Chapter 21: Formatted Input/Output

Section 21.1: Conversion Specifiers for printing

Conversion Specifier	Type of Argument	Description
`i`, `d`	int	prints decimal
`u`	unsigned int	prints decimal
`o`	unsigned int	prints octal
`x`	unsigned int	prints hexadecimal, lower-case
`X`	unsigned int	prints hexadecimal, upper-case
`f`	double	prints float with a default precision of 6, if no precision is given (lower-case used for special numbers `nan` and `inf` or `infinity`)
`F`	double	prints float with a default precision of 6, if no precision is given (upper-case used for special numbers `NAN` and `INF` or `INFINITY`)
`e`	double	prints float with a default precision of 6, if no precision is given, using scientific notation using mantissa/exponent; lower-case exponent and special numbers
`E`	double	prints float with a default precision of 6, if no precision is given, using scientific notation using mantissa/exponent; upper-case exponent and special numbers
`g`	double	uses either `f` or `e` *[see below]*
`G`	double	uses either `F` or `E` *[see below]*
`a`	double	prints hexadecimal, lower-case
`A`	double	prints hexadecimal, upper-case
`c`	char	prints single character
`s`	char*	prints string of characters up to a `NUL` terminator, or truncated to length given by precision, if specified
`p`	void*	prints `void`-pointer value; a non`void`-pointer should be explicitly converted ("cast") to `void*`; pointer to object only, **not** a function-pointer
`%`	n/a	prints the `%` character
`n`	int *	***write*** the number of bytes printed so far into the `int` pointed at.

Note that length modifiers can be applied to `%n` (e.g. `%hhn` indicates that *a following `n` conversion specifier applies to a pointer to a `signed char` argument*, according to the ISO/IEC 9899:2011 §7.21.6.1 ¶7).

Note that the floating point conversions apply to types `float` and `double` because of default promotion rules — §6.5.2.2 Function calls, ¶7 *The ellipsis notation in a function prototype declarator causes argument type conversion to stop after the last declared parameter. The default argument promotions are performed on trailing arguments.*) Thus, functions such as `printf()` are only ever passed `double` values, even if the variable referenced is of type `float`.

With the `g` and `G` formats, the choice between `e` and `f` (or `E` and `F`) notation is documented in the C standard and in the POSIX specification for `printf()`:

> The double argument representing a floating-point number shall be converted in the style `f` or `e` (or in the style `F` or `E` in the case of a `G` conversion specifier), depending on the value converted and the precision. Let `P` equal the precision if non-zero, 6 if the precision is omitted, or 1 if the precision is zero. Then, if a conversion with style `E` would have an exponent of `X`:

> - If P > X >= -4, the conversion shall be with style `f` (or `F`) and precision `P - (X+1)`.
> - Otherwise, the conversion shall be with style `e` (or `E`) and precision `P - 1`.

> Finally, unless the '#' flag is used, any trailing zeros shall be removed from the fractional portion of the result and the decimal-point character shall be removed if there is no fractional portion remaining.

Section 21.2: The printf() Function

Accessed through including **<stdio.h>**, the function `printf()` is the primary tool used for printing text to the console in C.

```
printf("Hello world!");
// Hello world!
```

Normal, unformatted character arrays can be printed by themselves by placing them directly in between the parentheses.

```
printf("%d is the answer to life, the universe, and everything.", 42);
// 42 is the answer to life, the universe, and everything.

int x = 3;
char y = 'Z';
char* z = "Example";
printf("Int: %d, Char: %c, String: %s", x, y, z);
// Int: 3, Char: Z, String: Example
```

Alternatively, integers, floating-point numbers, characters, and more can be printed using the escape character `%`, followed by a character or sequence of characters denoting the format, known as the *format specifier*.

All additional arguments to the function `printf()` are separated by commas, and these arguments should be in the same order as the format specifiers. Additional arguments are ignored, while incorrectly typed arguments or a lack of arguments will cause errors or undefined behavior. Each argument can be either a literal value or a variable.

After successful execution, the number of characters printed is returned with type `int`. Otherwise, a failure returns a negative value.

