1 Declaring uninitialized data

Let's look at a program:

```
section .bss
        num RESB 3

section .text
        global _start
        _start:
                MOV bl, 1
                MOV [num], bl
```

- **RESB** → means reserve bytes.
- **RESW** → means reserve word.
- **RESD** → means reserve double words.
- **RESQ** → means reserve quadwords.

Now, why did we not just do: **MOV [num], 1** ? Beacause x86 doesn't know how big **num** is. That's why we need to store a byte in(in this case 1) into a register first so as to provide that setting of having some definite size so that x86 can perform the **MOV** operation on **num**. Basically **num** is actually a byte array that can store 3 bytes. So as we know(from C/C++) in an array, **the array name points to the starting address of the array**. Similarly, **num** points to the first memory location.

Instructions like MOV depend on the size of the operands to know how many bytes to transfer. So, here in the instruction **MOV [num], bl** we are basically moving the one byte stored in **bl** to the byte memory slot of **num**.

As mentioned earlier, **num** points to the first memory location, if we want to get to the second memory location we will write: **[num+1]** and if we want to get to the third memory location we will write **[num+2]**. Now, this offset depends on the type of data we are working with. Here we are working with bytes. If we were working with words, then to get to the second memory location then we will write: **[num+2]**(since a word is 2 bytes in size) and to get to the third memory location we will write: **[num+4]**.

Let's start debugging the program.

Layout

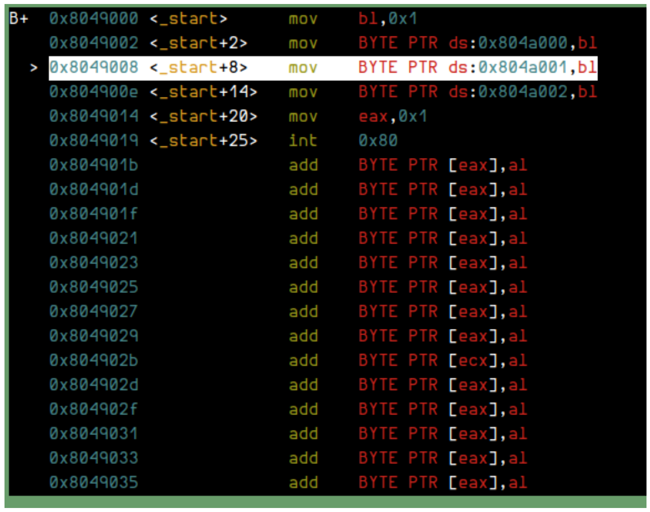

Figure 1: Layout

What has changed in the layout? → The instructions are written in intel assembly language syntax. Generally, the instructions are written in AT&T syntax. To use this layout in gdb, we need to run the following command:

```
$ echo "set disassembly-flavor intel" > ~/.gdbinit
```

Basically, here we are configuring GDB to use intel syntax. To stop the intel layout, we can just open the **.gdbinit** file and remove the command(**"set disassembly-flavor intel"** from there.

So, we can see from the figure below that initially **0x804a000** contained 0(the addresses **0x804a000**, **0x804a001** and **0x804a002** are addresses of the three memory blocks that were reserved for **num**).

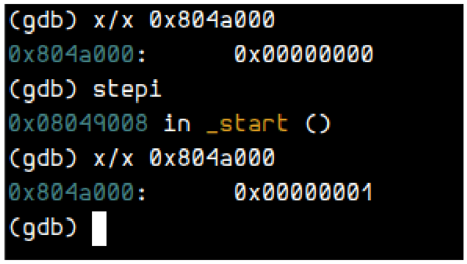

Figure 2: Value in **0x804a000**

After we ran the **stepi** command, the instruction **MOV [num], bl** got executed and then the value **0x01** was stored in the address **0x804a000**.

So, as we go on executing the next instruction the value start getting stored into the memory blocks.

