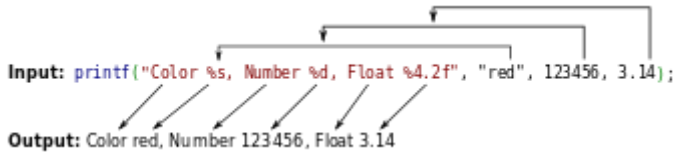
printf format string

printf format string refers to a control parameter used by a class of functions in the input/output libraries of C and many other programming languages. The string is written in a simple template language: characters are usually copied literally into the function's output, but **format specifiers**, which start with a % character, indicate the location and method to translate a piece of data (such as a number) to characters.



An example of the printf function.

"printf" is the name of one of the main C output functions, and stands for "*print f*ormatted". printf format strings are complementary to scanf format strings, which provide formatted input (parsing). In both cases these provide simple functionality and fixed format compared to more sophisticated and flexible template engines or parsers, but are sufficient for many purposes.

Many languages other than C copy the printf format string syntax closely or exactly in their own I/O functions.

Mismatches between the format specifiers and type of the data can cause crashes and other vulnerabilities. The format string itself is very often a string literal, which allows static analysis of the function call. However, it can also be the value of a variable, which allows for dynamic formatting but also a security vulnerability known as an uncontrolled format string exploit.

Contents

History

Early programming languages such as Fortran used special statements with completely different syntax from other calculations to build formatting descriptions:

```
      WRITE OUTPUT TAPE 6, 601, IA, IB, IC, AREA
601   FORMAT (4H A= ,I5,5H  B= ,I5,5H  C= ,I5,8H  AREA= ,F10.2, 13H SQUARE UNITS)
```

ALGOL 68 had more function-like api, but still used special syntax (the $ delimiters surround special formatting syntax):

```
printf(($"Color "g", number1 "6d,", number2 "4zd,", hex "16r2d,", float "-d.2d,", unsigned
value"-3d"."l$,
         "red", 123456, 89, BIN 255, 3.14, 250));
```

But using the normal function calls and data types simplifies the language and compiler, and allows the implementation of the input/output to be written in the same language. These advantages outweigh the disadvantages (such as a complete lack of type safety in many instances) and in most newer languages I/O is not part of the syntax.

C's `printf` has its origins in BCPL's `writef` function (1966). `*N` is a newline, and the order of field width and type are swapped from printf:[1]

```
WRITEF("%I2-QUEENS PROBLEM HAS %I5 SOLUTIONS*N", NUMQUEENS, COUNT)
```

Probably the first copying of the syntax to outside the C language was the Unix `printf` shell command, which first appeared in Version 4, as part of the porting to C:[2]

Format placeholder specification

Formatting takes place via placeholders within the format string. For example, if a program wanted to print out a person's age, it could present the output by prefixing it with "Your age is ". To denote that we want the integer for the age to be shown immediately after that message, we may use the format string:

```
printf("Your age is %d", age);
```

Syntax

The syntax for a format placeholder is

```
%[parameter][flags][width][.precision][length]type
```

Parameter field

This is a POSIX extension and not in C99. The Parameter field can be omitted or can be:

Character	Description
n$	*n* is the number of the parameter to display using this format specifier, allowing the parameters provided to be output multiple times, using varying format specifiers or in different orders. If any single placeholder specifies a parameter, all the rest of the placeholders MUST also specify a parameter. For example, `printf("%2$d %2$#x; %1$d %1$#x",16,17)` produces `17 0x11; 16 0x10`.

