char * get_specifier(char *spe

1 = flg
0 = width
0 = precision
0 = modifier
1 = specifier

%hhd

%#hd

%d

%10-h.d

%#d

```
char *get_specifier(char *spc, fmt_opts_t * fop)
{
    check_flag(&spc, fop)
    check_width( &spc, fop)
    check_precision( &spc, fop)
    check_modifier(&spc, fop)
    check_specifier(&spc)

    return (spc)
}
```

```
void check_flag(char **s, fmt_opts_t *f)
{
    int is_flg = 1;

    while (**s != '\0')
    { switch (**s)
      {case '-'
          f->minus_flg = 1;
        break
        case '+'
          f->plus_flg = 1;
        break
        case '0'
          f->zero_flg = 1
        break;
        case '#':
          f->hash_flg = 1
        case ' ':
          f->blank_flg = 1
        break
        default:
          is_flg = 0
      }
      if (is_flag)
        ++(*s);
    }
}
```

```
void check_width (char **s, fmt_opts_t *f)
{
    f->width = 0;
    while (**s != '\0')
    {
        if (isdigit(**s))
        {
            f->width = (f->width * 10) + (**s - '0')
            ++(*s);
            continue;
        }
        break;
    }
}
void check_precision (char **s, fmt_opts_t *f)
{
    if (**s != '.')
        return;
    ++(*s);
    f->precision = 0;
    while (**s != '\0')
    {
        if (isdigit(**s))
        {
            f->precision = (f->precision * 10) + (**s - '0')
            ++(*s);
            continue;
        }
        break;
    }
}
```

```
void check_modifier(char **s, fmt_opts_t *f)
{
    int is_modifier = 1;

    while (**s != '\0')
    {
        switch (**s)
        {
            case 'h':
                if (*(*s+1) == 'h')
                    f->modifier = strdup("hh");
                else
                    f->modifier = strdup("h");
                break;
            case 'l':
                if (*(*s+1) == 'l')
                    f->modifier = strdup("ll");
                else
                    f->modifier = strdup("l");
                break;
                :
            default:
                is_modifier = 0;
        }
        if (is_modifier)
        {
            ++(*s);
            break;
        }
}
```

++(*s)

++(*s)

```
void check_specifier(char **s)
{
    char specifier[] = "cs%";

    if (**s == '\0' || strchr(specifier, **s) == NULL)
        *s = NULL;
}
```

Number Systems

d → decimal

b → binary ⟶ sizeof(int) / 1 = 32 / 1 = 32

o → octal ⟶ 32 / 3 = 11

x → hex ⟶ 32 / 4 = 8

X → HEX

Converting an unsigned → binary

uint → 4 bytes = 32 bits

so max_space for binary_buf = 32 + 1

e.g 7284

7284 % 2 = 0

3642 % 2 = 0

1821 % 2 = 1

910 % 2 = 0

455 % 2 = 1

227 % 2 = 1

113 % 2 = 1

56 % 2 = 0

28 % 2 = 0

14 % 2 = 0

7 % 2 = 1

3 % 2 = 1

1 % 2 = 1

0

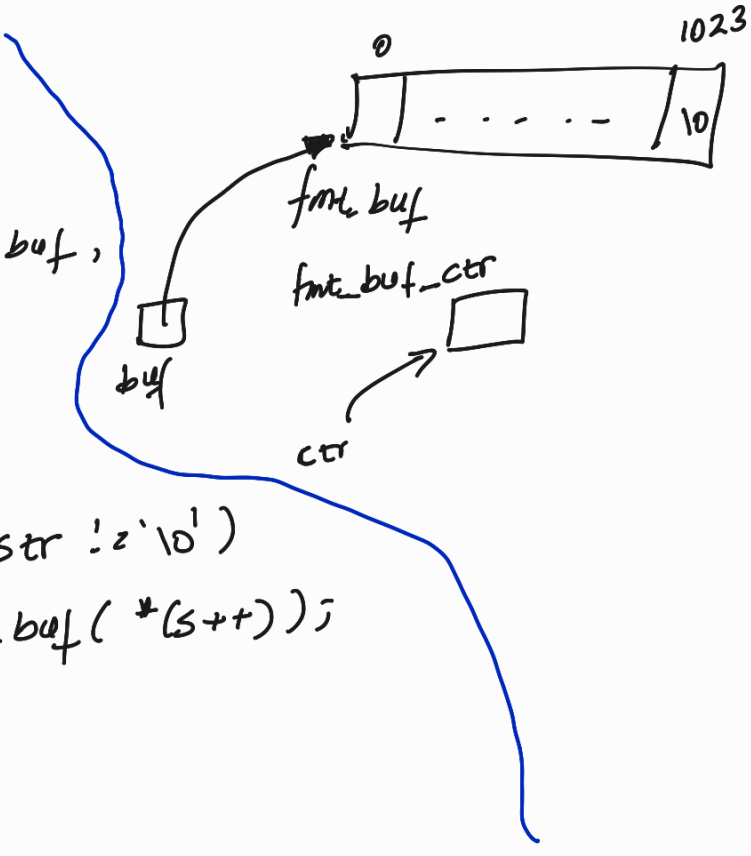
1 1 1 0 0 0 1 1 1 0 1 0 0

```
int _putchar_buf(int ch, char *buf, int *ctr)
{
    int printed_chars = 0;

    if ( ch == -1 || ctr == BUFF_SIZE - 1 )
    {
        printed_chars += write(1, *buf, *ctr );

        *ctr = 0;
    }
    if ( ch != -1 )
        (*buf)[ctr++] = ch;

    return (printed_chars);
}
```



