ft_printf: recode libc printf

Instructions 1
Resources 2
Notes 3

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~~~

ft_printf: recode libc printf

Resources:

Man printf:

"Printf format string": https://en.wikipedia.org/wiki/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

Man stdarg:

"stdarg(3) - Linux man page": https://linux.die.net/man/3/stdarg

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

"C library macro - va_start()": https://www.tutorialspoint.com/c_standard_library/c_macro_va_start.htm

- Usage of va_start

Flags, specifiers, etc.:

"Printf - C++ Reference" http://www.cplusplus.com/reference/cstdio/printf/

- Describes each flag, specifier, etc.
- Gives information on combined flags

"Secrets of 'printf'": https://www.cypress.com/file/54441/download

- 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

ft_printf Tests:

"Ftprintfdestructor" by Tom Marx, 42 Paris: https://github.com/t0mm4rx/ftprintfdestructor

- 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 https://github.com/charMstr/printf_lover_v2

- 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: https://github.com/Mazoise/42TESTERS-PRINTF

"PFT" by Gavin Fielder, 42 SV: https://github.com/gavinfielder/pft

- **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**

Structures in C:

"Structures in C": https://www.geeksforgeeks.org/structures-c/

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

"C - Structures": https://www.tutorialspoint.com/cprogramming/c_structures.htm

- Same as above but tutorialspoint

"Struct (C programming language)": https://en.wikipedia.org/wiki/Struct_(C_programming_language)

- Same as above but wikipedia

ft_printf: recode libc printf

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. Thanks :)

What ft_printf will do in very simple terms

- Reads string
- Identifies when there is a %_ which is an argument
    - Identifies flags, specifiers, etc.
    - Formats arguments accordingly
    - Adds arguments with format to new string
- Rest of string is copied to new string as written in original string
- Prints final string at end

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

The *format specifier* can also contain sub-specifiers: *flags*, *width*, *.precision* and *modifiers* (in that order), which are optional and follow these specifications:

flags	description
–	Left-justify within the given field width; Right justification is the default (see *width* sub-specifier).
+	Forces to preceed the result with a plus or minus sign (+ or –) even for positive numbers. By default, only negative numbers are preceded with a – sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with `o`, `x` or `X` specifiers the value is preceeded 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.
0	Left-pads the number with zeroes (`0`) instead of spaces when padding is specified (see *width* sub-specifier).

width	description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The *width* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	description
.number	For integer specifiers (`d`, `i`, `o`, `u`, `x`, `X`): *precision* specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A *precision* of `0` means that no character is written for the value `0`. For `a`, `A`, `e`, `E`, `f` and `F` specifiers: this is the number of digits to be printed **after** the decimal point (by default, this is 6). For `g` and `G` specifiers: This is the maximum number of significant digits to be printed. For `s`: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for *precision*, `0` is assumed.
.*	The *precision* is not specified in the *format* string, but as an additional integer value argument preceding the argument that has to be formatted.

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

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)
- Undefined behavior if used with %s so you can make it work or not on your project

Width:

- **Always goes after flags** unless is specified with the [*] flag
- Can be any number (cannot start with 0)
- If length of string is less than width, rest is filled with spaces
    - Default is right aligned
- If length of string is more than width, width is ignored
- If flag [s] [c] or [%] and width < precision, printf will allocate space for width, not precision
    - Opposite goes for the rest of the flags or the "number" flags

Precision:

- The number of digits printed after the decimal point, for the floating-point formats %e, %f, and %g; or
- 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.
- 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
- If precision is less than the length of the number, the number is NOT cut down (see to the right).
    - i.e if you have precision 2 for 123, will it not print 123 instead of 12 because the natural precision is 3 (credit: Teva Dagai, 42 Silicon Valley)

```
int i = 123;
printf("%.2d\n", i);
-----------------------
Outcome:
123$
```

