VIDHIT
TECHNOLOGIES

BUILD.MARKET.SECURE.SCALE

Campus to Corporate

Your tested road for placement success

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

What Do I Need to Qualify into the race?

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)	

Agenda

Revisiting C Basics

Fun Code Samples

Problem Statement

GitHub

Creating Workspace

https://github.com/ranjit27/campus2connect

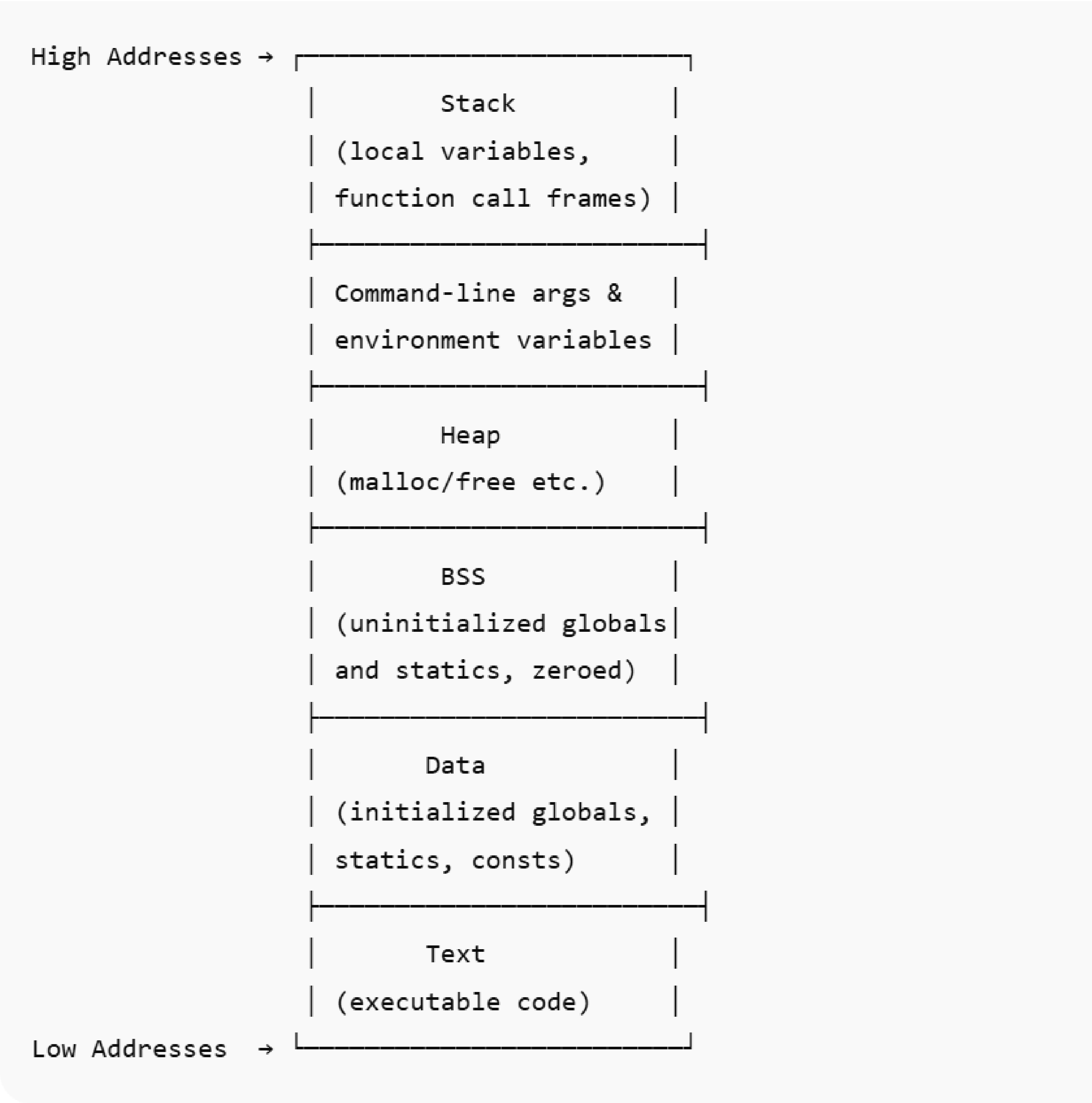
CPR

The Toolchain

Stage	Tool	Input	Output	GCC flag
Preprocess	cpp	.c	.i	-E
Compile	cc1/cc1plus	.i	.s	-S
Assemble	as	.s	.o	-c
Link	ld	.o	Executable	(none)

CPR

Memory Layout of a C Program



```
High Addresses → ┌─────────────────────────┐
                  │        Stack            │
                  │ (local variables,       │
                  │ function call frames)   │
                  ├─────────────────────────┤
                  │ Command-line args &     │
                  │ environment variables   │
                  ├─────────────────────────┤
                  │        Heap             │
                  │ (malloc/free etc.)      │
                  ├─────────────────────────┤
                  │        BSS              │
                  │ (uninitialized globals│
                  │ and statics, zeroed)    │
                  ├─────────────────────────┤
                  │       Data              │
                  │ (initialized globals,   │
                  │ statics, consts)        │
                  ├─────────────────────────┤
                  │       Text              │
                  │ (executable code)       │
Low Addresses  → └─────────────────────────┘
```

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.

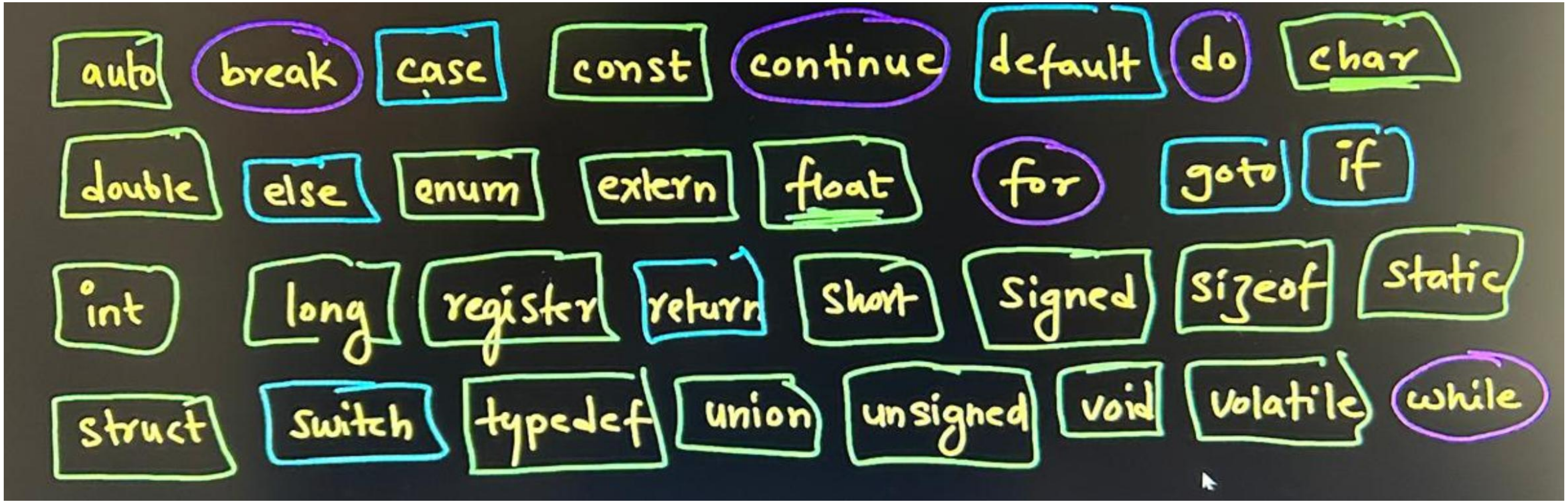
Summary

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

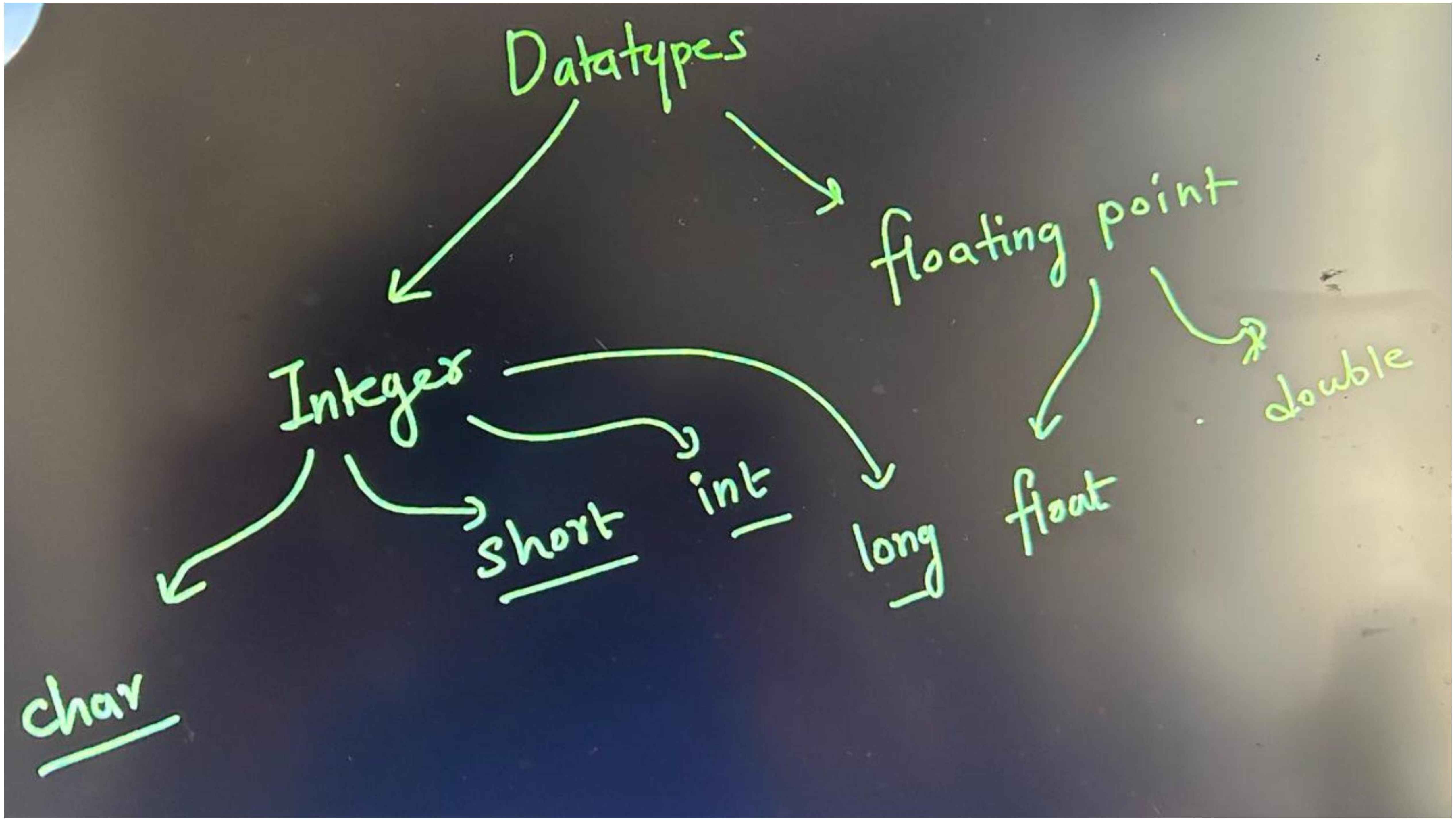
C Keywords



auto	break	case	const	continue	default	do	char
double	else	enum	extern	float	for	goto	if
int	long	register	return	Short	Signed	sizeof	Static
struct	Switch	typedef	union	unsigned	void	volatile	while

