Coffee Into Bugs: libc from Scratch

Part 3: printf

Dave Allison

*“A software engineer is a device for converting coffee into bugs”*

I’m writing my own version of *libc* (the standard C library) from scratch for my 6502 project, a small 6502 based computer and with a C compiler. This is the first part of a series of articles describing how to write a simple version of *libc*.

All this code will be in C (the C99 standard) and the hope is that it is useful as a learning tool for those wanting to learn how to program in C, or those writing their own libc. This is all my own work and I have not copied anything from anywhere else. It’s also not related to my employer.

This article will present the popular **printf** functions, the C library’s formatted output facility.

# Formatted Output

The standard C library contains a set of functions to read from and write to FILE pointers, generally attached to a file on disk. This allows the program to read and write raw bytes, but if any formatting is needed, the programmer would have to do that themselves.

Enter C formatted input and output, in the form of the ***printf*** and ***scanf*** functions. The **f** at the end means *formatted*, so it should be obvious that *printf* means *print formatted*.

Both these functions take a character string describing the format of the output and a variable number of arguments that provide either the addresses of where to put the input (for scanf) or the values to be output (for printf). The format string uses the percent (%) symbol to indicate a formatting command and all non-formatting commands are treated as a literal, read or written as they appear in the format string.

This article will look at *printf* with *scanf* kept for the future. I doubt there is a C programmer who wouldn’t recognize this:

void Foo(int x) {

printf("The value is %d\n", x);

}

Here, the format string contains a single formatting command “%d” which means “convert to decimal” and the second argument contains the value to convert. The “%d” is a simple formatting command, but you can expand upon it with field widths, padding characters, different output conversions, etc. For example, to print the value in hex, using 8 digits with ‘0’ as the left padding character, the function would look like:

void Foo(int x) {

printf("The value is %08x\n", x);

}

# Variable argument functions

The *printf* function takes a variable number of arguments. This is a C language facility that allows the function prototype to specify that there may be any number of additional arguments after this point. The prototype for *printf* looks like this:

int printf(const char\* format, ...);

The ellipsis (…) token says: “there may be any number of arguments here, including none”.

Therefore, the caller to *printf* must provide at least one argument (the format string) but can then add any number of additional arguments. So how does *printf* know how many arguments are provided? It looks at the formatting commands in the format string and for each one, consumes another argument. This begs the question of what happens if you don’t provide sufficient arguments? Most modern compilers can detect this error, but in the absence of compiler support, *printf* will gladly consume whatever happens to be on the stack unaware of anything being wrong.

## How argument passing works in C

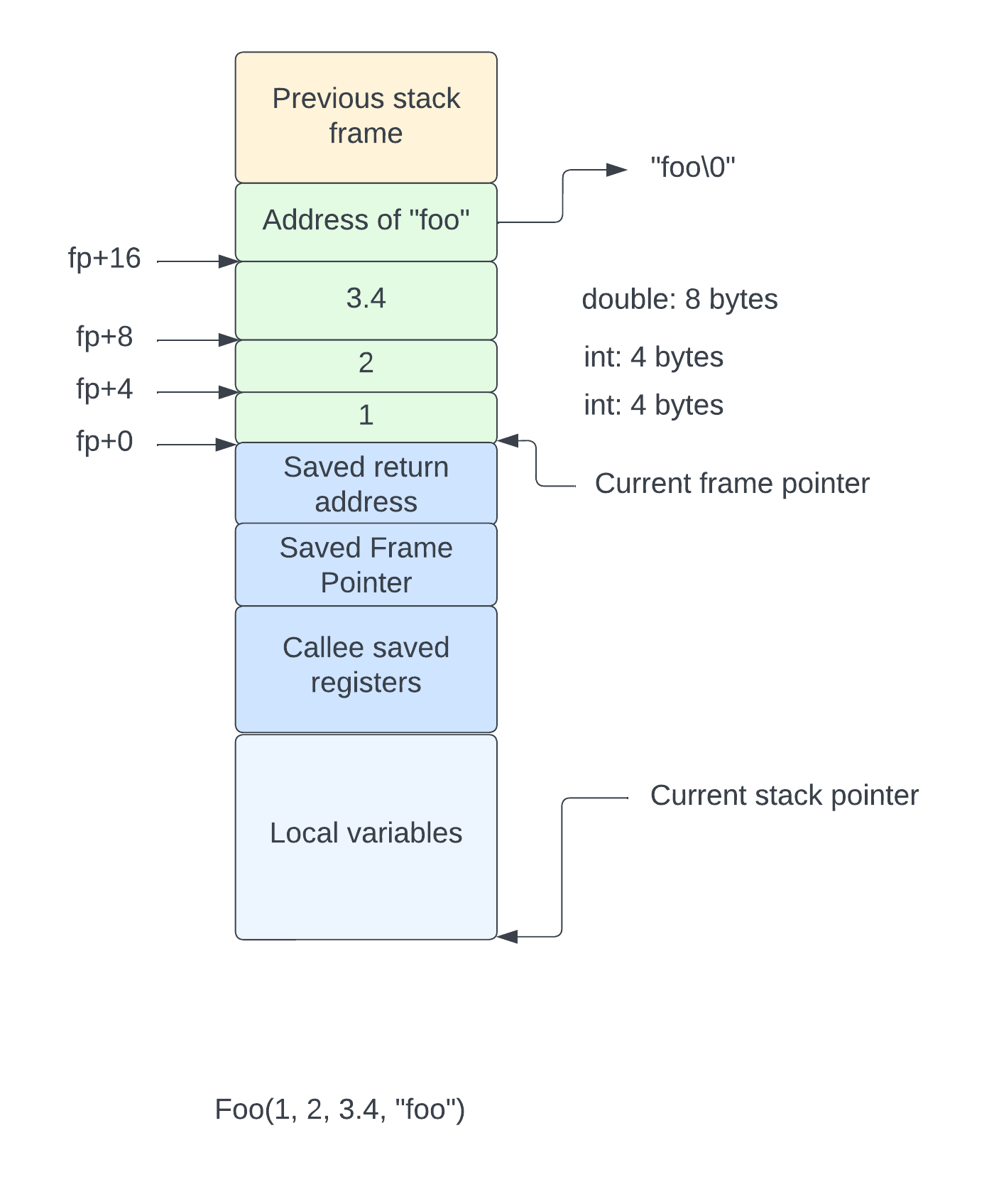
An *Application Binary Interface* (ABI) document defines a set of rules describing how arguments are passed to functions. Traditionally, arguments were all pushed onto a runtime stack from right to left with the leftmost argument being at the top of the stack. Most modern CPU architectures allow some arguments to be passed in registers to speed up most procedure calls, but all will define that any arguments that don’t fit into the small number of argument registers will be pushed onto the runtime stack.

