

Ft_printf

A função deve funcionar como a printf.

Resumo da estratégia usada:

1. Encontre o caractere de especificação da conversão;
2. Caso encontre, imprima e contabilize em outra função que dependerá de cada tipo de variável. Se não, imprima caractere por caractere.
3. Usará va_list por usarmos diversas variáveis com tipos diferentes.
4. Deve retornar a contagem (tamanho da string) e a impressão.

Man page do printf

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

```
int printf(const char *restrict format, ...);
```

DESCRIÇÃO

As funções da família printf () produzem saída de acordo com um **formato** conforme os especificadores de conversão.

Escreve a saída sob o controle de uma **string de formato** que especifica como os argumentos subsequentes são convertidos para saída.

Cada especificação de conversão é **introduzido pelo caractere %** e termina com um especificador de conversão.

No meio pode haver (nesta ordem) zero ou mais sinalizadores , uma largura de campo mínima opcional , uma precisão opcional e um modificador de comprimento opcional .

A sintaxe geral de uma especificação de conversão é:

`%[$][flags][largura][.precisão][modificador de comprimento]conversão`

Especificadores para serem implementados na `ft_printf`

You have to implement the following conversions:

- `%c` Prints a single character.
- `%s` Prints a string (as defined by the common C convention).
- `%p` The void * pointer argument has to be printed in hexadecimal format.
- `%d` Prints a decimal (base 10) number.
- `%i` Prints an integer in base 10.
- `%u` Prints an unsigned decimal (base 10) number.
- `%x` Prints a number in hexadecimal (base 16) lowercase format.
- `%X` Prints a number in hexadecimal (base 16) uppercase format.
- `%%` Prints a percent sign.

Especificadores de conversão Um caractere que especifica o tipo de conversão a ser aplicado. Os especificadores de conversão e seus significados são:

d , i

O argumento `int` é convertido em notação decimal com sinal. A precisão, se houver, fornece o número mínimo de dígitos que deve aparecer; se o valor convertido exigir menos dígitos, é preenchido à esquerda com zeros. O padrão precisão é 1. Quando 0 é impresso com um explícito precisão 0, a saída está vazia.

o , u , x , X

O argumento `unsigned int` é convertido em unsigned octal (`o`), notação decimal sem sinal (`u`) ou hexadecimal sem sinal (`x` e `X`). As letras **abcdef** são usadas para conversões `x` ; as letras **ABCDEF** são usadas para `X` conversões. A precisão, se houver, fornece o mínimo número de dígitos que devem aparecer; se o valor convertido requer menos dígitos, é preenchido à esquerda com zeros. A precisão padrão é 1. Quando 0 é impresso com uma precisão explícita 0, a saída está vazia.

c

Se nenhum modificador **l** estiver presente, o argumento `int` será convertido para um *caractere não assinado* e o caractere resultante é escrito.

%

Um `'%'` é escrito. Nenhum argumento é convertido. O completo especificação de conversão é `'%%'`.

p

The `void *` pointer argument is printed in hexadecimal (as if by `%#x` or `%#1x`).

Para alocar uma string suficientemente grande e imprimir nela

```
char * make_message(const char *fmt, ...)
{
    int n = 0;
    size_t size = 0;
    char *p = NULL;
    va_list ap;

    /* Determina o tamanho necessário. */
    va_start(ap, fmt);
    n = vsnprintf(p, size, fmt, ap);
    va_end(ap);

    if (n < 0)
        return NULL;

    size = (size_t) n + 1; /* Um byte extra para '\0' */
    p = malloc(size);
    if (p == NULL)
        return NULL;

    va_start(ap, fmt);
    n = vsnprintf(p, size, fmt, ap);
    va_end(ap);

    if (n < 0)
    {
        free(p);
        return NULL;
    }
    return p;
}
```

VALOR DE RETORNO

Após o retorno bem-sucedido, essas funções retornam o número de caracteres impressos (excluindo o byte nulo usado para finalizar a saída para cordas).

Por que usar va_list

Quando não se sabe quantos argumentos serão passados para a função, há duas maneiras de criar a função: Uma maneira seria criar um ponteiro para uma matriz. Outra maneira seria escrever uma função que pode **receber qualquer número de argumentos**, como `avg(4, 12,2, 23,3, 33,3, 12,1)`.