- Works differently when paired with other flags
    - If used with width, will default right align
    - Can have a 0 precision (width must start with number between 1-9)
    - Use tests for help with edge cases but I personally would start by making it work with simpler
    - 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_____

```
int i = 123;
printf("%.5d\n", i);
-----------------------
Outcome:
00123$
-----------------------
printf("%-.5d\n", i);
-----------------------
Outcome:
00123$
```

Structs:

- Can be used to assign and access for formats
    - I used one for zero, left align, width, and precision
- 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

Real Printf Outputs:

I decided to test a bunch of the same string / number with different combinations of flags using the REAL printf. This is helping me figure out what I can group together to create more efficient code. While trying to go through what I had already written and updated as I did smaller tests, I kept running into the problem of breaking another formatting rule which resulted in a test I already accounted for not working. So I decided that figuring out patterns can really help keep things organized rather than testing and updating as you go. You can use this as a base and then once you have the most basic formatting done, start to add more special cases.

Examples of how printf prints strings:

Same works for %c and %% HOWEVER using precision and flag [0] with %c is undefined behavior and using precision and flag [0] are not undefined behavior for specifier %% HOWEVER precision has no effect on the %% specifier.

UPDATE TO THIS: IT GETS MUCH MORE COMPLEX THAN THIS. ONLY USE THIS FOR A START IF YOU ARE HAVING TROUBLE. THERE ARE MANY SPECIAL CASES THAT ARE NOT SHOWN LIKE BELOW.

w = width, p = precision, s = strlen, string = ivank

Zero = 1 (undefined behavior for strings - you can include the 0's or not, printf does write the 0's but gives a warning)
Leftal = 0

p>w>s, 15>10>5	output: "00000ivank" (5 zeros)
p>s>w, 10>5>3	output: "ivank"
w>p>s, 15>10>5	output: "0000000000ivank" (10 zeros)
w>s>p, 10>5>3	output: "0000000iva" (7 zeros)
s>p>w, 5>4>3	output: "ivan"
s>w>p, 5>4>3	output: "0iva" (1 zero)

Zero = 0
Leftal = 0

p>w>s, 15>10>5 output: "_____ivank" (5 spaces)
p>s>w, 10>5>3 output: "ivank"
w>p>s, 15>10>5 output: "__________ivank" (10 spaces)
w>s>p, 10>5>3 output: "_______iva" (7 spaces)
s>p>w, 5>4>3 output: "ivan"
s>w>p, 5>4>3 output: "_iva" (1 space)

Zero = 1 (undefined behavior for strings, zeros only show up while right aligned)
Leftal = 1
p>w>s, 15>10>5 output: "ivank_____" (5 spaces)
p>s>w, 10>5>3 output: "ivank"
w>p>s, 15>10>5 output: "ivank__________" (10 spaces)
w>s>p, 10>5>3 output: "iva_______" (7 spaces)
s>p>w, 5>4>3 output: "ivan"
s>w>p, 5>4>3 output: "iva_" (1 space)

Zero = 0
Leftal = 1
p>w>s, 15>10>5 output: "ivank_____" (5 spaces)
p>s>w, 10>5>3 output: "ivank"
w>p>s, 15>10>5 output: "ivank__________" (10 spaces)
w>s>p, 10>5>3 output: "iva_______" (7 spaces)
s>p>w, 5>4>3 output: "ivan"
s>w>p, 5>4>3 output: "iva_" (1 space)

Examples of how printf prints decimals:
THIS IS ONLY ONE EXAMPLE USED WITH %d. IT MIGHT BE DIFFERENT FOR OTHER THINGS LIKE %p.
UPDATE TO THIS: IT GETS MUCH MORE COMPLEX THAN THIS. ONLY USE THIS FOR A START IF YOU ARE HAVING TROUBLE. THERE ARE MANY SPECIAL CASES THAT ARE NOT SHOWN LIKE BELOW.
w = width, p = precision, s = strlen, string = (-)12345
Negative: total stringlen: 5
Zero = 1
Leftal = 0
p>w>s, 15>10>5 output: "-000000000012345" (10 zeros, allocated extra space for "-")
p>s>w, 10>5>3 output: "-0000012345" (5 zeros, allocated extra space for "-")
w>p>s, 15>10>5 output: "____-0000012345" (4 spaces, 5 zeros, no extra allocated space)
w>s>p, 10>5>3 output: "____-12345" (4 spaces, no extra allocated space)
s>p>w, 5>4>3 output: "-12345"
s>w>p, 5>4>3 output: "-12345"