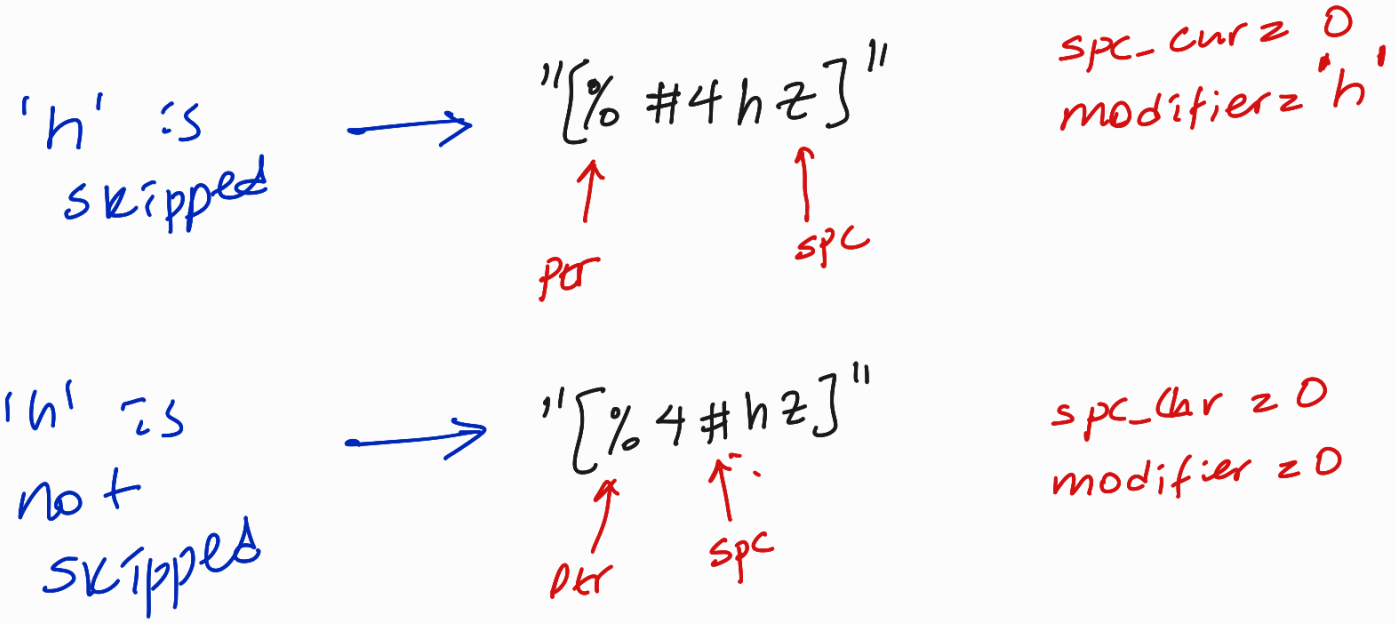

```
int _puts_buf(char *str, char *buf,
              int *ctr)
{
    int printed_chars = 0;

    while ( str != NULL && *str != '\0' )
        printed_chars += _putchar_buf( *(s++) );

    return (printed_chars);
}
```

printing invalid syntax

int _print_invalid_syntax(char *p, va*

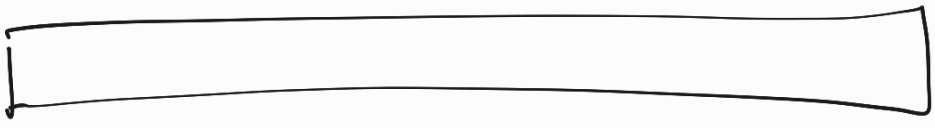


Field Width

width >= strlen ✓ → format

width < strlen ✗

formatting



memset(padding)