A vantagem dessa abordagem é que é muito mais fácil alterar o código se você quiser alterar o número de argumentos. Algumas funções de biblioteca podem aceitar uma **lista variável de argumentos**, como o `printf`. Para usar essas funções, precisamos de uma variável capaz de armazenar uma lista de argumentos de tamanho variável → essa variável será do tipo `va_list`.

Sempre que uma função é declarada como tendo um **número indeterminado de argumentos**, no lugar do **último argumento você deve colocar uma reticência** (que se parece com `'...'`), então, `int a_function (int x, ...);` diria ao compilador que a função deve aceitar quantos argumentos o programador usar, desde que seja pelo menos um, sendo o primeiro, o `x`.



Precisaremos usar algumas **macros** (que funcionam como funções, e você pode tratá-las como tal) do arquivo de cabeçalho `stdarg.h` para extrair os valores armazenados no argumento variável `list--va_start`, que inicializa a lista, `va_arg`, que retorna o próximo argumento na lista e `va_end`, que limpa a lista de argumentos variáveis.

Por exemplo, o código a seguir declara uma `va_list` que pode ser usada para armazenar um número variável de argumentos.

`va_start` é uma macro que aceita dois argumentos:

1. uma `va_list`; = `a_list`
2. e o nome da variável que precede diretamente as reticências ("...").

Assim, na função `a_function`, para inicializar `a_list` com `va_start`, você escreveria `va_start (a_list, x);`

```
1 | int a_function ( int x, ... )  
2 | {  
3 |     va_list a_list;    
4 |     va_start( a_list, x );  
5 | }
```

`va_arg` recebe um `va_list` e um tipo de variável, e retorna o próximo argumento na lista na forma de qualquer tipo de variável informado.

Em seguida, ele se move para baixo na lista para o próximo argumento. Por exemplo, `va_arg (a_list, double)` retornará o próximo argumento, supondo que ele exista, na forma de um `double`.

Na próxima vez que for chamado, ele retornará o argumento após o último número retornado, se houver. Observe que você precisa saber o tipo de cada argumento - **isso é parte do motivo pelo qual printf requer uma string de formato!**

Quando terminar, use `va_end` para limpar a lista: `va_end(a_list);`

```

1  #include <stdarg.h>
2  #include <stdio.h>
3
4  /* this function will take the number of values to average
5     followed by all of the numbers to average */
6  double average ( int num, ... )
7  {
8     va_list arguments;
9     double sum = 0;
10
11     /* Initializing arguments to store all values after num */
12     va_start ( arguments, num );
13     /* Sum all the inputs; we still rely on the function caller to tell us how
14        * many there are */
15     for ( int x = 0; x < num; x++ )
16     {
17         sum += va_arg ( arguments, double );
18     }
19     va_end ( arguments );           // Cleans up the list
20
21     return sum / num;
22 }
23
24 int main()
25 {
26     /* this computes the average of 13.2, 22.3 and 4.5 (3 indicates the number of values to average) */
27     printf( "%f\n", average ( 3, 12.2, 22.3, 4.5 ) );
28     /* here it computes the average of the 5 values 3.3, 2.2, 1.1, 5.5 and 3.3 */
29     printf( "%f\n", average ( 5, 3.3, 2.2, 1.1, 5.5, 3.3 ) );
30 }

```

Resumo VA_LIST

A `va_list` é usada para armazenar informações de argumentos variáveis

Este tipo é usado como parâmetro para as macros definidas em `<stdarg.h>`, que serve para recuperar os argumentos adicionais de uma função.

A `va_start` inicializa um objeto desse tipo, de forma que as chamadas subsequentes recuperam sequencialmente os argumentos adicionais passados para a função pela `va_arg`.

Antes de uma função que inicializou uma `lista_va` com `va_start` para retornar, o `va_end` macro deve ser invocado.

As especificidades desse tipo dependem da implementação da biblioteca específica. Objetos deste tipo só devem ser usados como argumento para o `va_start`, `va_arg`, `va_end`, `va_copy` macros ou funções que as usam, como as funções de argumento variável em `<stdio.h>`.

