ft_printf: recode libc printf

Welcome, fellow coders! I started putting this together for personal use but realized it is good information for anyone to have. I hope this is helpful for you during your ft_printf journey. If you see anything that is wrong or missing or could use some work, please leave a comment so I can update. Thank you!

- epines-s

ft_printf: recode libc printf

INSTRUCTIONS:

The prototype of ft_printf should be:

int ft_printf(const char *, ...);

- You have to recode the libc's printf function
- It must not do the buffer management like the real printf
- It will manage the following conversions: c s p d i u x X %
- It will manage any combination of the following flags: '-0.*' and minimum field width with all conversions
- It will be compared with the real printf

Bonus:

- If the Mandatory part is not perfect don't even think about bonuses
- You don't need to do all the bonuses
- Manage one or more of the following conversions: nfge
- Manage one or more of the following flags: l ll h hh
- Manage all the following flags: '# [space] +' (yes, one of them is a space)

External functions allowed:
Libc:
Malloc, free, write, va_start, va_arg, va_copy, va_end

Projects allowed:
libft

~~~well-structured and good extensible code~~~

GETTING STARTED:

You've just read through the pdf and you're thinking: now what? Well, this document is full of resources and information to help you get through this project. If you ask most people in the 42 network where to start, most people will tell you to google "man stdarg" and "man printf". Before even thinking about writing anything down, it is good to know what exactly printf does, how it works, and how it uses stdargs. In the Resource section, there are many options for information that you can use to start learning about printf and stdargs.

What I did next is google "implementing my own printf" and found this article that goes over printf, how it works internally, and gives an example of a **very very** simple function of printf with three specifiers and no flags. Although it is a simple example, it helped give me an idea of how to start writing my own printf. From here, there are many obstacles to face while writing your printf, but hopefully this helped give a better understanding on how to start this very daunting project.

Everyone has their own way of starting and writing printf. If you still feel like you don't know where to start, do more research, find more resources. Everyone struggles starting and everyone finds their own path on how they want to approach this project. Good luck!

RESOURCES:

Man printf:

"printf" :

- Really great starting resource
- Goes over how printf works, character sequence, and descriptions of every flag, specifier, modifier, etc.
- Trying command+f and searching for "convert" or "ignore"
    - A huge part of printf is converting argument

"Printf format string":

- Overview of printf - how it is used, what the flags mean
- Good starting resource but doesn't go into good detail
- Helps figure out what else you need to search for for more information

"How printf and scanf function works in c internally? (implementation of printf and scanf)":

- Implementation of printf
- Gives a very very very simple example of printf and how it uses stdargs
- Good starting point for those who don't really know where to begin!
- Also goes over scanf, but just ignore that part unless you're interested

Man stdarg:

"stdarg.h" :

- Simple overview of stdarg

"stdarg(3) - Linux man page":

- stdarg, va_start, va_arg, va_end, va_copy - variable argument lists

"C library macro - va_start()":

- Usage of va_start

Flags, specifiers, etc.:

"Printf - C++ Reference"

- Contrary to the title, it is for printf in C
- Describes each flag, specifier, etc.
- Gives information on combined flags

"Secrets of 'printf'":

- Describes flags in detail (probably more than needed)
- This article is to help you understand how to set up flags, width, specifiers for using printf to get desired results
- Gives many examples for each flag
- Gives information on combined flags
- **Does not** go over the [*] flag
    - THIS PROJECT DOES NOT REQUIRE PRECISION WITH FLOATING NUMBERS UNLESS YOU DO THE BONUS

Null inputs:

"Passing NULL to printf in C":

- Covers %s, %d, and %c

"NULL pointer in C"

- Somewhat goes over %p (output as "(nil)") and covers %d

Structures in C:

"Structures in C":

- Very basic overview of structs in C - how to define and use them and pointers to structs

"C - Structures":

- Same as above but tutorialspoint

"Struct (C programming language)":

- Same as above but wikipedia

ft_printf Tests:

"Ftprintfdestructor" by Tom Marx, 42 Paris:

- This project is a script that generates thousands of tests for the school 42 project ft_printf
- It doesn't test %p flag

"Printf_lover_v2": by charmstr, 42 Paris

- This checker generates .c files automatically according to the bonuses you select or not.
- Does not check for leaks

"42TESTERS-PRINTF" by mchardin, 42 Paris:

"PFT" by Gavin Fielder, 42 SV:

- Used in 42FileChecker, would start here, easiest to use and gives easy to read results
- I tried using the other tests before I tried this one with 42 File Checker, did not go well

"42-Tests" by Kwevan:

- This link includes many tests for the new curriculum.
    - These can be a bit overwhelming because they have a lot of tests in them, but once you have gotten a bit into the project, it is really useful for finding specific issues your program might be having!

Based on 42-Tests, I came up with this list of four easy to run, extensive tests, all inside the root directory of ft_printf:

