Output Functions

Invoke standard library procedures to stream data to users

"printf ... is usually the easiest and most concise way to perform output."
(GNU C Library Reference Manual, 2013)

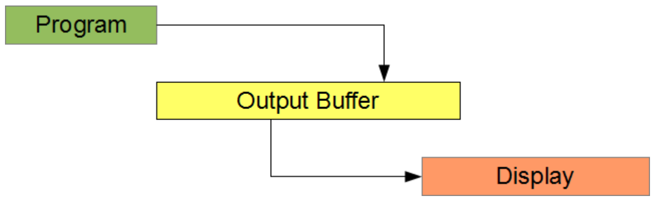
Buffering | Unformatted | Formatted | Exercises

The adequate provision of a user interface is an important aspect of software development: an interface that consists of user friendly input and user friendly output. The output facilities of a programming language convert the data in memory into a stream of characters that is read by the user. The **`stdio`** module of the C language provides such facilities.

This chapter describes two functions in the **`stdio`** module that provide formatted and unformatted buffered support for streaming output data to the user and demonstrates in detail how to format output for a user friendly interface.

BUFFERING

Standard output is line buffered. A program outputs its data to a buffer. That buffer empties to the standard output device separately. When it empties, we say that the buffer *flushes*.



Output buffering lets a program continue executing without having to wait for the output device to finish displaying the characters it has received.

The output buffer flushes if

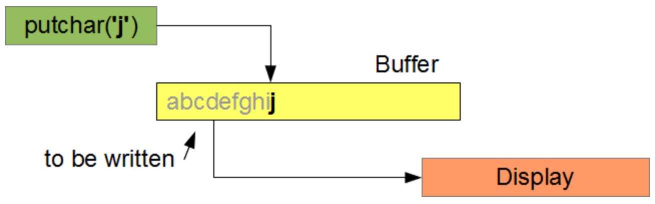
- it is full
- it receives a newline (**`\n`**) character
- the program terminates

Two functions in the **`stdio`** module that send characters to the output buffer are

- **`putchar()`** - unformatted
- **`printf()`** - formatted

UNFORMATTED OUTPUT

The **`putchar()`** function sends a single character to the output buffer. We pass the character as an argument to this function. The function returns the character sent or **`EOF`** if an error occured.



The prototype for `putchar()` is

```
int putchar (int);
```

To send the character `'a'` to the display device, we write

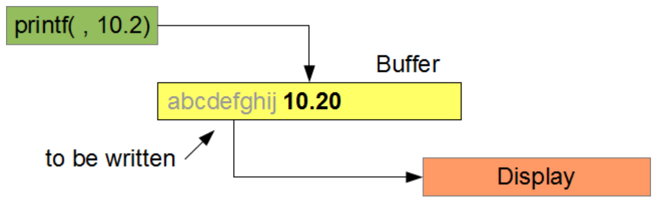
```
// Single character output
// putchar.c

#include <stdio.h>

int main(void)
{
        putchar('a');
        return 0;
}
```

```
a
```

Note that `putchar()` can take `EOF` as an argument.

FORMATTED OUTPUT

The `printf()` function sends data to the output buffer under format control and returns the number of characters sent.



The prototype for the `printf()` function is

```
int printf(format, argument, ... );
```

format is a set of characters enclosed in double-quotes that may consist of any combination of plain characters and conversion specifiers. The function sends the plain characters as is to the buffer and uses the conversion specifiers to translate each value passed as an argument in the function call. The ellipsis indicates that the number of arguments can vary. Each conversion specifier corresponds to one argument.

Conversion Specifiers

A conversion specifier begins with a `%` symbol and ends with a *conversion character*. The conversion character defines the formatting as listed in the table below

Specifier	Format As	Use With Type ...	Common
`%c`	character	`char`	*
`%d`	decimal	`char, int, short, long, long long`	*
`%o`	octal	`char, int, short, long, long long`	
`%x`	hexadecimal	`char, int, short, long, long long`	
`%f`	floating-point	`float, double, long double`	*
`%g`	general	`float, double, long double`	
`%e`	exponential	`float, double, long double`	

For example, the following code snippet yields the output on the right

```
int i = 15;
float x = 3.141593f;
printf("i is %d; x is %f\n", i, x);
```

```
i is 15; x is 3.141593
```