Section 21.3: Printing format flags

The C standard (C11, and C99 too) defines the following flags for `printf()`:

Flag	Conversions	Meaning
`-`	all	The result of the conversion shall be left-justified within the field. The conversion is right-justified if this flag is not specified.
`+`	signed numeric	The result of a signed conversion shall always begin with a sign ('+' or '-'). The conversion shall begin with a sign only when a negative value is converted if this flag is not specified.
<space>	signed numeric	If the first character of a signed conversion is not a sign or if a signed conversion results in no characters, a **<space>** shall be prefixed to the result. This means that if the **<space>** and '+' flags both appear, the **<space>** flag shall be ignored.
`#`	all	Specifies that the value is to be converted to an alternative form. For `o` conversion, it shall increase the precision, if and only if necessary, to force the first digit of the result to be a zero (if the value and precision are both 0, a single 0 is printed). For `x` or `X` conversion specifiers, a non-zero result shall have `0x` (or `0X`) prefixed to it. For `a`, `A`, `e`, `E`, `f`, `F`, `g`, and `G` conversion specifiers, the result shall always contain a radix character, even if no digits follow the radix character. Without this flag, a radix character appears in the result of these conversions only if a digit follows it. For `g` and `G` conversion specifiers, trailing zeros shall not be removed from the result as they normally are. For other conversion specifiers, the behavior is undefined.

`0`	numeric	For d, i, o, u, x, X, a, A, e, E, f, F, g, and G conversion specifiers, leading zeros (following any indication of sign or base) are used to pad to the field width rather than performing space padding, except when converting an infinity or NaN. If the '0' and '-' flags both appear, the '0' flag is ignored. For d, i, o, u, x, and X conversion specifiers, if a precision is specified, the '0' flag shall be ignored. ⌦ If the '0' and **<apostrophe>** flags both appear, the grouping characters are inserted before zero padding. For other conversions, the behavior is undefined. ⌫

These flags are also supported by Microsoft with the same meanings.

The POSIX specification for `printf()` adds:

Flag	Conversions	Meaning
`'`	i, d, u, f, F, g, G	The integer portion of the result of a decimal conversion shall be formatted with thousands' grouping characters. For other conversions the behavior is undefined. The non-monetary grouping character is used.

Section 21.4: Printing the Value of a Pointer to an Object

To print the value of a pointer to an object (as opposed to a function pointer) use the `p` conversion specifier. It is defined to print `void`-pointers only, so to print out the value of a non `void`-pointer it needs to be explicitly converted ("casted*") to `void*`.

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h>  /* for printf() */

int main(void)
{
  int i;
  int * p = &i;

  printf("The address of i is %p.\n", (void*) p);

  return EXIT_SUCCESS;
}
```

Version ≥ C99

Using `<inttypes.h>` and `uintptr_t`

Another way to print pointers in C99 or later uses the `uintptr_t` type and the macros from **`<inttypes.h>`**:

```
#include <inttypes.h> /* for uintptr_t and PRIXPTR */
#include <stdio.h>    /* for printf() */

int main(void)
{
  int  i;
  int *p = &i;

  printf("The address of i is 0x%" PRIXPTR ".\n", (uintptr_t)p);

  return 0;
}
```

In theory, there might not be an integer type that can hold any pointer converted to an integer (so the type `uintptr_t` might not exist). In practice, it does exist. Pointers to functions need not be convertible to the `uintptr_t` type — though again they most often are convertible.

If the `uintptr_t` type exists, so does the `intptr_t` type. It is not clear why you'd ever want to treat addresses as

signed integers, though.

Version = K&R Version < C89

Pre-Standard History:

Prior to C89 during K&R-C times there was no type `void*` (nor header **`<stdlib.h>`**, nor prototypes, and hence no `int main(void)` notation), so the pointer was cast to `long unsigned int` and printed using the `lx` length modifier/conversion specifier.

The example below is just for informational purpose. Nowadays this is invalid code, which very well might provoke the infamous Undefined Behaviour.

