

# printf

▼ Class	2nd Circle
🕒 Created	@Dec 12, 2020 5:17 PM

## Summary:

This project is pretty straight forward. You will recode printf. Because putnbr and putstr aren't enough. Hopefully you will be able to reuse it in future project without the fear of being flagged as a cheater. You will mainly learn how to use variadic arguments.



The key to a successful ft\_printf is a **well-structured** and **good extensible code**.

## Prototype: `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: cspdiuxX%
- 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

## Dissecting `printf`

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

### Print formatted data to stdout

Writes the C string pointed by format to the standard output (`stdout`). If format includes format specifiers (subsequences beginning with %), the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers.

## 1. printf can take many arguments → variadic functions

```
printf("Hello world! %s %d %f..", str , int ... many more);
      arg1                arg2   arg3   argX
```

To handle this variable argument problem, we have to use a standard library called `stdarg.h`.

### Variadic functions

Unlike your traditional function that has an established set of arguments, a variadic function is able to receive an unknown amount of arguments and you will be able to cycle through them using a certain library.

You must declare your function prototype as you normally would and add three dots (...) as the last argument. What this does is it signals the possibility that there can be additional arguments and they are not specifically mandatory for the function.

In order to start creating a program that uses this structure you must include `<stdarg.h>` in the header section of your program along with the following macros so you are actually able to access all those data points a user might have entered:

The first macro you need to know about is `va_list`: this one will help you initialize the arguments pointer.

Second, you must use `va_start`: this one will point to the first element (of the mandatory one(s)). `va_start` will connect our argument list. The C library macro `void va_start(va_list ap, last_arg)` initializes `ap` variable to be used with the `va_arg` and `va_end` macros. The `last_arg` is the last known fixed argument being passed to the function i.e. the argument before the ellipsis (...). This macro must be called before using `va_arg` and `va_end`.

Third, use `va_arg` : this is the one that will help you point to the first of the optional arguments the user entered and every time it is called it will move over to the next argument.

Fourth, also optional, the `va_end` macro: this one is not really necessary since GCC will not even notice it but you could still add it in case you use a different compiler.

SO;

```
va_list someArgumentPointer;
va_start( someArgumentPointer, numberOfElements );
va_arg( someArgumentPointer, someType );
va_end( someArgumentPointer );

Example;
int addingNumbers( int nHowMany, ... )
{
    int          nSum =0;

    va_list      intArgumentPointer;
    va_start( intArgumentPointer, nHowMany );
    for( int i = 0; i < nHowMany; i++ )
        nSum += va_arg( intArgumentPointer, int );
    va_end( intArgumentPointer );

    return nSum;
}
```

## 2. Format specifiers

A format specifier follows this prototype:

`%[flags][width][.precision][length]specifier`

Most of these fields are optional, other than providing a conversion specifier, which you've already seen (for example, using `%d` to print out a decimal number).

Understanding this formatting is best done by working backward, starting with the conversion specifier and working outward. Where the specifier character at the end is the most significant component, since it defines the type and the interpretation of its corresponding argument:

### 2.1. Conversion specifiers

For our assignment > `cspdiuxX%`

The available conversion specifiers are:

- `%c` a single character.
- `%s` a string of characters.
- `%p` Pointer address
- `%d or i` signed decimal integers.
- `%u` unsigned decimal integers
- `%x` unsigned hexadecimal integer in lowercase letters
- `%X` unsigned hexadecimal integer in uppercase letters
- `%` % followed by another % character will write a single % to the stream.

Not included for our assignment but available on real printf

- `%o` unsigned octal integers
- `%f` decimal floating point, lowercase
- `%F` decimal floating point, uppercase
- `%a` Hexadecimal floating point, lowercase
- `%A` Hexadecimal floating point, uppercase
- `%n` Nothing printed

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

## 2.2. Length modifier

The length modifier is perhaps oddly-named; it does not modify the length of the output. Instead, it's what you use to specify the length of the input. Say you have:

```
long double d = 3.1415926535;
printf ( "%g", d );
```

Here, d is the input to printf; and what you're saying is that you want to print d as a double; but d is not a double, it is a long double. A long double is likely to be 16 bytes (compared to 8 for a double), so the difference matters. Try running that small snippet and you'll find that you get garbage output that looks something like this:

```
4.94066e-324
```

Remember, the bytes that are given to printf are being treated like a double--but they aren't a double, they're a long double. The length is wrong, and the results are ugly!

