Section 1 / `switch` and Jump Tables

If section 1 is about bridging your knowledge of C and C++ backwards to assembly language, how the heck do jump tables fit in?

Jump tables are one key way in which switch statements work. In this chapter we'll explore three ways in which `switch` statements are very very clever.

Naively, you might imagine all `switch` statements are implemented as long chains of `if / else` constructions. This is the case often, for small `switch` statements or where the `case` values have no or little pattern with few if any values being consecutive.

The long chain of `if / else` isn't covered here. Instead see the section on if statements.

When the C++ optimizer is enabled, it will look at your cases and choose between three different constructs for implementing your `switch`.

1. It may emit a long string of `if / else` constructs.
2. It may find the right `case` using a *binary search*.
3. Finally, it might use a **jump table**.

And, it can use any combination of the above! Compiler writers are smart!

Jump Tables

Suppose our cases are largely consecutive. Given that all branch instructions are the same length in bytes, we can do math on the switch variable to somehow derive the address of the case we want.

For example, take the following C / C++ code:

```
# include <stdlib.h>
# include <stdio.h>
# include <time.h>

int main()
{
    int r;

    srand(time(0));
    r = rand() & 7;
    switch (r)
    {
        case 0:
            puts("0 returned");
            break;

```

```c
        case 1:
            puts("1 returned");
            break;

        case 2:
            puts("2 returned");
            break;

        case 3:
            puts("3 returned");
            break;

        case 4:
            puts("4 returned");
            break;

        case 5:
            puts("5 returned");
            break;

        case 6:
            puts("6 returned");
            break;

        case 7:
            puts("7 returned");
            break;
    }
    return 0;
}
```

When run, the program will calculate a random number from 0 to 7. Then, using this value, it will enter a `switch` statement with cases for values 0 through 7. The appropriate `case` will be executed.

Notice that the `case` values are all, in this case, consecutive.

Why bother going through the sequential search of chained `if / else` statements when we can gain direct access to the case we want?

What about this block of code?

```
jt:     b       0f
        b       1f
        b       2f
        b       3f
        b       4f
        b       5f
```

```
        b       6f
        b       7f
```

At address `jt` there are a sequence of branch statements... jumps if you will. Being in a sequence, this is an example of a jump table. We'll compute the index into this *array of instructions* and then branch to it.

AARCH64 makes it easy for us since all instructions are the same length, 4 bytes. Suppose our random number were 3. We'd calculate 3 time 4 yielding 12. At 12 bytes from label `jt` we'll find the fourth branch in the table. If we branch to that address, we'll land on this instruction: `b 3f` which in turn jumps us to the case for the value of 3.

Let's examine this code assuming that our number between 0 and 7 inclusive is already in `x0`:

```
lsl     x0, x0, 2                   // 1
ldr     x1, =jt                     // 2
add     x1, x1, x0                  // 3
br      x1                          // 4
```

Line 1 multiplies our number by 4 by shifting it left by 2 bits. Shifting is a fast way of multiplying by powers of 2. We're doing this because each branch instruction in the jump table is exactly 4 bytes long.

Line 2 loads the base address of the "instruction array" starting at address `jt`.

Line 3 adds the two values together putting the result in `x1`. This register now contains the address of one of the branch instructions found at label `jt`.

Line 4 stands for `branch using register`. It loads the program counter with the value found in `x1`.

We land on one of the unconditional branches which immediately causes us to land on the code for the `case` we want.

Here is a complete program demonstrating this.

The program also hints at a further optimization that works with this code only because the length of the code for each case is the same. The hinted at optimization would NOT work if the code in each case were different lengths.

How to implement falling through?

If there is no `break` following the code for a `case`, control will simply fall through to the next `case`.

Here is a snippet from the program linked just above.

```
0:      ldr     x0, =ZR                 // 1
        bl      puts                    // 2
        b       99f                     // 3
```

```
                                        // 4
1:      ldr     x0, =ON                 // 5
        bl      puts                    // 6
        b       99f                     // 7
```

If we wanted case 0 to fall through into case 1, simply remove line 3. Then, landing at the 0 case, we execute lines 1 and 2 and happily continue on to the next case.

How about implementing gaps?

In our example, we present 8 consecutive cases. What if there was no code for case 4? In other words, what if case 4 simply didn't exist?

Thinking naively, this would seem to screw up our nice little approach we have going on. Does this doom us to a chain of `if / else`?

Nope.

When we're using a jump table that has gaps here and there, just implement stubs for the missing cases. Here's an example… let's model this strategy with a missing case 4.

```
2:      ldr     x0, =TW
        bl      puts
        b       99f

3:      ldr     x0, =TH
        bl      puts
        b       99f

4:      b       99f

5:      ldr     x0, =FV
        bl      puts
        b       99f
```

Our jump table remains the same.

More strategies for implementing `switch`

As indicated above, an optimizer has at least three tools available to it to implement complex `switch` statements. And, it can combine these tools.

For example, suppose your cases boil down to two ranges of fairly consecutive values. For example, you have cases 0 to 9 and also cases 50 to 59. You can implement this as two jump tables with an `if / else` to select which one you use.

Suppose you have a large `switch` statement with widely ranging `case` values. In this case, you can implement a binary search to narrow down to a small range in which another technique becomes viable to narrow down to a single `case`.

You might have need to implement hierarchical jump tables, for example.

This sounds complicated but it isn't given some thought.

The bottom line

With some thought you can avoid long chains of `if / else`.

If you DO use a long chain of `if / else`

If you do choose to implement a long chain of `if / else` statements, consider how frequently a given case might be chosen. Put the most common cases at the top of the `if / else` sequence.

This is known as making the common case fast.

Making the common case fast is one of the Great Ideas in Computer Science. One, you would do well to remember no matter what language you're working with.