CPR

C Data Types



CPR

void

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

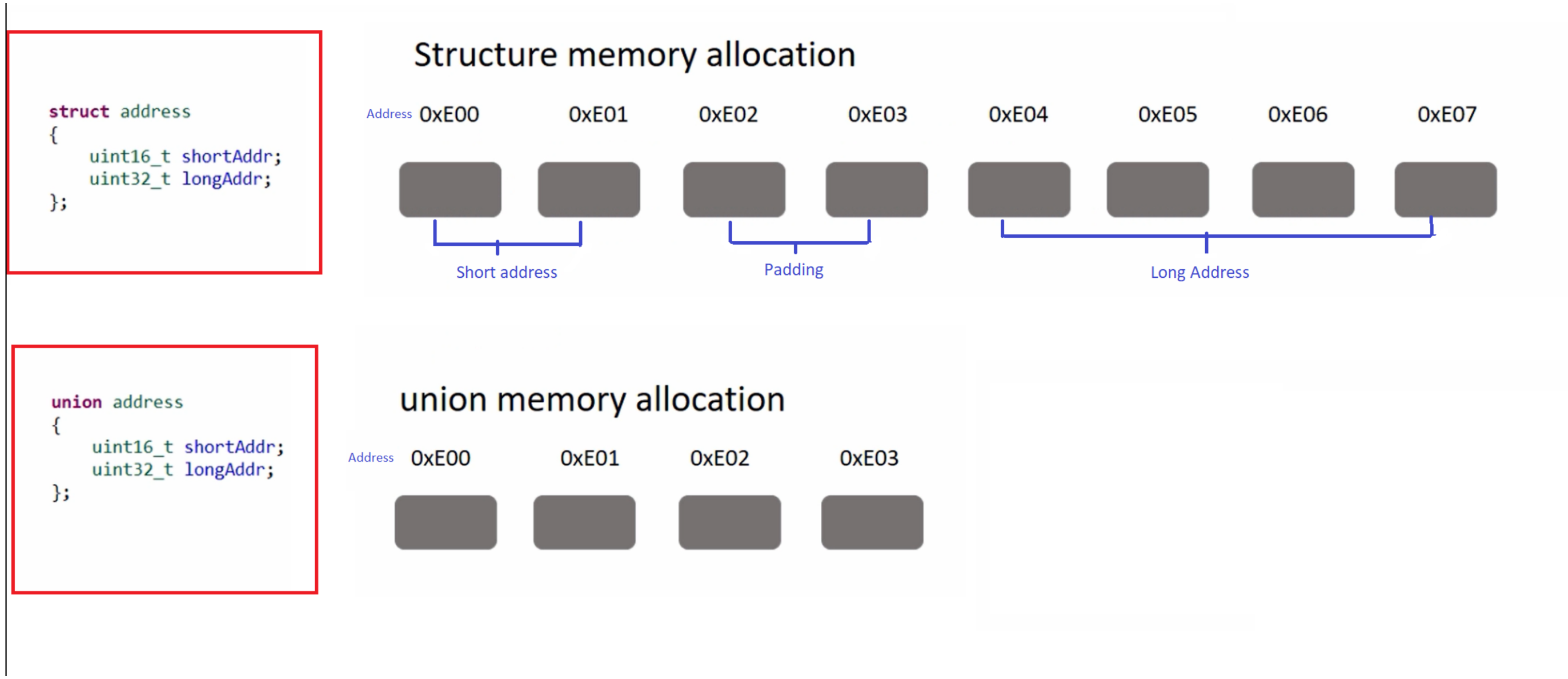
Storage classes

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

Struct & unions

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



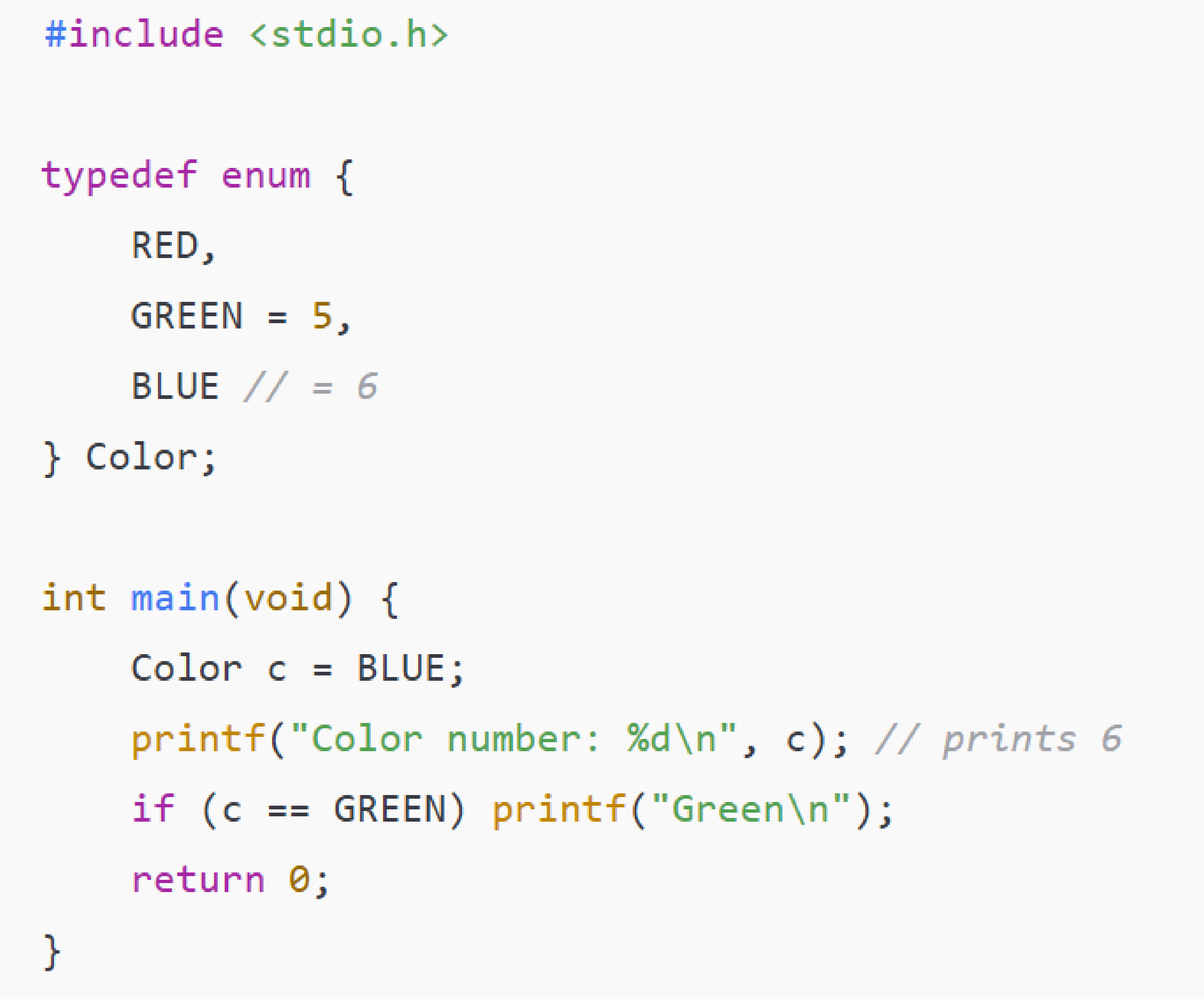
```
struct address
{
    uint16_t shortAddr;
    uint32_t longAddr;
};
```

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

CPR

enums

When to use enum	When to avoid enum
Related integer constants	Floats, strings, large integer constants
Values used in switches	Global namespace conflicts
State machines, flags	Strict type safety required



```c
#include <stdio.h>

typedef enum {
    RED,
    GREEN = 5,
    BLUE // = 6
} Color;

int main(void) {
    Color c = BLUE;
    printf("Color number: %d\n", c); // prints 6
    if (c == GREEN) printf("Green\n");
    return 0;
}
```

CPR

Branching Statements

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

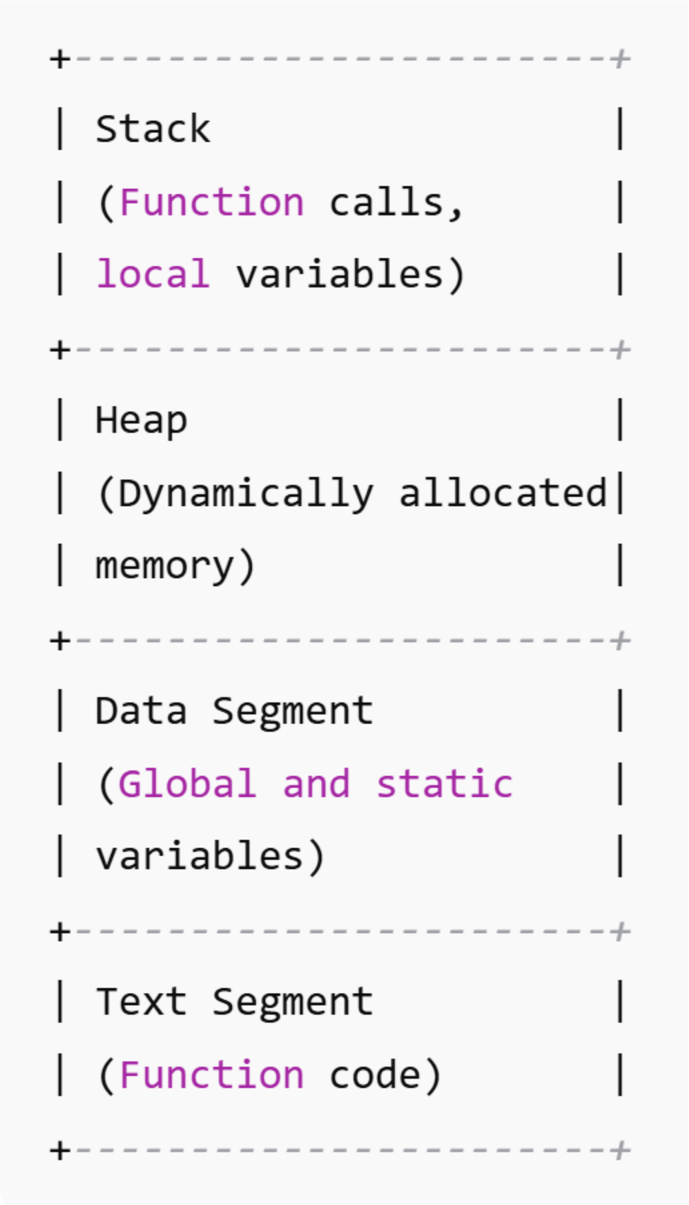
CPR

Loops

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

Functions



```
+---------------------+
| Stack               |
| (Function calls,    |
| local variables)    |
+---------------------+
| Heap                |
| (Dynamically allocated|
| memory)             |
+---------------------+
| Data Segment        |
| (Global and static  |
| variables)          |
+---------------------+
| Text Segment        |
| (Function code)     |
+---------------------+
```

Functions in C are essential for:

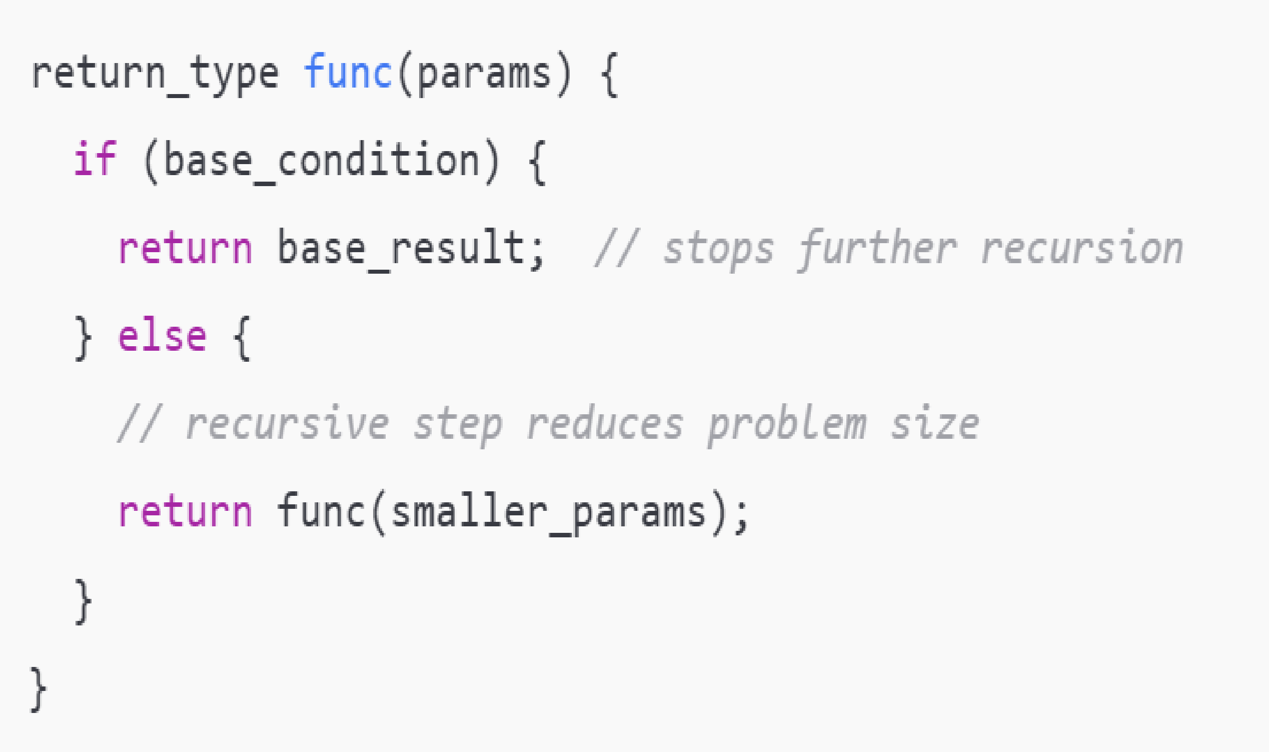
- Organizing code into logical blocks.
- Enhancing code reuse and maintainability.
- Simplifying complex tasks into manageable components.
- By defining and using functions effectively, you can write cleaner, more efficient, and more understandable C programs.

CPR

Recursion

What Is Recursion?