```
git clone https://github.com/Kwevan/FT_PRINTF_TESTER tests451 && cd tests451
&& bash run.sh

git clone https://github.com/Mazoise/42TESTERS-PRINTF && cd 42TESTERS-PRINTF
&& bash runtest.sh

git clone https://github.com/gavinfielder/pft pft_2019 && echo "pft_2019/"
>> .gitignore && cd pft_2019 && rm unit_tests.c && rm options-config.ini &&
git clone https://github.com/cclaude42/PFT_2019 temp && cp temp/unit_tests.c
. && cp temp/options-config.ini . && rm -rf temp && ./test

git clone https://github.com/charMstr/printf_lover_v2 && cd printf_lover_v2
&& bash printf_lover.sh
```

Bonus:

"What is the use of %n in printf()?":

- Explains use of %n

NOTES:

Disclaimer: these are my personal notes I have written as I go through the project. It is what I have gathered while writing this function and it may be wrong or not complete. Do not use this as end all be all - tests are your friend. Thanks :)

Required Specifiers:
[c] [s] [p] [d] [i] [u] [x] [X] [%]
c = character
s = string
p = pointer address
d = decimal
i = integer
u = unsigned decimal integer
x = unsigned hexadecimal int
X = unsigned hexadecimal int (capital)
% = % character

Required Flags:
[-] [0] [.] [*]
- = left alignment, spaces
0 = zero fill option, zeros default right aligned
. = precision
* = The width is not specified in the format string, but as an additional integer value argument (given by the next va_arg) preceding the argument that has to be formatted.
.* = precision is an additional integer value argument given by the next va_arg

SPECIFIERS:

[s] string

- **Precision**
  - The *maximum* number of characters to be printed, for %s
  - If precision > strlen, len = strlen, if precision < strlen, len = precision
- **Width**
  - If width > len, allocate for width, else allocate for len
- **Zero**
  - Zero flag is undefined behavior with a string.
  - If you choose to include it, it will only work when the string is right aligned
- **Left Aligned**
  - only works when width > len
  - does not work with zero flag
- **Null string (str = NULL)**
  - String becomes string literal “(null)”
  - Flags are formatted just as any other string

[c] character, [%] percent character

- **Precision**
  - Precision does not affect how a character is printed.
- **Width**
  - If the width > 1, allocate for width, else allocate for 1
- **Zero**

    - Must have width
    - Must be right aligned
- **Left Aligned**
    - Only works when width > 1
    - Does not work with zero flag
- **Null characters (c = 0 or c = '\0'):**
    - Will be printed but cannot actually be seen by us
    - If you use a function to create your final string to be printed or if you print as you go, make sure that whatever function you use does not use null termination to determine the end of a string because if you do, it will see the null character and stop printing aka it will not print the null character.
        - i.e. what we see when it is printed "", what the computer sees: "\x00"

[p] pointers

- should include "0x" before pointer
- given in hexadecimal form
    - ft_itoa_hex, which means base 16
- **Precision**
    - if precision > strlen – 2, len = precision, else len = strlen – 2
    - -2 for the 0x required at the beginning
    - 0x always goes at the beginning, so if the precision > strlen – 2, 0s will be added as padding
        - i.e. 0x[pointer] vs. 0x0000000[pointer]
    - does not include the "0x"
- **Width**
    - if width > len, allocate for width, else allocate for len
    - includes the "0x"
- **Zero**
    - works with right aligned
    - only works when precision is less than width
    - "0x" always goes at the beginning so if width > len, 0s will be added as padding
        - i.e. 0x[pointer] vs. 0x0000000[pointer]
- **Left Aligned**
    - Only works when width > len
    - Does not work with zero flag
- **Null input and zero input**
    - if the input is zero, it should be written as 0x0 and formatted correctly with flags, width, and precision
    - if the input is NULL, should be written as string literal "(nil)"
