Embedded C Interview Question and Answer.

Set - 4

[Linkedin](www.linkedin.com/in/uttam-basu/)

Owner	UttamBasu
Author	Uttam Basu
Linkedin	www.linkedin.com/in/uttam-basu/

Level - Easy

1) **What is the difference between const char *p, char *const p, and const char *const p? Explain with examples.**

A) const char *p

- **Meaning**: Pointer to a constant character.
- **The data is constant**, the pointer is not.
- You **cannot modify the value** being pointed to, but you **can change** the pointer to point somewhere else.

```
const char *p = "Hello";
p = "World";    // OK - changing the pointer
*p = 'h';       // ERROR - trying to modify read-only data
```

B) char *const p

- **Meaning**: Constant pointer to a character.
- **The pointer is constant**, the data is not.
- You **cannot change** the pointer, but you **can modify** the value it points to.

```
char str[] = "Hello";
char *const p = str;
*p = 'h';       // OK - modifying the data
p = str + 1;    // ERROR - changing the pointer
```

C) const char *const p

- **Meaning**: Constant pointer to a constant character.
- **Both the pointer and the data are constant**.
- You **cannot modify** the data or change the pointer.

```
const char *const p = "Hello";
p = "World";    // ERROR - cannot change pointer
*p = 'h';       // ERROR - cannot change value
```

Summary Table:

Syntax	Pointer Modifiable?	Data Modifiable?
const char *p	✅ Yes	❌ No
char *const p	❌ No	✅ Yes
const char *const p	❌ No	❌ No

[Uttam Basu]

2) What are the different storage classes in C (auto, register, static, extern)? How do they affect variable scope, lifetime, and linkage?

A) `auto` (Automatic Storage):

- **Default** for local variables (you don't usually write it explicitly).
- **Scope**: Local to the block (function or `{}`).
- **Lifetime**: Created when the block is entered, destroyed when it's exited.
- **Linkage**: None (not visible outside the function).

```
void func() {
    auto int x = 10;  // same as just "int x = 10;"
}
```

✅ **Used for**: Ordinary local variables
🚫 **Can't be accessed outside the function/block**

B) `register` (Register Storage)

- Suggests storing variable in a **CPU register** for faster access.
- **Scope**: Local to the block.
- **Lifetime**: As long as the block exists.
- **Linkage**: None.
- **Limitations**:
    - Can't take the **address** of a register variable.
    - It's just a **suggestion** to the compiler—may be ignored.

```
void func() {
    register int i = 0;
    // printf("%p", &i); // ❌ Error: can't take address
}
```

✅ **Used for**: Loop counters or frequently accessed variables
🚫 **Limited in modern compilers – often ignored**

C) `static` (Static Storage)

- Variable retains its value **across function calls**.
- **Scope**: Depends on where declared:
    - Inside a function: Local scope.
    - Outside all functions: Global scope.
- **Lifetime**: Entire program duration.

- **Linkage**:
  - **Internal linkage** if defined outside a function (not accessible from other files).
  - **No linkage** if inside a function (still private to that function).

```
void func() {
    static int count = 0;
    count++;
    printf("%d\n", count);
}
```

✅ **Used for**: Persisting state between function calls, or file-local globals
💡 Think of static like a "hidden global" when used inside a function

D) `extern` (External Linkage)

- Declares a variable defined **in another file or location**.
- **Scope**: Global.
- **Lifetime**: Entire program duration.
- **Linkage**: **External** – can be accessed across different files.

Declaration in one file:

```
extern int sharedVar;  // declared, not defined
```

Definition in another file:

```
int sharedVar = 42;    // definition and initialization
```

✅ **Used for**: Sharing global variables between files
🚫 **Only a declaration; must be defined elsewhere**

Summary Table:

Storage Class	Scope	Lifetime	Linkage
auto	Local block	Block	None
register	Local block	Block	None
static	Function/File local	Whole program	None/internal
extern	Global	Whole program	External

[Uttam Basu]

3) Can a void * pointer be dereferenced directly? Why or why not? How can you use it safely?

No, you cannot dereference a void * pointer directly.

Why not?