Recursion occurs when a function in C calls itself, directly or indirectly, repeating the process until it meets a **base case** that ends the recursion

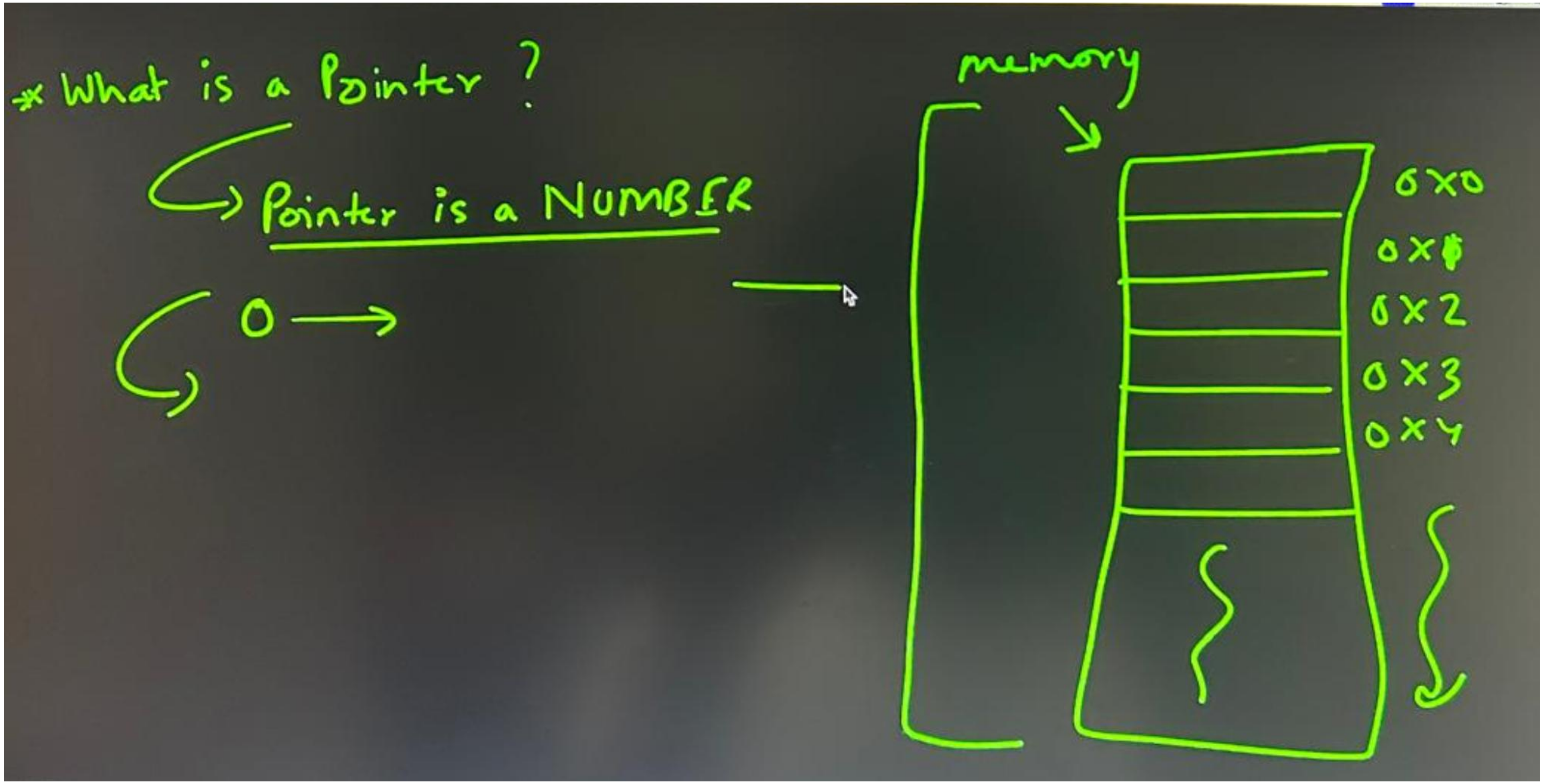


```
return_type func(params) {
  if (base_condition) {
    return base_result;  // stops further recursion
  } else {
    // recursive step reduces problem size
    return func(smaller_params);
  }
}
```

✅ When to Recursively Use	⚠️ When to Avoid Recursion
Natural recursive structures (trees, graphs)	When efficiency and memory are critical
Clean, simple code conception	When recursion depth is uncertain
Backtracking & divide-and-conquer	If compiler lacks tail call optimization

CPR

Pointers

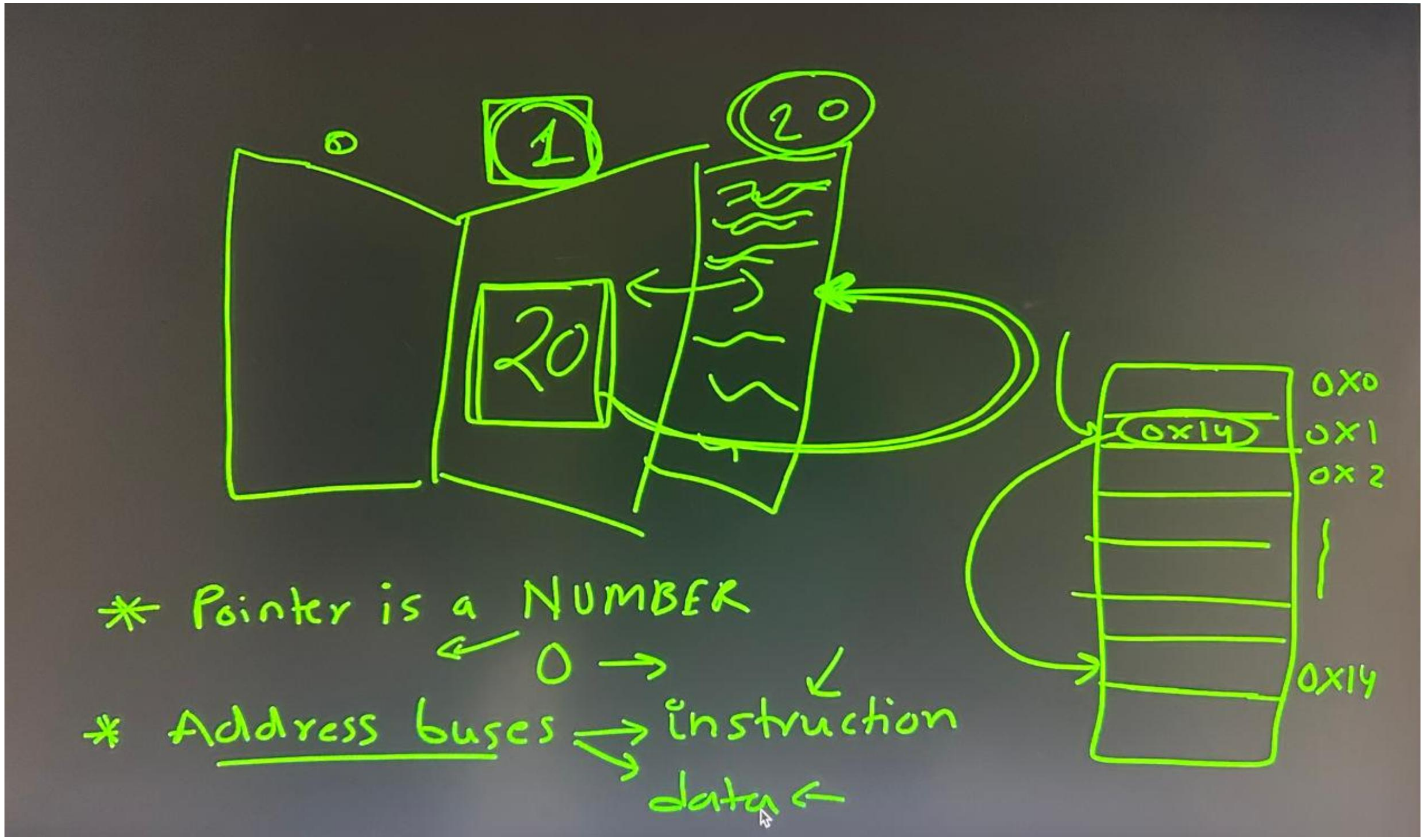


* What is a Pointer ?