        - I'm not sure how accurate this is because when I send NULL through, it shows up at "0x0". I would do your own tests and base what you do off that.

d: decimal, i: integer, u: unsigned, x: hexadecimal (lowercase), X; hexadecimal (uppercase)

[u]unsigned:

- can be taken from va_arg as int64_t type as long as it is type casted to unsigned before going into ft_itoa_unsigned function
    - or if you'd rather take va_arg as unsigned int that works too
    - as long as it is not an int
- Converted from decimal to unsigned
    - Which means final form does not have a negative sign
- Written as an unsigned int
    - ft_itoa_unsigned, which uses base 10 but must be converted with unsigned int type

[x] [X] hexadecimal:

- converted from decimal to hexadecimal
    - ft_itoa_hex, uses base 16
- x gives lowercase, X gives uppercase

[d] [i] decimal/integer:

- use ft_itoa for formatting
- if negative, must have negative sign in front in final formatting
- these two are pretty much interchangeable

FORMATTING

Positive:

- **Precision**
    - The *minimum* number of digits to be printed, for the integer formats %d, %o, %x, and %u.
    - If precision < strlen, len = strlen, if precision > strlen, len = precision
- **Width**
    - If width > len, allocate for width, else allocate for len
- **Zero**
    - Only works when right aligned
    - Only works when width > len and precision is set
        - Otherwise, precision already adds zeros if its greater than the length
- **Left Aligned**
    - Left aligns number
    - does not work with zero flag, becomes undefined behavior
- **NULL and 0**
    - If it is null, print nothing
    - If it is 0, print 0
        - UNLESS PRECISION IS SET TO 0, then you print nothing
    - Do tests to confirm these.

Negative: only matters for decimal and integer arguments

- **Precision**
    - If precision < strlen - 1, len = strlen , if precision > strlen - 1, len = precision
        - (- 1 for the negative sign)
    - Negative sign is always at the beginning if there is padding

        - i.e. -000004 or _____-00004 (if width > len)
- **Width**
    - If width > len, allocate for width, else allocate for len
    - If allocating for the len, you must add 1 for the negative sign, else allocation is just the width
        - i.e. precision = 15, width = 10, strlen = 5, string = "-12345", you should allocate for 16, and the output will be "-000000000012345"
        - i.e. precision = 10, width = 15, strlen = 5, string = "-12345" you should allocate for 15 and the output will be "____-0000012345" (_ is a space)
    - default right aligned
- **Zero**
    - Only works when right aligned
    - Only works when precision is not assigned
    - Negative sign is always at the beginning if there is 0 padding
        - i.e. -000004
- **Left Aligned**
    - Left aligns number
    - Does not work with zero flag, becomes undefined behavior

FLAGS:

[-] Left Alignment:

- Aligns string on the left and fills the rest with spaces (unless [0] flag is present)
- Does not work with the [0] flag

[0] Zero:

- For negative numbers, negative sign stays at the beginning of the string and the number gets right aligned (unless [-] flag is present, then it is left aligned)
- Will only work when you **do not** have a precision
- Undefined behavior if used with %s so you can make it work or not on your project

[1-9 or *] Width:

- **Always goes after flags** unless is specified with the [*] flag
- Can be any number but **cannot** start with 0
    - Must start with number between 1-9

[0-9 or .*] Precision:

- The *maximum* number of characters to be printed, for %s; or
- When precision < strlen, strlen = precision, when precision > strlen, strlen = strlen
- The *minimum* number of digits to be printed, for the integer formats %d, %o, %x, and %u.
- If specified with the [*] flag, it is taken using va_arg
- A ZERO PRECISION DOES EXIST. IF YOU INITIALIZE YOUR PRECISION TO ZERO IT MIGHT NOT WORK OUT WELL BECAUSE ZERO PRECISION VS UNASSIGNED PRECISION DO DIFFERENT THINGS. krantoverthnxbye.
- For this project, precision is not used with floating numbers (i.e. 3.1415 - a number with a decimal point)
