Chapter 32: Variable arguments

Parameter	Details
`va_list` ap	argument pointer, current position in the list of variadic arguments
last	name of last non-variadic function argument, so the compiler finds the correct place to start processing variadic arguments; may not be declared as a `register` variable, a function, or an array type
type	**promoted** type of the variadic argument to read (e.g. `int` for a `short int` argument)
`va_list` src	current argument pointer to copy
`va_list` dst	new argument list to be filled in

Variable arguments are used by functions in the printf family (`printf`, `fprintf`, etc) and others to allow a function to be called with a different number of arguments each time, hence the name *varargs*.

To implement functions using the variable arguments feature, use `#include <stdarg.h>`.

To call functions which take a variable number of arguments, ensure there is a full prototype with the trailing ellipsis in scope: `void err_exit(const char *format, ...);` for example.

Section 32.1: Using an explicit count argument to determine the length of the va_list

With any variadic function, the function must know how to interpret the variable arguments list. With the `printf()` or `scanf()` functions, the format string tells the function what to expect.

The simplest technique is to pass an explicit count of the other arguments (which are normally all the same type). This is demonstrated in the variadic function in the code below which calculates the sum of a series of integers, where there may be any number of integers but that count is specified as an argument prior to the variable argument list.

```
#include <stdio.h>
#include <stdarg.h>

/* first arg is the number of following int args to sum. */
int sum(int n, ...) {
    int sum = 0;
    va_list it; /* hold information about the variadic argument list. */

    va_start(it, n); /* start variadic argument processing */
    while (n--)
      sum += va_arg(it, int); /* get and sum the next variadic argument */
    va_end(it); /* end variadic argument processing */

    return sum;
}

int main(void)
{
    printf("%d\n", sum(5, 1, 2, 3, 4, 5)); /* prints 15 */
    printf("%d\n", sum(10, 5, 9, 2, 5, 111, 6666, 42, 1, 43, -6218)); /* prints 666 */
    return 0;
}
```

Section 32.2: Using terminator values to determine the end of va_list

With any variadic function, the function must know how to interpret the variable arguments list. The “traditional” approach (exemplified by `printf`) is to specify number of arguments up front. However, this is not always a good idea:

```
/* First argument specifies the number of parameters; the remainder are also int */
extern int sum(int n, ...);

/* But it's far from obvious from the code. */
sum(5, 2, 1, 4, 3, 6)

/* What happens if i.e. one argument is removed later on? */
sum(5, 2, 1, 3, 6)  /* Disaster */
```

Sometimes it's more robust to add an explicit terminator, exemplified by the POSIX `execlp()` function. Here's another function to calculate the sum of a series of `double` numbers:

```
#include <stdarg.h>
#include <stdio.h>
#include <math.h>

/* Sums args up until the terminator NAN */
double sum (double x, ...) {
  double sum = 0;
  va_list va;

  va_start(va, x);
  for (; !isnan(x); x = va_arg(va, double)) {
    sum += x;
  }
  va_end(va);

  return sum;
}

int main (void) {
  printf("%g\n", sum(5., 2., 1., 4., 3., 6., NAN));
  printf("%g\n", sum(1, 0.5, 0.25, 0.125, 0.0625, 0.03125, NAN));
}
```

Good terminator values:

- integer (supposed to be all positive or non-negative) — `0` or `-1`
- floating point types — `NAN`
- pointer types — `NULL`
- enumerator types — some special value

Section 32.3: Implementing functions with a `printf()`-like interface

One common use of variable-length argument lists is to implement functions that are a thin wrapper around the `printf()` family of functions. One such example is a set of error reporting functions.

errmsg.h

```
#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h>    // C11

void verrmsg(int errnum, const char *fmt, va_list ap);
noreturn void errmsg(int exitcode, int errnum, const char *fmt, ...);
void warnmsg(int errnum, const char *fmt, ...);

#endif
```

This is a bare-bones example; such packages can be much elaborate. Normally, programmers will use either `errmsg()` or `warnmsg()`, which themselves use `verrmsg()` internally. If someone comes up with a need to do more, though, then the exposed `verrmsg()` function will be useful. You could avoid exposing it until you have a need for it (YAGNI — you aren't gonna need it), but the need will arise eventually (you *are* gonna need it — YAGNI).