copy str to form from bace to fromt

or fram front to bacie

TODO

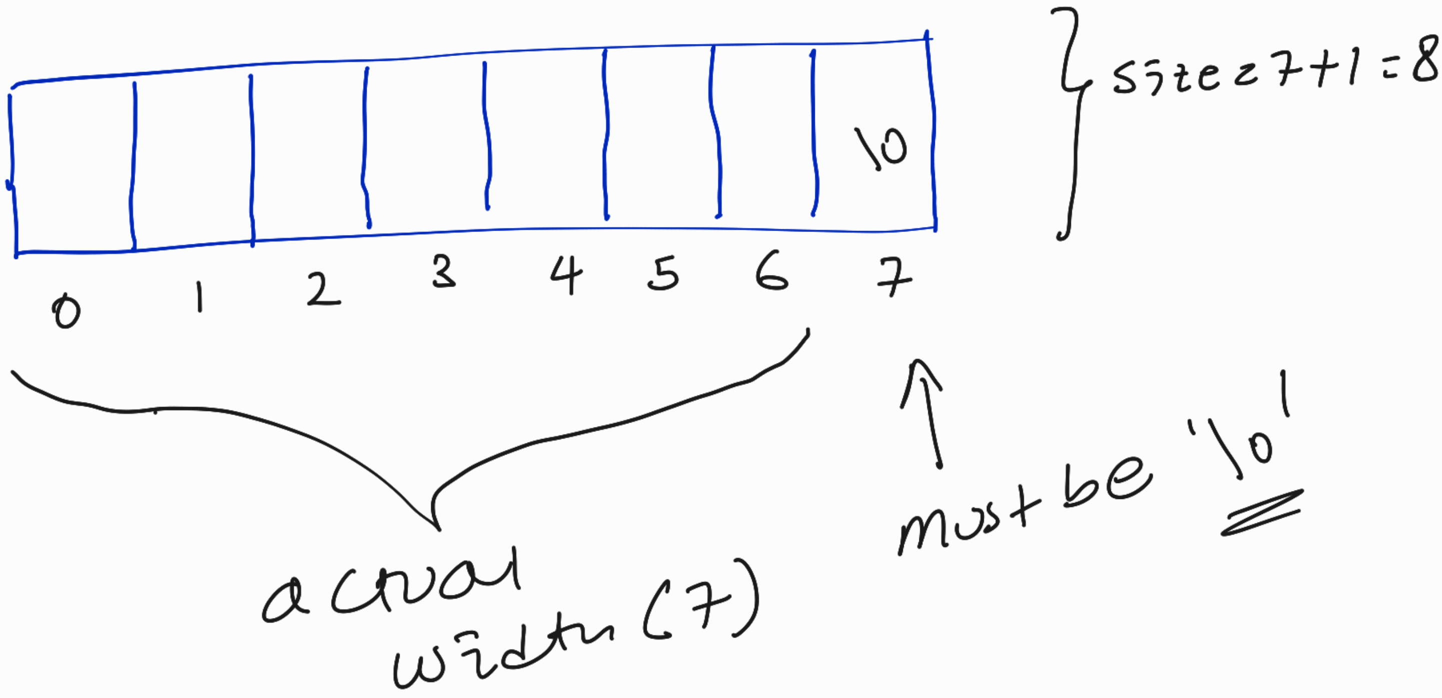
move args → format_data

- [x] percent
- [x] char
- [x] strings
- [x] integer
- [x] unsigned
- [x] octal
- [x] hex
- [x] pointer

standard

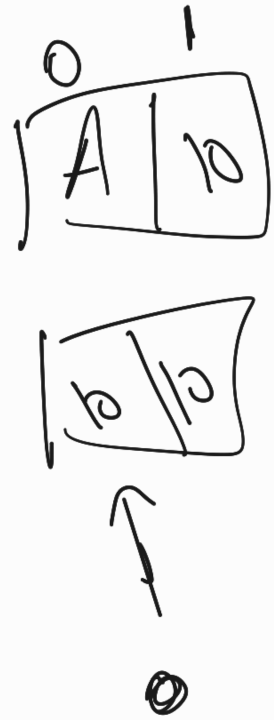
- [x] binary
- [x] custom_string

custom

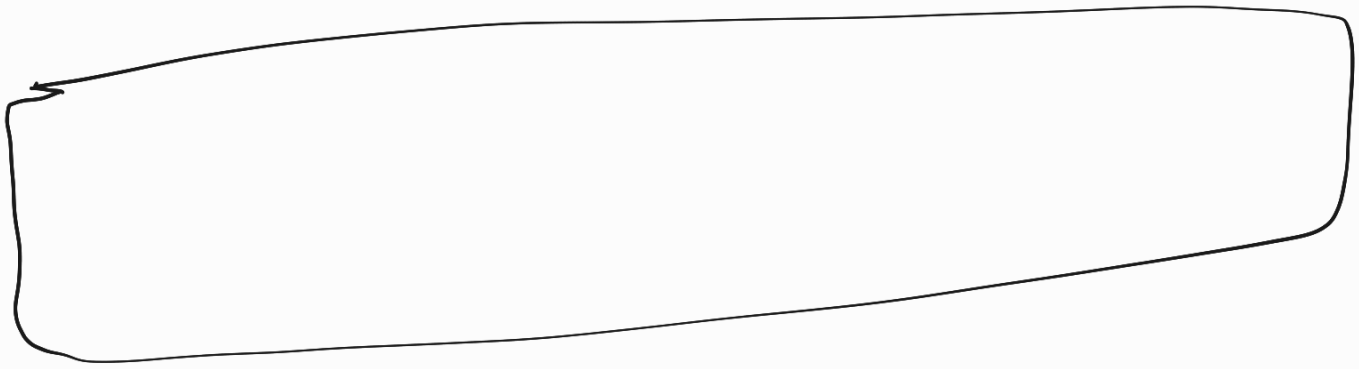


A \0

\0



Heno.



