Output formatting

The function "printf()" delivers character stream to the standard output file `stdout`, which is usually connected to the screen. It controls the printing by simple conversion specifications, or formats. The argument list to "printf()" has two parts: a control string and other arguments. For example, consider the statement

```
printf("Sum of %d and %d is %d\n",2,3,5);
```

Here `"Sum of %d and %d is %d\n"` is the control string and `2,3,5` are other arguments. The expressions in other arguments are evaluated and converted according to the formats in the control string, and then placed in the output stream. Characters in the control string that are not part of a format are placed directly in the output stream. A format follows the following prototype

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

Here specifier is the most significant one that defines the type and the interpretation of the value of the corresponding argument. Note that a format must contain `%` and `specifier`.

specifier	How the corresponding argument is printed
c	as a character
d or i	as a decimal integer
u	as an unsigned decimal integer
o	as an unsigned octal integer
x	as an unsigned hexadecimal integer
X	as an unsigned hexadecimal integer (capital latters)
e	as a floating-point number in scientific notation using 'e' character
E	as a floating-point number in scientific notation using 'E' character
f	as a decimal floating-point number
g	use the shorter of %e or %f
G	use the shorter of %E or %f
s	as a string
p	the corresponding argument is a pointer to `void`, its value is printed as a hexadecimal number.
n	Nothing printed. The argument must be a pointer to an `int`, where the number of characters written so far is stored. The argument is not converted.
%	with the format %%, a single % is printed; there is no corresponding argument to be converted.

Some of the specifiers have already been encountered before. Few will be discussed later. Note that, similar to decimal and binary systems, we also have an octal and hexadecimal system. In an octal system, the base is 8 and a number is represented using $0, 1, 2, \cdots, 7$. Similarly, in a hexadecimal system, the base is 16 and a number is represented using $0, 1, 2, \cdots, 9, A, B, C, D, E, F$. For example, see the following code segment:

```
printf("%x\n",123); // 7b is printed
printf("%o\n",123);  // 173 is printed
```

width	How the corresponding argument is printed
(number)	Defines the minimum number of characters to print. If the data has fewer characters, spaces (or zeros, if the '0' flag is used) are added to fill the gap. If the data is longer than the specified width, the field expands to accommodate the entire value, as values are never truncated.
*	The * symbol acts as a placeholder for an int argument that determines the minimum field width at runtime. The width value must be provided as an additional argument before the actual value to be printed.

flag	How the corresponding argument is printed
-	Left-justify within the given field width.
+	It forces the + sign to be printed for nonnegative numbers
(space)	If no sign is going to be written, a blank space is inserted before the value
#	• Used with o, x, and X specifiers, the value is preceded with 0, 0x, and 0X, respectively, for values different than zero. • Used with e, E, and f, it forces the output to contain a decimal point even if no digits would follow (by default, if no digits follow, no decimal point is written). • Used with g or G, it causes the trailing zeros to be printed.
0	Left-pads the number with zeros (0) instead of spaces where padding is specified.

.precision	How the corresponding argument is printed
.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. • For e, E, and f specifiers: this is the number of digits to be printed after the decimal point. • 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 `'\0'` is encountered. • For c type: it has no effect
*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

length	How the corresponding argument is printed
h	The argument is interpreted as a `short int` or `unsigned short int` (only applies to integer specifiers: i, d, o, u, x and X)
l	The argument is interpreted as a `long int` or `unsigned long int` for integer specifiers (i, d, o, u, x and X), and as a wide character or wide character string for specifiers c and s.
L	The argument is interpreted as a `long double` (only applies to floating point specifiers: e, E, f, g and G).

Now, we provide a few examples showing the widely used formats for `int` and `double` variables.

```
#include<stdio.h>
int main()
{ int a=3456,b=-345;
printf("Formatting integers\n");
printf("$$$$@$$$$@$$$$@$$$$@$$$$@\n");// each @ is at 5th place
printf("%d\n",a); //need width (w) 4, uses width (w) 4
printf("%10d\n",b);//w 10 reserved, uses 4 right justified
printf("%+010d\n",a);//w 10 reserved, w 5 needed for + and 3456, padded 5 zeros
printf("%-10d\n",a);//w 10 reserved, uses 4 at left, left justified due to -
printf("%-010d\n",a);//w 10 reserved, uses 4 left justified due to -, 0 flag ignored
printf("%10.3d\n",b);//w 10 reserved, print min 3 digits, right justified
printf("%10.4d\n",b);//w 10 reserved, print min 4 digits, right justified
printf("%010.4d\n",b);//0 flag ignored in cases of precision
return 0;
}
```

Listing 13: C program "fmti.c"

Compiling with `$ gcc fmti.c -lm ↩` and running the default executable with `$ ./a.out ↩` produce the following output.

```
Formatting integers
$$$$@$$$$@$$$$@$$$$@$$$$@
3456
      -345
+000003456
3456
3456
      -345
     -0345
     -0345
```