Pointer is a NUMBER

CPR

Pointers

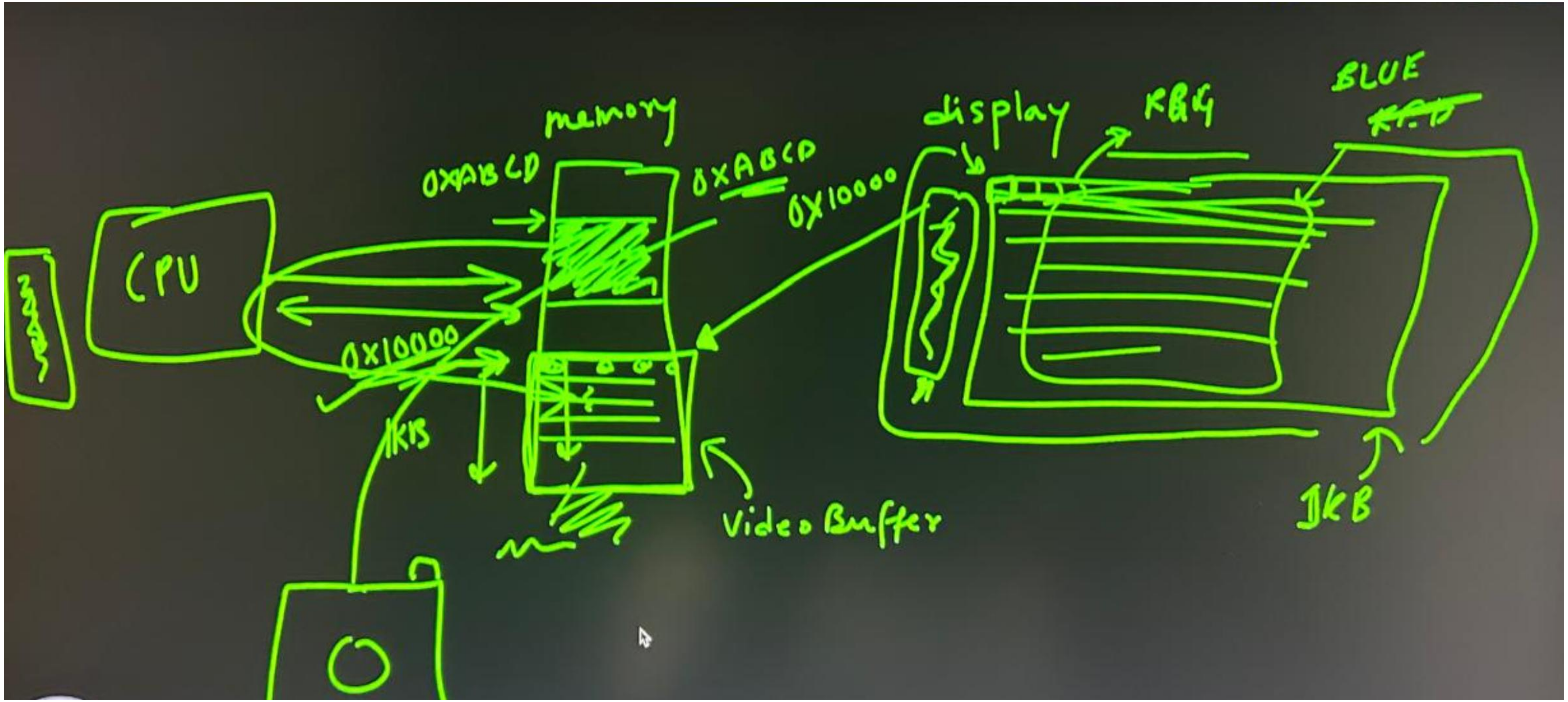


* Pointer is a NUMBER

* Address buses → instruction
→ data

CPR

Pointers



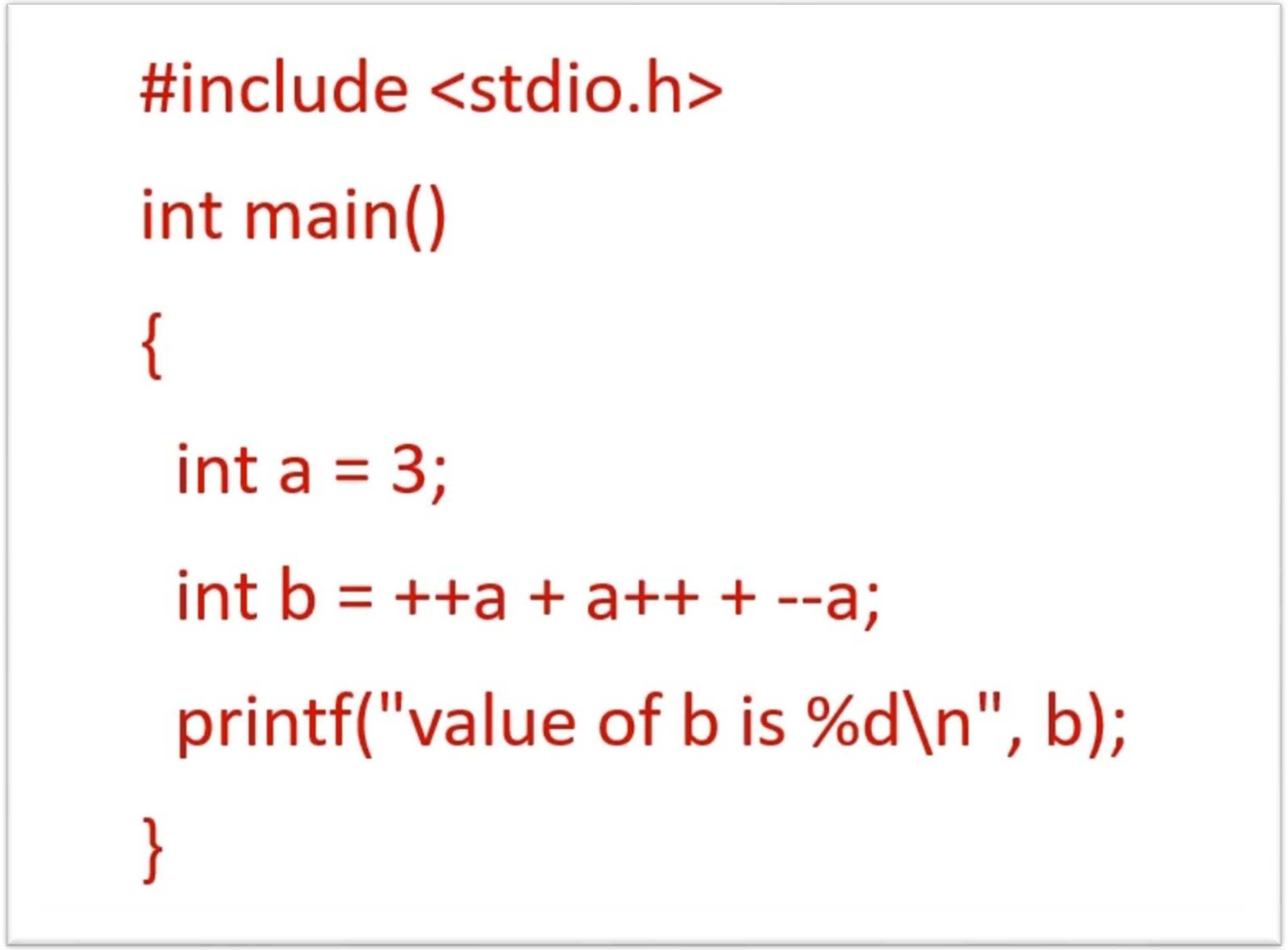
CPR

Pointers

DO PROGRAMS

CPR

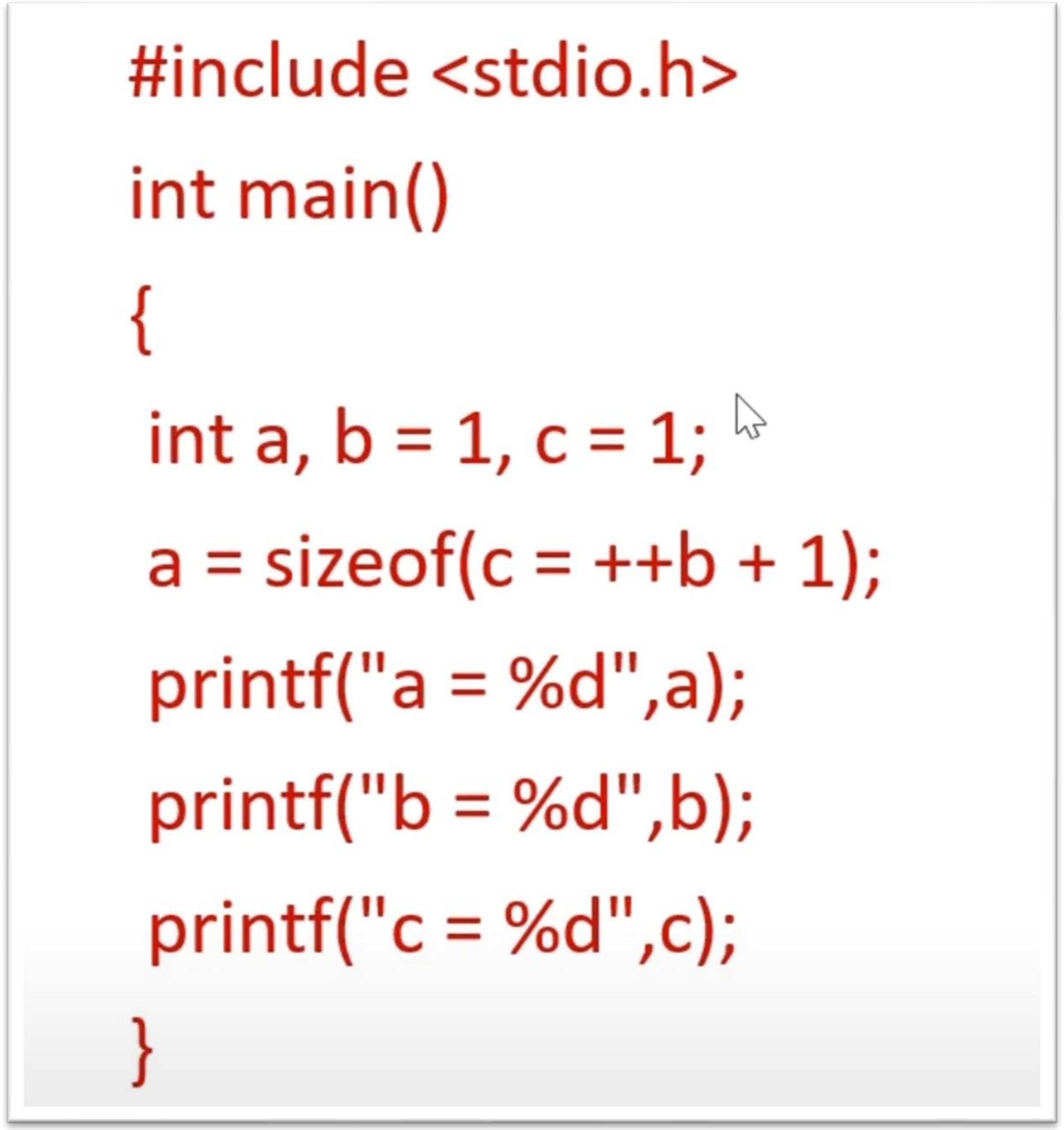
Test Questions



```
#include <stdio.h>
int main()
{
  int a = 3;
  int b = ++a + a++ + --a;
  printf("value of b is %d\n", b);
}
```

CPR

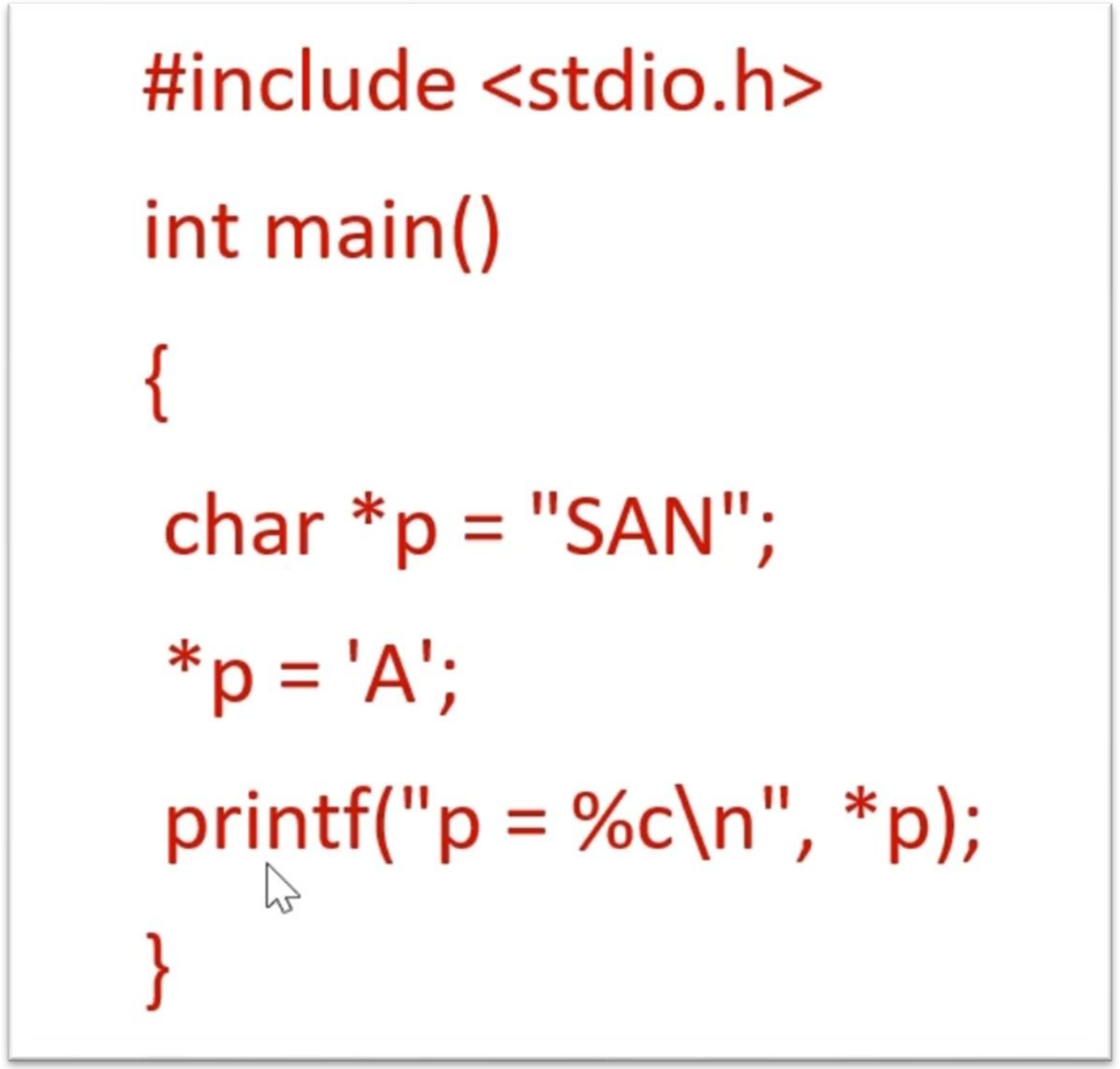
Test Questions



```
#include <stdio.h>
int main()
{
 int a, b = 1, c = 1;
 a = sizeof(c = ++b + 1);
 printf("a = %d",a);
 printf("b = %d",b);
 printf("c = %d",c);
}
```

CPR

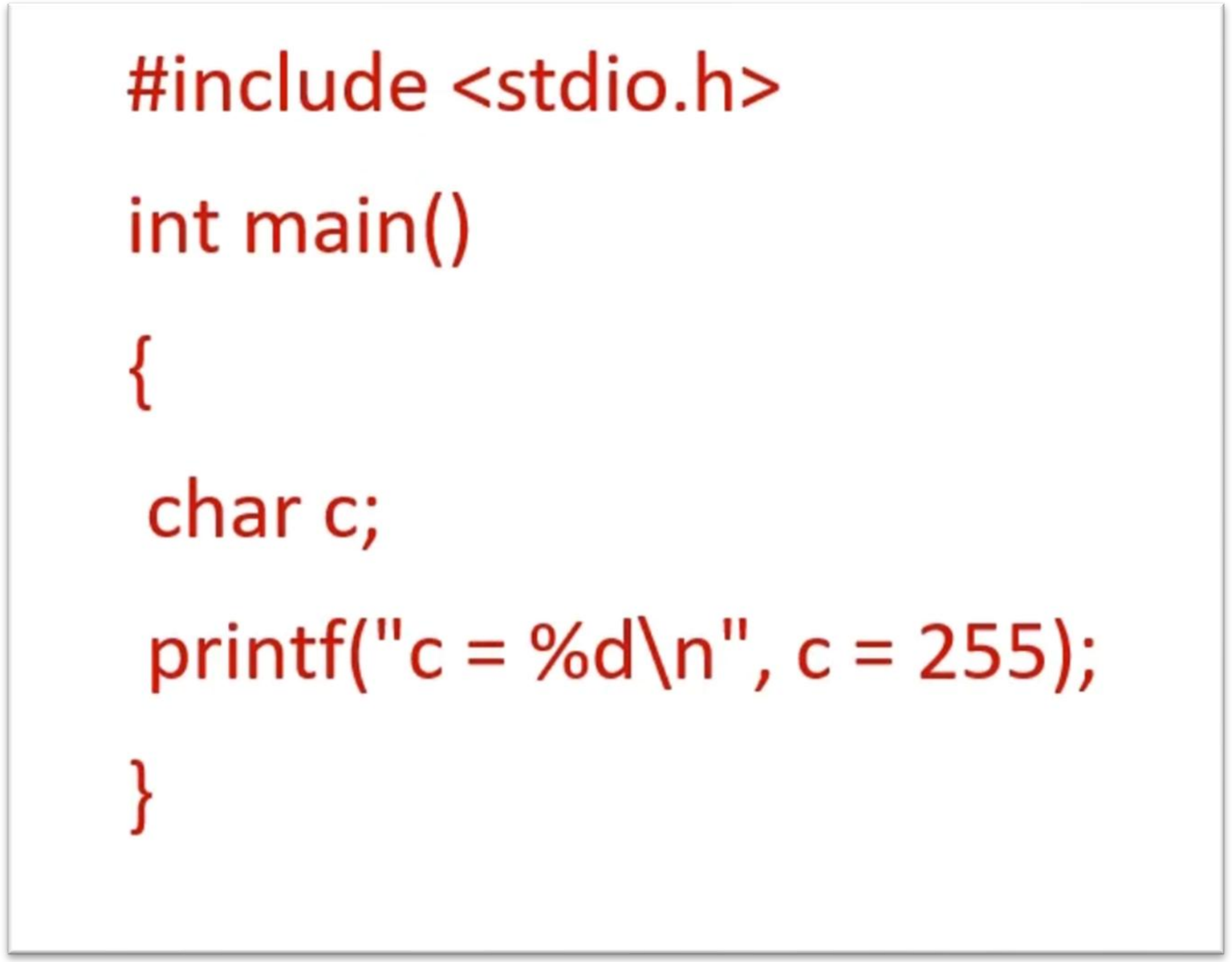
Test Questions



```
#include <stdio.h>
int main()
{
 char *p = "SAN";
 *p = 'A';
 printf("p = %c\n", *p);
}
```