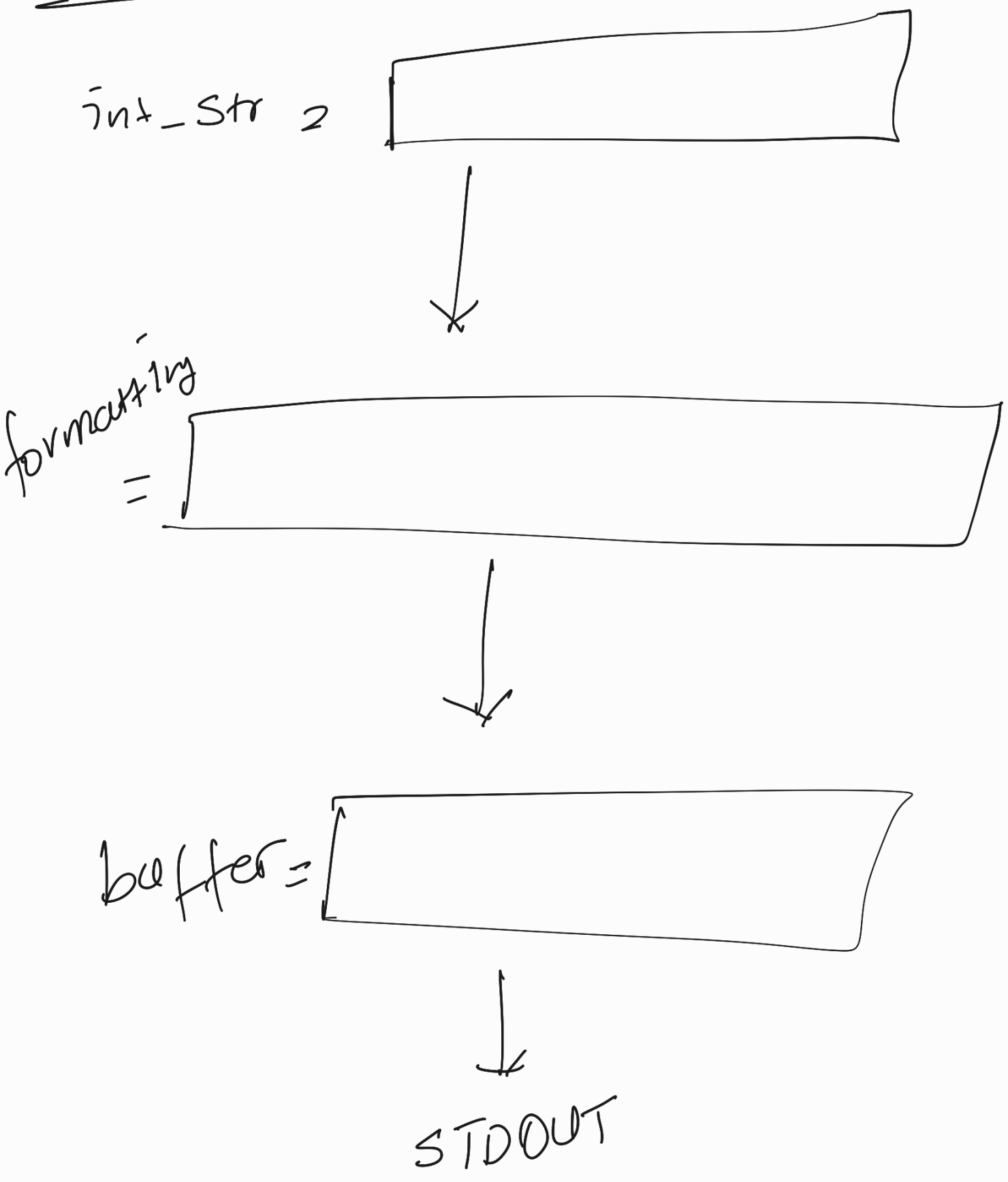
AIX = 3 per 2

trun = 1

if(str != NULL && f->prec < strlen(st)
str[f->prec] = '\0';