Flags field

The Flags field can be zero or more (in any order) of:

Character	Description
`-` (minus)	Left-align the output of this placeholder. (The default is to right-align the output.)
`+` (plus)	Prepends a plus for positive signed-numeric types. positive = `+`, negative = `-`. (The default doesn't prepend anything in front of positive numbers.)
(space)	Prepends a space for positive signed-numeric types. positive = , negative = `-`. This flag is ignored if the `+` flag exists. (The default doesn't prepend anything in front of positive numbers.)
`0` (zero)	When the 'width' option is specified, prepends zeros for numeric types. (The default prepends spaces.) For example, `printf("%2X",3)` produces ` 3`, while `printf("%02X",3)` produces in `03`.
# (hash)	Alternate form: For `g` and `G` types, trailing zeros are not removed. For `f`, `F`, `e`, `E`, `g`, `G` types, the output always contains a decimal point. For `o`, `x`, `X` types, the text `0`, `0x`, `0X`, respectively, is prepended to non-zero numbers.

Width field

The Width field specifies a *minimum* number of characters to output, and is typically used to pad fixed-width fields in tabulated output, where the fields would otherwise be smaller, although it does not cause truncation of oversized fields.

The width field may be omitted, or a numeric integer value, or a dynamic value when passed as another argument when indicated by an asterisk `*`. For example, `printf("%*d", 5, 10)` will result in `  10` being printed, with a total width of 5 characters.

Though not part of the width field, a leading zero is interpreted as the zero-padding flag mentioned above, and a negative value is treated as the positive value in conjunction with the left-alignment `-` flag also mentioned above.

Precision field

The Precision field usually specifies a *maximum* limit on the output, depending on the particular formatting type. For floating point numeric types, it specifies the number of digits to the right of the decimal point that the output should be rounded. For the string type, it limits the number of characters that should be output, after which the string is truncated.

The precision field may be omitted, or a numeric integer value, or a dynamic value when passed as another argument when indicated by an asterisk `*`. For example, `printf("%.*s", 3, "abcdef")` will result in `abc` being printed.

Length field

The Length field can be omitted or be any of:

Character	Description
`hh`	For integer types, causes `printf` to expect an `int`-sized integer argument which was promoted from a `char`.
`h`	For integer types, causes `printf` to expect an `int`-sized integer argument which was promoted from a `short`.
`l`	For integer types, causes `printf` to expect a `long`-sized integer argument. For floating point types, this has no effect.[3]
`ll`	For integer types, causes `printf` to expect a `long long`-sized integer argument.
`L`	For floating point types, causes `printf` to expect a `long double` argument.
`z`	For integer types, causes `printf` to expect a `size_t`-sized integer argument.
`j`	For integer types, causes `printf` to expect a `intmax_t`-sized integer argument.
`t`	For integer types, causes `printf` to expect a `ptrdiff_t`-sized integer argument.

Additionally, several platform-specific length options came to exist prior to widespread use of the ISO C99 extensions:

Characters	Description
`I`	For signed integer types, causes `printf` to expect `ptrdiff_t`-sized integer argument; for unsigned integer types, causes `printf` to expect `size_t`-sized integer argument. *Commonly found in Win32/Win64 platforms.*
`I32`	For integer types, causes `printf` to expect a 32-bit (double word) integer argument. *Commonly found in Win32/Win64 platforms.*
`I64`	For integer types, causes `printf` to expect a 64-bit (quad word) integer argument *Commonly found in Win32/Win64 platforms.*
`q`	For integer types, causes `printf` to expect a 64-bit (quad word) integer argument *Commonly found in BSD platforms.*

ISO C99 includes the `inttypes.h` header file that includes a number of macros for use in platform-independent `printf` coding. These must be outside double-quotes, e.g. `printf("%" PRId64 "\n", t);`

Example macros include:

