



Assembly: More GDB Commands!



(gdb) layout split is a neat way to view both the assembly + GDB panel!

display: Reprints something automatically every time you step, e.g., **(gdb) display \$rax** will print out the value in **\$rax** each time we move forward

► **Conditional breakpoints:** Pauses only if a specific condition is met

- > **(gdb) break [function or instruction #] if (condition)**
- > **Example:** Can help for **loops** if you want to stop only on the last iteration, not every!

Watchpoints: Special and nifty breakpoint — breaks only when variable, expression, or memory location value changes

- > **Example:** **(gdb) watch *(int *) (\$rsp + 0x4)**
- > This monitors an **int** variable stored at address **\$rsp + 0x4** for any changes!

🍀 I've personally found **conditional breakpoints / watchpoints** a life-saver on **Assign5/6!**

Exercise: GDB Printing of String Arrays



Premise: In some program far far away, we know that **%rbx** holds an address for an **array of strings** (i.e. is a pointer to the first element, which is a **char***).

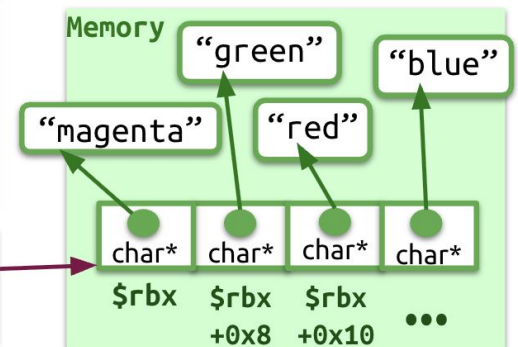
Question: How could we **print in GDB** to see the **index-2 element** in this array?

Solution: As **%rbx** is a pointer to a **char***, we need to **cast it to a char**** data type before we can index / dereference.

```
(gdb) print ((char **) $rbx) [2]
```

Extra: To print out the entire array in GDB, we can write:

char** **rbx** **CPU**



```
(gdb) print ((char **) $rbx) [0] @ number of elements
```

Registers: With great power comes great responsibility



Many registers are given designated roles or responsibilities, e.g., for function calls:

- ☐ The first 6 args passed in are **\$rdi, \$rsi, \$rdx, \$rcx, \$r8, \$r9**, rest are on stack.
- ☐ The return value of a function (if any) is written to **\$rax** before it ends

Register	Conventional use	Low 32-bits	Low 16-bits	Low 8-bits
%rax	Return value, callee-owned	%eax	%ax	%al
%rdi	1st argument, callee-owned	%edi	%di	%dil
%rsi	2nd argument, callee-owned	%esi	%si	%sil
%rdx	3rd argument, callee-owned	%edx	%dx	%dl
%rcx	4th argument, callee-owned	%ecx	%cx	%cl
%r8	5th argument, callee-owned	%r8d	%r8w	%r8b
%r9	6th argument, callee-owned	%r9d	%r9w	%r9b
%r10	Scratch/temporary, callee-owned	%r10d	%r10w	%r10b
%r11	Scratch/temporary, callee-owned	%r11d	%r11w	%r11b
%rsp	Stack pointer, caller-owned	%esp	%sp	%spl

💖 *Full list can be found on course's x86-64 guide!*

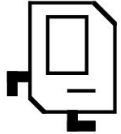
Advice: When you see a function you know (e.g., **strcmp**), check:

- > How many arguments, and so the relevant registers
- > Which data type for each to detect what's being passed in :)

Also, function
calls in assembly!

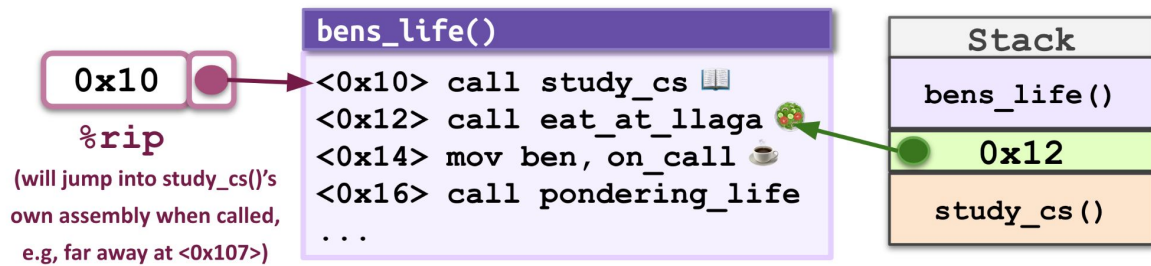


Special Register: Instruction Pointer `%rip`



`%rip` stores the **address of the next instruction** to execute:

- If we jump to another instruction or call a new function, `%rip` is updated
- When we call / jump into another function, **we must save the caller's next instruction** to execute **so we can resume there** when the callee finishes
- `call` instruction does this for us automatically by **storing it on the stack**, and the `ret` instruction pops the  value / breadcrumb off into `%rip`



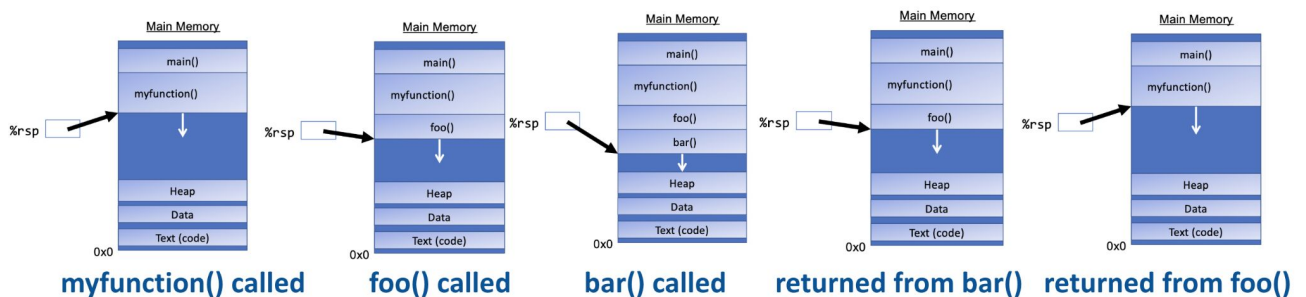
“When Ben is done studying CS, his life resumes at Llaga dining”

Special Register: Stack Pointer `%rsp`



`%rsp` stores the **address of the current top of the stack** (note: visually,  the stack grows downward). Instructions can adjust `%rsp` by:

- **Explicitly incrementing / decrementing** it to make more or less space
- **Pushing or popping** things on/off the stack using the `push` and `pop` instructions, or `call`, which pushes the return address onto stack



Buffer overflow vulnerability with C library function `gets()`



Our `greet()` function relies on `gets(char* buf)` to read an input string from the command line, and store it in a C string `buf`.



`gets()` also adds a `\0` at the end, which the buffer should accommodate.

```
[bbyan@myth65:~/CS107_TA/assignments/lab6$ ./smash
You've got 5 tries to blow this popsicle stand. G
What is your name? Mclovin
Hello, Mclovin! Buffer has space for 16 chars, yo
```

```
Mclovin\0 ...
char buf[16];
```

However, we don't know how many characters the user will enter, and `gets()` will try writing all of them in memory, even if it overflows `buf`!

Thus, `gets()` is very dangerous security-wise, so use `fgets()` instead :) 