

Ben Reads Assembly for Pleasure (totally)



Ben's 'Waterfall Method' of Reading Assembly

General Guidelines:

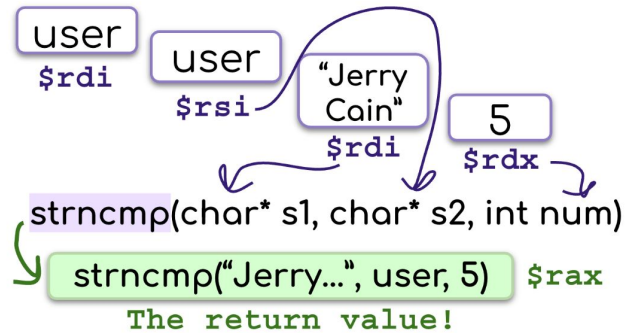
(may have to adapt to specific assembly task!)

- ❑ Write each register value (or corresponding variable) in a separate, labeled box.
- ❑ Assembly is poetry, flowing top to bottom – for each line, write a new box lower than the previous one and indented to the right, with the updated value (don't cross out old boxes!).
- ❑ Treat function calls as a stanza break – write each function call explicitly, and draw arrows from the registers being passed in. Write a box for return `$rax` after, and reset indentation.

```

🐻 is_not_poohbear(user) :
<+0> mov $rdi, $rsi
<+4> mov $0x106a106b, $rdi
<+8> mov $0x5, $rdx
<+12> call strncmp
<+16> ret
    
```

Small toy example! Assume `$0x106a106b` corresponds to the string "Jerry Cain".



Tips for Assembly & C Reverse-Engineering

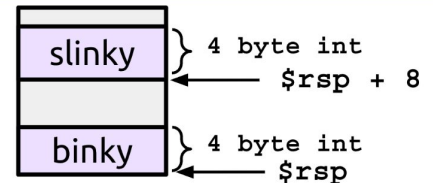


Try to understand what the assembly is doing certainly, but be more concerned with mapping your understanding to **equivalent C code**, and not necessarily the original.

Look for registers that alternate between being updated and being read (**think very active, lively registers**), because they typically correspond to key **local variables**.

"If it looks like a `char*`, swims like a `char*`, and quacks like a `char*`, it's probably a `char*`"
 👉 Assembly operates directly on raw data/bits, but **context can help in determining type!**

📖 **Keep a record / sketch of the stack frame.** And when portions of it are written to or updated (e.g., a 4-byte sliver at address `$rsp` to store an integer variable there), **update your sketch accordingly!**



🧩 **Decomposition!** If you encounter a long af stream of assembly, use function calls to **segment the instructions into smaller, more manageable sections**. And when you see a loop 🌀, first aim to divide it into 3 or so parts: a **pre-loop, within-loop, and post-loop**.

Additional Tips: Loops

✏ Includes Jerry's own advice!

You can usually ignore all push and pop operations when reading, as they're almost always boilerplate in place to save & ultimately restore caller-owner register values.

When you suspect there's an **if test** or an **if/else test**:

- Code branches off, so look for where/how it branches, i.e. any **conditional jumps** and **unconditional jumps** to higher addresses
- Use that to demarcate what is part of the **if test**, the **if body**, and **potentially the body of a corresponding else branch**.

```

check condition
Jump to [IfBody] if condition true
[Else]:
<If false statements>
Jump to [EndIf]
[IfBody]:
<If true statements>
[EndIf]
    
```

🏠 **More on for/while loops:** When you see one, **decompose it!**

- Look for the **conditional jump forward** and the **unconditional jump backwards** (can be the reverse! i.e. conditional jump back)
- Above case: **backwards jump keeps us in loop, forward jump exits**
- Use this to demarcate which instructions are **within the body of the loop**, which ones precede it, and **which ones come after it**.

```

[Initialize] (e.g., int i = 0)
[Test]:
Check OPPOSITE of loop condition
Jump to [LoopEnd] if true
[LoopBody]:
<statements>
<Update> (e.g., i++)
Jump to [Test]
[LoopEnd]:
everything else
    
```

