functions



Functions in C are essential for:

- Organizing code into logical blocks.
- Enhancing code reuse and maintainability.
- Simplifying complex tasks into manageable components.
- By defining and using functions effectively, you can write cleaner, more efficient, and more understandable C programs.

CPR

Problem Statement 1

VIDHIT TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE

Project 1: Calculator in C

Program Overview

The calculator program will:

1. Prompt the user to enter two numbers.
2. Ask the user to select an operation (e.g., +, -, *, /).
3. Perform the requested calculation and display the result.
4. Handle basic error cases (e.g., division by zero).
5. Optionally, allow the user to perform multiple calculations in a loop.

Whiteboard

VIDHIT
TECHNOLOGIES
BUILD.MARKET.SECURE.SCALE