The length modifier is all about helping printf deal with cases where you're using unusually big (or unusually small) variables. The best way to think about length modifiers is to say: what variable type do I have, and do I need to use a length modifier for it? Here's a table that should help you out:

Variable type	Length Modifier	Example
short int, unsigned short int	h	short int i = 3; printf( "%hd", i );
long int, unsigned long int	l	long int i = 3; printf( "%ld", i );
wide characters or strings	l	printf( "%ls", wide_str );
long double	L	long double d = 3.1415926535; printf( "%Lg", d );

I'd like to make special mention about the wide character handling. If you write

```
wchar_t wide_str = L "Wide String" ;
printf ( "%s", wide_str );
```

without the l, the result will be to print a single W to the screen. The reason is that wide characters are two bytes, and for simple ASCII characters like W, the second byte is 0. Therefore, printf thinks the string is done! You must tell printf to look for multibyte characters by adding the l: %ls.

(If you happen to be using wprintf, on the other hand, you can simply use %s and it will natively treat all strings as wide character strings.)

## 2.3. Precision

The "precision" modifier is written ".number", and has slightly different meanings for the different conversion specifiers (like d or f).

For **floating point numbers** (e.g. %f), it controls the number of digits printed after the decimal point:

```
printf( "%.3f", 1.2 );
will print:
1.200
```

If the number provided has more precision than is given, it will round. For example:

```
printf( "%.3f", 1.2348 );
will display
1.235
```

Interestingly, for g and G, it will control the number of significant figures displayed. This will impact not just the value after the decimal place but the whole number.

```
printf( "%.3f\n%.3g\n%.3f\n%.3g\n", 100.2, 100.2, 3.1415926, 3.1415926 );
100.200 // %.3f, putting 3 decimal places always
100     // %.3g, putting 3 significant figures
```

```
3.142 // %.3f, putting 3 decimal places again
3.14  // %.3g, putting 3 significant figures
```

For **integers**, on the other hand, the precision it controls the minimum number of digits printed:

```
printf( "%.3d", 10 );
// will print number with 3 digits
010
```

There's one **special case for integers**--if you specify '.0', then the number zero will have no output:

```
printf( "%.0d", 0 );
// has no output!!
```

Finally, for **strings**, the precision controls the maximum length of the string displayed:

```
printf( "%.5s\n", "abcdefg" );
// will display
abcde
```

## 2.4. Width

The width field is almost the opposite of the precision field. Precision controls the max number of characters to print, width controls the minimum number, and has the same format as precision, except without a decimal point:

```
printf( "%5s\n", "abc" );
// will print
abc
```

The blank spaces go at the beginning, by default.


You can combine the precision and width, if you like: <width>.<precision>

```
printf( "%8.5f\n", 1.234 );
// will print
1.23400
```

## 2.5. Flags

The flag setting controls 'characters' that are added to a string, such whether to append 0x to a hexadecimal number, or whether to pad numbers with 0s.


**The Minus Sign Flag: -**

-  Left-justify within the given field width; Right justification is the default

Finally, the minus sign will cause the output to be left-justified. This is important if you are using the width specifier and you want the padding to appear at the end of the output instead of the beginning:

```
printf( "|%-5d|%-5d|\n", 1, 2 );
// will print
|1    |2    |
```

**The Zero Flag: 0**

-  Left-pads the number with zeroes (0) instead of spaces when padding is specified

Using 0 will force the number to be padded with 0s. This only really matters if you use the width setting to ask for a minimal width for your number. For example, if you write:

```
printf( "%05d\n", 10 );
//
00010
```

### The Pound Sign: #

- **#** (bonus) 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.