The output is self-explanatory. The second line in the output shows 25-width, where each `@` is at the consecutive 5th place. Thus, the second `@` denotes 10th place in the width. In the 4th "printf()" statement "`printf("%10d\n",b);`," width 10 has been specified. To print `-345`, we need 4 places: one for `-` and three for `345`. By default, a number is printed right-justified. Hence, the digit '5' is printed at 10th place (match with 2nd `@`), '4' is printed at 9th place, etc. Other outputs can be explained using the short comments provided. Here, 'w' in the comment implies width.

```
#include<stdio.h>
int main()
{
 double a=423.76;
 printf("Formating real numbers: f-format \n");
 printf("$$$$@$$$$@$$$$@$$$$@$$$$@\n");
 printf("%f\n",a);//default 423.760000 use 10 w
 printf("%011f\n",a);//w=11 423.760000 use 10 w paadded 1 zero
 printf("%9.3f\n",a);//w 9 423.760 use 7 w right justified
 printf("%+011.3f\n",a);//w=11 +423.760 use 8 w padded 3 zeros
 printf("%+-011.3f\n",a);//use 8 w left justified 0 flag ignored
 printf("%.0f\n",a);//0 precision 424 (rounded) use 3 w
 printf("%#.0f\n",a);//424 (rounded) print 424. . due to #
 printf("Formating real numbers: e-format \n");
 printf("$$$$@$$$$@$$$$@$$$$@$$$$@\n");
 printf("%e\n",a);//default 4.237600e+02 use 12 w
 printf("%010E\n",a);//w=10 4.237600E+02 need 12 w print using 12 w
 printf("%12.3e\n",a);// w=12 4.238e+02 use 9 w right justified
 printf("%+011.3E\n",a);//w=11 +4.238E+02 needs 10 w  padded 1 zero
 printf("%+-011.3E\n",a);// 0 flag ignored due to -
 printf("%.0E\n",a);//0 precision 4E+02 needs 5 w
 printf("%#.0E\n",a);//0 precision 4.E+02 . due to #
 return 0;
}
```

Listing 14: C program "fmtf.c"

Compiling with `$ gcc fmtf.c -lm ↩` and running the default executable with `$ ./a.out ↩` produce the following output.

```
Formatting real numbers: f-format
$$$$@$$$$@$$$$@$$$$@$$$$@
423.760000
```

```
0423.760000
  423.760
+000423.760
+423.760
424
424.
Formatting real numbers: e-format
$$$$@$$$$@$$$$@$$$$@$$$$@
4.237600e+02
4.237600E+02
   4.238e+02
+04.238E+02
+4.238E+02
4E+02
4.E+02
```

Any real number is rounded depending on the number of digits after the decimal place. For example, the real `423.76` in scientific notation is `4.2376e+02`. If we print this in `e` format with precision 3, such as `%.3e`, then we are printing `4.238e+02`.

```
#include <stdio.h>
int main(void)
{
  double a=12345.6789,b=10005.00005;
  printf("Formatting real numbers: g-format \n");
  printf("$$$$@$$$$@$$$$@$$$$@$$$$@\n");
  printf("%g\n",a);//default six significant digits 6 w
  printf("%9g\n",a);// w 9 use 7 w right justfied
  printf("%09g\n",a);// w 9 use 7 w right just. padd 2 zero
  printf("%9.3G\n",a);//w 9 sig. dig. 3 E format 1.23E+02 right just.
  printf("%.5g\n",a);//w 5 sig. dig. 5 f format 123456
  printf("%#.5g\n",a);//. due to #
  printf("%#9.5g\n",a);//w 9 sig. dig. 5 f format 123456. right just.
  printf("%5.4g\n",a);//w 5 sig. dig. 4 g format 1.235+e02. right just.
  printf("%5.2g\n",b);//w 5 sig. digit 2 e format 1.0e+04 last 0 not printed
  printf("%#10.2g\n",b);//w 5 sig. digit 2 printed 0 due to #
  printf("%.0g\n",b);// 0 sig. digit=> 1 sig. digit
  printf("%#.0g\n",b);// . due to #
  return(0);
}
```

Listing 15: C program "fmtg.c"

Compiling with $ gcc fmtg.c -lm ↩ and running the default executable with $ ./a.out ↩ produce the following output.

```
Formatting real numbers: g-format
$$$$@$$$$@$$$$@$$$$@$$$$@
12345.7
  12345.7
0012345.7
 1.23E+04
12346
12346.
   12346.
1.235e+04
1e+04
   1.0e+04
1e+04
1.e+04
```

Any real number is rounded depending on the number of significant digits in the g-format. For example, the real `4236.76` in scientific notation is `4.23676e+03`. If we print this using `%#.3g` format with precision 3, then we are printing `4.24e+03`, which has 3 significant digits (the last digit is rounded). If we print using `%#.4g` format with precision 4, then we are printing `4237.`, which has 4 significant digits (the last digit is rounded).

Return to Lecture Topics