CPR

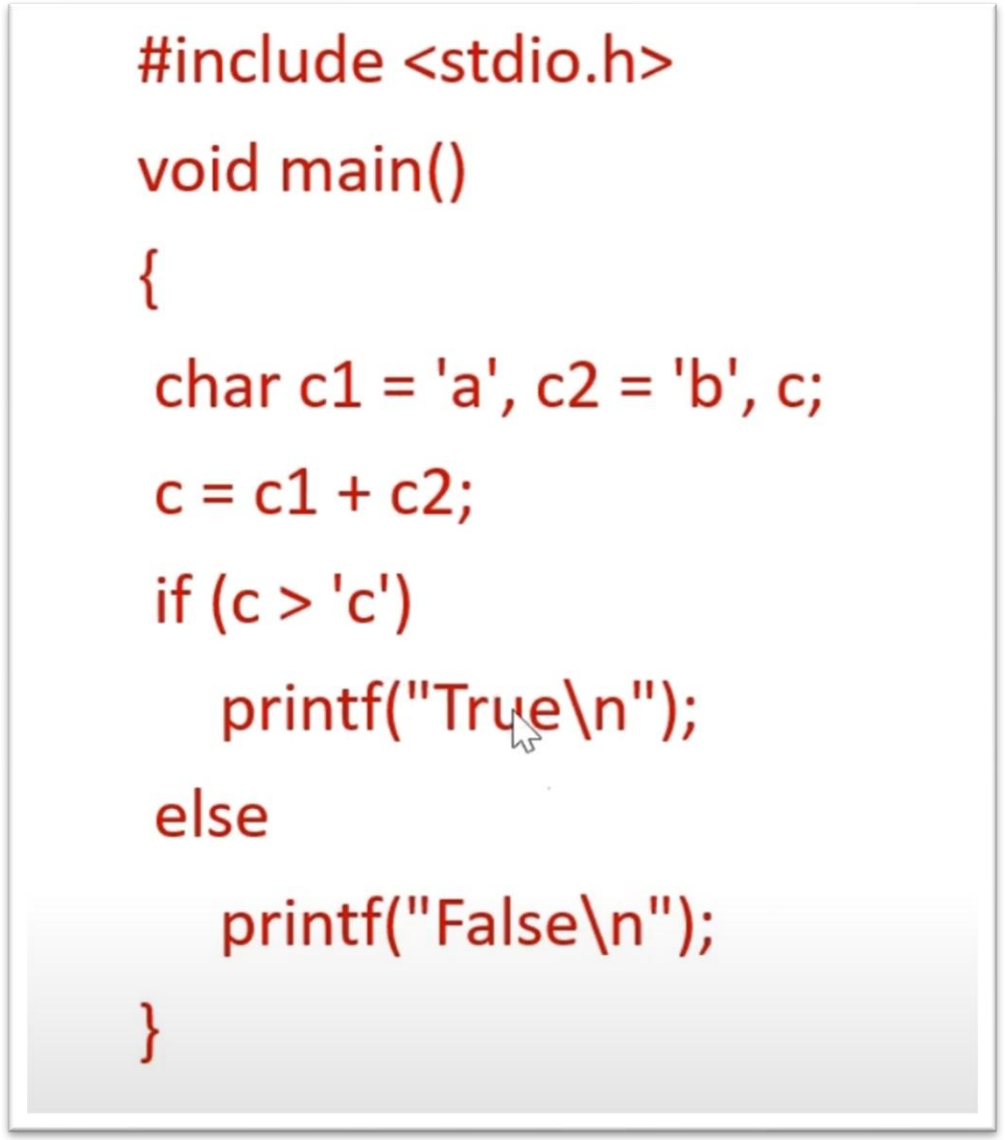
Test Questions



```
#include <stdio.h>
int main()
{
 char c;
 printf("c = %d\n", c = 255);
}
```

CPR

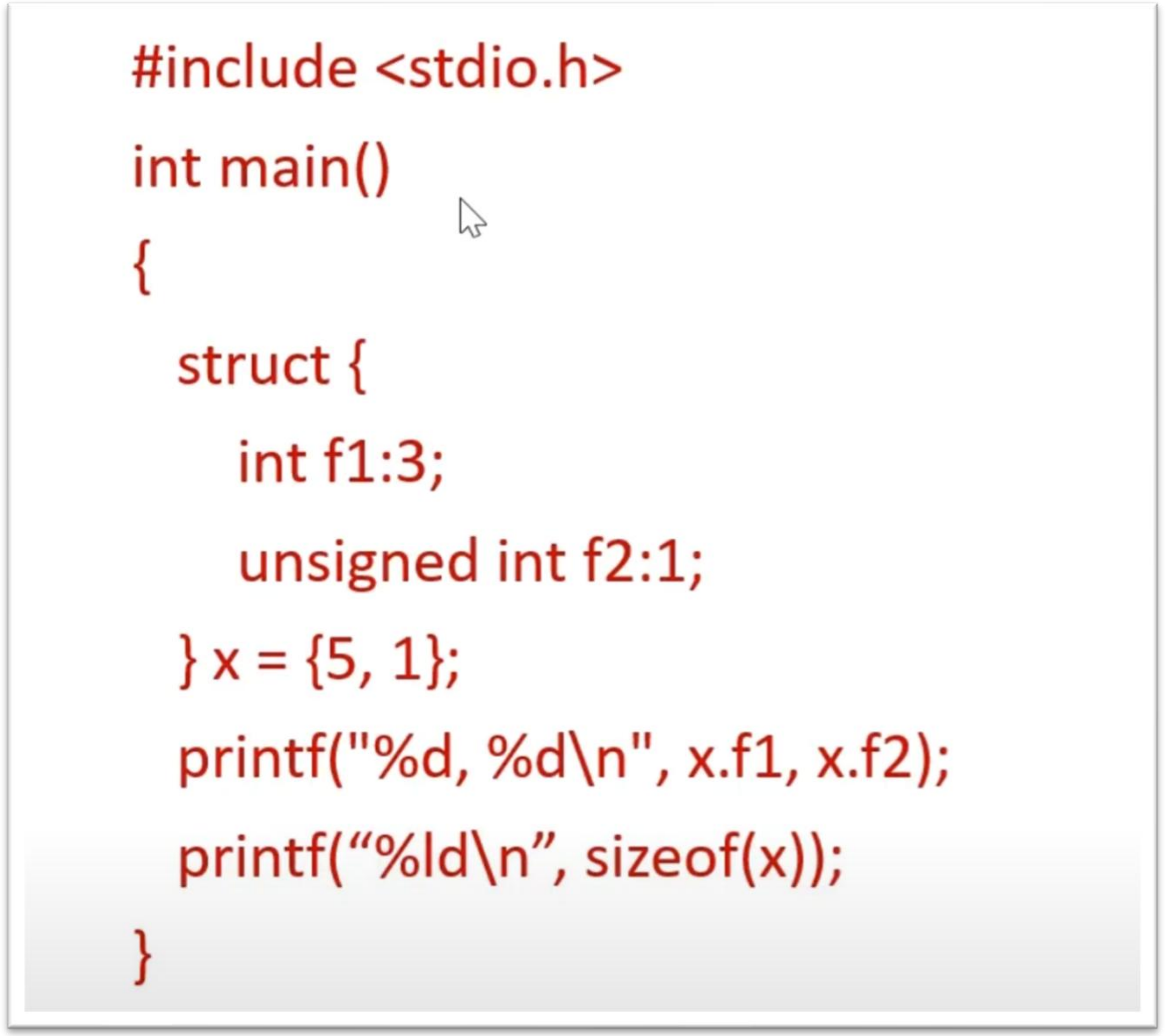
Test Questions



```
#include <stdio.h>
void main()
{
 char c1 = 'a', c2 = 'b', c;
 c = c1 + c2;
 if (c > 'c')
     printf("True\n");
 else
     printf("False\n");
}
```

CPR

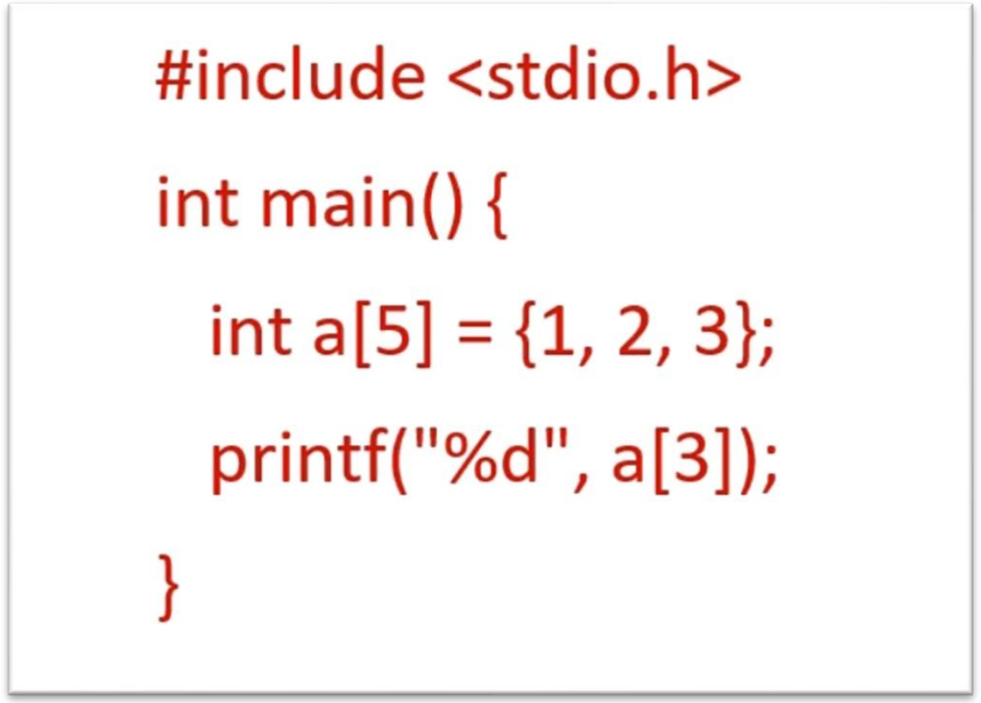
Test Questions



```
#include <stdio.h>
int main()
{
  struct {
     int f1:3;
     unsigned int f2:1;
  } x = {5, 1};
  printf("%d, %d\n", x.f1, x.f2);
  printf(“%ld\n”, sizeof(x));
}
```

CPR

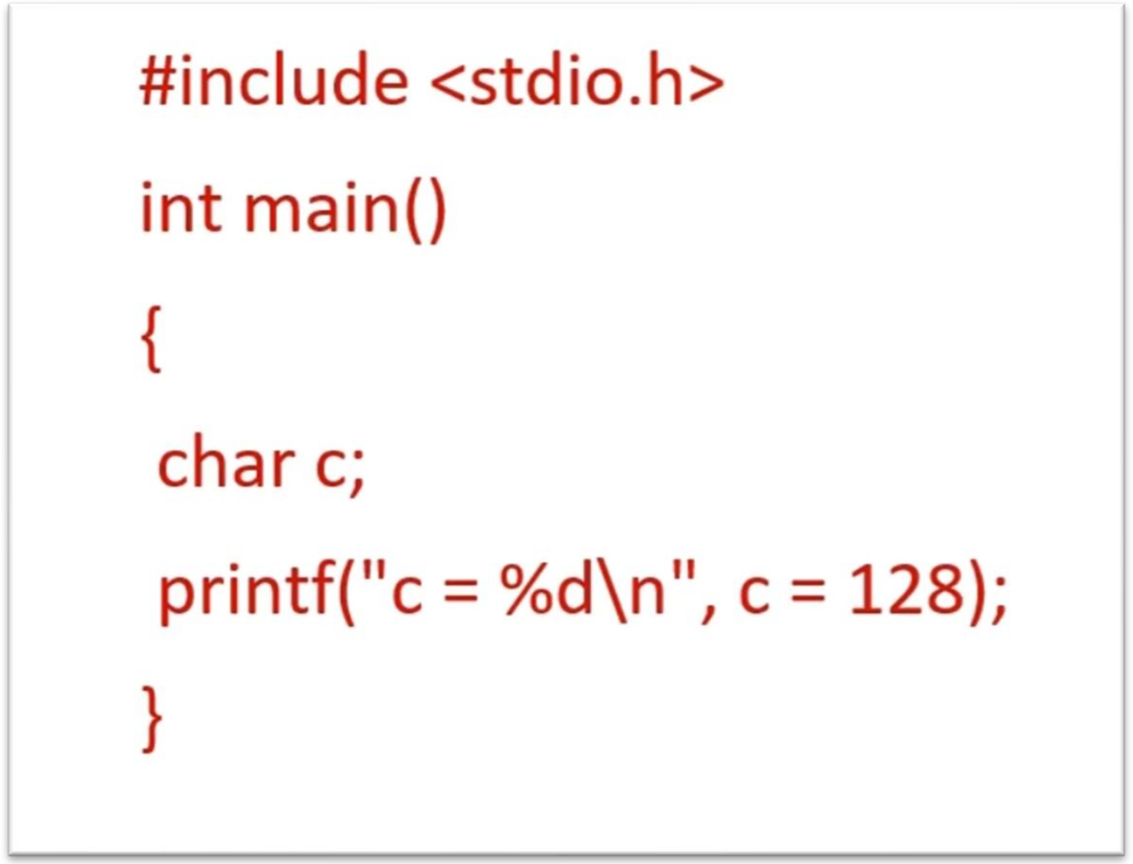
Test Questions



```
#include <stdio.h>
int main() {
  int a[5] = {1, 2, 3};
  printf("%d", a[3]);
}
```