Because a `void *` (void pointer) is a **generic pointer**—it doesn't know what **type** of data it's pointing to, and therefore:

- It doesn’t know **how many bytes** to read or write.
- It has **no type information** for the data it points to.

```
void *ptr;
// *ptr = 10;     ❌ ERROR: invalid use of void expression
```

How to use it safely?

You need to **cast** the `void *` to the correct type **before dereferencing**:

```
int x = 42;
void *ptr = &x;

int y = *(int *)ptr;  // ✅ Safe: cast to int* before dereferencing
printf("%d\n", y);    // Outputs: 42
```

Rule of Thumb

> Always cast a `void *` to the **appropriate type** before using it.

Real-world Example

Used in **generic functions** like `qsort` or `memcpy`:

```
void printValue(void *data) {
    printf("Value: %d\n", *(int *)data);
}
```

Uttam Basu

Summary:

Property	void *
Generic pointer type	✅ Yes
Can be assigned any pointer	✅ Yes
Can be dereferenced directly	❌ No
Needs casting before use	✅ Yes (e.g., (int *), (char *))

4) Explain how function pointers work in C. Can you write a function that takes a function pointer as an argument and uses it?

A **function pointer** is a pointer that stores the **address of a function**.

Here's the basic syntax:

```
return_type (*pointer_name)(parameter_types);
```

Example:

```
int (*funcPtr)(int, int);  // pointer to a function taking two ints, returning int
```

You can assign it like this:

```
int add(int a, int b) {
    return a + b;
}
funcPtr = add;             // OR funcPtr = &add;
int result = funcPtr(2, 3);  // Call the function via pointer
```

Function Pointer as Argument Example

Let's write a **higher-order function**: it takes a function pointer and applies it to two numbers.

```c
#include <stdio.h>

// Two basic math operations
int add(int a, int b) {
    return a + b;
}

int multiply(int a, int b) {
    return a * b;
}

// Higher-order function: takes a function pointer as an argument
int compute(int x, int y, int (*operation)(int, int)) {
    return operation(x, y);  // Call the passed-in function
}

int main() {
    int result1 = compute(4, 5, add);
    int result2 = compute(4, 5, multiply);

    printf("Add: %d\n", result1);      // Outputs: 9
    printf("Multiply: %d\n", result2); // Outputs: 20

    return 0;
}
```

Why Use Function Pointers?

- Callbacks (e.g., `qsort()`)
- Strategy pattern (pick behavior dynamically)
- Implementing generic libraries
- Runtime flexibility

Syntax Cheatsheet:

Usage	Syntax
Declare a function pointer	int (*fptr)(int, int);
Assign a function to it	fptr = add;
Call function via pointer	fptr(3, 4); or (*fptr)(3, 4);
Pass function pointer to a func	int compute(int, int, int (*op)(int, int))

[Uttam Basu]()

5) Explain the #pragma directive. Can you give some real-world compiler-specific examples where it is useful?

What is #pragma?

- `#pragma` is part of the **preprocessor**.
- It's used to **enable/disable features or optimizations**.
- Behavior is **compiler-specific** – different compilers support different pragmas.
- If the compiler doesn't recognize a particular pragma, it **ignores** it (that's part of the design!).

General Syntax

```
#pragma directive_name
```

Real-World Examples

A) Suppress Warnings (GCC, Clang, MSVC)

```
#pragma GCC diagnostic ignored "-Wunused-variable"
```

Tells GCC to **ignore a specific warning** (in this case, unused variable).

Full usage:

```
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused-variable"

void foo() {
    int unused = 42; // No warning!
}

#pragma GCC diagnostic pop
```

Use `push`/`pop` to locally modify diagnostics.

B) Structure Packing (GCC/MSVC)

Control alignment of struct members to save memory or match external formats.

GCC / Clang:

```
#pragma pack(push, 1)
struct PackedStruct {
    char a;
    int b;
};
#pragma pack(pop)
```

MSVC:

```
#pragma pack(1)
```

Packed structs are useful when interfacing with **hardware or network protocols**.

C) Once-only Header Inclusion (#pragma once)

This is a **modern alternative** to include guards.

```
#pragma once
// Same effect as #ifndef/#define/#endif include guards
```

Supported by most compilers now (GCC, Clang, MSVC).

