Command Line Arguments in C

Command line arguments are values passed to a program when it is executed from the terminal (command prompt).
They allow us to provide **inputs to the program without using scanf or reading from a file**.

Example run:

```
./a.out hello 123 world
```

Here:

- `./a.out` → program name
- `hello`, `123`, `world` → command line arguments

Syntax of main() with Arguments

```
int main(int argc, char *argv[]);
```

Parameters:

1. argc (argument count)

An integer that stores the number of arguments passed.
Always ≥ 1 (since program name itself counts as the first argument).

2. argv (argument vector)

An array of strings (`char*[]`).
Each element stores one argument as a C-style string.

Example:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Number of arguments: %d\n", argc);

    for (int i = 0; i < argc; i++) {
        printf("Argument %d: %s\n", i, argv[i]);
    }
    return 0;
}
```

Run:

```
./a.out apple banana 42
```

Output:

```
Number of arguments: 4
Argument 0: ./a.out
Argument 1: apple
Argument 2: banana
Argument 3: 42
```

Important Points

- `argv[0]` → program name (may include path).
- `argv[1]` to `argv[argc-1]` → user-supplied arguments.
- All command line arguments are strings (`char*`), even numbers.
- To use as integers, convert with `atoi()` or `strtol()`.

Converting Arguments

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

int main(int argc, char *argv[]) {
    if (argc == 3) {
        int a = atoi(argv[1]);   // convert string to int
        int b = atoi(argv[2]);
        printf("Sum = %d\n", a + b);
    } else {
        printf("Usage: %s num1 num2\n", argv[0]);
    }
    return 0;
}
```

Run:

```
./a.out 10 20
```

Output:

```
Sum = 30
```

Function Pointers in C

- A **function pointer** is a pointer that stores the **address of a function**.
- Just like pointers can point to variables, they can also point to functions.
- Useful for calling functions dynamically, passing functions as arguments, or implementing callback mechanisms.

Example:

```
int (*funcPtr)(int, int);
```

`funcPtr` is a pointer to a function that takes two `int` arguments and returns an `int`.

Assigning a Function to a Pointer

- Function name itself represents its address.
- So `funcPtr = function_name;` or `funcPtr = &function_name;` both are valid.

Example:

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

funcPtr = sum;   // or &sum
```

Calling a Function using Pointer

Two equivalent ways:

```
(*funcPtr)(5, 7);   // Explicit dereference
funcPtr(5, 7);      // Shortcut, preferred
```

Example Program

```
#include <stdio.h>

int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

int main() {
    int (*operation)(int, int);

    operation = add;
    printf("Sum = %d\n", operation(10, 5));

    operation = sub;
    printf("Difference = %d\n", operation(10, 5));

    return 0;
}
```

Output:

```
Sum = 15
Difference = 5
```

Array of Function Pointers

We can store multiple function pointers in an array for easier selection.

```
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 main() {
    int (*ops[3])(int, int) = { add, sub, mul };

    printf("Add: %d\n", ops[0](3, 2));
    printf("Sub: %d\n", ops[1](3, 2));
    printf("Mul: %d\n", ops[2](3, 2));

    return 0;
}
```

Function Pointers as Arguments (Callbacks)

Functions can receive function pointers as parameters.

```
#include <stdio.h>

void execute(int x, int y, int (*operation)(int, int)) {
    printf("Result: %d\n", operation(x, y));
}

int add(int a, int b) { return a + b; }
int mul(int a, int b) { return a * b; }

int main() {
    execute(4, 5, add);
    execute(4, 5, mul);
    return 0;
}
```

Output:

```
Result: 9
Result: 20
```

Use Cases of Function Pointers

1. **Callbacks** (passing functions to other functions).
2. **Menu-driven programs** (selecting operation at runtime).
3. **Dynamic behavior** (deciding which function to call at runtime).
4. **Implementing event-driven systems** (like GUI libraries, OS kernels).

Memory Leak in C

- A **memory leak** happens when a program allocates memory dynamically (using `malloc`, `calloc`, or `realloc`) but does not free it with `free()`.
- The memory stays reserved, even though the program no longer needs it.
- Over time, this reduces available memory and may cause performance issues or program crashes.

Causes of Memory Leaks

1. **Not freeing allocated memory**

```
int *ptr = malloc(sizeof(int) * 5);
// used ptr...
// forgot free(ptr);
```

2. **Losing reference to allocated memory**

```
int *ptr = malloc(sizeof(int));
ptr = malloc(sizeof(int)); // old memory lost (leak)
free(ptr);                 // frees only latest allocation
```

3. **Returning pointer without freeing inside functions**

```
char* func() {
    char *s = malloc(20);
    // no free() here
    return s; // caller must remember to free
}
```

Example of Memory Leak

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

int main() {
    int *arr = malloc(100 * sizeof(int)); // memory allocated

    arr[0] = 10;
    arr[1] = 20;

    // forgot to free(arr);

    return 0;
}
```

When program ends, OS reclaims memory, but during long runs (like servers) this creates big problems.

How to Prevent Memory Leaks

1. **Always free memory** after use:

```
free(ptr);
```

2. **Set pointer to NULL** after freeing:

```
free(ptr);
ptr = NULL;
```

(prevents accidental reuse of dangling pointer).

3. **Avoid losing pointer references**:

```
int *ptr = malloc(10 * sizeof(int));
int *temp = ptr;   // backup pointer
free(temp);
```

4. **Use tools to detect leaks**:

`valgrind` (Linux)
`AddressSanitizer` (gcc/clang option)

Correct Example

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

int main() {
    int *arr = malloc(100 * sizeof(int)); // allocate memory
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    arr[0] = 42;
    printf("First element: %d\n", arr[0]);

    free(arr);    // free memory
    arr = NULL;   // avoid dangling pointer

    return 0;
}
```