In the simple case where arguments are all passed on the stack, if the running program wants to see the arguments in a function, it can simply obtain the value of the *Frame Pointer* (usually in one of the CPU registers) and read the memory above that address.

In C, integer values are passed as an *int* type. Types smaller than an *int* are promoted to an *int* before being pushed onto the stack. Pointers are passed as a native address (8 bytes on a 64-bit CPU). Integer values larger than an *int* (long and long long) are passed as their native size. Floating point numbers are all passed as *double* precision types.

These rules simplify the task of reading arguments in variable-argument functions, since you know that, on a modern CPU, most integer values will be passed as a 32-bit number, pointers and longs as 64-bit and floating point as 64-bit doubles.

The following diagram shows the stack situation after during a call to a function *Foo*, with 4 arguments:



The *frame pointer* register points typically points to the address just above the saved return address (the value of the *stack pointer* before the call was made). The arguments are pushed onto the stack in reverse order, and each occupies either 4 or 8 bytes of memory (assuming a typical 64-bit machine where an *int* is 32-bits and a *double* is 64-bits).

Therefore, the arguments are usually (in absence of passing them in registers) just above the *frame pointer* and all local variables are below the *frame pointer* and above the *stack pointer*.

On a smaller computer, like the 6502, due to memory limits and the cost of multi-byte values, *ints* are usually 16 bits. Also, since the address bus is only 16 bits wide, pointers are also 16 bits. Depending on the compiler and runtime library, larger numbers may or may not be supported.

## The <stdarg.h> facility

In C, you gain access to the arguments passed to variable argument functions using the macros defined in the header file <stdarg.h>. This file provides a couple of facilities, the most common of which are:

1. Obtain the address of the first argument beyond the named arguments (*va\_start*)[[1]](#footnote-1)
2. Obtain the value of the next argument (*va\_arg*)

The idea is that you would call *va\_start* to initialize a local variable to the address of the additional arguments and then call *va\_arg* to read the value of each argument. For example, this function calculates the sum of a variable set of integers:

int Add(int n, ...) {

va\_list ap;

va\_start(ap, n);

int sum = 0;

while (n > 0) {

sum += va\_arg(ap, int);

}

return sum;

}

It could be called like thisL

int v = Add(4, 56, 78, 23, 45);

Resulting in the value of 202 in v.

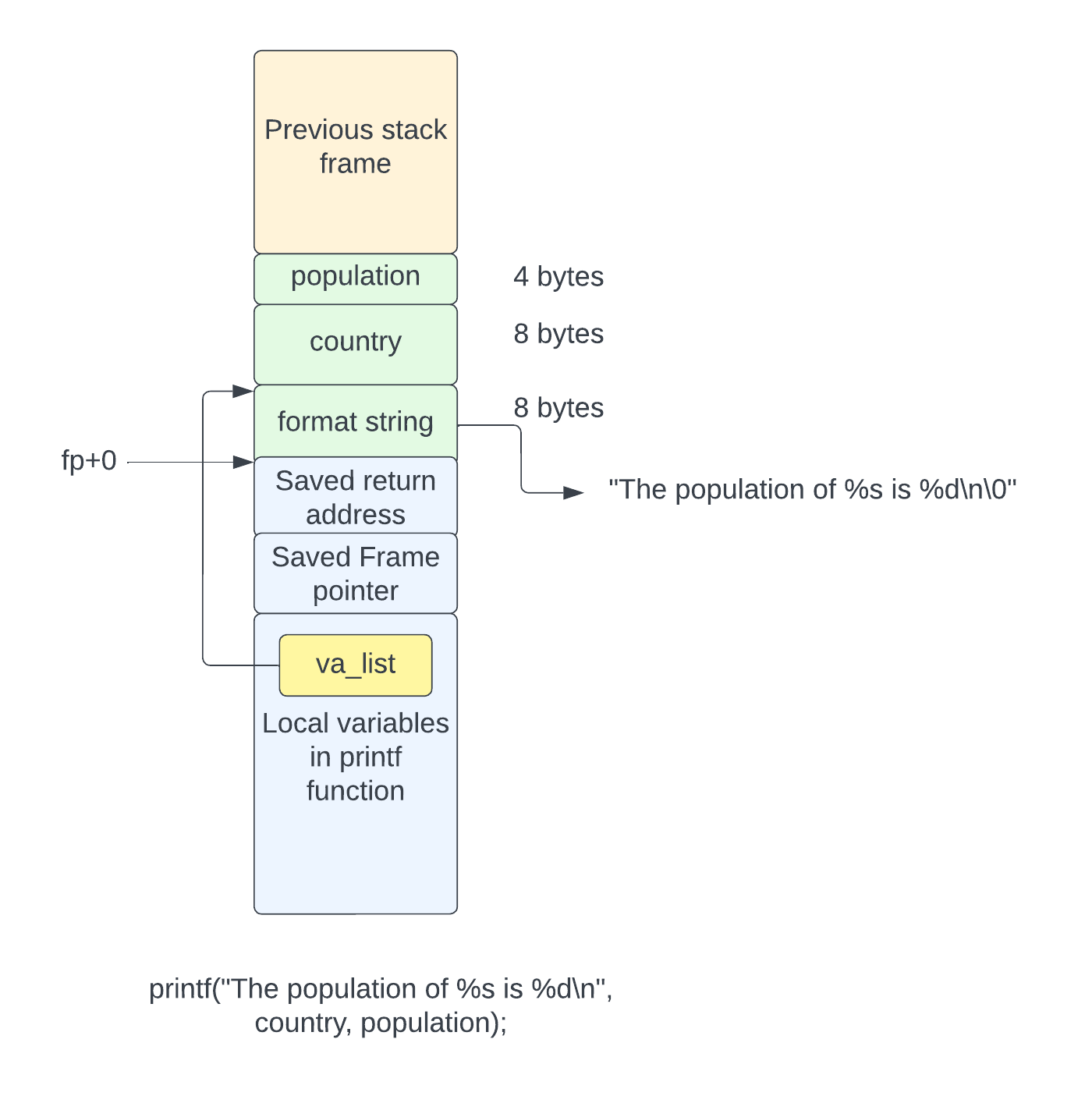
The local variable **ap**, of type *va\_list* is initialized to refer to the address of the first argument beyond **n**, which says how many additional arguments will be available. Then, for each argument, we read the value as an *int*, using *va\_arg*, and add it to the sum.