CPR

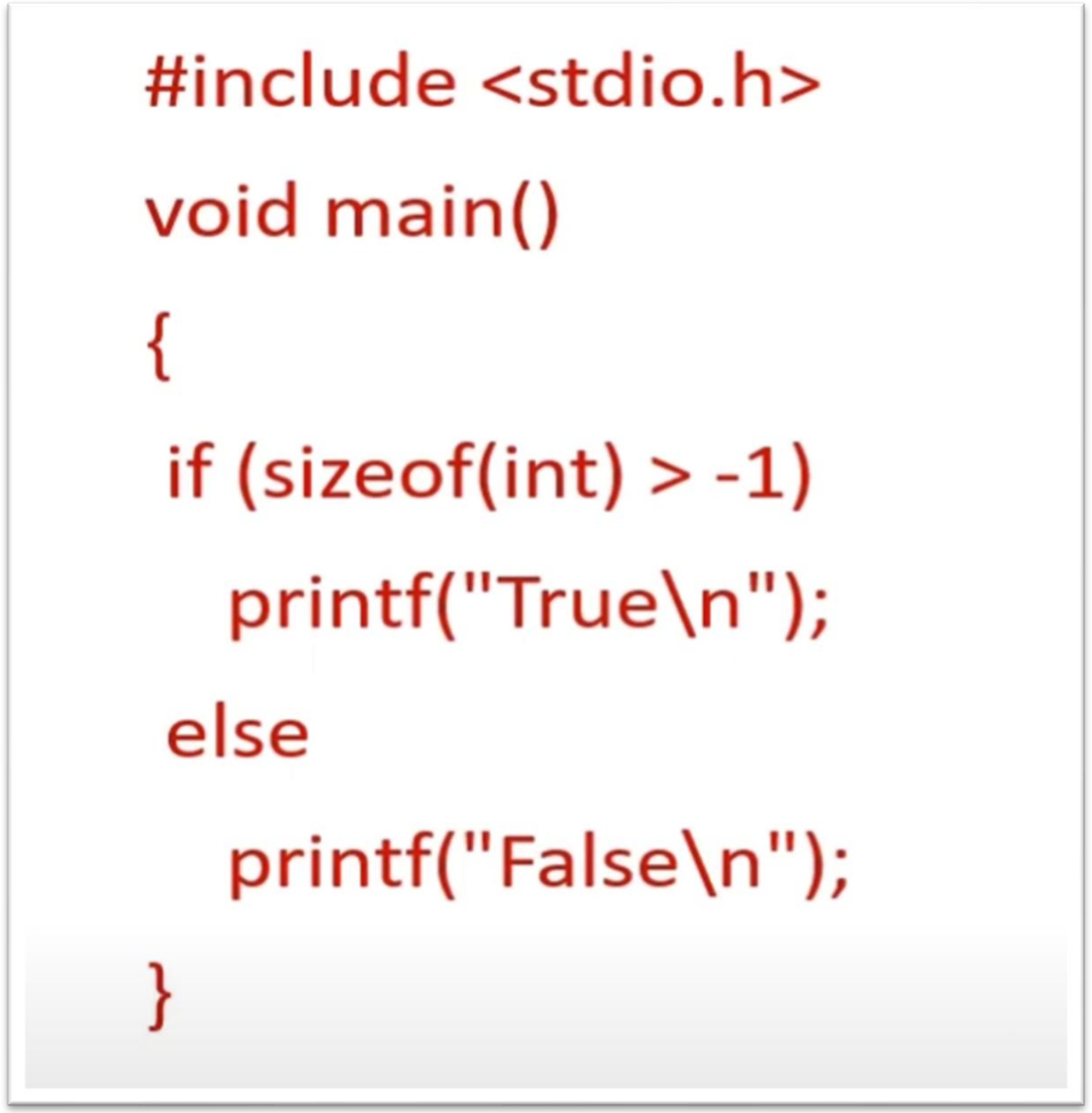
Test Questions



```
#include <stdio.h>
int main()
{
 char c;
 printf("c = %d\n", c = 128);
}
```

CPR

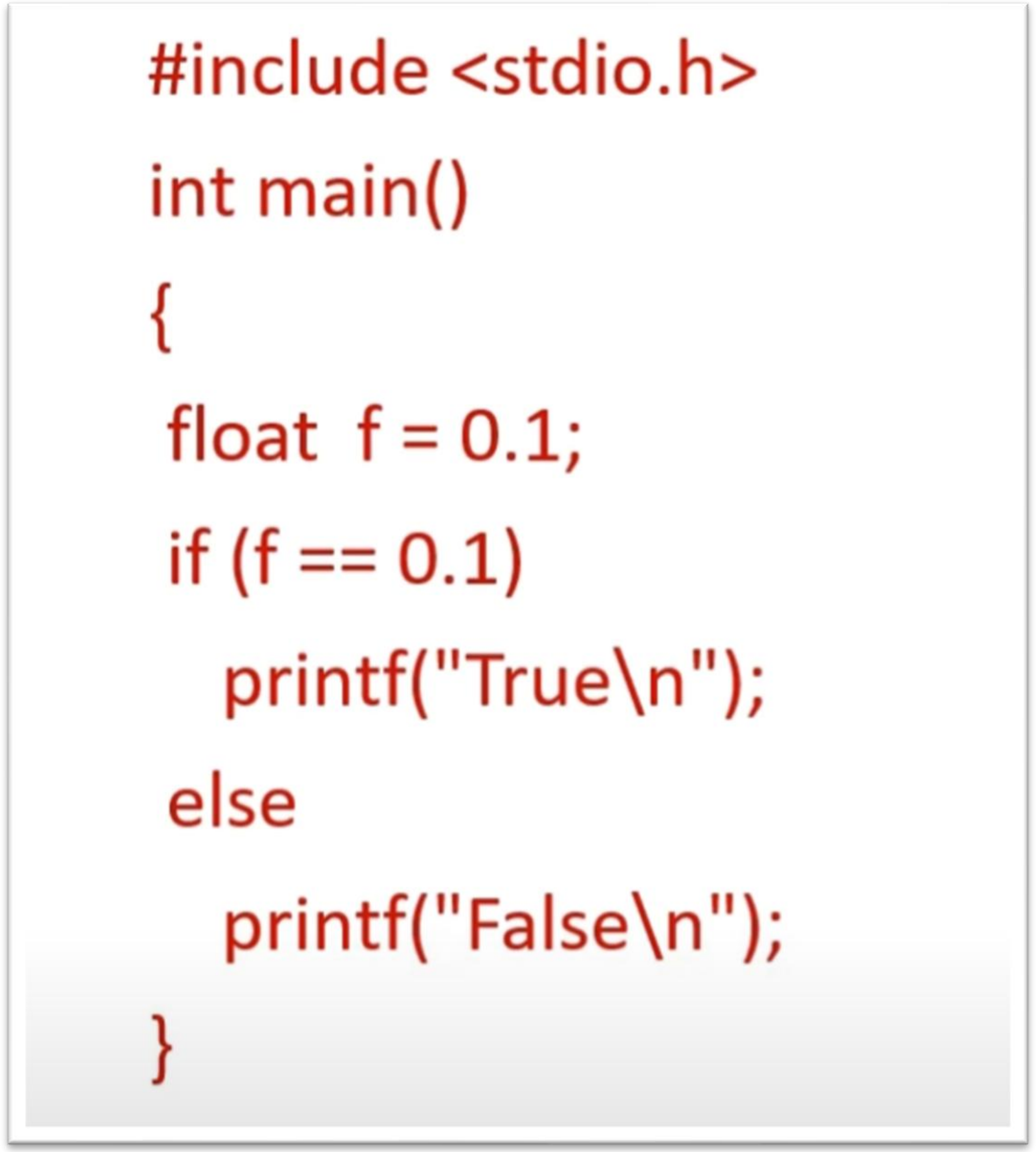
Test Questions



```
#include <stdio.h>
void main()
{
 if (sizeof(int) > -1)
    printf("True\n");
 else
    printf("False\n");
}
```

CPR

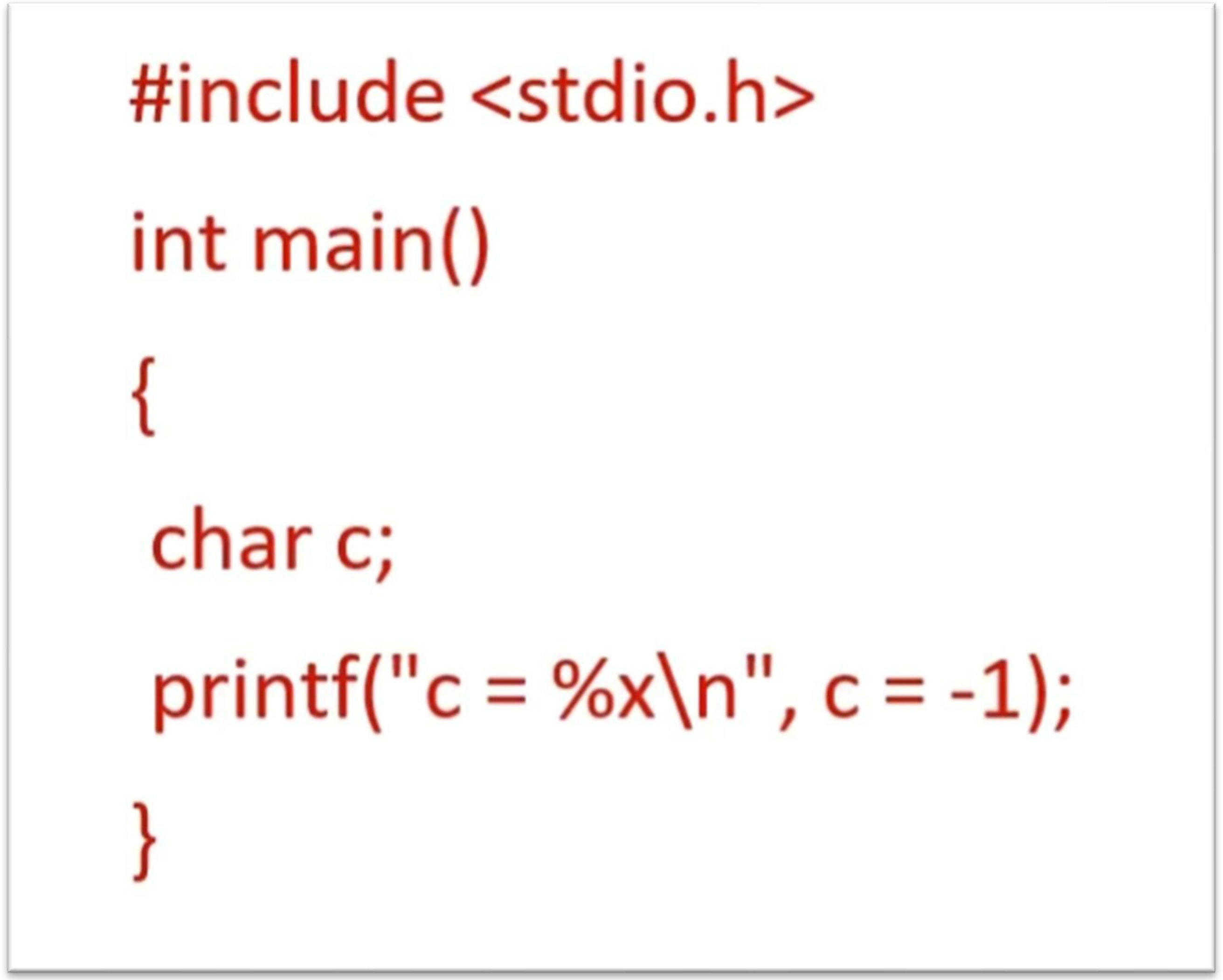
Test Questions



```
#include <stdio.h>
int main()
{
 float  f = 0.1;
 if (f == 0.1)
   printf("True\n");
 else
   printf("False\n");
}
```