This is how formatting works

Integer

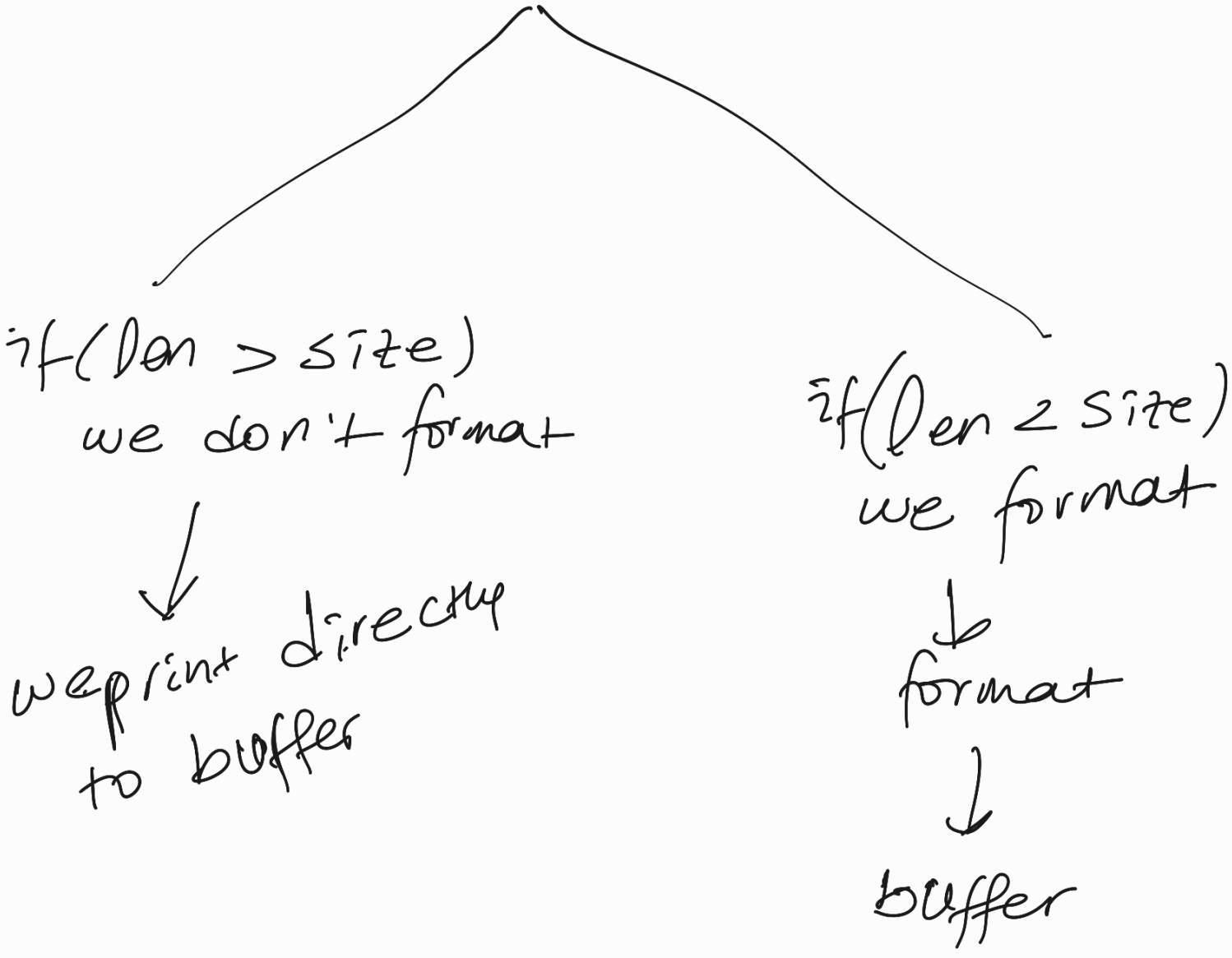


First determine format size

size = Max(width, precision)

len = strlen(int_str)