<code>va_start</code>	Inicialize uma lista de argumentos de variável (macro)
<code>va_arg</code>	Recuperar o próximo argumento (macro)
<code>va_end</code>	Terminar usando a lista de argumentos de variável (macro)


```
int ft_printf(const char *str, ...)
```

... = nº indeterminado de argumentos.

```
{
```

```
va_list args;
```

A va_list é usada para armazenar informações de argumentos variáveis.

```
int i;
```

```
int count;
```

```
i = 0;
```

```
count = 0;
```

```
va_start(args, str);
```

Irá inicializar a va_list args, com char *str.

```
while (str[i])
```

```
{
```

```
if (str[i] == '%' && ft_strchr("cspdiuxX%", str[i + 1]))
```

Se i estiver na posição em que a variável seja igual a % e que a posição i + 1 seja um caractere.

```
{
```

```
count += print_format(args, str[i + 1]);
```

Irá contabilizar os itens printados por outra função (print_format). Porque o retorno da printf é o tamanho do impresso.

```
i++;
```

```
}
```

```
else
```

```
count += print_char(str[i]);
```

Se não possui %ch, irá imprimir caractere por caractere. E o resultado será a quantidade de caracteres.

```
i++;
```

```
}
```

```
va_end(args);
```

Encerra a limpa a lista. E retorna a contagem.

```
return (count);
```

```
}
```

```

int print_format(va_list args, const char format)
{
    int count;

    count = 0;
    if (format == '%')
        count += print_char(format);
    if (format == 'c')
        count += print_char(va_arg(args, int));
    if (format == 's')
        count += print_str(va_arg(args, char *));
    if (format == 'd' || format == 'i')
        count += print_nbr(va_arg(args, int));
    if (format == 'u')
        count += print_nbr_u(va_arg(args, unsigned int));
    if (format == 'x')
        count += print_hex_lower(va_arg(args, unsigned int));
    if (format == 'X')
        count += print_hex_upper(va_arg(args, unsigned int));
    if (format == 'p')
        count += print_ptr(va_arg(args, void *));
    return (count);
}

```

A função `print_format` irá printar e contabilizar de acordo com a especificação de conversão.

O `va_args` é utilizado para receber a `va_list` (*va_list args que foi definida na função `ft_printf`*) e um tipo de variável (*format*), e retorna o próximo argumento na lista na forma de qualquer tipo de variável informada.

Para printar cada formato, é necessário uma função que contabilize e printe.

```
int print_char(char c) %c
{
    write (1, &c, 1);
    return (1);
}
```

```
int print_str(char *str) %s
{
    int len;

    if (!str)
    {
        write (1, "(null)", 6);
        return (6);
    }
    len = ft_strlen(str);
    write (1, str, len);
    return (len);
}
```

```
int print_nbr(int nbr) %d ou %i
{
    char *str;
    int len;

    str = ft_itoa(nbr);
    len = print_str(str);
    free(str);
    return (len);
}
```

```
int print_nbr_u(unsigned int nbr) %u
{
    char *str;
    int len;

    str = ft_uitoa(nbr);
    len = print_str(str);
    free(str);
    return (len);
}
```

É usado o itoa para converter de número para string.

Printf retorna escrito (null), no caso de NULL.

É usado o uitoa, que nada mais é do que o itoa mas com unsigned int, para converter de número para string.

```
int print_hex_lower(unsigned int value)
{
    int    counter;
    char   *str;

    str = itoa_base(value, HEX_BASE_LOWER);
    counter = print_str(str);
    free(str);
    return (counter);
}
```

```
int print_hex_upper(unsigned int value)
{
    int    counter;
    char   *str;

    str = itoa_base(value, HEX_BASE_UPPER);
    counter = print_str(str);
    free(str);
    return (counter);
}
```

ft_print_hex.c

A função `print_hex` irá imprimir e contabilizar de acordo com o formato `x` ou `X`.

Para esta função, foi necessário criar a função `itoa_base`, em que especifica que não será decimal como no `itoa`, mas sim em hexadecimal.

