C
PROGRAMMING

TRAIN YOUR MIND WITH PRACTICE

theDirector

AYMANE LAKSIMI

C Mastery

Train Your Mind, Command the Code

First edition

This book was professionally typeset on Reedsy.
Find out more at reedsy.com

Contents

1

The Flow Begins – What is ft_atoi

You can’t master code by copying it — you master it by understanding what it wants from you.

What is ft_atoi?

In C, you often receive numbers in the form of text — maybe as command-line arguments (argv[]) or inputs. But computers don’t *calculate* on strings. They calculate on **integers**. So, you need to convert the string ” -42” into the real number -42. That’s what ft_atoi stands for:

> *atoi* → ***ASCII to Integer***

Our goal is to write a function that does this conversion **from scratch**, without using any standard library functions like atoi().

The Final Result We Want:

We want our function to take:

```
char *str = "   -42";
```

And return:

```
int result = -42;
```

Breaking Down the Logic

Before we write a single line of code, **let's understand** the steps a human would take to read that string and figure out the number:

Step 1: Skip All Useless Spaces

Like " -42", you don't want the function to panic about the spaces.

Step 2: Handle Sign

What if it's -42 or +19? We need to detect that.

Step 3: Read Digits

Only digits matter after that. Once we see a non-digit like a letter or symbol, we stop.

Step 4: Build the Number

We need to convert '4' and '2' to the actual number 42, not ASCII codes.

Anatomy of ft_atoi

Here’s the full function, then we’ll **dissect it line by line**:

```
int ft_atoi(char *str)
{
    int sign = 1;
    int result = 0;
    int i = 0;

    // 1. Skip whitespaces
    while (str[i] == ' ' || str[i] == '\n' || str[i]
    == '\t' ||
           str[i] == '\v' || str[i] == '\f' || str[i]
           == '\r')
        i++;

    // 2. Handle sign
    if (str[i] == '-')
    {
        sign = -1;
        i++;
    }
    else if (str[i] == '+')
        i++;

    // 3. Read digits and build number
    while (str[i] >= '0' && str[i] <= '9')
    {
        result = result * 10 + (str[i] - '0');
        i++;
    }

    return sign * result;
}
```

Let’s Understand Every Line

int sign = 1;

We **assume** the number is positive. If we see -, we’ll flip it.

Skip All Whitespaces

```
while (str[i] == ' ' || str[i] == '\n' || str[i] ==
'\t' ||
       str[i] == '\v' || str[i] == '\f' || str[i] ==
       '\r')
    i++;
```

Why this many? Because strings might have:

- space ’ ’
- tab ‘\t’
- newline ‘\n’
- vertical tab ‘\v’
- form feed ‘\f’
- carriage return ‘\r’

We’re cleaning up the string before reading it.

Handle + and -

```
if (str[i] == '-')
{
    sign = -1;
    i++;
}
else if (str[i] == '+')
```

```
    i++;
```

If you see a -, it means the number is negative. We flip the sign to -1. If it’s a +, we keep going — no change.

Read the Digits and Build the Number

```
while (str[i] >= '0' && str[i] <= '9')
{
    result = result * 10 + (str[i] - '0');
    i++;
}
```

Let’s say you’re reading ‘4’ and ‘2’:

- ‘4’ - ‘0’ → 52 - 48 → 4
- ‘2’ - ‘0’ → 50 - 48 → 2

We multiply the current result by 10 and add the new digit each time.
Example:

- ‘4’ → result = 0 × 10 + 4 = 4
- ‘2’ → result = 4 × 10 + 2 = **42**

Final Return

```
return sign * result;
```

If the sign was -1, we return -42. Otherwise, it’s just 42.

Test with a main

```
#include <stdio.h>

int ft_atoi(char *str);

int main(void)
{
    printf("%d\n", ft_atoi("  -42"));     // -42
    printf("%d\n", ft_atoi("  +1234"));   // 1234
    printf("%d\n", ft_atoi("42"));         // 42
    printf("%d\n", ft_atoi("  987abc"));  // 987
    (stops at 'a')
    printf("%d\n", ft_atoi("  -00123"));  // -123
}
```

Summary

Steps * Action
1 Skip whitespaces
2 Detect sign (+ or -)
3 Read digits and build int
4 Multiply by sign and return

When you understand ft_atoi, you understand:

- String reading
- ASCII manipulation
- Clean parsing
- Control flow logic

2

Strings – The Hidden Power Behind Text

Strings are not just characters. They're living sequences with a purpose — to tell the machine what humans mean.

The Magic of ft_strcpy

Let's begin by solving a simple challenge:

> *"Copy a string from one place to another, manually."*

In C, you don't just say dest = src; — you have to **copy each character** yourself, because you're working with memory directly

he Goal of ft_strcpy

Let's say:

```
char src[] = "Aymane";
char dest[100];
```

After calling:

```
ft_strcpy(dest, src);
```

We want dest to contain the same characters as src, including the null terminator \0.

The Full Code

```
char *ft_strcpy(char *dest, char *src)
{
    char *start = dest;

    while (*src)
    {
        *dest++ = *src++;
    }

    *dest = '\0';

    return start;
}
```

Line-by-Line Breakdown

char *start = dest;

We save the **beginning** of dest because by the time the copy is done, dest will be pointing to the end. We want to return the full copied string, so we return the **start address** we saved.

while (*src)

This means:

> *"Keep going as long as the character in src isn't the null terminator \0."*

So, we loop over each character in src.

*dest++ = *src++;

Here's what this does:

- *src → get current char from src
- *dest = *src → assign it to dest
- src++ → move to next character in source
- dest++ → move to next position in destination

*dest = '\0';

After copying all characters, we add the null terminator \0 to **finish** the string. Without it, the string is not complete and may cause bugs when printed.

return start;

Why return start?

Because dest is now pointing to the **end** of the copied string.

start gives us the full beginning — the actual copy.

Test It with a main

```
#include <stdio.h>

char *ft_strcpy(char *dest, char *src);

int main(void)
{
    char src[] = "Aymane the Director";
    char dest[100];

    ft_strcpy(dest, src);
    printf("Copied string: %s\n", dest);

    return 0;
}
```

Real Understanding Check:

- What does *src++ mean? → Read the char, then move to the next.
- Why do we return start, not dest? → dest moves forward; start holds the beginning.
- What happens if we forget \0? → The string becomes **undefined**, leading to garbage output or crashes.

Related Functions Coming Soon:

- ft_strncpy – copy with a limit
- ft_strcmp – compare two strings
- ft_strlen – find how long a string is

All of these build on this foundation.

Summary

*Concept *What You Learned
*src++ and *dest++ How pointers move through strings
Null terminator \0 Why it matters and where to add it
Return start address Why returning the original pointer gives you the full string

3

ft_swap – Switching Values in Memory

Now that you've grasped what memory is and how variables live in that space, it's time to make your first bold move: **swap values directly in memory** using **pointers**.

Goal

Write a function that swaps the values of two integers using their memory addresses.

The Code:

```
void ft_swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Let’s Break It Down:

- int *a: A pointer that holds the **address** of an integer a.
- int *b: A pointer to integer b.
- *a and *b: These access the actual **values** stored in those addresses.
- temp = *a: Temporarily store the value of a.
- *a = *b: Copy the value of b into a’s location.
- *b = temp: Move the original value of a into b’s location.

This is **direct manipulation** of RAM. You just changed two boxes in memory.

Test It with a main

```c
#include <stdio.h>

void ft_swap(int *a, int *b);

int main(void)
{
    int x = 10;
    int y = 20;

    printf("Before: x = %d, y = %d\n", x, y);
    ft_swap(&x, &y);
    printf("After:  x = %d, y = %d\n", x, y);

    return 0;
}
```

Output:

```
Before: x = 10, y = 20
After:  x = 20, y = 10
```

Wisdom

You didn’t just pass variables. You passed **addresses**. You now **own memory**. Welcome to the club.

Next, we’ll explore how strings work in C—starting with calculating the length of a string manually.

4

ft_strlen – Finding the Length of a String

You’ve heard that C strings end with \0. That’s a null terminator, marking where the string stops.

Goal

Write a function that returns the length of a string (number of characters before \0).

The Code:

```
int ft_strlen(char *str)
{
    int length = 0;
    while (str[length] != '\0')
    {
        length++;
    }
    return length;
}
Let's Break It Down:
char *str: This points to the start of the string.
The loop goes character-by-character until it sees \0.
```

Let's Break It Down:

- char *str: This points to the start of the string.
- The loop goes character-by-character until it sees \0.
- length++ counts how many steps until the null terminator.

Test It with a main

```
#include <stdio.h>

int ft_strlen(char *str);

int main(void)
{
    char *text = "Aymane is learning C!";
    int len = ft_strlen(text);
    printf("Length: %d\n", len);
    return 0;
}
```

Output:

```
Length: 21
```

5

ft_strcpy – Copying Strings Like a Machine

In C, copying strings is a rite of passage. You don’t just say str1 = str2. You go byte-by-byte

Goal

Write a function that copies the string from src to dest, including the null terminator.

The Code:

```
char *ft_strcpy(char *dest, char *src)
{
    char *start = dest;
    while (*src)
    {
        *dest = *src;
        dest++;
        src++;
    }
    *dest = '\0';
    return start;
}
```

Let’s Break It Down:

*char *start = dest: Saves the start of destination to return it.*

**dest = *src: Copies character by character.*

*The loop stops when *src is \0.*

```
*dest = '\0': Ensures the copied string is
null-terminated.
```

Test It with a main

```
#include <stdio.h>

char *ft_strcpy(char *dest, char *src);

int main(void)
```

```
{
    char src[] = "Let's go!";
    char dest[20];

    ft_strcpy(dest, src);
    printf("Copied: %s\n", dest);
    return 0;
}
```

Output:

```
Copied: Let's go!
```

6

ft_strncpy – Controlled Copy

What if you want to copy only the first N characters? That's where ft_strncpy shines.

Goal

Copy up to n characters from src to dest. If src is shorter than n, pad with \0.

The Code:

```
char *ft_strncpy(char *dest, char *src, unsigned int
n)
{
    unsigned int i = 0;

    while (i < n && src[i])
    {
        dest[i] = src[i];
        i++;
    }
    while (i < n)
    {
        dest[i] = '\0';
        i++;
    }
    return dest;
}
```

Let's Break It Down:

- First loop copies actual characters.
- Second loop fills the rest with null bytes if src was shorter.
- Handles string truncation or padding.

Test It with a main

```
#include <stdio.h>

char *ft_strncpy(char *dest, char *src, unsigned int
n);

int main(void)
{
    char src[] = "Aymane";
    char dest[20];

    ft_strncpy(dest, src, 3);
    printf("Copied (3 chars): %s\n", dest);

    ft_strncpy(dest, src, 10);
    printf("Copied (10 chars): %s\n", dest);

    return 0;
}
```

Output:

```
Copied (3 chars): Aym
Copied (10 chars): Aymane
```

7

ft_strcmp – Comparing Strings Byte-by-Byte

Time to compare strings, byte-by-byte, ASCII-by-ASCII

Goal

Write a function that returns:

- 0 if strings are equal
- <0 if s1 is less than s2
- 0 if s1 is greater than s2

The Code:

```
```

Let's Break It Down:

Loop continues as long as characters are equal and not null.

When mismatch or null, subtract the two characters.

Test It with main():

```
#include <stdio.h>

int ft_strcmp(char *s1, char *s2);

int main(void)
{
    printf("Compare: %d\n", ft_strcmp("abc", "abc"));
    // 0
    printf("Compare: %d\n", ft_strcmp("abc", "abd"));
    // < 0
    printf("Compare: %d\n", ft_strcmp("abd", "abc"));
    // > 0
    return 0;
}
```

Output:

```
Compare: 0
Compare: -1
Compare: 1
```

8

ft_putstr – Printing a String with Low-Level Power

Now it's time to **print a string**, the C way. No fancy formatting—just raw output using low-level operations.

Goal

Create a function that displays a string character by character, using only **write()** from <unistd.h>.

The Code:

```
#include <unistd.h>

void ft_putstr(char *str)
{
    while (*str)
    {
        write(1, str, 1);
        str++;
    }
}
```

Let's Break It Down:

- char *str: Pointer to the beginning of the string.
- while (*str): Loop until it hits the null terminator (\0).
- write(1, str, 1):
- 1 = file descriptor for **standard output (stdout)**.
- str = address of the current character.
- 1 = number of bytes to write.

You're writing **one byte at a time**—like a typewriter.

Test It with main():

```
#include <unistd.h>

void ft_putstr(char *str);

int main(void)
{
    ft_putstr("Let's master C, step by step.\n");
    return 0;
}
```

Output:

```
Let's master C, step by step.
```

Why Not printf?

Because printf is **high-level**. It uses **a lot** of extra functionality behind the scenes. But in **system-level programming**, we stay close to the metal. write() is raw. It’s **power**.