Adding a # will cause a '0' to be prepended to an octal number (when using the o conversion specifier), or a 0x to be prepended to a hexadecimal number (when using a x conversion specifier). For most other conversion specifiers, adding a # will simply force the inclusion of a decimal point, even if the number has no fractional part.

```
printf( "%#x", 12 );  
//will print  
0xc  
printf( "%x", 12 );  
//will print  
c
```

### Combining all together

For any given format specifier, you can provide must always provide the percent sign and the base specifier. You can then include any, or all, of the flags, width and precision and length that you want. You can even include multiple flags together. Here's a particularly complex example demonstrating multiple flags that would be useful for printing memory addresses as hexadecimal values.

```
printf ( "%#010x\n", 12 );
```

The easiest way to read this is to first notice the % sign and then read right-to-left--the x indicates that we are printing a hexadecimal value; the 10 indicates we want 10 total characters width; the next 0 is a flag indicating we want to pad with 0s instead of spaces, and finally the # sign indicates we want a leading 0x. Since we start with 0x, this means we'll have 8 digits--exactly the right amount for printing out a 32 bit memory address.

The final result is:

```
0x0000000c
```

## 4. Return value

Returns the count of printed characters when the function is successful and -1 when the function fails.

### NOT NEEDED

**Buffer management:** stdio functions store the written data in a buffer before really writing it, to avoid calling too much write() syscall which is pretty heavy in terms of execution time. This buffer is emptied when:

- The buffer size limit is reached (typically 1024 bytes);
- The filestream is attached to a terminal (stdout, stderr) and a newline is reached (so printing to the screen line by line);
- The user calls the function fflush() that force-writes the buffer to the file;
- The filestream is closed (either at the end of the program or by calling fclose()).

Because reproducing this behaviour requires global variables (to retain the fd, the buffer, the fd status [closed/on terminal/etc]), it is not required to do this for printf. If you're concerned about performance you can still store all your output in a buffer and print it all at once just before exiting your function (so storing all format conversions in a byte array and doing only 1 write when all the string is parsed).

### How to structure your code?

When printf processes its arguments, it starts printing the characters it finds in the first argument, one by one. When it finds a percent it knows it has a format specification. It goes to the next argument and uses its value, printing it according to that format specification. It then returns to printing a character at a time (from the first argument).

My thoughts in the beginning of this project: It's maybe easy to understand how printf works but it's really hard to start writing your own version because it needs some serious consideration on how to structure your code and it takes some courage to decide where to begin!

## my structure as an example:

PARSE > save all the info about format specifiers (parse flag, get width&precision, parse type specifier)

CONVERT > get the argument based on its type specifier (handle string or char etc.)

FORMAT > format the argument based on the width&precision and flags (alignment etc.)

PRINT > once the argument is formatted print it with write() call

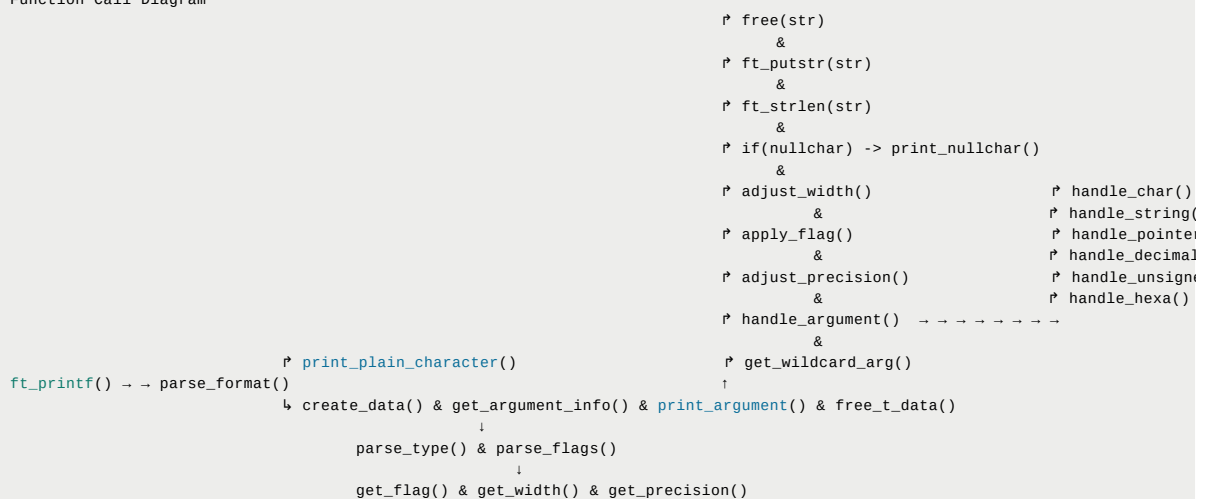
RETURN > at the very end get the length of formatted argument and add it to the total of format length return value

```
HOW?
options..

1) write if it's not %...
   parse if it's %..

2) save everything in format somewhere and parse as you go
   write at the end
```