CPR

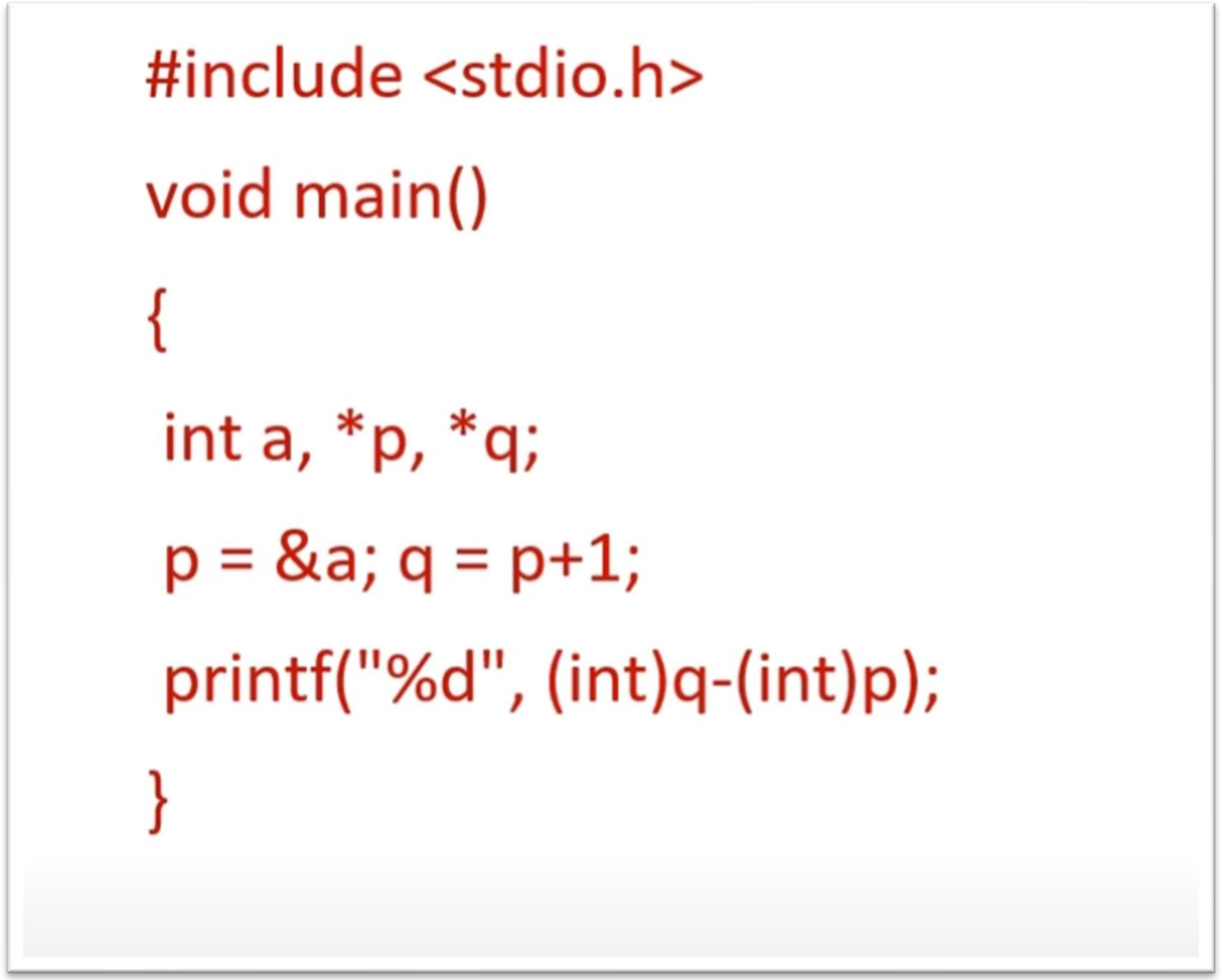
Test Questions



```
#include <stdio.h>
int main()
{
 char c;
 printf("c = %x\n", c = -1);
}
```

CPR

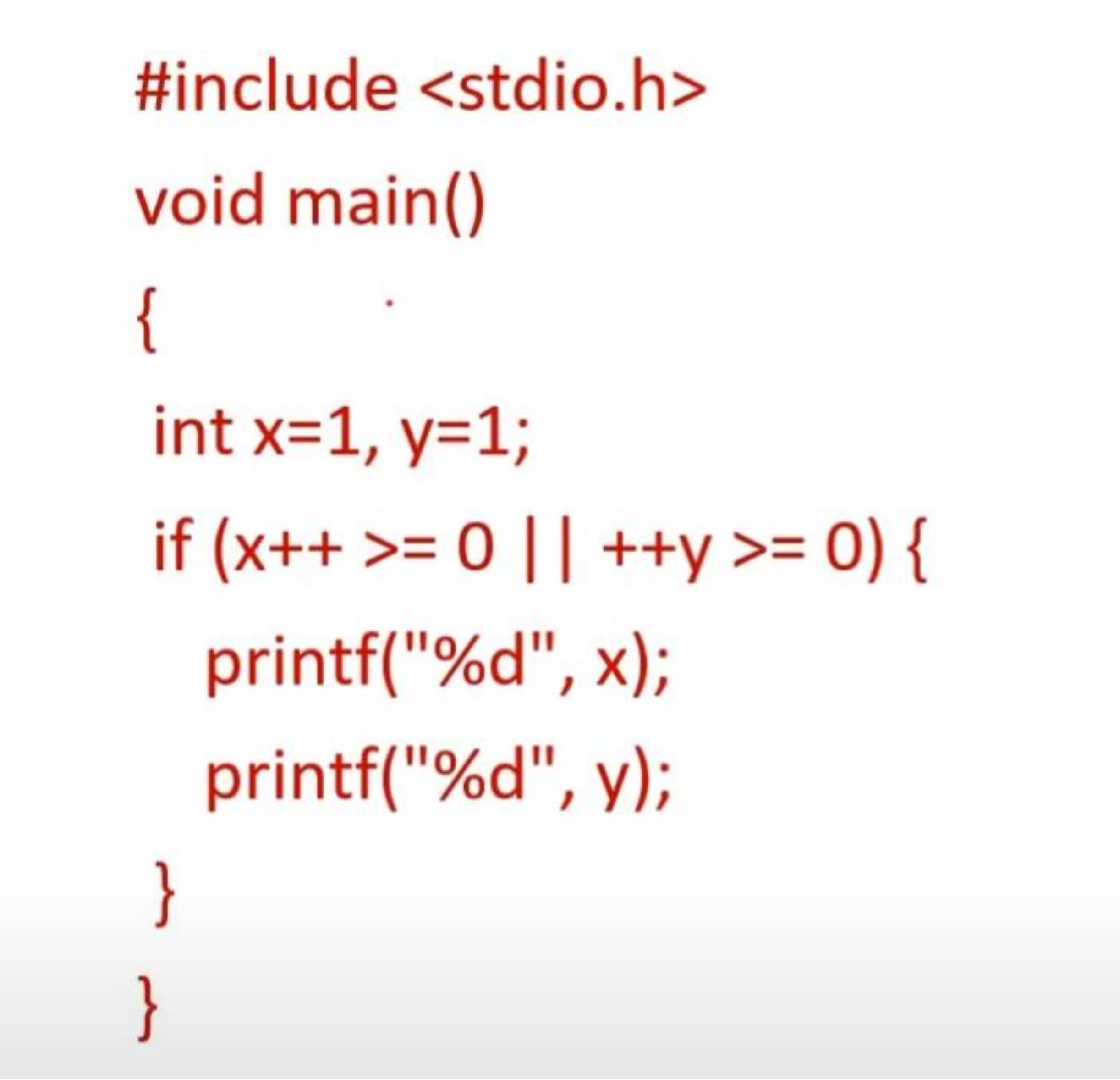
Test Questions



```
#include <stdio.h>
void main()
{
 int a, *p, *q;
 p = &a; q = p+1;
 printf("%d", (int)q-(int)p);
}
```

CPR

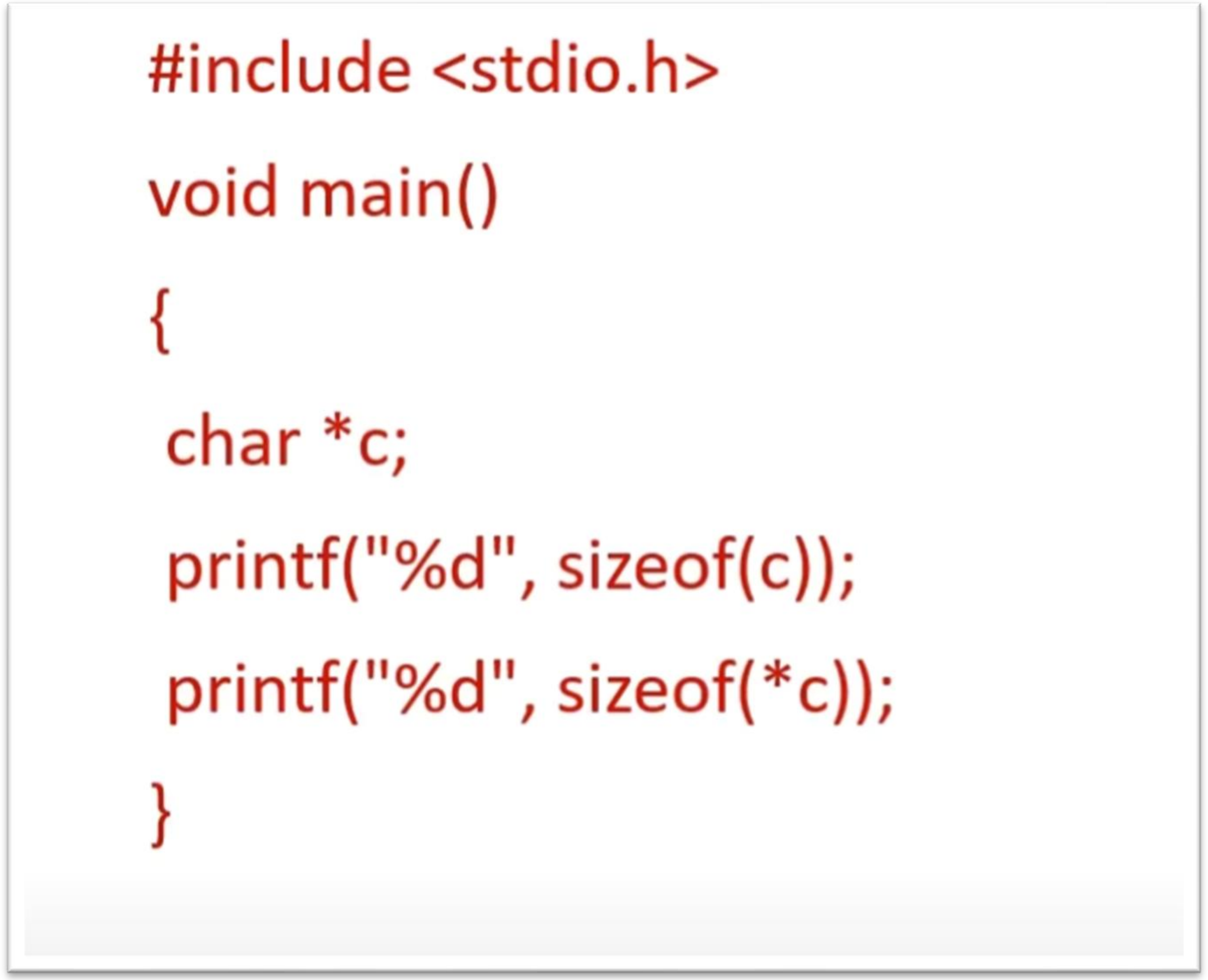
Test Questions



```
#include <stdio.h>
void main()
{
 int x=1, y=1;
 if (x++ >= 0 || ++y >= 0) {
    printf("%d", x);
    printf("%d", y);
 }
}
```

CPR

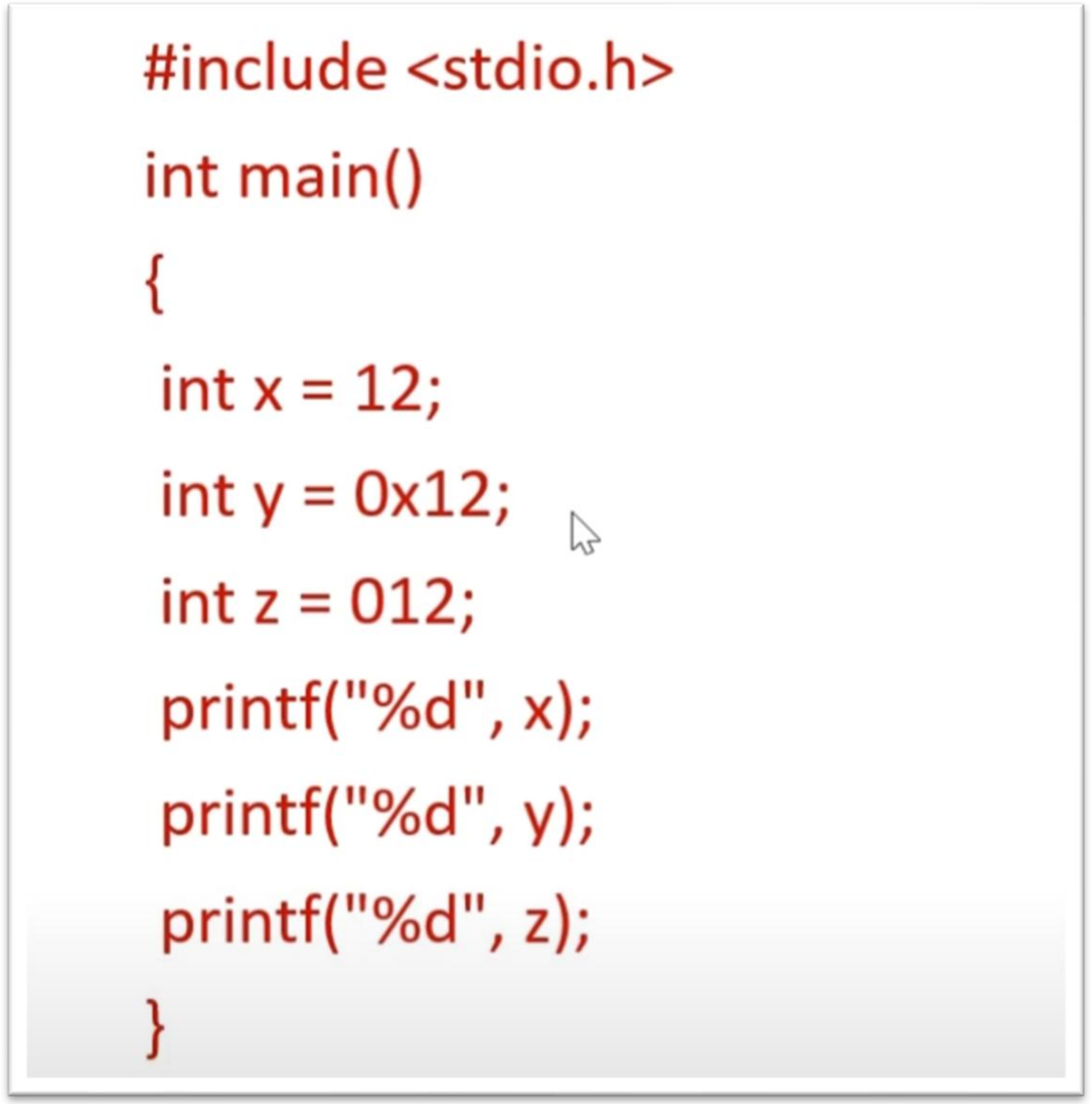
Test Questions



```
#include <stdio.h>
void main()
{
 char *c;
 printf("%d", sizeof(c));
 printf("%d", sizeof(*c));
}
```

CPR

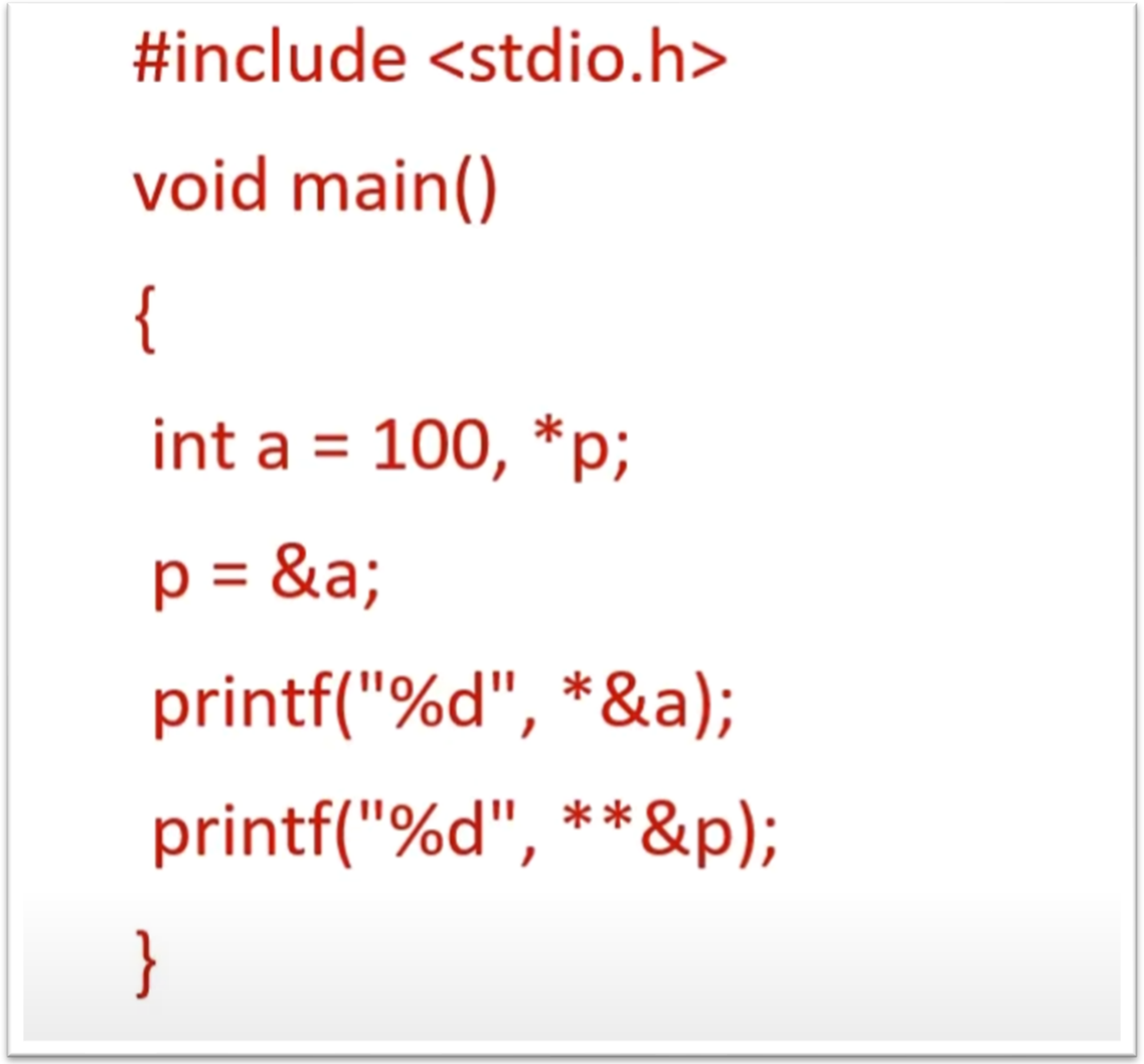
Test Questions



```
#include <stdio.h>
int main()
{
 int x = 12;
 int y = 0x12;
 int z = 012;
 printf("%d", x);
 printf("%d", y);
 printf("%d", z);
}
```

CPR

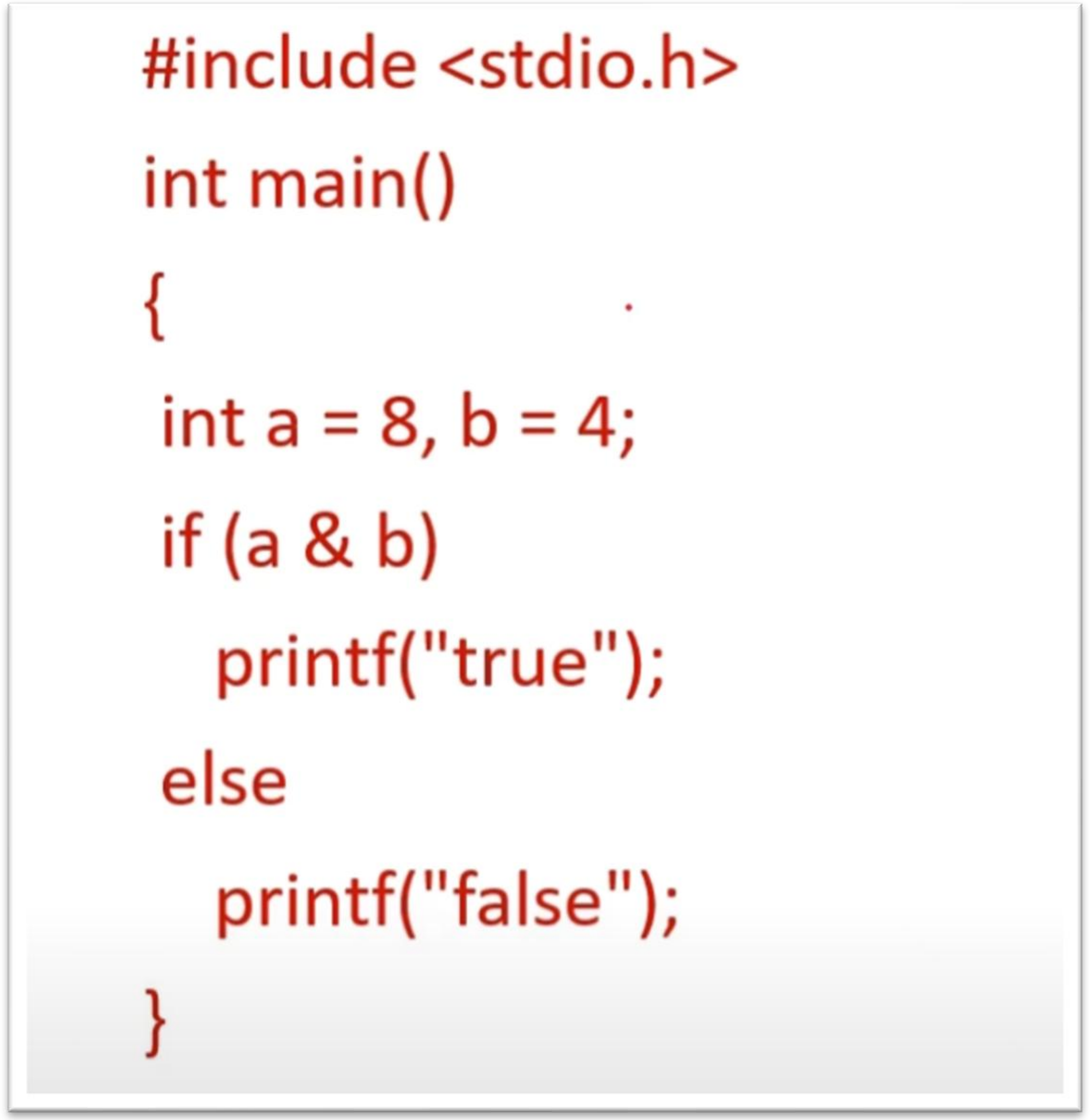
Test Questions



```
#include <stdio.h>
void main()
{
 int a = 100, *p;
 p = &a;
 printf("%d", *&a);
 printf("%d", **&p);
}
```

CPR

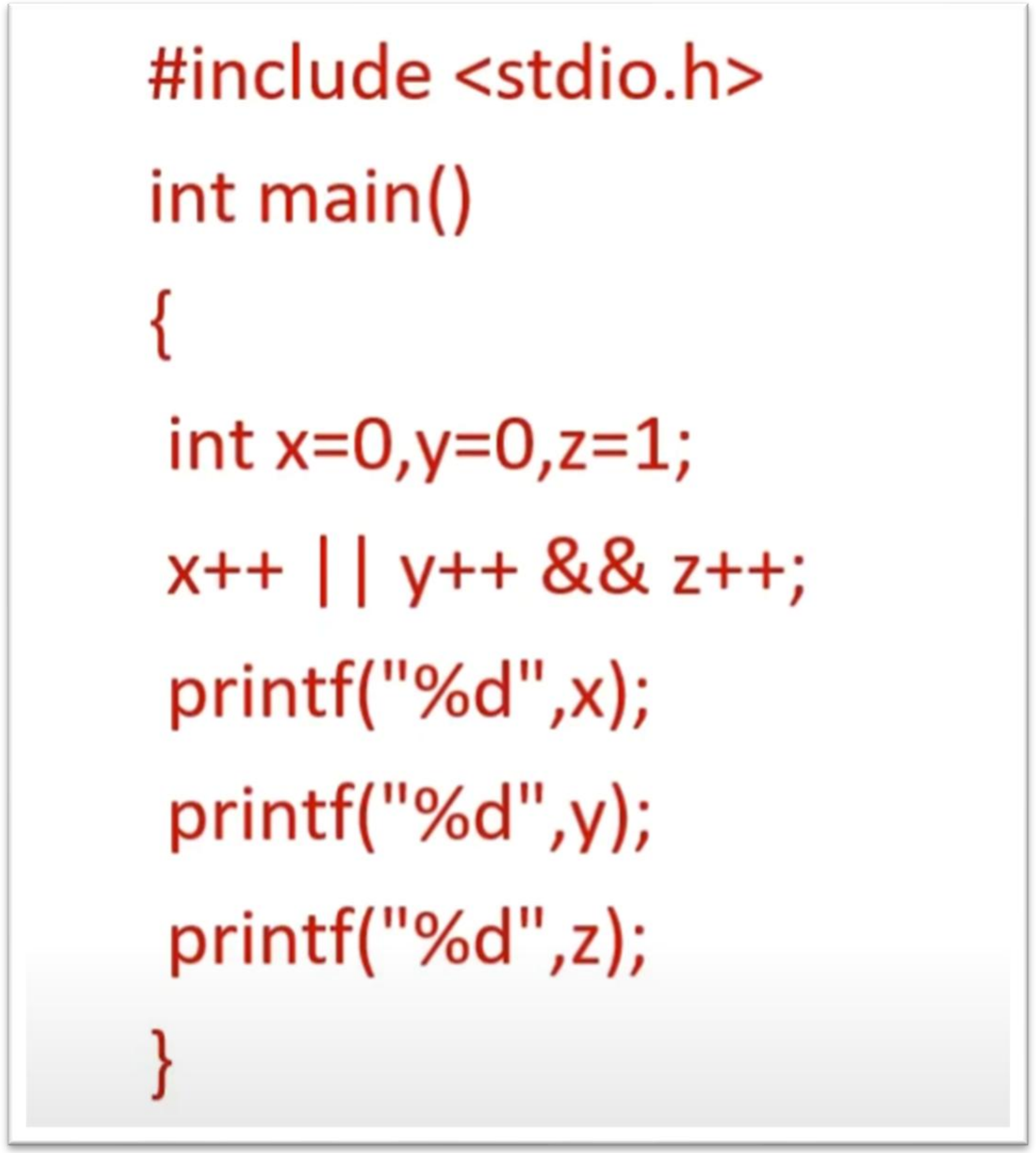
Test Questions



```
#include <stdio.h>
int main()
{
 int a = 8, b = 4;
 if (a & b)
    printf("true");
 else
    printf("false");
}
```

CPR

Test Questions



```
#include <stdio.h>
int main()
{
 int x=0,y=0,z=1;
 x++ || y++ && z++;
 printf("%d",x);
 printf("%d",y);
 printf("%d",z);
}
```

Whiteboard