Macro	Description
`PRId32`	Typically equivalent to `I32d` (*Win32/Win64*) or `d`
`PRId64`	Typically equivalent to `I64d` (*Win32/Win64*), `lld` (*32-bit platforms*) or `ld` (*64-bit platforms*)
`PRIi32`	Typically equivalent to `I32i` (*Win32/Win64*) or `i`
`PRIi64`	Typically equivalent to `I64i` (*Win32/Win64*), `lli` (*32-bit platforms*) or `li` (*64-bit platforms*)
`PRIu32`	Typically equivalent to `I32u` (*Win32/Win64*) or `u`
`PRIu64`	Typically equivalent to `I64u` (*Win32/Win64*), `llu` (*32-bit platforms*) or `lu` (*64-bit platforms*)
`PRIx32`	Typically equivalent to `I32x` (*Win32/Win64*) or `x`
`PRIx64`	Typically equivalent to `I64x` (*Win32/Win64*), `llx` (*32-bit platforms*) or `lx` (*64-bit platforms*)

Type field

The Type field can be any of:

Character	Description
%	Prints a literal % character (this type doesn't accept any flags, width, precision, length fields).
d, i	int as a signed decimal number. %d and %i are synonymous for output, but are different when used with scanf() for input (where using %i will interpret a number as hexadecimal if it's preceded by 0x, and octal if it's preceded by 0.)
u	Print decimal unsigned int.
f, F	double in normal (fixed-point) notation. f and F only differs in how the strings for an infinite number or NaN are printed (inf, infinity and nan for f; INF, INFINITY and NAN for F).
e, E	double value in standard form ([-]d.ddd e[+/-]ddd). An E conversion uses the letter E (rather than e) to introduce the exponent. The exponent always contains at least two digits; if the value is zero, the exponent is 00. In Windows, the exponent contains three digits by default, e.g. 1.5e002, but this can be altered by Microsoft-specific _set_output_format function.
g, G	double in either normal or exponential notation, whichever is more appropriate for its magnitude. g uses lower-case letters, G uses upper-case letters. This type differs slightly from fixed-point notation in that insignificant zeroes to the right of the decimal point are not included. Also, the decimal point is not included on whole numbers.
x, X	unsigned int as a hexadecimal number. x uses lower-case letters and X uses upper-case.
o	unsigned int in octal.
s	null-terminated string
c	char (character).
p	void * (pointer to void) in an implementation-defined format.
a, A	double in hexadecimal notation, starting with 0x or 0X. a uses lower-case letters, A uses upper-case letters.[4][5] (C++11 iostreams have a hexfloat that works the same).
n	Print nothing, but writes the number of characters successfully written so far into an integer pointer parameter. Java: indicates a platform neutral newline/carriage return.[6] Note: This can be utilized in Uncontrolled format string exploits.

Custom format placeholders

There are a few implementations of printf-like functions that allow extensions to the escape-character-based mini-language, thus allowing the programmer to have a specific formatting function for non-builtin types. One of the most well-known is the (now deprecated) glibc's register_printf_function() However, it is rarely used due to the fact that it conflicts with static format string checking. Another is Vstr custom formatters, which allows adding multi-character format names.

Some applications (like the Apache HTTP Server) include their own printf-like function, and embed extensions into it. However these all tend to have the same problems that register_printf_function() has.

The Linux kernel printk function supports a number of ways to display kernel structures using the generic %p specification, by *appending* additional format characters.[7] For example, %pI4 prints an IPv4 address in dotted-decimal form. This allows static format string checking (of the %p portion) at the expense of full compatibility with normal printf.

Most non-C languages that have a printf-like function work around the lack of this feature by just using the %s format and converting the object to a string representation. C++ offers a notable exception, in that it has a printf function inherited from its C history, but also has a completely different mechanism that is preferred.

Vulnerabilities

Invalid conversion specifications

If the syntax of a conversion specification is invalid, behavior is undefined, and can cause program termination. If there are too few function arguments provided to supply values for all the conversion specifications in the template string, or if the arguments are not of the correct types, the results are also undefined. Excess arguments are ignored. In a number of cases, the undefined behavior has led to "Format string attack" security vulnerabilities.