### Function Call Diagram



## Questions to take into account and some bugs I had to solve while writing the project

What kind of struct do I need to keep format data?

Do I need a linked list or creating one struct each time and delete it once it's written?

- my xtoa does not work with negative integers? > negative ints for hexadecimal undefined behaviour
- I had a bug in my `adjust_int_precision` because it was returning `to_print` as if it was `tmp` then make it equal to `tmp` in other function. Then, free `to_print` and return `tmp` but this did not work because it was the same pointer and it freed `to_print` and `tmp` at the same time!
- CHECK INPUT VALIDATION FOR EACH FUNCTION
- `int = 0 & mfw` > handle exceptions! = `adjust_width || apply_flag` funcs > actually there was a problem with `adjust_precision` > when no `'.'` > `data->precision` was `""` but `adjust_precision` did execute when it was `""` > SO = I added another `*(data->precision) == '\0';` > right now it works!
- negative numbers for adjust width (when there is 0 padds) = DONE
- max unsigned int was printed as -1 > I used `ft_itoa` in `handle_unsigned` > I wrote my own `ft_uitoa` and it's fixed!
- create a `ft_strjoin_free` to cut down on lines for extra frees after `strjoin`!
- precision with `char` == precision used with `'c'` conversion specifier, resulting in undefined behavior, mine prints the char which is undefined in itself as well!
- ▼ negative width & precision? + return value when negative precision?

a negative field width is treated as a left adjustment (test : `&- -5d`) flag followed by a positive field width;

a negative precision is treated as though it were missing.

If a single format directive mixes positional (nn\$) and non-positional arguments, the results are undefined.

- nullterm char with different widths > return value = wrote an exception func
- makefile bonus
- for any undefined behaviour > you can handle that any way you want (just dont segfault and dont leak)
- protecting 'write()' calls and return (-1) : Well, technically yes. What sometimes is done is that write() calls to FD 1 (stdout) are done blindly, and writing to specific FD's is protected. If you are writing a shell for example, not protecting your write() could give unwanted behaviour, but if you are using ft\_printf in a program and you use that output in another program, it's also a big problem if something's not right. I hate to do it too, but the "better" thing to do is protect your write() calls indeed. Even though, I am not sure if I would fail anyone on it. I am willing to bet that 90% of the C programs in the world don't have protection for this. If you are writing software which runs on a spaceshuttle, yes, you should protect it, in "les lethal" usecases, I usually decide on the spot and come up with valid arguments on why / why not to do it.
- how to differentiate between valid input of a string pointing to NULL and failed memory allocation? i protect everything w/ if (!variable) return (whatever needs to be returned when something goes wrong) > For instance in `ft_printf("%s", NULL);` your variadic arguments points to NULL, and the behavior of BSD printf (which is used by the Codam iMacs) would be to print "(null)" in that case (IIRC). If you fail to allocate memory somewhere, terminating your `ft_printf` early and returning an error would be the correct approach. I would suggest checking the value of your variadic argument before calling something like `ft_strdup` to differentiate between the cases you described + i add it in the very beginning that if the string is NULL, then i write (null) as the original function. then moving forward you are sure the string is not NULL and if you get NULL it must be failed memory allocation
- error check
- norm check

## Makefile

Since libft functions are allowed in this project, we have to add that to our directory and find a way to create libft library as well as libftprintf. So, how to compile my libftprintf.a with the libft.a its relying on library?