There are cases where we don't format

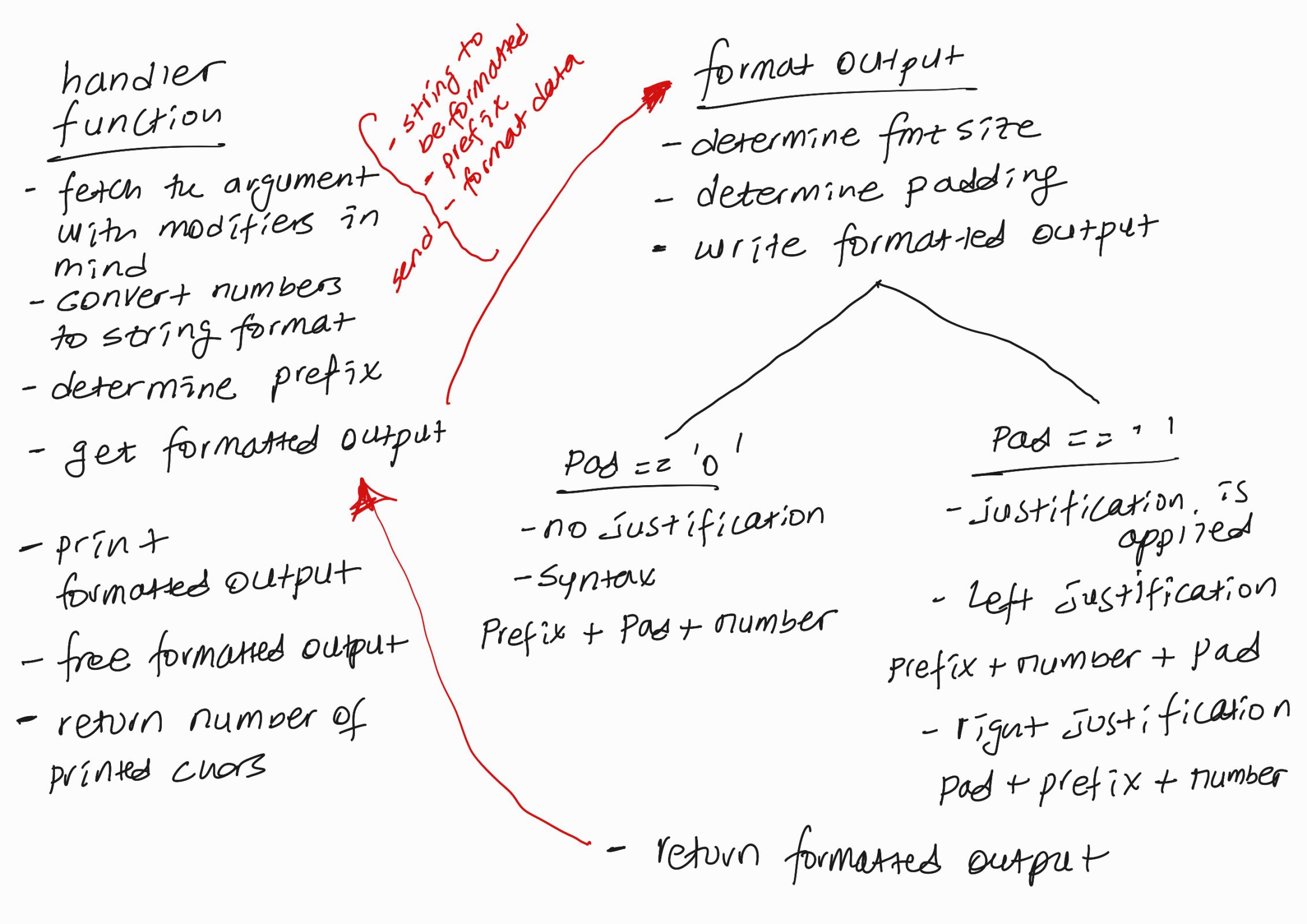


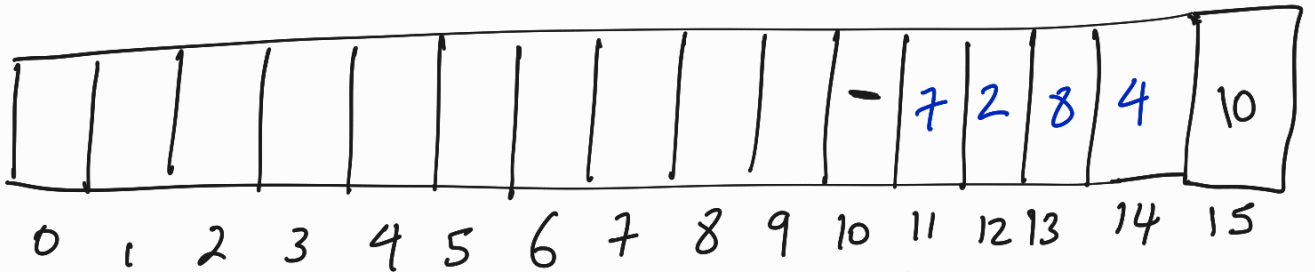
Steps in formatting

1. Determine format size
   - → fmtsize = max { width, precision, length of number + pre
   - → if fmtsize == precision
     fmtsize += strlen (prefix)

2. Determine Padding
   - Pad = ' ' — by default
   - Pad = 0 → if
     - precision is set or
     - zero is set & minus flag is not set

3. write


handler function
- fetch the argument with modifiers in mind
- convert numbers to string format
- determine prefix
- get formatted output
- print formatted output
- free formatted output
- return number of printed chars
- string to be formatted
- prefix
- format data
send
format output
- determine fmt size
- determine padding
- write formatted output
Pad == '0'
- no justification
- Syntax
Prefix + Pad + number
Pad == ' '
- justification is applied
- Left justification
prefix + number + Pad
- right justification
Pad + prefix + number
- return formatted output



→ fmt_output
size = 16

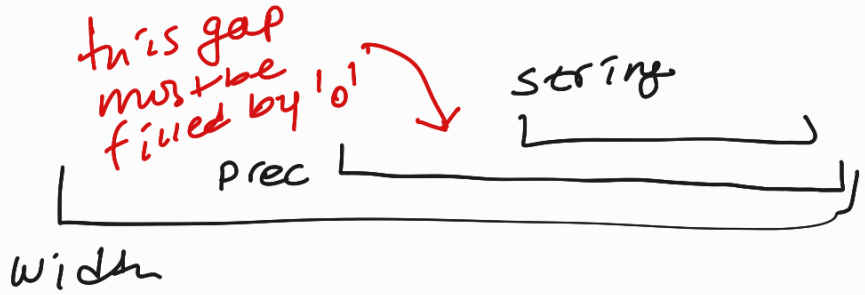
number = 7284
prefix = -

4 ←

e.g.

fout_len = 15 (1) (4) int
int_len = 5 (pre + str)

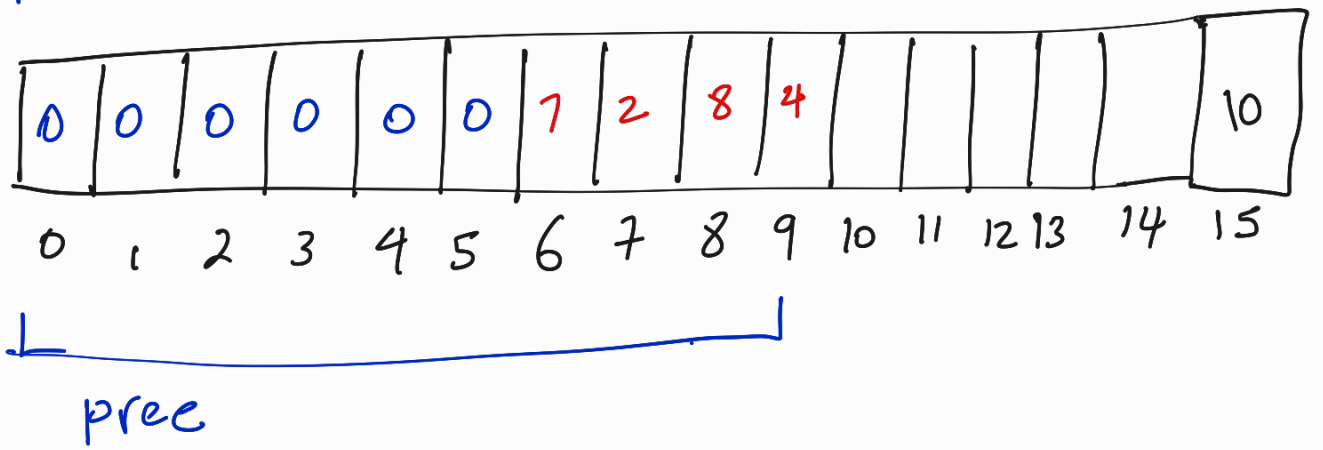
fmt (fout_len - str_len)
(15) (4)