`errmsg.c`

This code only needs to forward the variadic arguments to the `vfprintf()` function for outputting to standard error. It also reports the system error message corresponding to the system error number (`errno`) passed to the functions.

```
#include "errmsg.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void
verrmsg(int errnum, const char *fmt, va_list ap)
{
    if (fmt)
        vfprintf(stderr, fmt, ap);
    if (errnum != 0)
        fprintf(stderr, ": %s", strerror(errnum));
    putc('\n', stderr);
}

void
errmsg(int exitcode, int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
    exit(exitcode);
}

void
warnmsg(int errnum, const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    verrmsg(errnum, fmt, ap);
    va_end(ap);
}
```

Using `errmsg.h`

Now you can use those functions as follows:

```
#include "errmsg.h"
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    char buffer[BUFSIZ];
    int fd;
    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    const char *filename = argv[1];

    if ((fd = open(filename, O_RDONLY)) == -1)
        errmsg(EXIT_FAILURE, errno, "cannot open %s", filename);
    if (read(fd, buffer, sizeof(buffer)) != sizeof(buffer))
        errmsg(EXIT_FAILURE, errno, "cannot read %zu bytes from %s", sizeof(buffer), filename);
    if (close(fd) == -1)
        warnmsg(errno, "cannot close %s", filename);
    /* continue the program */
    return 0;
}
```

If either the `open()` or `read()` system calls fails, the error is written to standard error and the program exits with exit code 1. If the `close()` system call fails, the error is merely printed as a warning message, and the program continues.

Checking the correct use of `printf()` formats

If you are using GCC (the GNU C Compiler, which is part of the GNU Compiler Collection), or using Clang, then you can have the compiler check that the arguments you pass to the error message functions match what `printf()` expects. Since not all compilers support the extension, it needs to be compiled conditionally, which is a little bit fiddly. However, the protection it gives is worth the effort.

First, we need to know how to detect that the compiler is GCC or Clang emulating GCC. The answer is that GCC defines `__GNUC__` to indicate that.

See common function attributes for information about the attributes — specifically the `format` attribute.

Rewritten `errmsg.h`

```
#ifndef ERRMSG_H_INCLUDED
#define ERRMSG_H_INCLUDED

#include <stdarg.h>
#include <stdnoreturn.h>    // C11

#if !defined(PRINTFLIKE)
#if defined(__GNUC__)
#define PRINTFLIKE(n,m) __attribute__((format(printf,n,m)))
#else
#define PRINTFLIKE(n,m) /* If only */
```

```
#endif /* __GNUC__ */
#endif /* PRINTFLIKE */

void verrmsg(int errnum, const char *fmt, va_list ap);
void noreturn errmsg(int exitcode, int errnum, const char *fmt, ...)
        PRINTFLIKE(3, 4);
void warnmsg(int errnum, const char *fmt, ...)
        PRINTFLIKE(2, 3);

#endif
```

Now, if you make a mistake like:

```
errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
```

(where the `%d` should be `%s`), then the compiler will complain:

```
$ gcc -O3 -g -std=c11 -Wall -Wextra -Werror -Wmissing-prototypes -Wstrict-prototypes \
>     -Wold-style-definition -c erruse.c
erruse.c: In function ‘main’:
erruse.c:20:64: error: format ‘%d’ expects argument of type ‘int’, but argument 4 has type ‘const char
*’ [-Werror=format=]
         errmsg(EXIT_FAILURE, errno, "Failed to open file '%d' for reading", filename);
                                                               ~^
                                                               %s
cc1: all warnings being treated as errors
$
```

Section 32.4: Using a format string

Using a format string provides information about the expected number and type of the subsequent variadic arguments in such a way as to avoid the need for an explicit count argument or a terminator value.

The example below shows a a function that wraps the standard `printf()` function, only allowing for the use of variadic arguments of the type `char`, `int` and `double` (in decimal floating point format). Here, like with `printf()`, the first argument to the wrapping function is the format string. As the format string is parsed the function is able to determine if there is another variadic argument expected and what it's type should be.

```
#include <stdio.h>
#include <stdarg.h>

int simple_printf(const char *format, ...)
{
    va_list ap; /* hold information about the variadic argument list. */
    int printed = 0; /* count of printed characters */

    va_start(ap, format); /* start variadic argument processing */

    while (*format != '\0') /* read format string until string terminator */
    {
        int f = 0;

        if (*format == '%')
        {
            ++format;
            switch(*format)
            {
                case 'c' :
```

```
                    f = printf("%d", va_arg(ap, int)); /* print next variadic argument, note type
promotion from char to int */
                    break;
                case 'd' :
                    f = printf("%d", va_arg(ap, int)); /* print next variadic argument */
                    break;

                case 'f' :
                    f = printf("%f", va_arg(ap, double)); /* print next variadic argument */
                    break;
                default :
                    f = -1; /* invalid format specifier */
                    break;
            }
        }
        else
        {
            f = printf("%c", *format); /* print any other characters */
        }

        if (f < 0) /* check for errors */
        {
            printed = f;
            break;
        }
        else
        {
            printed += f;
        }
        ++format; /* move on to next character in string */
    }

    va_end(ap); /* end variadic argument processing */

    return printed;
}

int main (int argc, char *argv[])
{
    int x = 40;
    int y = 0;

    y = simple_printf("There are %d characters in this sentence", x);
    simple_printf("\n%d were printed\n", y);
}
```