`cp libft.a libftprintf.a` and then use `ar -rc` to archive your other .o files into the libftprintf.a

## Number bases

For this assignment, you will need a good understanding of number bases. You will be writing code to express integers in a variety of number bases (e.g., binary, hexadecimal, etc.). However, most of the algorithms you see online will give you the digits from right-to-left. You will need to print them left-to-right.

> In mathematics, a "base" or a "radix" is the number of different digits or combination of digits and letters that a system of counting uses to represent numbers.

For example,

- Base 10 (*Decimal*) — Represent any number using 10 digits [0–9]
- Base 2 (*Binary*) — Represent any number using 2 digits [0–1]
- Base 8 (*Octal*) — Represent any number using 8 digits [0–7]
- Base 16 (*Hexadecimal*) — Represent any number using 10 digits and 6 characters [0–9, A, B, C, D, E, F]

An `integer` constant is a decimal (base 10), octal (base 8), or hexadecimal (base 16) number that represents an integral value.

Basic format specifiers for our project:

- `%d` or `i` signed decimal integers.
- `%u` unsigned decimal integers
- `%x` unsigned hexadecimal integer in lowercase letters
- `%X` unsigned hexadecimal integer in uppercase letters

## Integer Types

Every integer constant is given a type based on its value and the way it's expressed. You can force any integer constant to type long by appending the letter l or L to the end of the constant; you can force it to be type unsigned by appending u or U to the value. The lowercase letter l can be confused with the digit 1 and should be avoided.

The type you assign to a constant depends on the value the constant represents. A constant's value must be in the range of representable values for its type. A constant's type determines which conversions are performed when the constant is used in an expression or when the minus sign (-) is applied. This list summarizes the conversion rules for integer constants.

- The type for a decimal constant without a suffix is either `int`, `long int`, or `unsigned long int`. The first of these three types in which the constant's value can be represented is the type assigned to the constant.
- The type assigned to octal and hexadecimal constants without suffixes is `int`, `unsigned int`, `long int`, or `unsigned long int` depending on the size of the constant.
- The type assigned to constants with a `u` or `U` suffix is `unsigned int` or `unsigned long int` depending on their size.
- The type assigned to constants with an `l` or `L` suffix is `long int` or `unsigned long int` depending on their size.
- The type assigned to constants with a `u` or `U` and an `l` or `L` suffix is `unsigned long int`.

### Different numeric type examples:

```
/* Decimal Constants */
int      dec_int    = 28;
unsigned dec_uint   = 4000000024u;
long     dec_long   = 2000000022l;
unsigned long dec_ulong = 4000000000ul;
long long dec_llong  = 9000000000LL;
unsigned long long dec_ullong = 90000000001ull;

/* Octal Constants */
int      oct_int    = 034;
unsigned oct_uint   = 04000000024u;
long     oct_long   = 02000000022l;
unsigned long oct_ulong = 04000000000ul;
long long oct_llong  = 044000000000011;
unsigned long long oct_ullong = 0444000000000001ull;

/* Hexadecimal Constants */
int      hex_int    = 0x1c;
unsigned hex_uint   = 0xA00000024u;
long     hex_long   = 0x20000022l;
unsigned long hex_ulong = 0xA0000021uL;
long long hex_llong  = 0x8a000000000011;
unsigned long long hex_ullong = 0x8A4000000000010uLL;
```

### References

<https://medium.com/@SergioPietri/what-are-variadic-functions-in-c-6205e26c729f>

<https://www.thegeekstuff.com/2017/05/c-variadic-functions/>

<http://www.cplusplus.com/reference/cstdio/printf/>

<https://www.cprogramming.com/tutorial/printf-format-strings.html>