Conversion Controls

We refine the output by inserting control characters between the `%` symbol and the conversion character. The general form of a conversion specification is

```
% flags width . precision size conversion_character
```

The five control characters are

- ***flags***
  - `-` prescribes left justification of the converted value in its field
  - `0` pads the field width with leading zeros
- ***width*** sets the minimum field width within which to format the value (overriding with a wider field only if necessary). Pads the converted value on the left (or right, for left alignment). The padding character is space or `0` if the padding flag is on
- `.` separates the field's width from the field's precision
- ***precision*** sets the number of digits to be printed after the decimal point for `f` conversions and the minimum number of digits to be printed for an integer (adding leading zeros if necessary). A value of `0` suppresses the printing of the decimal point in an `f` conversion
- ***size*** identifies the size of the type being output

Integral values:

Size Specifier	Use With Type
none	`int`
`hh`	`char`
`h`	`short`
`l`	`long`
`ll`	`long long`

Floating-point values:

Size Specifier	Use With Type
none	`float`
`l`	`double`
`L`	`long double`

Special Characters

To insert the special characters `\`, `'`, and `"`, we use their escape sequences. To insert the special character `%` into the format, we use the `%` symbol:

```
// Outputting special characters
// special.c

int main(void)
{
        printf("\\ \' \" %%\n");
        return 0;
}
```

```
\ ' " %
```

Reference Example

The following program produces the output listed on the right for the ASCII collating sequence

```
// Playing with output formatting
//  printf.c
#include <stdio.h>

int main(void)
```

```
{
    /* integers */
    printf("\n* ints *\n");
    printf("00000000011\n");
    printf("12345678901\n");
    printf("------------------------\n");
    printf("%d|<--        %%d\n",4321);
    printf("%10d|<--  %%10d\n",4321);
    printf("%010d|<--  %%010d\n",4321);
    printf("%-10d|<--  %%-10d\n",4321);
    /* floats */
    printf("\n* floats *\n");
    printf("00000000011\n");
    printf("12345678901\n");
    printf("------------------------\n");
    printf("%f|<-- %%f\n",4321.9876546);
    /* doubles */
    printf("\n* doubles *\n");
    printf("00000000011\n");
    printf("12345678901\n");
    printf("------------------------\n");
    printf("%lf|<-- %%lf\n",4321.9876546);
    printf("%10.3lf|<--  %%10.3lf\n",4321.9876);
    printf("%010.3lf|<--  %%010.3lf\n",4321.9876);
    printf("%-10.3lf|<--  %%-10.3lf\n",4321.9876);
    /* characters */
    printf("\n* chars *\n");
    printf("00000000011\n");
    printf("12345678901\n");
    printf("------------------------\n");
    printf("%c|<--           %%c\n",'d');
    printf("%d|<--         %%d\n",'d');
    printf("%x|<--          %%x\n",'d');
    return 0;
}
```

```

* ints *
00000000011
12345678901
------------------------
4321|<--        %d
      4321|<--  %10d
0000004321|<--  %010d
4321      |<--  %-10d

* floats *
00000000011
12345678901
------------------------
4321.987655|<-- %f

* doubles *
00000000011
12345678901
------------------------
4321.987655|<-- %lf
  4321.988|<--  %10.3lf
004321.988|<--  %010.3lf
4321.988  |<--  %-10.3lf

* chars *
00000000011
12345678901
------------------------
d|<--           %c
100|<--         %d
64|<--          %x
```

Note that

- `double`s and `float`s round to the requested precision before being displayed;
- `double` data may be displayed using `%f` (`printf()` converts `float` values to `double`s for compatibility with legacy programs);
- character data displays in various formats including character, decimal and hexadecimal.

Portability Note (Optional)

Character data is encoded on many computers using the ASCII standard, but not all computers use this sequence. A program is portable across sequences if it refers to character data in its symbolic form (`'A'`) and to special characters - such as newline, tab, and formfeed - by their escape sequences (`'\n'`, `\t`, `\f`, etc.) rather than by their decimal or hexadecimal values.

EXERCISES

- Experiment with the `printf.c` program by changing flags, widths and precisions in the conversion specifiers