D) OpenMP Parallelism (GCC, Clang, MSVC with OpenMP)

Enable automatic parallelism on loops:

```
#include <omp.h>

#pragma omp parallel for
for (int i = 0; i < 1000; ++i) {
    // Runs in parallel on multiple threads
}
```

Great for **multi-core CPU parallelism** in numerical computing.

E) Deprecation Warning (MSVC)

Warn the user if they use a deprecated function:

```
#pragma deprecated(oldFunction)

void oldFunction() {
    // ...
}
```

Summary Table:

Use Case	Example	Compiler
Suppress warnings	#pragma GCC diagnostic ignored	GCC / Clang
Struct packing	#pragma pack(1)	GCC / MSVC
Header protection	#pragma once	All modern
Parallel for loop	#pragma omp parallel for	OpenMP enabled
Deprecation notice	#pragma deprecated(func)	MSVC

6) How does the C compiler resolve a function call when both a function prototype and a macro with the same name exist?

The macro wins.

When a function prototype and a macro have the **same name**, the **macro takes precedence**—because macros are handled **before compilation** during the **preprocessing phase**.

Why? Here's what happens under the hood:

1. **The preprocessor** runs first and replaces all macro invocations.
2. Only **after that**, the compiler sees the resulting code.

3. So if a macro and a function have the same name, the macro is expanded **before** the compiler ever sees the function name.

Example:

```
#include <stdio.h>

int square(int x) {
    return x * x;
}

#define square(x) ((x) * (x))

int main() {
    int result = square(5);
    printf("%d\n", result);
    return 0;
}
```

Output:

This will **not call the `square()` function**. Instead, the macro is expanded:

```
int result = ((5) * (5));
```

How to call the function anyway?

If you want to **force a function call**, you can **undefine** the macro:

```
#undef square  // Remove the macro
int result = square(5);  // Now this calls the actual function
```

Or just call the function **indirectly via a pointer**:

```
int (*squareFunc)(int) = square;
int result = squareFunc(5);  // This will call the function, not the macro
```

[Uttam Basu]()

Summary:

Conflict Type	Outcome
Function vs Macro (same name)	✅ Macro is used (preprocessor wins)
Want function?	❌ `#undef macro`, or use a function pointer

7) How does a union differ from a struct in terms of memory layout? When is a union useful?

Memory Layout: Struct vs Union

`struct`: All members exist simultaneously

- Memory is allocated for **each member separately**.
- Total size = **sum of all members** (plus padding).
- You can use **all members at the same time**.

```
struct MyStruct {
    int a;      // 4 bytes
    float b;    // 4 bytes
    char c;     // 1 byte + padding
};
```

Size ≈ 12 bytes (depends on alignment)

`union`: One memory block shared by all members

- All members **share the same memory location**.
- Memory is allocated for the **largest member only**.
- Only **one member can hold a valid value at a time**.

```
union MyUnion {
    int a;      // 4 bytes
    float b;    // 4 bytes
    char c;     // 1 byte
};
```

Size = 4 bytes (largest member)

Access Example

```
union MyUnion u;
u.a = 10;
printf("%d\n", u.a);    // OK
u.b = 5.5;
printf("%d\n", u.a);    // ❌ Undefined behavior (you changed the union type)
```

You can't use multiple fields at once safely—just one at a time.

When is a Union Useful?

A) Memory-efficient storage

- Useful in **embedded systems**, where memory is tight.
- You store different types in the **same space**, just not at the same time.

B) Tagged unions / Variants

- Store multiple possible data types, like a **variant** or **discriminated union**.
- Combine with an enum to track active member:

```
enum TypeTag { INT_TYPE, FLOAT_TYPE };

struct Variant {
    enum TypeTag tag;
    union {
        int i;
        float f;
    } data;
};
```

Now you can safely know **what the union currently holds**.

C) Bit-level manipulation

- Union lets you interpret the same memory in different ways.

```
union Data {
    float f;
    unsigned int i;
};

union Data d;
d.f = 3.14;
printf("Raw bits: %X\n", d.i);  // See binary representation of float
```

Summary Table:

Feature	struct	union
Members stored	All at once	One at a time
Memory usage	Sum of members	Size of largest member
Use cases	Grouping related data	Type variants, memory-saving
Safe multi-access	✅ Yes	❌ No (undefined behavior)
Type tracking	❌ Manual	✅ Use with enum tag + union