Section 1 / Calling and Returning From Functions

Calling functions, passing parameters to them and receiving back return values is basic to using `C` and and `C++`.

Calling *methods* (which are functions connected to objects) is similar but with enough differences to warrant its own discussion to be provided later in the chapter on structs.

Be sure to read this for information about passing parameters to functions.

Bottom Line Concept

The name of a (non-inline) function is a label to which a branch with link ('bl') can be made.

The `bl` instruction is stands for **B**ranch with **L**ink. The **link** concept is what enables a function (or method) to **return** to the instruction after the call.

A Trivial Function

In `C`, here is a trivial function:

```
void func() {
}
```

The function `func()` takes no parameters, does nothing and returns nothing.

Here it is in assembly language:

```
func: ret
```

Notice that `func` is a label. The only instruction in the function is `ret`. Strictly speaking, the assembly language function might more explicitly look like this in `C`:

```
void func() {
    return;
}
```

To call this function in `C` you would do this:

```
func();
```

This would be done this way in assembly language:

```
bl func
```

Notice that calling a function **is** a branch. But it is a special branch instruction - *branch-with-link*. Again, it is the *link* that allows the function to `ret`urn.

bl

Branch-with-link computes the address of the instruction following it.

It places this address into register `x30` and then branches to the label provided. It makes one link of breadcrumbs to follow to get back following a `ret`.

This is why it is absolutely essential to backup `x30` inside your functions if they call other functions themselves.

How about this trivial program:

```
        .text                                           // 1
        .global main                                    // 2
        .align  2                                       // 3
                                                        // 4
main:   ldr     x0, =hw                                 // 5
        bl      puts                                    // 6
        ret                                             // 7
                                                        // 8
        .data                                           // 9
hw:     .asciz  "Hello World!"                          // 10
                                                        // 11
        .end                                            // 12
```

What could possibly go wrong?

Here is a listing from `gdb` since running the program hangs:

```
gdb) b main
Breakpoint 1 at 0x798: file not_backing_up_x30.s, line 5.
(gdb) run
Starting program: /media/psf/Home/asm_book/section_1/funcs/a.out

Breakpoint 1, main () at not_backing_up_x30.s:5
5   main:   ldr     x0, =hw
(gdb) n
6           bl      puts
(gdb) n
Hello World!
7           ret
(gdb) n
^C
Program received signal SIGINT, Interrupt.
main () at not_backing_up_x30.s:7
7           ret
```

The program hung and had to be killed with ^C. Why?

Think about it...

Somebody called `main()` - it's a function and someone called it with a `bl` instruction. At the moment `main()` entered, the address to which it needed to return was sitting in `x30`.

Then, `main()` called a function - in this case `puts()` but which function doesn't matter - it called a function. In doing so, it overwrote the address to which `main()` needed to return with the address of line 7 in the code. That is where `puts()` needs to return.

So, when line 7 executes it puts the contents of `x30` into the program counter and branches to it.

And the problem with this is?

Hint: notice where `gdb` put us after the control-C. Still on line 7. An infinite loop of returning to the return statement.

Here is a fixed version of the code:

```
        .text                                   // 1
        .global main                            // 2
        .align  2                               // 3
                                                // 4
main:   str     x30, [sp, -16]!                 // 5
        ldr     x0, =hw                         // 6
        bl      puts                            // 7
        ldr     x30, [sp], 16                   // 8
        ret                                     // 9
                                                // 10
        .data                                   // 11
hw:     .asciz  "Hello World!"                  // 12
                                                // 13
        .end                                    // 14
```

The address to which `main()` should return is pushed onto the stack on line 5. It should be safe there.

It is recovered from the stack on line 8 and used by line 9's `ret`.

Returning Values

First, let's take a trip back in time to the early days of C.

Stephen Bourne was writing `sh`, the first shell for Unix. He noticed that every function had to return a value - even functions that had no reason to return a value.

In these early days, `void` functions did not yet exist.

Bourne argued that an instruction could be saved per function if the concept of `void` functions were added to C. Saving one instruction per function was really

valuable - so that's how we get `void` functions that return no value.

What about functions that do return a value?

In the AARCH64 Linux style calling convention, values are returned in `x0` and sometimes also returned in `x1` though this is uncommon.

Note that `x0` and `x1` could also be `w0` and `w1` or even the first and second floating point registers if the function is returning a `float` or `double`.

Here are samples, first in C / C++ then in the corresponding assembly language:

```
                                                    // 1
int ReturnsAnInt() {                                // 2
    return 16;                                      // 3
}                                                   // 4
                                                    // 5
long ReturnsALong() {                               // 6
    return 16;                                      // 7
}                                                   // 8
                                                    // 9
float ReturnsAFloat() {                             // 10
    return 16.0f;                                   // 11
}                                                   // 12
                                                    // 13
double ReturnsADouble() {                           // 14
    return 16.0;                                    // 15
}                                                   // 16
```

Here it is in assembly language:

```
ReturnsAnInt:                                       // 1
    mov    w0, 16                                   // 2
    ret                                             // 3
                                                    // 4
ReturnsALong:                                       // 5
    mov    x0, 16                                   // 6
    ret                                             // 7
                                                    // 8
ReturnsAFloat:                                      // 9
    fmov    s0, 16                                  // 10
    ret                                             // 11
                                                    // 12
ReturnsADouble:                                     // 13
    fmov    d0, 16                                  // 14
    ret                                             // 15
```

Note, the use of the floating point move instruction as well as the single precision and double precision registers.

Inline functions

Functions that are declared as *inline* don't actually make function calls. Instead, the code from the function is type checked and inserted directly where the "call" is made after adjusting for parameter names.

Repeating the TL;DR

If your functions call *any* other functions, `x30` must be backed up on the stack and then restored into `x30` before returning.