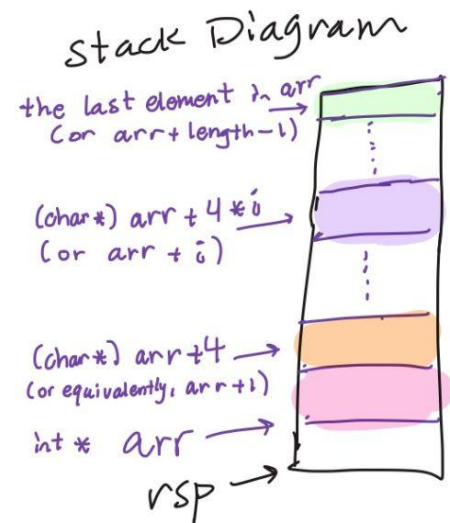
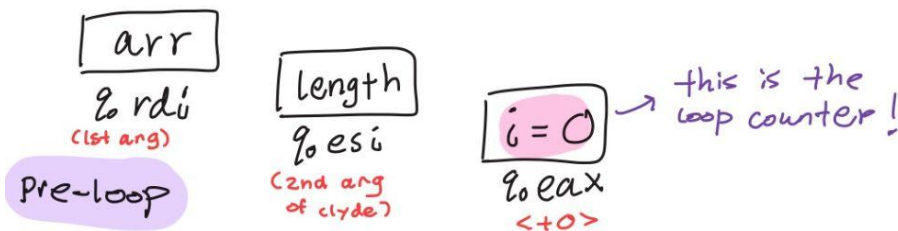


Sample waterfall trace on a loop that increments each element in an array by 1. Each box is labeled with its corresponding line of assembly, e.g., `<+0>`, and note pointer arithmetic in assembly works in terms of bytes offsets, hence the `(char*)` casts.

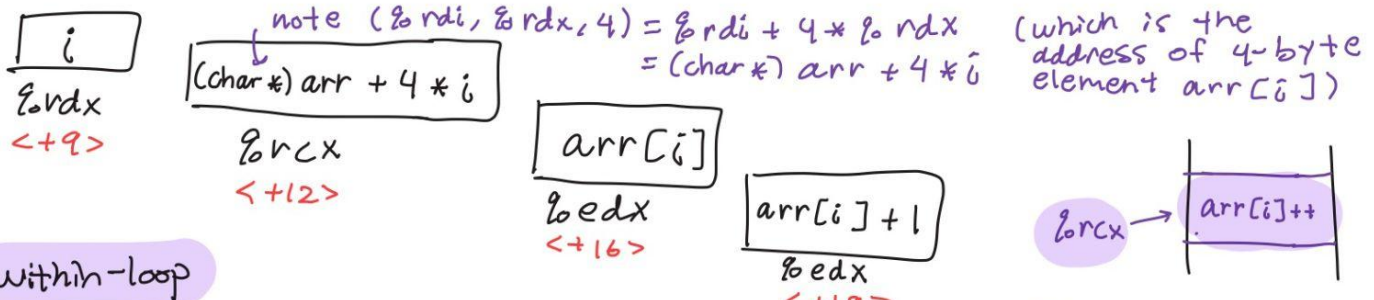
```

<+0>:  mov    $0x0,%eax
<+5>:  cmp     %esi,%eax           // task: fill in the blanks! :)
<+7>:  jge     0x401154 <clyde+28> int clyde(int arr[], int length) {
<+9>:  movslq  %eax,%rdx           for (int i = 0; i < length; i++) {
<+12>: lea     (%rdi,%rdx,4),%rcx   arr[i]++;
<+16>: mov     (%rcx),%edx
<+18>: add     $0x1,%edx
<+21>: mov     %edx, (%rcx)
<+23>: add     $0x1,%eax
<+26>: jmp     0x40113d <clyde+5>    }
<+28>: movslq  %esi,%rsi
<+31>: mov     -0x4(%rdi,%rsi,4),%eax
<+35>: ret

```



<+5> Compare `%esi` (`length`) with `%eax` (`i`) and with **<+7>** `jge`, if `i >= length`, jump to , otherwise if `i < length`, proceed from here.



<+21> The next line moves `%edx`, or `arr[i] + 1`, into `%rcx`, or `&arr[i]`

(increments loop counter!) `i + 1` $\rightarrow$ `%eax` `<+23>`

<+26> now, jump back to loop check 