In its simplest form, *va\_list* is a pointer that contains the address of the next argument to be read from the stack. Each time *va\_arg* is called, it obtains the value of the argument at the address held in the *va\_list* and increments the address held in the *va\_list* by an amount to move it to the next argument. If the size of the argument is less than the size of an *int*, the increment is by the size of an *int* (4 bytes in this case). Otherwise the increment is by 8.

The ABI also specifies how non-scalar types are to be passed. This can either be done by copying them onto the stack directly or by pushing them earlier and passing a reference to their stack address. However, functions that take variable arguments, like *printf,* don’t really work with aggregate types (structs).

In a more complex form (as specified in the *x86\_64* and *Aarch64* ABIs), the *va\_list* contains a couple of areas that hold the values of the arguments passed in registers. This complicates the *va\_arg* macro as it has to determine if the next argument comes from the *va\_list* struct or from the stack. For the sake of brevity, we won’t cover how that all operates here, but instead assume that the arguments are all on the stack. The compiler can always arrange for this to be the case (this is exactly what Apple’s AArch64 ABI does as far as I can tell).

Let’s now look at the stack when *printf* is called. Remember that *printf* always takes a known first argument, the *format string*. This is normally the address of a string literal (in a read-only address known by the linker). As you might have guessed, *printf* uses the facilities in <stdarg.h> to read the arguments, so we also show a *va\_list* variable pointing to the first argument, in the local variables part of stack frame.



As you might recall, *printf’s* signature is:

int printf(const char\* format, ...);

The *va\_list* is set up by *printf* by a call to *va\_start* as follows:

va\_list ap;

va\_start(ap, format);

This sets the value of *ap* to the address of the *format* argument plus its size (8 bytes in this case), making *ap* contain the address of the value of *country[[2]](#footnote-2)*.

Each call to *va\_arg* will retrieve the value at *ap* and increment *ap* to point to the next argument.

# The Various *printf* Functions

It should be reasonably obvious how *printf* works. It goes through each character in the format string and either writes it to the output directly, or if it’s a format command beginning with a % sign, it retrieves the argument of the type specified by the format command, converts it to ASCII as necessary, and writes it to the output. The function terminates when the end of the format string is reached.

For straight *printf*, the output is written to the FILE pointer *stdout*. Other variants (*fprintf*, *snprint*,…) write to other places. Since there are multiple versions of *printf* that write to different locations, a common function that can write to anywhere would be an appropriate design. Let’s define a functor to pass to the common *printf* function as follows:

typedef int (\*Writer)(const char\* s, size\_t len, void\* data);

This is a function pointer that takes a pointer to something to write, its length, and an opaque pointer to data that holds information about the thing being written to. It returns the number of bytes written to the output.

For those versions of *printf* that write to a FILE pointer, we can define a *Writer* for it as:

static int FILEWriter(const char\* s, size\_t len, void\* data) {

FILE\* fp = data;

return fwrite(s, 1, len, fp);

}

In this case, the *data* argument is simply a *FILE* pointer and it calls the *fwrite* function directly.

For the *printf* variants that write to memory (*snprintf*, etc.), we need another struct to hold the address to write to, and the length of the memory we can use[[3]](#footnote-3).

typedef struct {

char\* p; // Current place to write to.

ssize\_t len; // Remaining space.

} StringData;

static int StringWriter(const char\* s, size\_t len, void\* data) {

StringData\* str = data;

if (str->len > 0 && len > str->len) {

len = str->len;

}

memcpy(str->p, s, len);

str->p += len;

str->len -= len;

return len;

}

The common *printf* function is defined with the following signature:

static int Printf(Writer writer, void\* data, const char\* format, va\_list ap);

It’s a static function so that it’s not visible as a global function and takes the parameters:

1. A pointer to the writer function used to write to the destination
2. An opaque pointer to the data for the writer function
3. The *format* string pointer
4. A variable argument struct that has been set to the first argument following the *format* string in the original call.

Given this common function, we can define all the variants we need:

* *printf* and *vprintf*: prints to *stdout*
* *fprintf* and *vfprintf*: prints to a FILE pointer
* *sprintf* and *vsprintf*: prints to memory with no bounds (not recommended)
* *snprintf* and *vsnprintf*: prints to memory with bounds

The functions starting with ‘***v***’ take a *va\_list* instead of a set of arguments.

Here are the various *printf* functions that call the common *Printf* function.

int fprintf(FILE\* fp, const char\* format, ...) {

va\_list ap;

va\_start(ap, format);

int v = Printf(FILEWriter, fp, format, ap);

va\_end(ap);

return v;

}

int vfprintf(FILE \* restrict stream,

const char \* restrict format, va\_list arg) {

return Printf(FILEWriter, stream, format, arg);

}

int printf(const char\* format, ...) {

va\_list ap;

va\_start(ap, format);

int v = Printf(FILEWriter, stdout, format, ap);

va\_end(ap);

return v;

}

int vprintf(const char \* restrict format, va\_list arg) {

return Printf(FILEWriter, stdout, format, arg);

}

int sprintf(char\* s, const char\* format, ...) {

va\_list ap;

va\_start(ap, format);

StringData data = {s, -1};

int v = Printf(StringWriter, &data, format, ap);

va\_end(ap);

return v;

}

int vsprintf(char \* restrict s,

const char \* restrict format, va\_list arg) {

StringData data = {s, -1};

return Printf(StringWriter, &data, format, arg);

}

int snprintf(char\* s, size\_t len, const char\* format, ...) {

va\_list ap;

va\_start(ap, format);

StringData data = {s, len};

int v = Printf(StringWriter, &data, format, ap);

va\_end(ap);

return v;

}

int vsnprintf(char \* restrict s, size\_t n,

const char \* restrict format, va\_list arg) {

StringData data = {s, n};

return Printf(StringWriter, &data, format, arg);

}

## The Common *Printf* Function

So, now let’s take a look at how the common *Printf* function works. As stated in the previous section, the function looks at each character in the format string and either prints it out or interprets it as a formatting command.

Formatting commands begin with a **%** sign and are followed by a fairly complex sequence of format characters. In its simplest form, the % is followed by a single letter that specifies the type of conversion to be done. This is one of:

* **%d, %i, %u**: convert int to decimal
* **%x, %X**: convert int to hexadecimal
* **%o**: convert int to octal
* **%s**: print pointer as a string of characters
* **%c**:print int as character
* **%e, %f** ,**%g**: convert double to decimal in various forms
* **%p**: print pointer as an address
* **%n:** write the number of conversions to the address provided

If more complex formatting is required, the formatting command may be enhanced and may include:

1. A field width, with either left or right justification in the field
2. A precision, specifying the number of digits after the decimal point for floating point values or the width of the value for integers (padded left with 0)
3. Whether to use a space or 0 for padding characters to fill the field before the value.
4. The field width and precision can be obtained from an argument instead of being coded into the command.

The rules for formatting are quite complex and can be found in the C99 standard, or by typing “**man fprintf”** on the shell in a Mac or Linux computer.

The format command can contain the following, all of which are optional.

1. A ‘-‘ to specify left justification
2. A‘+’ which means to prepend a + for positive decimals
3. A space, which means to use a space instead of sign
4. An integral field width or \*.
5. A dot, followed by an integral precision or \*.

A couple of examples might make this clearer. The ¬ character represents a space.

|  |  |  |
| --- | --- | --- |
| Format string | Meaning | Example |
| %6d | Decimal right justified in a 6 character field | ¬¬¬100 |
| %-8s | String field left justified in 8 character field | hello¬¬¬ |
| %.6d | Decimal padded to 6 digits with 0 | 001234 |
| %08x | Hexadecimal padded to 8 digits with 0 | 0000feed |
| %4.2f | Floating point with 2 digits after the decimal point | 3.14 |
| %\*d | Decimal with field with as next argument | (args 6, 1234)  ¬¬1234 |
| %-\*s | String left justified in field width from next arg | (args 8, “hello”)  hello¬¬¬ |

In addition to the width and precision, the format also allows a *modifier* to applied. This specifies additional information about the type of the argument passed. For example, the “**ll**” modifier says that argument is a *long long* type and not an int. This is because there is no way for the *printf* function to know what the type of the next argument on the stack is, so we have to tell it.

Let’s just dive in with the full *Printf* function as it’s not too long. I think the only thing missing from this implementation is the %a and %A formatting commands because I don’t think I’ve ever used them. Maybe I’ll write them one day.

static int Printf(Writer writer, void\* data, const char\* format, va\_list ap) {

char buf[256];

char\* v;

const char\* end = buf + sizeof(buf);

const char\* p = format;

int count = 0;

long long value\_ll;

const char\* value\_s;

char value\_c;

double value\_f;

void\* value\_p;

while (\*p != '\0') {

if (\*p == '%') {

p++;

ConversionFormat fmt = {0};

p = CollectFormat(p, &fmt);

bool negative = false;

bool is\_unsigned = false;

// If \* is specified for field width or precision, fetch them

// from the arg list now.

ResolveFieldWidth(&fmt, &ap);

ResolvePrecision(&fmt, &ap);

// Get value from arg list based on conversion.

GetNextArgument(\*p, &fmt, &ap, &value\_ll, &value\_s, &value\_c,

&value\_f, &value\_p, &is\_unsigned, &negative);

// Convert the arg value and write it.

switch (\*p) {

case 'd':

case 'i':

case 'u':

p++;

v = ConvertDecimalLongLong(value\_ll, buf, sizeof(buf));

count += WriteFormatted(writer, data, &fmt, v, end - v,

!is\_unsigned && negative, true);

break;

case 'x':

case 'X': {

bool upper = \*p == 'X';

p++;

v = ConvertHexLongLong(value\_ll, buf, sizeof(buf), upper);

fmt.fill\_zero = false;

count += WriteFormatted(writer, data, &fmt, v, end - v, false, true);

break;

}

case 'o':

p++;

v = ConvertOctalLongLong(value\_ll, buf, sizeof(buf));

count += WriteFormatted(writer, data, &fmt, v, end - v, false, true);

break;

case 'p':

p++;

v = ConvertHexPointer(value\_p, buf, sizeof(buf));

count += WriteFormatted(writer, data, &fmt, v, end - v, false, true);

break;

// **TODO: support %a**

case 'f':

FixFloatPrecision(&fmt, sizeof(buf) - 2);

p++;

v = \_\_PrintFloatFormat(value\_f, fmt.precision, buf, sizeof(buf));

count +=

WriteFormatted(writer, data, &fmt, v, strlen(v), false, false);

break;

case 'e':

FixFloatPrecision(&fmt, sizeof(buf) - 5);

p++;

v = \_\_PrintScientificFormat(value\_f, fmt.precision, buf, sizeof(buf));

count +=

WriteFormatted(writer, data, &fmt, v, strlen(v), false, false);

break;

case 'g':

FixFloatPrecision(&fmt, sizeof(buf) - 5);

p++;

v = \_\_PrintGeneralFormat(value\_f, fmt.precision, buf, sizeof(buf));

count +=

WriteFormatted(writer, data, &fmt, v, strlen(v), false, false);

break;

case 'c': {

p++;

char b[1] = {value\_c};

// Don't fill or prepend characters.

RemoveFormatting(&fmt);

count += WriteFormatted(writer, data, &fmt, b, 1, false, false);

break;

}

case 's': {

p++;

// Don't fill or prepend strings.

RemoveFormatting(&fmt);

// Limit width to precision.

size\_t len = strlen(value\_s);

if (fmt.precision != kWidthDefault) {

if (fmt.precision < len) {

len = fmt.precision;

}

}

count +=

WriteFormatted(writer, data, &fmt, value\_s, len, false, false);

break;

}

case 'n':

p++;

WriteCount(&fmt, count, value\_p);

break;

default:

count += writer(p++, 1, data);

break;

}

} else {

count += writer(p++, 1, data);

}

}

return count;

}

It’s basically a single loop, iterating over the characters in format, in the pointer ***p***. For each one, if it’s a %, a conversion is made by extracting the next argument from the stack, converting it and formatting it according to the command. If it’s not a %, it’s written directly to the output.

The formatting command is handled after a % is seen and is parsed using *CollectFormat*. This looks at the current characters in the format string, collects and parses them into a *ConversionFormat* struct, and returns the next character after the end of the formatting characters.

The *ConversionFormat* struct tells the formatter what to do and is defined as:

typedef enum {

kWidthDefault = -1,

kWidthNextArg = -2,

kWidthExplicit = 0,

} Width;

typedef enum {

kModNone = 0,

kModChar,

kModShort,

kModLong,

kModLongLong,

kModLongDouble,

kModIntMax,

kModSize\_t,

kModPtrdiff\_t

} Modifier;

typedef struct {

Width field\_width;

Width precision;

bool left\_justify;

bool fill\_zero;

bool prepend\_sign;

bool prepend\_space;

bool alternate\_form;

Modifier modifier;

} ConversionFormat;

The *Width* enumeration holds a supplied width, which can be missing, come from the next argument, or specified explicitly in the format string. This is used for the *field\_width* and *precision* members. The *Modifier* enumeration holds the type modifier specified in the format string, allowing the printf function to know what type to expect as the next argument.

The boolean fields are pretty self-explanatory[[4]](#footnote-4).

To parse a formatting command into this struct, we have the *CollectFormat* function.

static const char\* CollectFormat(const char\* p, ConversionFormat\* format) {

format->field\_width = kWidthDefault;

format->precision = kWidthDefault;

format->left\_justify = false;

format->fill\_zero = false;

format->modifier = kModNone;

int n = 0;

bool done = false;

while (!done && \*p != '\0') {

switch (\*p) {

case '-':

p++;

format->left\_justify = true;

break;

case '0':

if (!format->left\_justify) {

format->fill\_zero = true;

}

p++;

break;

case '#':

format->alternate\_form = true;

p++;

break;

case ' ':

format->prepend\_space = true;

p++;

break;

case '+':

format->prepend\_sign = true;

p++;

break;

default:

done = true;

break;

}

}

// Field width.

if (\*p == '\*') {

p++;

format->field\_width = kWidthNextArg;

} else if (isdigit(\*p)) {

while (isdigit(\*p)) {

n = n \* 10 + \*p++ - '0';

}

format->field\_width = n;

}

// Precision.

if (\*p == '.') {

p++;

if (\*p == '\*') {

p++;

format->precision = kWidthNextArg;

} else if (isdigit(\*p)) {

n = 0;

while (isdigit(\*p)) {

n = n \* 10 + \*p++ - '0';

}

format->precision = n;

}

}

// Modifier (length field).

switch (\*p) {

case 'l': // l or ll

if (p[1] == 'l') {

format->modifier = kModLongLong;

p++;

} else {

format->modifier = kModLong;

}

p++;

break;

case 'h': // h or hh

if (p[1] == 'h') {

format->modifier = kModChar;

p++;

} else {

format->modifier = kModShort;

}

p++;

break;

case 'L':

format->modifier = kModLongDouble;

p++;

break;

case 'z':

format->modifier = kModSize\_t;

p++;

break;

case 'j':

format->modifier = kModIntMax;

p++;

break;

case 't':

format->modifier = kModPtrdiff\_t;

p++;

break;

default:

break;

}

return p;

}

It’s fairly long due to the number of options it needs to handle, but it’s pretty easy to follow. It uses the pointer ‘***p***’ to refer to the current character and returns the address of the character after the parse has completed. The first loop collects the boolean fields, then the field width and precision are collected, followed by the modifier. It does not collect the final format specifier.

If the *ConversionFormat* specified that the field with and/or precision comes from the next argument, we call the *ResolveFieldWidth* and *ResolvePrecision* function to read the value from an integer argument:

static void ResolvePrecision(ConversionFormat\* fmt, va\_list\* ap) {

if (fmt->precision == kWidthNextArg) {

fmt->precision = va\_arg(\*ap, int);

}

}

static void ResolveFieldWidth(ConversionFormat\* fmt, va\_list\* ap) {

if (fmt->field\_width == kWidthNextArg) {

fmt->field\_width = va\_arg(\*ap, int);

}

}

Notice that these functions take a pointer to a *va\_list* as their argument. This is to allow them to modify the *va\_list* so that it consumes an argument that the caller won’t see again. It is quite ambiguous whether this is legitimate or not. The standard says you can pass a pointer to a *va\_list*, but other sources say that passing the *va\_list* by value is needed. I’ve gone with passing a pointer and that seems to work on everything I’ve tried it on.

After resolving the field width and precision arguments, we then collect the actual argument to be converted using the *GetNextArgument* function.

static void GetNextArgument(char cmd, ConversionFormat\* fmt, va\_list\* ap,

long long\* value\_ll, const char\*\* value\_s,

char\* value\_c, double\* value\_f, void\*\* value\_p,

bool\* is\_unsigned, bool\* negative) {

switch (cmd) {

case 'd':

case 'u':

case 'i':

case 'x':

case 'X':

case 'o':

\*is\_unsigned = cmd == 'u' || cmd == 'x' || cmd == 'X';

switch (fmt->modifier) {

case kModLong:

\*value\_ll = va\_arg(\*ap, long);

break;

case kModLongLong:

\*value\_ll = va\_arg(\*ap, long long);

break;

case kModChar:

\*value\_ll = (int)va\_arg(\*ap, int) & 0xff;

break;

case kModShort:

\*value\_ll = (int)va\_arg(\*ap, int) & 0xffff;

break;

case kModSize\_t:

\*value\_ll = (int)va\_arg(\*ap, size\_t);

\*is\_unsigned = true;

break;

case kModIntMax:

\*value\_ll = (int)va\_arg(\*ap, intmax\_t);

break;

case kModPtrdiff\_t:

\*value\_ll = (int)va\_arg(\*ap, ptrdiff\_t);

break;

case kModLongDouble:

\*value\_f = (int)va\_arg(\*ap, long double);

break;

default:

\*value\_ll = (int)va\_arg(\*ap, int);

break;

}

if (!\*is\_unsigned && \*value\_ll < 0) {

\*negative = true;

\*value\_ll = -\*value\_ll;

}

break;

case 'p':

case 'n':

\*value\_p = va\_arg(\*ap, void\*);

break;

case 'f':

case 'g':

case 'e':

\*value\_f = va\_arg(\*ap, double);

break;

case 's':

\*value\_s = va\_arg(\*ap, const char\*);

break;

case 'c':

\*value\_c = va\_arg(\*ap, int);

break;

default:

break;

}

}

Again, due to the number of conversion options, this is a longish function but pretty easy to follow. It takes the conversion format char (‘d’, ‘s’, etc.) and uses that to call va\_arg to retrieve an argument of the correct type. If the argument doesn’t actually match the type specified, all bets are off. Most modern compilers check for this type of error.

The result is written into one of the *value\_\** pointers depending on the type of the argument. For example, an integer argument is written into *\*value\_ll*, a string is written to *\*value\_s*, etc. The function also writes into two boolean variables: \**is\_unsigned* and \**negative*. If the integer is negative, it is negated so that the caller always sees a positive value.

After collecting the argument for the conversion, it then performs the conversion based on the specifier and writes the output using the *ConversionFormat* value.

## Converting The Value

Each type of argument has its own conversion function. The conversion functions write the ASCII characters into a buffer, return the address of the lowest converted character. This is because most conversions are done backwards. The conversion function called depends on the type of conversion needed (decimal, hex, etc.).

### Converting an Integer to Decimal

The algorithm for converting an integer to decimal is easier to understand if you first consider how to convert an integer to binary. The binary output for an integer is a series of the characters ‘0’ or ‘1’ with the rightmost character being the least significant bit. To do this conversion, we look at the rightmost bit of the number. If it’s a 1, we output the character ‘1’, otherwise we output ‘0’. We then shift the number to the right and continue until the number is 0.

Like this:

char\* ConvertBinary(unsigned long long v, char\* buf,

int buflen) {

char\* p = &buf[buflen - 1];

if (v == 0) {

\*p = '0';

return p;

}

while (v != 0) {

char ch = (v & 1) + '0';

\*p-- = ch;

v >>= 1;

}

// p is one less than the first char.

return p + 1;

}

As we are using the lowest order bit and shifting right, the output is generated backwards, therefore we start at the end of the buffer and move down in memory towards the start, returning the address of the last conversion done.

The bitwise AND and shift operations are exactly equivalent of a modulus with 2 and a division by 2 respectively. The expression **v & 1** takes the bottom bit, which is the same as **v % 2**. Likewise, the expression **v >>= 1** is equivalent to **v /= 2**.

Binary is base 2 and decimal is base 10, so we can use the same algorithm to convert to decimal instead of binary. This is simply done by replacing the modulus and division by 10 instead of 2. However, there’s a function provided by the C library that performs both the division and modulus in one operation instead of two[[5]](#footnote-5). This function is *lldiv* which returns the struct:

typedef struct {

long long int quot;

long long int rem;

} lldiv\_t;

Here’s the function to convert a *long long* into decimal, writing into the buffer, optimized to use the *lldiv* function.

static char\* ConvertDecimalLongLong(unsigned long long v, char\* buf,

int buflen) {

char\* p = &buf[buflen - 1];

if (v == 0) {

\*p = '0';

return p;

}

while (v != 0) {

lldiv\_t qr = lldiv(v, 10);

char ch = qr.rem + '0';

\*p-- = ch;

v = qr.quot;

}

// p is one less than the first char.

return p + 1;

}

### Converting an Integer to Hexadecimal

Conversion from integer to hex is a little easier than the decimal conversion. Each hexadecimal digit is 4 bits of the integer value. This gives 16 possible values, so we use the numbers 0…9 and then the letters A…F (in upper or lower case).

Like the conversion to decimal, we do it backwards, starting at the least significant digit and moving up until the value is zero. Each digit is 4 bits so the conversion to hex is very similar to the binary conversion we looked at in the previous section, but instead of masking and shifting with 1 bit, we do it with 4.

Here’s the function to convert a 64-bit integer into hexadecimal. It takes a boolean parameter to tell it whether to use lower or upper case for the letters.

static char\* ConvertHexLongLong(unsigned long long v, char\* buf, int buflen,

bool upper) {

char\* p = &buf[buflen - 1];

if (v == 0) {

\*p = '0';

return p;

}

while (v != 0) {

uint8\_t n = v & 0xf;

char ch;

if (n > 9) {

ch = n - 10 + (upper ? 'A' : 'a');

} else {

ch = n + '0';

}

\*p-- = ch;

v >>= 4;

}

return p + 1;

}

### Converting an integer to Octal

Whereas hexadecimal uses 4 bits per digit, octal uses just 3 and each digit is in the range 0…7. The conversion to octal is almost identical to the conversion to binary or to hex:

static char\* ConvertOctalLongLong(unsigned long long v, char\* buf, int buflen) {

char\* p = &buf[buflen - 1];

if (v == 0) {

\*p = '0';

return p;

}

while (v != 0) {

uint8\_t n = v & 0x7;

char ch = n + '0';

\*p-- = ch;

v >>= 3;

}

return p + 1;

}

### Converting a pointer to hex

The **%p** formatting command converts a pointer into hexadecimal with a prefix of “0x”. This is very similar to the conversion to hex, except that we always use lower case for the letters and we always write a “0x” at the start of the output. The standard says that the %p conversion is implementation defined so we can choose our output format. The form chosen here matches common implementations. If the pointer value is NULL, we write the string “(null”) instead of “0x0”.

static char\* ConvertHexPointer(void\* ptr, char\* buf, int buflen) {

char\* p;

if (ptr == NULL) {

// Write (null) (choice - it's up to the implementation what is printed.

p = &buf[buflen - 6];

strcpy(p, "(null)");

return p;

}

p = &buf[buflen - 1];

uintptr\_t v = (uintptr\_t)ptr;

while (v != 0) {

uint8\_t n = v & 0xf;

char ch;

if (n > 9) {

ch = n - 10 + 'a';

} else {

ch = n + '0';

}

\*p-- = ch;

v >>= 4;

}

\*p-- = 'x';

\*p = '0';

return p;

}

### Converting a floating point value

The conversion from double precision floating point into decimal is reasonably complex, and indeed, I covered this in the first **Coffee Into Bugs** article I wrote. It’s entitled “**Coffee Into Bugs: Floating point from Scratch”** and can be found at

<https://www.linkedin.com/posts/allisondave_coffee-into-bugs-floating-point-from-scratch-activity-6984288747764813824-mjVy?utm_source=share&utm_medium=member_desktop>

This implementation of *printf* uses the functions provided in the section “**Converting to ASCII Decimal**” on page 37. Please take a look at that article if you are interested in understanding how floating point works.

The functions \_\_*PrintFloatFormat*, \_\_*PrintScientificFormat* and \_\_*PrintGeneralFormat* are prefixed with a double underscore because they are in a different source file and the underscores prevent namespace pollution. For the sake of completeness, here are those functions, along with a function to print the %g for a fixed point number. I suggest going back and reading the article on floating point if you want to understand these functions.

// According to C99:

// Let P = precision

// Let X = the exponent calculated for the %e conversion.

// If P > X and X >= -4 then use %f with precision = P - (X + 1)

// Otherwise use %e with precision P - 1

// Trailing zeroes are removed and also the dot if it's trailing.

STATIC char\* PrintFixedPointGeneral(struct FloatPrinter\* printer, int precision, char\* buf, size\_t size) {

if (printer->naninf != 0) {

return WriteNanInf(printer->sign, printer->naninf, buf, size);

}

if (\_\_IsZero(printer->fx)) {

char\* p = buf + size - 1;

\*p-- = '\0';

\*p = '0';

return p;

}

// Need to work out the base10 exponent but we can't do that if we

// are going to use %f output.

int exp = CalculateExponent(printer->fx);

bool f\_format = false;

char\* p;

if (precision > exp && exp >= -4) {

f\_format = true;

p = PrintFixedPoint(printer, precision - (exp + 1), buf, size);

} else {

// We've already normalized the fixed point number so we can use that.

p = PrintFixedPointScientific(printer, precision - 1, exp, buf, size);

}

// The C99 spec says that we strip trailing zeroes and the decimal

// point. The string returned in p is terminated by a NUL.

if (f\_format) {

char\* s = buf + size - 2;

while (\*s == '0') {

\*s = '\0';

--s;

}

if (\*s == '.') {

\*s = '\0';

}

} else {

char\* e = buf + size - 5; // e+[X]XX

if (\*e != 'e') {

e--;

}

char\* s = e - 1; // Last digit in fraction.

// Remove trailing zeroes.

while (\*s == '0') {

\*s = '\0';

--s;

}

// Remove trailing point.

if (\*s == '.') {

\*s = '\0';

}

s++;

// Copy the exponent down.

if (s < e - 1) {

while (\*e != '\0') {

\*s++ = \*e++;

}

}

\*s = '\0';

}

return p;

}

char\* \_\_PrintFloatFormat(double f, int precision, char\* buf, size\_t size) {

struct FloatPrinter printer = {0};

FixFloat(&printer, f);

return PrintFixedPoint(&printer, precision, buf, size);

}

char\* \_\_PrintScientificFormat(double f, int precision, char\* buf, size\_t size) {

struct FloatPrinter printer = {0};;

FixFloat(&printer, f);

// Calculate the exponent.

int exp = CalculateExponent(printer.fx);

return PrintFixedPointScientific(&printer, precision, exp, buf, size);

}

// %g format.

char\* \_\_PrintGeneralFormat(double f, int precision, char\* buf, size\_t size) {

struct FloatPrinter printer = {0};

FixFloat(&printer, f);

return PrintFixedPointGeneral(&printer, precision, buf, size);

}

The floating point conversions place the data at the beginning of the buffer and won’t go beyond its end.

### Other conversions

There are a few other conversions that are pretty simple:

1. **%c**: write an integer out as an ASCII character
2. **%s**: assume the pointer value is a pointer to a character array and write out its contents
3. **%n**: write the number of characters written so far into the integer pointed to by the argument
4. %<anything else>: write the character out verbatim.

The **%s** conversion has to observe the precision specifier, which limits the number of characters printed in the string. The **%c** and **%s** specifiers both observe the field width formatting.

For the **%s** conversion, the precision causes truncation to occur if the string is longer than the precision value.

For the **%n** command, we write the number of characters we’ve written to the output (the current result of the call to printf) into the signed integral value pointed to by the argument. This is done by the *WriteCount* function defined as follows:

static void WriteCount(ConversionFormat\* fmt, int count, void\* p) {

switch (fmt->modifier) {

case kModLong:

\*(long\*)p = count;

break;

case kModLongLong:

\*(long long\*)p = count;

break;

case kModChar:

\*(char\*)p = count;

break;

case kModShort:

\*(short\*)p = count;

break;

case kModSize\_t:

\*(size\_t\*)p = count;

break;

case kModIntMax:

\*(intmax\_t\*)p = count;

break;

case kModPtrdiff\_t:

\*(ptrdiff\_t\*)p = count;

break;

case kModLongDouble:

\*(long double\*)p = count;

break;

default:

\*(int\*)p = count;

break;

}

}

The output modifiers can be used to change the type of the integer that will contain the value. By default, it’s an *int*. This is a weird addition to the formatting set, and I’ve read, makes *printf* a Turing Complete language. For a completely crazy and utterly amazing (ab)use of this feature, please check out one of the winners of the 27th International Obfuscated C Code Contest (2020):

<https://www.ioccc.org/2020/carlini/prog.c>

The **%n** is used like this:

int p;

int x = printf("%d %n\n", 1234, &p);

The result of this is **p** is set to the value 5 and **x** to 6. The characters output when the %n is encountered is “1234 “, which is 5 bytes long. The %n command results in no additional output being produced, so the total number of characters returned as the value of the *printf* call is 6 (“1234 \n”).

## Formatting The Output

Now we come to the heart of the formatting algorithm. We have seen how to generate a sequence of ASCII characters for the conversions, placed in a local buffer as necessary. Now we need to send them to the output formatted according to the *ConversionFormat*.

Each formatting command specifies:

1. An optional *field width*. Think of this as a column in a table that has a certain width in characters. The value being output can be either justified to the left or the right of this field. It is used to line up the columns on the output. Numbers can be padded to the left by a zero or a space.
2. An optional *precision*. The meaning of this varies according to the type being formatted. For a floating point value, it specifies the number of digits to be printed after the decimal point. For an integer, it specifies the minimum number of digits to be output. For a string, it specifies the maximum number of characters. The precision will never cause an integer to be truncated but will pad it to the left with zeroes[[6]](#footnote-6). For a string, the precision can cause it to be truncated.

The output formatter already knows the length of the value to be printed and, if a field width is specified, ensures that the value is padded to the left or the right with the appropriate padding character (space or zero).

The function to write a formatted value to the output is *WriteFormatted*, defined as follows:

static int WriteFormatted(Writer writer, void\* data, ConversionFormat\* fmt,

const char\* s, size\_t len, bool negative,

bool use\_precision) {

if (fmt->field\_width == kWidthDefault) {

Prepend(writer, data, fmt, negative);

int num\_chars = PadToPrecision(writer, data, fmt, len, use\_precision);

return num\_chars + writer(s, len, data);

}

// There is a field width, pad as necessary.

int num\_chars = 0;

int padding = fmt->field\_width - (int)len;

if (fmt->left\_justify) {

// Left justify. Pad to the right with spaces.

if (AnyPrependNeeded(fmt, negative)) {

Prepend(writer, data, fmt, negative);

--padding;

}

int n = PadToPrecision(writer, data, fmt, len, use\_precision);

num\_chars += n;

padding -= n;

num\_chars += writer(s, len, data);

num\_chars += Pad(writer, data, padding, false);

} else {

// Right justify. Pad to left with space or '0'. The sign is

// prepended to the left if padding with zeroes and after the

// padding if padding with spaces.

if (AnyPrependNeeded(fmt, negative)) {

if (fmt->fill\_zero) {

// Filling with zeroes, add prepend character now.

Prepend(writer, data, fmt, negative);

}

--padding; // One less padding character now.

}

if (use\_precision && fmt->precision != kWidthDefault) {

int diff = fmt->precision - (int)len;

if (diff > 0) {

padding -= diff;

}

}

num\_chars += Pad(writer, data, padding, fmt->fill\_zero);

if (!fmt->fill\_zero) {

Prepend(writer, data, fmt, negative);

}

num\_chars +=

PadToPrecision(writer, data, fmt, len, use\_precision);

num\_chars += writer(s, len, data);

}

return num\_chars;

}

The *use\_precision* flag is used to disable the use of the precision in the formatting. The floating point conversions have already dealt with it, so there’s nothing for the *WriteFormatted* function to do.

### No Field Width

The easiest path in the function is taken if there is no field width specified.

The first operation is to write out the prepend character using the *Prepend* function. If the number is negative, we write out a minus sign. If it’s positive we can write out either a + sign, a space or nothing. Here’s the *Prepend* function:

static void Prepend(Writer writer, void\* data, ConversionFormat\* fmt,

bool negative) {

if (negative) {

writer("-", 1, data);

return ;

}

if (fmt->prepend\_sign) {

writer("+", 1, data);

return ;

}

if (fmt->prepend\_space) {

writer(" ", 1, data);

return;

}

}

This uses the *ConversionFormat* value and a boolean flag to determine what character to write. The three options that result in a prepended character are:

1. The *negative* flag is set: write a minus sign.
2. The *prepend\_sign* flag is set in the *ConversionFormat* and the number is positive: write a plus sign.
3. The *prepend\_space* flag is set in the *ConversionFormat* and we haven’t prepended a minus or plus, write a space.

Even though there is no field width padding to be produced, we still might need to pad the value to the number of characters specified in the precision. Integers are padded to the left with zeroes if the precision is greater than the number of digits in the integer. We never truncate integers.

This precision padding is done by the function *PadToPrecision*, defined as follows. It returns the number of characters written to the output.

static int PadToPrecision(Writer writer, void\* data, ConversionFormat\* fmt,

size\_t len, bool enabled) {

if (!enabled) {

return 0;

}

if (fmt->precision == kWidthDefault) {

return 0;

}

int diff = fmt->precision - (int)len;

if (diff <= 0) {

return 0;

}

return Pad(writer, data, diff, true);

}

Only certain conversions use the precision in this way. Floating point precision is handled by the floating point conversion functions as they have to perform rounding. The *enabled* flag allows the precision padding to be switched off by the caller. The function is told the length of the value to print, and if this is less than the precision specified in *ConversionFormat*, it is padded to the left with zeroes.

This uses the *Pad* function to write out a number of padding characters. This is defined as:

static const char spaces[] = " ";

static const char zeroes[] = "00000000";

static int Pad(Writer writer, void\* data, int n, bool zero) {

const size\_t kBlockSize = sizeof(spaces) - 1;

const char\* pad = zero ? zeroes : spaces;

int count = 0;

while (n > 0) {

int len = n;

if (len > kBlockSize) {

len = kBlockSize;

}

writer(pad, len, data);

n -= len;

count += len;

}

return count;

}

A reasonably easy function, this writes out either spaces or zeroes to the output and returns the number of characters written. Rather than writing one character at a time, it tries to optimize this a little by bunching the padding characters in groups of 8.

After the precision padding is done, the *WriteFormatted* function writes the value to the output, returning the total number of characters written to the output.

### Field Width Specified

If the conversion format specified a field width, we have more work to do. We know the length of the value we want to print and if this is less than the field width, we print padding characters to fill out the field. If we are right justifying, we can use either a space or a zero as the padding character, but we never pad to the right with zeroes (that would look very odd and change the values of numbers significantly).

At this point, negative integers are held as a positive decimal string and a flag specifying whether it is negative or positive (a sign and value encoding). The *AnyPrependNeeded* function determines whether we need to add a prepended character. Here’s the function:

// Do we need to prepend a character?

static bool AnyPrependNeeded(ConversionFormat\* fmt,

bool negative) {

return negative || fmt->prepend\_sign || fmt->prepend\_space;

}

The actual prepend character is written using the *Prepend* function, as in the case of no field width.

Left justification is easier than right justification and is just a matter of writing:

1. The prepended character if necessary (minus, plus or space)
2. Padding with zero to the precision as necessary.
3. The value itself.
4. Spaces to pad to the field width.

We know the length of the value and the field width, so we can calculate how many padding characters are needed for the precision and the field width.

Right justification allows the padding to use spaces or zeroes and we need to be smart about where the prepended character is written. If we are padding with spaces we write the prepend character (think about the minus sign) after the padding characters, but if we are padding with zeroes we write it before the padding. Consider the following:

printf("%08d\n", -1234);

The output from this should be “-0001234” and not “000-1234”. For a space-padded call:

printf("%8d\n", -1234);

The output is “ -1234” and not “- 1234”.

The *WriteFormatted* function returns the number of characters sent to the output.

# Conclusion

This article has presented a possible implementation of the various *printf* functions provided by the standard C library. It is reasonably, but not fully complete and the hope is that you found it a useful description of how formatted output works in C.

Thanks for reading.

1. The header file also contains a *va\_end* macro which, to my knowledge, never did anything. [↑](#footnote-ref-1)
2. As mentioned earlier, on a 6502, pointers are 16-bit wide, as are, most likely, plain *ints*. [↑](#footnote-ref-2)
3. There is a *sprintf* function that takes a pointer without a length, but all modern code should use *snprintf*, which prevents buffer overflow errors. The *sprintf* function is considered hazardous. [↑](#footnote-ref-3)
4. The “alternate\_form” seems pretty useless to me and I’ve never seen it used in practice. We won’t support it and any use of # will be ignored, [↑](#footnote-ref-4)
5. On a small computer without hardware division (like the 6502) this makes a lot of difference since the algorithm for division automatically generates the remainder at the same time, allowing both to be calculated in one function call. [↑](#footnote-ref-5)
6. It would be really annoying if an integer was truncated, thus completely changing its value. For a floating point number, it only affects the fractional digits, so omitting trailing digits doesn’t change the value of the number much. [↑](#footnote-ref-6)