Zero = 1
Leftal = 1 (undefined behavior, [0] ignored when [-] flag present)
p>w>s, 15>10>5 output: “-000000000012345” (10 zeros, allocated extra space for “-”)
p>s>w, 10>5>3 output: “-0000012345” (5 zeros, allocated extra space for “-”)
w>p>s, 15>10>5 output: “-0000012345____” (5 zeros, 4 spaces, no extra allocated space)
w>s>p, 10>5>3 output: “-12345____” (4 spaces, no extra allocated space)
s>p>w, 5>4>3 output: “-12345”
s>w>p, 5>4>3 output: “-12345”

Zero = 0
Leftal = 0
p>w>s, 15>10>5 output: “-000000000012345” (10 zeros, allocated extra space for “-”)
p>s>w, 10>5>3 output: “-0000012345” (5 zeros, allocated extra space for “-”)
w>p>s, 15>10>5 output: “____-0000012345” (4 spaces, 5 zeros, no extra allocated space)
w>s>p, 10>5>3 output: “____-12345” (4 spaces, no extra allocated space)
s>p>w, 5>4>3 output: “-12345”
s>w>p, 5>4>3 output: “-12345”

Zero = 0
Leftal = 1
p>w>s, 15>10>5 output: “-000000000012345” (10 zeros, allocated extra space for ‘-’)
p>s>w, 10>5>3 output: “-0000012345” (5 zeros, allocated extra space for ‘-’)
w>p>s, 15>10>5 output: “-0000012345____” (5 zeros, 4 spaces, no extra allocated space)
w>s>p, 10>5>3 output: “-12345____” (4 spaces, no extra allocated space)
s>p>w, 5>4>3 output: “-12345”
s>w>p, 5>4>3 output: “-12345”

Positive: total stringlen: 6 stringlen without ‘-’: 5
Zero = 1
Leftal = 0
p>w>s, 15>10>5 output: “000000000012345” (10 zeros)
p>s>w, 10>5>3 output: “0000012345” (5 zeros)
w>p>s, 15>10>5 output: “_____0000012345” (5 spaces, 5 zeros)
w>s>p, 10>5>3 output: “_____12345” (5 spaces)
s>p>w, 5>4>3 output: “12345”
s>w>p, 5>4>3 output: “12345”

Zero = 1
Leftal = 1 (undefined behavior, [0] ignored when [-] flag present)
p>w>s, 15>10>5 output: “000000000012345” (10 zeros)
p>s>w, 10>5>3 output: “0000012345” (5 zeros)
w>p>s, 15>10>5 output: “0000012345_____” (5 spaces)
w>s>p, 10>5>3 output: “12345_____” (5 spaces)

s>p>w, 5>4>3 output: “12345”
s>w>p, 5>4>3 output: “12345”

Zero = 0
Leftal = 0
p>w>s, 15>10>5 output: “000000000012345” (10 zeros)
p>s>w, 10>5>3 output: “0000012345” (5 zeros)
w>p>s, 15>10>5 output: “_____0000012345” (5 spaces, 5 zeros)
w>s>p, 10>5>3 output: “_____12345” (5 spaces)
s>p>w, 5>4>3 output: “12345”
s>w>p, 5>4>3 output: “12345” (1 space)

Zero = 0
Leftal = 1
p>w>s, 15>10>5 output: “000000000012345” (10 zeros)
p>s>w, 10>5>3 output: “0000012345” (5 zeros)
w>p>s, 15>10>5 output: “0000012345_____” (5 zeros, 5 spaces)
w>s>p, 10>5>3 output: “12345_____” (5 spaces)
s>p>w, 5>4>3 output: “12345”
s>w>p, 5>4>3 output: “12345”

Debugging: