Guia Completo para o Projeto ft_printf (42)

1. Introdução

O projeto **ft_printf** tem como objetivo recriar a função `printf()` da biblioteca padrão C. Este é um dos projetos mais emblemáticos do currículo da 42, pois testa a tua capacidade de manipular *variadic functions*, formatação de texto e organização modular de código. A função deve imitar o comportamento da função original, imprimindo na saída padrão e retornando o número total de caracteres escritos.

---

2. Objetivos de Aprendizagem

- Compreender o funcionamento de funções variádicas (`stdarg.h`).
- Aprender a analisar strings de formato (`%d`, `%s`, etc.).
- Implementar conversões numéricas e de caracteres.
- Melhorar a estruturação modular do código em C.
- Garantir o uso correto de memória (sem *memory leaks*).

---

3. Recursos Úteis

Subject Oficial

- Lê cuidadosamente o *subject* do projeto (disponível na Intra da 42).
- Verifica as funções permitidas e as conversões obrigatórias:
- `%c`, `%s`, `%p`, `%d`, `%i`, `%u`, `%x`, `%X`, `%%`
- O retorno da função deve ser o número total de caracteres impressos.

Testers Populares

Para validar o teu código contra o `printf` real: - Tripouille/printfTester - sawyerf/ft_printftest - paulo-santana/ft_printf_tester

Repositórios de Exemplo (GitHub)

Consulta outros projetos apenas para referência de estrutura e organização: - ft_printf no GitHub (pesquisa global)

Tutoriais e Leituras Recomendadas

- Documentação do stdarg.h (cppreference)
- Man page do printf
- 42Docs/ft_printf

---

4. Desenvolvimento Passo a Passo

Passo 1: Compreender `stdarg.h`

Para usar funções variádicas:

```c
#include <stdarg.h>

int ft_printf(const char *format, ...)
{
    va_list args;
    va_start(args, format);
    // ... teu código
    va_end(args);
}
```

Funções principais: - `va_start(args, last_argument)` — inicializa a lista. - `va_arg(args, type)` — obtém o próximo argumento. - `va_end(args)` — termina o uso da lista.

Passo 2: Estruturação do Projeto

Organiza o código de forma modular: - `ft_printf.c` → Função principal, percorre o formato e chama os *handlers*. - `parser.c` → Analisa o formato e identifica especificadores. - `handlers.c` → Funções que tratam cada tipo (`%c`, `%s`, `%d`, etc.). - `utils.c` → Funções auxiliares (itoa, strlen, write, base conversions).

Exemplo básico de estrutura:

```c
int ft_printf(const char *format, ...)
{
    va_list args;
    int count = 0;
    va_start(args, format);
    while (*format)
    {
        if (*format == '%')
        {
            format++;
            count += handle_format(*format, args);
        }
        else
            count += write(1, format, 1);
        format++;
    }
    va_end(args);
```

```
    return count;
}
```

Passo 3: Implementar Conversões em Ordem Lógica

1. `%c` → imprime um caractere simples.
2. `%s` → imprime uma string.
3. `%d` / `%i` → imprime inteiros com sinal.
4. `%u` → imprime inteiros sem sinal.
5. `%x` / `%X` → imprime em hexadecimal.
6. `%p` → imprime endereços de ponteiros.
7. `%%` → imprime o símbolo `%`.

Passo 4: Implementar Flags e Width

Depois das conversões básicas, adiciona suporte para: - `-` → alinhamento à esquerda. - `0` → preenchimento com zeros. - `width` → largura mínima. - `.` (precisão) → número máximo de caracteres ou dígitos.

---

5. Testes e Debug

Usa testers e ferramentas para garantir o funcionamento: - **Testers:** Tripouille, sawyerf. - **Valgrind:** deteta memory leaks. - **AddressSanitizer:** ajuda a encontrar *buffer overflows*. - **diff ou colordiff:** compara saídas do teu printf com o original.

---

6. Boas Práticas

- Mantém o código limpo e bem comentado.
- Usa nomes de funções descritivos (`ft_puthex`, `ft_putnbr`, etc.).
- Evita duplicação de código (usa funções utilitárias comuns).
- Verifica todos os retornos de `write()`.
- Garante que todos os `malloc()` têm `free()` correspondente.

---

7. Estrutura Final do Projeto

```
ft_printf/
│
├── ft_printf.c
├── parser.c
├── handlers.c
├── utils.c
```

```
├── Makefile
└── ft_printf.h
```

Makefile exemplo:

```
NAME = libftprintf.a
SRC = ft_printf.c parser.c handlers.c utils.c
OBJ = $(SRC:.c=.o)
CC = gcc
CFLAGS = -Wall -Wextra -Werror

$(NAME): $(OBJ)
    ar rcs $(NAME) $(OBJ)

clean:
    rm -f $(OBJ)

fclean: clean
    rm -f $(NAME)

re: fclean $(NAME)
```

---

8. Exemplos de Testes

Testes Básicos

```c
ft_printf("Hello, world!\n");
// Saída: Hello, world!
// Retorno: 13

ft_printf("Caractere: %c\n", 'A');
// Saída: Caractere: A
// Retorno: 13

ft_printf("String: %s\n", "42 Lisboa");
// Saída: String: 42 Lisboa
// Retorno: 15

ft_printf("Decimal: %d\n", 1234);
// Saída: Decimal: 1234
// Retorno: 13

ft_printf("Unsigned: %u\n", 4294967295);
// Saída: Unsigned: 4294967295
```

```c
// Retorno: 22

ft_printf("Hex: %x | HEX: %X\n", 255, 255);
// Saída: Hex: ff | HEX: FF
// Retorno: 19

ft_printf("Pointer: %p\n", (void*)0x1234);
// Saída: Pointer: 0x1234
// Retorno: 15 (pode variar)

ft_printf("Percent: %%\n");
// Saída: Percent: %
// Retorno: 11
```

Testes Avançados

```c
ft_printf("%10d\n", 42);
// Saída: '        42'
// (10 caracteres de largura)

ft_printf("%-10d\n", 42);
// Saída: '42        '

ft_printf("%010d\n", 42);
// Saída: '0000000042'

ft_printf("%.5d\n", 42);
// Saída: '00042'

ft_printf("%8.5d\n", 42);
// Saída: '   00042'

ft_printf("%-8.5d\n", 42);
// Saída: '00042   '
```

---

9. Conclusão

Com o `ft_printf`, vais dominar conceitos essenciais de C: - Parsing de strings. - Manipulação de *variadic arguments*. - Impressão formatada e recursão. - Organização e modularidade.

> Segue um ritmo constante: entende o problema, testa frequentemente e refatora o teu código. O objetivo não é apenas replicar `printf`, mas compreender como ele funciona por dentro.