Importante lembrar que `x` ou `X` não deve ser negativos, portanto, é usado `unsigned int`.

Foram utilizadas as macros `HEX_BASE`, como descrito no header:

```
# define HEX_BASE_LOWER "0123456789abcdef"
# define HEX_BASE_UPPER "0123456789ABCDEF"
```

```
static int print_preceding_string(char *str)
{
    int counter;

    counter = print_str(str);
    return (counter);
}

int print_ptr(void *ptr)
{
    int counter;
    char *str;
    unsigned long addr;

    if (ptr == 0)
        return (print_str("(nil)"));
    addr = (unsigned long long)ptr;
    counter = print_preceding_string("0x");
    str = itoa_base(addr, HEX_BASE_LOWER);
    counter += print_str(str);
    free(str);
    return (counter);
}
```

É necessário uma função para contabilizar e printar o que antecede o endereço de memória: "0x".

Se o valor apontado pelo ponteiro for zero, a printf deve retornar (nil).

Addr é o endereço de memória do ponteiro ptr.

Contabiliza o que precede a string "0x"

Converte o int long long para string em hexadecimal;
Contabiliza e printa a string;
Esvazia a string para evitar memory leak; e
Retorna o contador (que é o tamanho da string completa)

Curiosidades que aprendi

`uintptr_t` é um tipo unsigned int que é capaz de armazenar um ponteiro de dados. O que normalmente significa que é do mesmo tamanho que um ponteiro.

Um motivo comum para querer um tipo inteiro que possa conter o tipo de ponteiro é executar operações específicas de inteiro em um ponteiro ou poder manipular como um número inteiro usual, depois converter de volta.

Para que pode ser usado?

Principalmente para operações bit a bit em ponteiros. Lembre-se que em C não se pode realizar operações bit a bit em ponteiros.

Assim, para fazer operações bit a bit em ponteiros, seria necessário manipular ponteiros através de `uintptr_t` e, em seguida, executar operações bit a bit.

Hexadecimais

Os programadores usam hexadecimais porque fica muito mais fácil de converter o número pra binário. Em outras palavras, é muito mais fácil saber quais os bits de um número hexadecimal do que de um número decimal. (Continuação no próximo slide).

Por exemplo, considere o número **18232** em decimal. Como que fica esse número em binário? É difícil saber e teríamos que gastar um bom tempo fazendo a conversão ou usar a calculadora.

O mesmo número em hexadecimal é **0x4738** (é comum colocar um "0x" ou "#" na frente de números hexa pra sabermos que é hexa e não decimal), e olhando pra esse número dá pra saber na hora que o binário é **100011100111000**.

Fica muito mais rápido converter pra binário do que usar uma calculadora.

A resposta objetiva então é que usamos hexacimais pois facilita muito nossa vida na hora de trabalharmos com binário.

Unsigned Hexadecimal for integer: %x, %X

```
#include <stdio.h>
int main()
{
    int a = 15;
    printf("%x\n", a);
    return 0;
}
```

Output:

f

	binário	Unicode	c	observação
0	00000000	U+0000	\0	byte nulo
1	00000001	U+0001		não usamos
2	00000010	U+0002		não usamos
3	00000011	U+0003		não usamos
4	00000100	U+0004		não usamos
5	00000101	U+0005		não usamos
6	00000110	U+0006		não usamos
7	00000111	U+0007	\a	apito
8	00001000	U+0008	\b	backspace
9	00001001	U+0009	\t	tabulação
10	00001010	U+000A	\n	fim de linha
11	00001011	U+000B	\v	tab vertical
12	00001100	U+000C	\f	fim de página
13	00001101	U+000D	\r	carriage return
14	00001110	U+000E		não usamos
15	00001111	U+000F		não usamos

Bibliografia

Listas de argumentos variáveis em C usando va_list:

<https://www.cprogramming.com/tutorial/c/lesson17.html>

printf(3) — Linux manual page:

<https://man7.org/linux/man-pages/man3/printf.3.html>

Por que programadores usam Hexadecimais?

<https://www.manualdocodigo.com.br/hexadecimais/>

Especificações em C:

<https://www.geeksforgeeks.org/format-specifiers-in-c/>