Some compilers, like the GNU Compiler Collection, will statically check the format strings of printf-like functions and warn about problems (when using the flags `-Wall` or `-Wformat`). GCC will also warn about user-defined printf-style functions if the non-standard "format" `__attribute__` is applied to the function.

Field width versus explicit delimiters in tabular output

Using only field widths to provide for tabulation, as with a format like `%8d%8d%8d` for three integers in three 8-character columns, will not guarantee that field separation will be retained if large numbers occur in the data. Loss of field separation can easily lead to corrupt output. In systems which encourage the use of programs as building blocks in scripts, such corrupt data can often be forwarded into and corrupt further processing, regardless of whether the original programmer expected the output would only be read by human eyes. Such problems can be eliminated by including explicit delimiters, even spaces, in all tabular output formats. Simply changing the dangerous example from before to `%7d %7d %7d` addresses this, formatting identically until numbers become larger, but then explicitly preventing them from becoming merged on output due to the explicitly included spaces. Similar strategies apply to string data.

Programming languages with printf

Languages that use format strings that deviate from the style in this article (such as AMPL and Elixir), languages that inherit their implementation from the JVM or other environment (such as Clojure and Scala), and languages that do not have a standard native printf implementation but have external libraries which emulate printf behavior (such as JavaScript) are not included in this list.

- awk (via sprintf)
- C
  - C++ (also provides overloaded shift operators and manipulators as an alternative for formatted output – see iostream and iomanip)
  - Objective-C
- D
- F#
- G (LabVIEW)
- GNU MathProg
- GNU Octave
- Go
- Haskell
- J
- Java (since version 1.5) and JVM languages
- Lua (string.format)
- Maple
- MATLAB
- Max (via the sprintf object)
- Mythryl
- PARI/GP
- Perl
- PHP
- Python (via % operator)
- R
- Red/System

- Ruby
- Tcl (via format command)
- Transact-SQL (via xp_sprintf)
- Vala (via `print()` and `FileStream.printf()`)
- The `printf` utility command, sometimes built in the shell like some implementations of the Korn shell (ksh), Bourne again shell (bash), or Z shell (zsh). These commands usually interpret C escapes in the format string.

See also

- Format (Common Lisp)
- C standard library
- Format string attack
- `iostream`
- ML (programming language)
- printf debugging
- `printf` (Unix)
- `printk` (print kernel messages)
- `scanf`
- string interpolation

References

1. "BCPL" (http://www.cl.cam.ac.uk/users/mr/BCPL.html). *www.cl.cam.ac.uk*. Retrieved 19 March 2018.
2. McIlroy, M. D. (1987). *A Research Unix reader: annotated excerpts from the Programmer's Manual, 1971–1986* (http://www.cs.dartmouth.edu/~doug/reader.pdf) (PDF) (Technical report). CSTR. Bell Labs. 139.
3. ISO/IEC (1999). *ISO/IEC 9899:1999(E): Programming Languages - C*. *§7.19.6.1 para 7*
4. "'"The GNU C Library Reference Manual", "12.12.3 Table of Output Conversions"'" (https://www.gnu.org/software/libc/manual/html_node/Table-of-Output-Conversions.html#Table-of-Output-Conversions) Gnu.org. Retrieved 2014-03-17.
5. "printf" (http://www.cplusplus.com/reference/cstdio/printf/) (%a added in C99)
6. "Formatting Numeric Print Output" (https://docs.oracle.com/javase/tutorial/java/data/numberformat.html) *The Java Tutorials*. Oracle Inc. Retrieved 19 March 2018.
7. "Linux kernel Documentation/printk-formats.txt" (https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/printk-formats.txt) Git.kernel.org. Retrieved 2014-03-17.

External links

- C++ reference for `std::fprintf`
- gcc printf format specifications quick reference
- `printf`: print formatted output – System Interfaces Reference, The Single UNIX Specification, Issue 7 from The Open Group
- The `Formatter` specification in Java 1.5
- GNU Bash `printf(1)` builtin