precision < string

we consider precision in determining the format field because if the precision is larger than the integer string then remaining space on the left must be padded with zeros

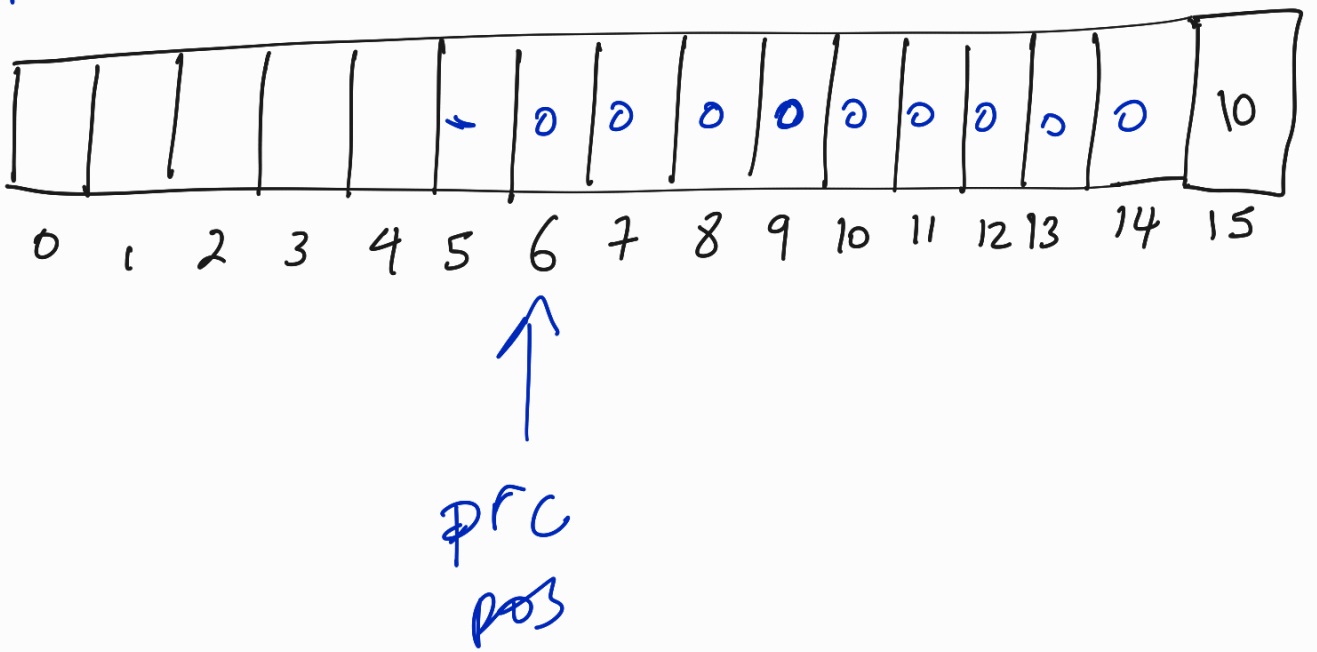
width = 14 num = 7284
precision = 10 jst = 1



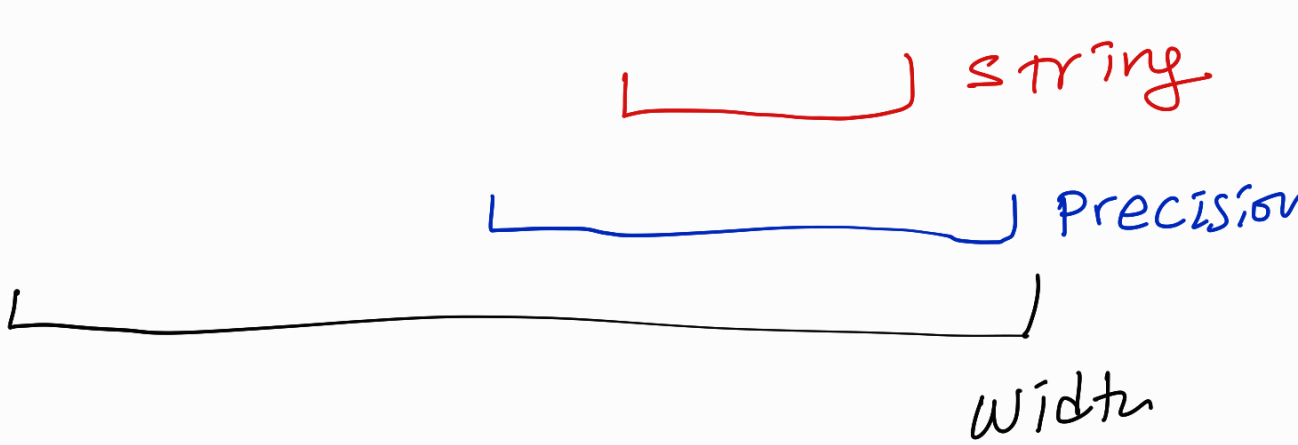
1. Pad the precision area with zeros
   memset (fout, '0', prec)
2. write prefix
   memcp (fout, prefix, pre_len)
3. write int_str
   memcpy (fout +