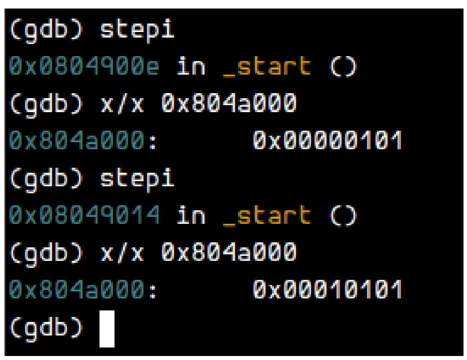

Figure 3: Values getting stored

The two values(hex) that we see in the image above(i.e **0x01** and the last **0x01**(1^{st} from the left)) get stored in the memory blocks pointed to by the addresses **0x804a001** and **0x804a002** respectively.

2 Can we do the same in the data section?

In the data section, we can only define initialized data.

```
section .data
        num DB 3 DUP(2)
```

This basically defines **num** as an array of 3 bytes where each byte stores the value 2. **DUP** here means "duplicate". That means 2 is duplicated three times.

2.1 Important

An assembly program:

```
section .bss
        num RESB 3

section .data
        num2 DB 3 DUP(2)

section .text
        global _start
        _start:
                MOV bl, 1
                MOV bl, [num2]
                MOV [num], bl
                MOV [num+1], bl
                MOV [num+2], bl

                MOV eax, 1
                INT 80h
```

The addresses in the image are 4 bytes apart.

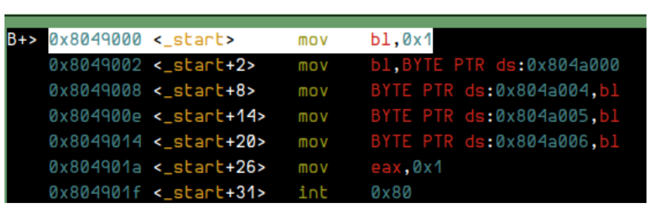

Figure 4: The addresses

Now, **num2** is points to the first memory location which is **0x804a000**(since it's a byte array).

Also, **num** is also a byte array which points to the first memory location which is **0x904a004**.

Now, we allocated only 3 bytes to **num2** then why is the address of the byte array **num** starting from **0x804a004**? It should have started from **0x804a003**.

Two reasons:

- **Memory alignment**
  - Variables are aligned to addresses that are multiples of their size.
  - **num** is 3 bytes long so it will be aligned to a 4-byte boundary i.e the smallest multiple of 4 that is greater than or equal to 3.
- **Section**
  - **num** and **num2** are placed in different sections.

For example:

```
section .bss
        num DB 3

section .data
        num2 DB 25 DUP(2)

section .text
        global _start
        _start:
                MOV bl, 1
                MOV bl, [num2]
                MOV [num], bl
                MOV [num+1], bl
                MOV [num+2], bl

                MOV eax, 1
                INT 80h
```

Now, we have initialized 25 bytes for **num2** and **num** is an array of 3 bytes.

Starting address of **num2** is **0x804a000**. We reserved 25 bytes. So, as per memory alignment we need to align to 4-byte boundary. Now, the smallest multiple of 4 that is greater than or equal to 25 is 28(4×7). Note that **num** is also aligned since it is only 3 bytes long. $(28)_{10} = (0x1c)_{16}$. So **0x804a000** and the starting address of **num** will be 28 bytes apart.

$$(0x804a000)_{16} = (134520832)_{10}$$

Now we will add 134520832 with 28, this gives us 134520860.

$$(134520860)_{10} = (804a01c)_{16}$$

And that is what we see when we run gdb.

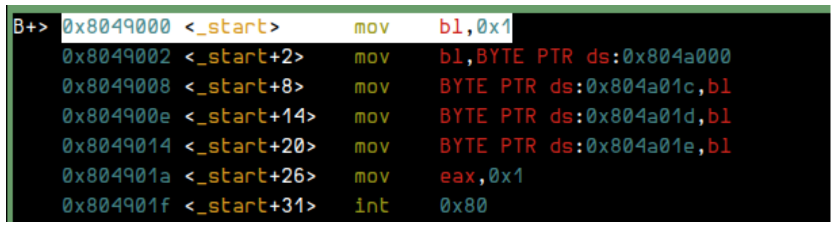

Figure 5: The addresses 28 bytes apart