```
#include <stdio.h> /* optional in pre-standard C - for printf() */

int main()
{
  int  i;
  int *p = &i;

  printf("The address of i is 0x%lx.\n", (long unsigned) p);

  return 0;
}
```

Section 21.5: Printing the Difference of the Values of two Pointers to an Object

Subtracting the values of two pointers to an object results in a signed integer *1. So it would be printed using *at least* the `d` conversion specifier.

To make sure there is a type being wide enough to hold such a "pointer-difference", since C99 **`<stddef.h>`** defines the type `ptrdiff_t`. To print a `ptrdiff_t` use the `t` length modifier.

Version ≥ C99

```
#include <stdlib.h> /* for EXIT_SUCCESS */
#include <stdio.h> /* for printf() */
#include <stddef.h> /* for ptrdiff_t */


int main(void)
{
  int a[2];
  int * p1 = &a[0], * p2 = &a[1];
  ptrdiff_t pd = p2 - p1;

  printf("p1 = %p\n", (void*) p1);
  printf("p2 = %p\n", (void*) p2);
  printf("p2 - p1 = %td\n", pd);

  return EXIT_SUCCESS;
}
```

The result might look like this:

```
p1 = 0x7fff6679f430
p2 = 0x7fff6679f434
p2 - p1 = 1
```

Please note that the resulting value of the difference is scaled by the size of the type the pointers subtracted point to, an `int` here. The size of an `int` for this example is 4.

*1If the two pointers to be subtracted do not point to the same object the behaviour is undefined.

Section 21.6: Length modifiers

The C99 and C11 standards specify the following length modifiers for `printf()`; their meanings are:

Modifier	Modifies	Applies to
hh	d, i, o, u, x, or X	`char`, `signed char` or `unsigned char`
h	d, i, o, u, x, or X	`short int` or `unsigned short int`
l	d, i, o, u, x, or X	`long int` or `unsigned long int`
l	a, A, e, E, f, F, g, or G	`double` (for compatibility with `scanf()`; undefined in C90)
ll	d, i, o, u, x, or X	`long long int` or `unsigned long long int`
j	d, i, o, u, x, or X	`intmax_t` or `uintmax_t`
z	d, i, o, u, x, or X	`size_t` or the corresponding signed type (`ssize_t` in POSIX)
t	d, i, o, u, x, or X	`ptrdiff_t` or the corresponding unsigned integer type
L	a, A, e, E, f, F, g, or G	`long double`

If a length modifier appears with any conversion specifier other than as specified above, the behavior is undefined.

Microsoft specifies some different length modifiers, and explicitly does not support `hh`, `j`, `z`, or `t`.

Modifier	Modifies	Applies to
I32	d, i, o, x, or X	`__int32`
I32	o, u, x, or X	`unsigned __int32`
I64	d, i, o, x, or X	`__int64`
I64	o, u, x, or X	`unsigned __int64`
I	d, i, o, x, or X	`ptrdiff_t` (that is, `__int32` on 32-bit platforms, `__int64` on 64-bit platforms)
I	o, u, x, or X	`size_t` (that is, `unsigned __int32` on 32-bit platforms, `unsigned __int64` on 64-bit platforms)
l or L	a, A, e, E, f, g, or G	`long double` (In Visual C++, although `long double` is a distinct type, it has the same internal representation as `double`.)
l or w	c or C	Wide character with `printf` and `wprintf` functions. (An `lc`, `lC`, `wc` or `wC` type specifier is synonymous with `C` in `printf` functions and with `c` in `wprintf` functions.)
l or w	s, S, or Z	Wide-character string with `printf` and `wprintf` functions. (An `ls`, `lS`, `ws` or `wS` type specifier is synonymous with `S` in `printf` functions and with `s` in `wprintf` functions.)

Note that the `C`, `S`, and `Z` conversion specifiers and the `I`, `I32`, `I64`, and `w` length modifiers are Microsoft extensions. Treating `l` as a modifier for `long double` rather than `double` is different from the standard, though you'll be hard-pressed to spot the difference unless `long double` has a different representation from `double`.