varargs

Mentioned elsewhere, how Apple and Linux handle variadic functions are quite different. In this chapter we will explore how variadic functions work by looking at an example in C++. Then, we'll review the differences between Apple and Linux.

How Variadic Functions Determine Number of Parameters

The TL;DR is that something among the *REQUIRED* arguments tells the function how many other arguments to expect. Commonly, it is the very first argument that contains this information but it is possible to encode this information in a parameter other than the first as long as no optional parameter precedes it.

For example, take our friend and yours: `printf`. The first argument is a template or formatting string that may contain zero or more instances of placeholders. For example:

Sample	Expected Arguments
`printf("Foo");`	None
`printf("%%d");`	None - notice the double %
`printf("%d %ld");`	An int and a long
`printf("%p %f");`	A pointer and a double

The `printf` function figures out how many arguments to expect by counting the number of naked `%` characters and what types to expect for each based on the letters coming after the `%`. These are found in the string pointed to by the first parameter.

Writing Variadic Functions in C or C++

You'll need to include `cstdarg` in C++ or `stdarg.h` in C++ or C.

Then, you'll use the **macros** `va_start()` and `va_end()` plus a variable of type `va_list`.

- `va_start()` is used to initialize the `va_list` variable with the address of the last required argument. This enables the variadic support functions to find the variadic parameters on the stack or in registers.
- `va_arg()`, another macro fetches the next variadic argument. The type of the argument is given as a parameter to `va_arg()`.
- `va_end()` is used to clean up the internal data structure allocated for you, behind your back.

Take a look at `Simple()` in this source code.

For a more sophisticated example, look at `LessSimple()` in the same file for something closer to `printf()`.

The above referenced source code contains ample commenting to explain what it is doing.

Differences Between Apple and Linux

Apple and Linux differ in how they implement parameter passing to variadic functions in assembly language. This is explained in more detail in the chapter on Apple Silicon.

Linux

Linux uses the first 8 scratch registers as normal. If you need to specify more than 8 arguments, the ninth and beyond go on the stack. It is up to you to determine how many arguments to expect (which you must do anyway) to determine where to find them.

Apple

Apple puts the first argument in `x0` as usual but all remaining arguments go on the stack in right to left order. There are some other restrictions - it is best to refer to this cryptically written document.

Here is our attempt at explaining variadics on the M series.

First, let's demonstrate what order values go on the stack using `stp` and `str`.

```
/*  A program to demonstrate the order in which registers are
    pushed onto the stack by stp and str. See text.
*/
        .text
        .p2align    2
        .global     main

main:   stp         xzr, x30, [sp, -16]!
        mov         x0, 0xF
        str         x0, [sp, -16]!
        ldp         xzr, x30, [sp], 16
        ret

        .end
```

Here is a `gdb` session showing the execution of the above:

```
(gdb) b main
Breakpoint 1, main () at stack_order.S:8
```

```
8   main:   stp         xzr, x30, [sp, -16]!
(gdb) s
9           mov         x0, 0xF
(gdb) s
10          str         x0, [sp, -16]!
(gdb) s
main () at stack_order.S:11
11          ldp         xzr, x30, [sp], 16
(gdb) x/4xg $sp
0xfffffffef40: 0x000000000000000f  0x0000fffff7e273c0
0xfffffffef50: 0x0000000000000000  0x0000fffff7e273fc
(gdb)
```

Traditionally, diagrams of memory place 0 at the top and higher addresses down below. Hence, the saying that "stack grows upwards."

In a diagram the results of `stp xzr, x30, [sp, -16]!` looks like this:

address	value
smaller address	0
larger address	x30

These can be found in the `gdb` session at the end beginning with `0xfffffffef50`. The second argument to `stp` goes on the stack first. The first argument to `stp` goes on the stack second.

For the `str` instruction, the lone argument goes on the stack second (at the smaller address). Using `str x0, [sp, -16]!`, you see the value in x0 (set to 0xF) appearing at the lower address. Surmises can be made about the value of the "second" register pushed but you should not succumb to the temptation to do so.

The C and C++ compilers on the Mac M series will put the very first argument into the zero register as per usual. All arguments past the first will go onto the stack from right to left, each argument consuming 8 bytes.

Combining the information provided here, you can put together how to use `varargs` in assembly language. One last comment: it might be easier to use `str` rather than `stp` to control the order going onto the stack but remember that there must be a net change in the stack pointer which is a multiple of 16.