Function Pointers in C

Introduction to Function Pointers

- In C, **a function pointer** is a variable that stores the address of a function.
- Just like normal pointers point to data, **function pointers point to code** (functions).
- Syntax:

```
return_type (*ptr_name)(parameter_types);
```

Example:

```c
int add(int a, int b) {
    return a + b;
}

int main() {
    int (*fptr)(int, int);  // function pointer declaration
    fptr = add;             // assign function address
    printf("%d\n", fptr(2, 3));  // call via pointer
}
```

Calling Functions via Function Pointers

There are **two equivalent ways** to call a function pointer:

```
(*fptr)(a, b);  // Explicit dereference
fptr(a, b);     // Implicit dereference
```

✅ Both are correct and equivalent in C.

Why useful?

- Enables dynamic behavior — decide at runtime which function to call.

Function Pointer Example – Calculator

```c
#include <stdio.h>

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }
int mul(int a, int b) { return a * b; }
int divide(int a, int b) { return b ? a / b : 0; }

int main() {
    int (*ops[])(int, int) = {add, sub, mul, divide};
    char symbols[] = {'+', '-', '*', '/'};

    for (int i = 0; i < 4; i++)
        printf("10 %c 5 = %d\n", symbols[i], ops[i](10, 5));
}
```

Key idea:

- Store multiple functions in an array of function pointers.

Passing Function Pointers as Arguments

You can pass a function pointer to another function to create **higher-order functions**.

```c
void compute_and_print(int (*op)(int, int), int x, int y) {
    printf("Result: %d\n", op(x, y));
}
```

Usage:

```c
compute_and_print(add, 5, 3);
compute_and_print(mul, 5, 3);
```

Real Use Case — Filtering Arrays

We can use function pointers to **customize filtering logic**.

```c
#include <stdio.h>

int is_even(int n) { return n % 2 == 0; }
int is_positive(int n) { return n > 0; }

void filter(int *arr, int size, int (*cond)(int)) {
    for (int i = 0; i < size; i++)
        if (cond(arr[i]))
            printf("%d ", arr[i]);
}

int main() {
    int nums[] = {1, -3, 4, 0, 5, -6, 8};
    printf("Even numbers: ");
    filter(nums, 7, is_even);

    printf("\nPositive numbers: ");
    filter(nums, 7, is_positive);
```

🧩 Sorting Structures in C using `qsort()` and Function Pointers

🧠 `qsort()` Function Prototype

```
void qsort(void *base, size_t nitems, size_t size, int (*compar)(const void *, const void *));
```

- `base` → pointer to the array
- `nitems` → number of elements
- `size` → size of each element
- `compar` → **function pointer** to comparison function

The comparison function must:

- Return **negative** if first < second
- Return **zero** if equal
- Return **positive** if first > second

🧱 Example 1 — Sorting Structures by One Field

Problem

We have an array of `Student` structs, and we want to sort by **marks**.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct {
    char name[20];
    int marks;
} Student;

int compare_by_marks(const void *a, const void *b) {
    const Student *s1 = (const Student *)a;
    const Student *s2 = (const Student *)b;
    return s2->marks - s1->marks;  // descending order
}

int main() {
    Student students[] = {
        {"Alice", 88},
        {"Bob", 95},
        {"Charlie", 72},
        {"Diana", 88}
    };
    int n = sizeof(students)/sizeof(students[0]);
```

🔠 Example 2 — Sorting by Multiple Fields

We'll sort primarily by **marks descending**, and if marks are equal, by **name ascending**.

```
int compare_by_marks_then_name(const void *a, const void *b) {
    const Student *s1 = (const Student *)a;
    const Student *s2 = (const Student *)b;

    if (s2->marks != s1->marks)
        return s2->marks - s1->marks; // higher marks first
    return strcmp(s1->name, s2->name); // alphabetical
}
```

Now just replace the comparator in `qsort()` with this one.

This demonstrates **multi-key sorting** via function pointers.

🧮 Example 3 — Sorting an Array of Structs by Enum Field

```c
typedef enum { FICTION, NONFICTION, SCIFI } Genre;

typedef struct {
    char title[30];
    Genre type;
    float rating;
} Book;

int compare_by_genre_then_rating(const void *a, const void *b) {
    const Book *b1 = a, *b2 = b;
    if (b1->type != b2->type)
        return b1->type - b2->type; // FICTION < NONFICTION < SCIFI
    if (b1->rating < b2->rating) return 1;
    if (b1->rating > b2->rating) return -1;
    return 0;
}
```

🧩 Example 4 — Sorting Dynamically Allocated Struct Array

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int id;
    double score;
} Record;

int cmp_score(const void *a, const void *b) {
    const Record *r1 = a, *r2 = b;
    return (r1->score < r2->score) - (r1->score > r2->score); // trick: compact cmp
}

int main() {
    int n = 5;
    Record *records = malloc(n * sizeof(Record));
    for (int i = 0; i < n; i++) {
        records[i].id = i + 1;
        records[i].score = (rand() % 100) / 10.0;
    }

    qsort(records, n, sizeof(Record), cmp_score);

    for (int i = 0; i < n; i++)
        printf("ID: %d Score: %.1f\n", records[i].id, records[i].score);

    free(records);
}
```

🧠 Example 5 — Passing Different Comparators at Runtime

You can define multiple comparison functions and **select one dynamically** (e.g., via menu).

```c
int main() {
    int choice;
    printf("Sort by: 1) Marks 2) Name\n");
    scanf("%d", &choice);

    int (*cmp)(const void*, const void*) =
        (choice == 1) ? compare_by_marks : compare_by_marks_then_name;

    qsort(students, n, sizeof(Student), cmp);
}
```

This is a real **function pointer use case** — dynamic behavior selection.

Function Pointer to Function Pointer

You can even have a **pointer to a function pointer**, though it’s rare:

```c
int add(int a, int b) { return a + b; }

int main() {
    int (*f)(int, int) = add;
    int (**pf)(int, int) = &f;

    printf("%d\n", (**pf)(10, 20));
}
```

Summary

Concept	Example	Use Case
Function Pointer	`int (*fp)(int, int)`	Store or pass a function
Array of Function Pointers	`ops[] = {add, sub}`	Dynamic dispatch
Function Pointer Argument	`filter(nums, n, is_even)`	Higher-order behavior
Function Pointer in Library	`qsort()`	Custom comparison