width = 14
precision = 10

num = -7284
jst = F



PFC
pos



Note on formatting

Sep 24, 2023

- In formatting an output the major steps undertaken are
    1. Determine the acutal format field width
    2. Determine the justification of the output
    3. Determine the padding character

1. Actual format field width

- Now we know that the user can give us a value for the format fied width in the format options.
- But the actual format fied width of the output depends on these three things
    a. the length of the string
    b. the given field width
    c. the given precision.
- And each of these properties have different behaviors based on the conversion specifier.
- So let's first understand these properties based on the conversion specifier handled.

a. the length of the string

- Each conversion specifier value, meaning value given by the optional arguments is converted to a string format in order to be printed.

char
string
int
unsigned int
hexadecimal
octal
pointer

} all is converted to a string format

e.g

1945 (int) ⟶ "1945" (str)

- So length of the string means the length of the converted string.

b. the given field width

- A field width value can be given by the user.
- Now eventhough a user supplied a field width, the actual field width of the formatted output may some other value
- This is because if the length of the string to be printed is larger than the given field width then printf will take the length of the string as the actual field width.

⟶ This means that the field width option never truncates the output (the string to be printed) if it is larger than the given field width.

- So

```
if (str_len > field_width)
    actual_field_width = str_len
else
    actual_field_width = field_width
```

c. the given precision

- Okay now precision doesn't directly affect the actual field width.
- In printf the precision format option defines how many characters of the output need to be printed.
- In the case of:
  i) character and string conversion
     - if precision > str_len, then the output string will be as-is
     - if precision < str_len, then the output string is truncated
       → truncated meaning if the output string length is 10 and if the precision is 7 then only the first 7 characters from the output string are printed.

  ii) Integer conversions
     - if precision > converted str_len, then the extra spaces will be padded with zeros.

       e.g.
       if the converted integer string = "1945" → 4 chars and the precision value is 7, then the extra 3 chars will be zero, padded from the left

       "1945" → "0001945"

       → This applies for all integer conversions.

     - if precision < converted str_len, then the converted integer string is printed as-is, it will not be truncated.

- So we see that the precision value affects the output string length. Because it can truncate it (for character & string conversions) or enlarge it for (for integer conversion).
- And the actual field width depends on the output string length, so we can see that the precision indirectly affects the actual field width

summary

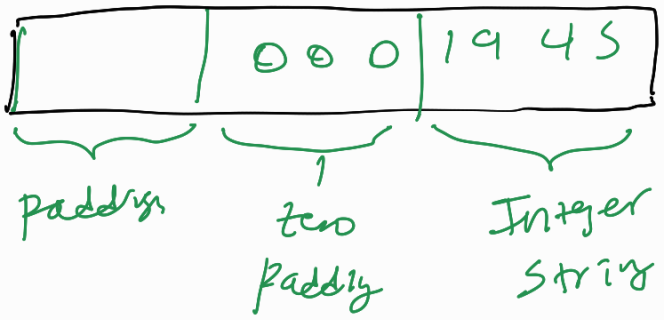
- so in summary the actual field width depends on the given field width and the length of the output string.
- And the output string length depends the preecision and the conversion specifier.
- I hope you get the idea :)

Idea

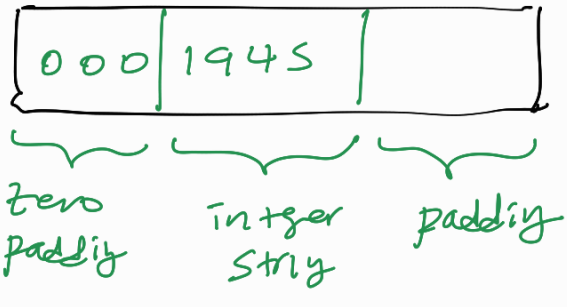
- I can divide the formatted output into different parts and direcly print those parts to the buffer without ever using malloc allocation.

The different parts are
- the padding
- the zero padding -> for integer conversions
- the output string

left justification



right justification



Example calculation

Integer Conversion (for all integer conversion) d, i, O, U, X, P

output_str_len = integer string length > Precision ? integer string len + precision : integer string len ;

actual_field width = output str len > field width ? output str len : field width ;

padding length = actual_field width − output_str_len ;

zero padding length = converted strig_length > precision ? 0 : Precision − Converted string len ;

String Conversion (same for characters)

output_str_len = string length > Precision ? string length : string length − precision;

actual_field width = output str len > field width ? output str len : field width ;

padding length = actual_field width − output_str_len ;

zero padding length = 0