    - Unless you do the bonus

- Can have a 0 precision (can start with any number between 0-9)
- Left align [-] flag does not work with the precision flag (see to the right). The 0's added to the number (assuming the specified precision is greater than the natural one) will always go on the left of the number, or in other words be right justified (credit: Teva Dagai, 42 Silicon Valley)
    - HOWEVER, if your width is greater than your precision, the whole number including the 0s included from precision are left justified
        - i.e. num = 12345 printf("%-15.*%d\n", 10, num) = 0000012345_____
        - this is just an example for %d, may differ for other specifiers, confirm using tests.

AND MORE

Structs:

- Can be used to assign and access for formats
- Can use a struct in a struct for easier use of navigating between functions and for memory management

Undefined Behaviors:

- About undefined behaviors: some say we're not supposed to reproduce them, in theory your program could do whatever it wants (including breaking, SEGV, SIGABRT) with undefined behaviors **[for example, using two incompatible flags (space and plus), using a flag that doesn't apply to the specified conversion, etc.]** but most cadets' testers include tests of undefined behaviors, which ends up leading oblivious cadets spending a lot of time dealing with them (credit: Amanda Pinha, 42 São Paulo)
- When it comes to undefined behaviors:
    - Feel free to replicate what printf does, but honestly, it is up to you how you want to deal with them.
    - They will not be tested by Moulinette
    - IF YOU FIND YOURSELF WORKING ON/WORRYING ABOUT UNDEFINED BEHAVIORS, REEVALUATE HOW IMPORTANT THEY ARE TO YOU BEFORE PROCEEDING

Debugging:

BONUS:
If you choose to do the bonus, it might be worth noting that the bonus is graded in the following order meaning if you get KOd at any point, even if you pass the rest, you will not get credit for them.
1. [n] 2. [f] 3. [g] 4. [e] 5. [l] 6. [ll] 7. [h] 8. [hh] 9. [#] 10. [ ] 11. [+]

[n] nothing:

- Nothing printed. The corresponding argument must be a pointer to a **signed int**. The number of characters written so far is stored in the pointed location.

[f] decimal floating, lowercase
[e] scientific notation (mantissa/exponent), lowercase
[g] use the shortest representation: %e or %f

- precision for the f case is actually (precision - amount of digits left of dot) with no trailing zeros
    - so say for example:
      dbl = 1234.56
      precision = 12
      it's going to pick f because the precision is actually 12 - 4 = 8 to ta maximum of 6 in this case cause that's the length of the number without trailing zeros
    - double dbl = 1234567.0
      int prec = 4

      Libc %e with precision of 4: |1.2346e+06|
      Libc %g with precision of 5: |1.2346e+06|
      ==========
      double dbl = 1234567.0
      int prec = 15

      Libc %e with precision of 15: |1.234567000000000e+06|
      Libc %f with precision of 0:
      |1234567|
      Libc %g with precision of 16:
      |1234567|

[l] long int [ll] long long int [h] short int [hh] signed char
See below for casting for each specifier:

	specifiers						
length	**d i**	**u o x X**	**f F e E g G a A**	**c**	**s**	**p**	**n**
(none)	int	unsigned int	double	int	char*	void*	int*
hh	signed char	unsigned char					signed char*
h	short int	unsigned short int					short int*
l	long int	unsigned long int		wint_t	wchar_t*		long int*
ll	long long int	unsigned long long int					long long int*
j	intmax_t	uintmax_t					intmax_t*
z	size_t	size_t					size_t*
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*
L			long double				

[#] pound sign

- Used with [o], [x] or [X] specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero.
- Used with [a], [A], [e], [E], [f], [F], [g] or [G] it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.

[ ] space

- If no sign is going to be written, a blank space is inserted before the value.

[+] plus sign

- Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.