CS 161 x86/C/GDB Cheat Sheet

Number Representation

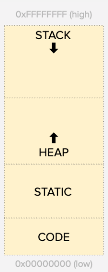
To begin, recall that **1 byte** is equal to **8 bits,** and **1 word** is equal to **4 bytes (32 bits)**. In this class, we do everything in a 32-bit address space. Thus, every address can be written using 1 word of memory.

One hexadecimal digit has sixteen possible values (0-9, A-F). To add/subtract addresses in hex, just use Python.

```
>>> hex(0x10 + 0x11)      // '0x21'
```

C Memory Layout

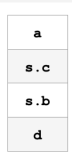
1. **Code**: contains machine code (x86 instructions).
2. **Static**: contains static variables & constants declared outside functions.
3. **Heap**: contains space for variables allocated using "malloc" - starts at low addresses & grows up.
4. **Stack**: contains function frames & local variables. Starts at high addresses and grows down.



Stack Diagram Reminders

1. We draw our diagrams such that the top is higher addresses, and the bottom is lower addresses.
2. Each row is typically one word.
3. The size of addresses and integers is one word.
4. When we declare **buffers**, we allocate one or more rows. When we write into them, we start at the lowest address (bottom-left) and work our way right and up (towards higher addresses).
5. Local variables declared later are at lower memory addresses, except for members of a **struct** (see example to right).

```
int a;
struct {
  int b;
  int c;
} s;
int d;
```



x86 Registers

EBP = "base pointer" (top of current stack frame)
ESP = "stack pointer" (bottom of current stack frame)
EIP = "instruction pointer" (current x86 instruction)

x86 Calling Convention

In C, the CPU executes a series of steps before and after every function call. This procedure is referred to as "calling convention," and the steps are as follows.

Remember, **push** refers to adding something to the stack, and **pop** refers to removing something.

1. Push arguments onto the stack (reverse order).
2. Push the old EIP onto the stack. This value becomes the **RIP (Return Instruction Pointer)**.
3. Move EIP to the first instruction of the function.
4. Push the old EBP onto the stack. This value becomes the **SFP (Saved Frame Pointer)**.
5. Move the EBP down to ESP to start a new frame.
6. Move the ESP down to allocate space for local vars.
7. Execute the function.
8. Move the ESP up to the EBP.
9. Pop the **SFP** and move the EBP to that value.
10. Pop the **RIP** and move the EIP to that value.
11. Remove arguments from the stack.

These steps can also be written using x86 notation and summarized into function prologues and epilogues.

Function Prologue (Steps 2-4)

```
push %ebp       // save previous frame
mov %esp %ebp   // start new frame
sub $X %esp     // move ESP down by X
```

Function Epilogue (Steps 8-10)

```
add $X %esp     // move ESP up by X
pop %ebp        // pop SFP and store in EBP
ret             // pop RIP and go there
```

GDB Tutorial & Shortcuts

On the CS 161 Project 1 VM, we use `./debug-exploit` to open GDB with our program. Here are a few helpful commands we can use to debug.

```
layout split     // show code
r                // run program
b [LINE | FN]    // break at line/function
n                // continue to next line (step over)
s                // continue to next line (step in)
c                // continue to next breakpoint
finish           // continue to end of function
p [VAR]          // print the value of a variable
p &[VAR]         // print the address of a variable
x/nwx [VAR]      // print "n" words of memory starting
                    at VAR in hex.
x/nwx [VAR]      // print "n" words of memory starting
                    at VAR in hex.
info registers   // display current ESP/EBP/EIP
info frame       // display location of SFP/RIP
refresh          // re-render the screen
```

We often use "x/..." to help us compare stack diagrams to the actual stack. For example, "x/16wx buf" will display 16 words of memory starting at buf.

Whereas "n" and "s" both continue to the next line, "n" steps over function calls while "s" steps into functions.

If we want to step out of a function that we've stepped into, we can use "finish" to do so.

To find the **address** of the SFP and RIP, we can use "info frame" and look specifically at the "Saved Registers" section for the Saved EBP and EIP, respectively. To find the **value** of the RIP, look at the "Saved EIP" section.

To count the space between a buffer and the RIP:

1. Identify the address of the RIP.
2. Identify the address of the buffer.
3. Subtract the two addresses.

"Info Frame" in GDB

The "B" indicates that we have a breakpoint set at this line number. The arrow ">" points to the next line that the CPU will execute.

If things look a little wonky for whatever reason (e.g. text is out of place, or you resized the GDB window), use the **refresh** command to re-render the screen!

Terminal — ssh -t cs161-tao@hive10.cs.berkeley.edu ~cs161/proj1/start — 8...

```
   ┌─orbit.c──────────────────────────────────────────────────────────────┐
   │2                                                                      │
   │3       void orbit()                                                   │
   │4       {                                                              │
   │5         char buf[8];                                                 │
B+>│6         gets(buf);                                                   │
   │7       }                                                              │
   │8                                                                      │
   │9       int main()                                                     │
   ┌──────────────────────────────────────────────────────────────────────┐
   │0xb7ffc4a5 <orbit>      push   %ebp                                    │
   │0xb7ffc4a6 <orbit+1>    mov    %esp,%ebp                               │
   │0xb7ffc4a8 <orbit+3>    sub    $0x18,%esp                              │
B+>│0xb7ffc4ab <orbit+6>    sub    $0xc,%esp                               │
   │0xb7ffc4ae <orbit+9>    lea    -0x10(%ebp),%eax                        │
   │0xb7ffc4b1 <orbit+12>   push   %eax                                    │
   │0xb7ffc4b2 <orbit+13>   call   0xb7ffc75e <gets>                       │
   │0xb7ffc4b7 <orbit+18>   add    $0x10,%esp                              │
   │0xb7ffc4ba <orbit+21>   nop                                            │
   └──────────────────────────────────────────────────────────────────────┘
native process 2075 In: orbit                             L6    PC: 0xb7ffc4ab
(gdb) info frame
Stack level 0, frame at 0xbffff860:
 eip = 0xb7ffc4ab in orbit (orbit.c:6); saved eip = 0xb7ffc4d3
 called by frame at 0xbffff880
 source language c.
 Arglist at 0xbffff858, args:
 Locals at 0xbffff858, Previous frame's sp is 0xbffff860
 Saved registers:
  ebp at 0xbffff858, eip at 0xbffff85c
(gdb)
```

The addresses that you see here are the addresses that these instructions reside at. Remember, this is in the **code** section of memory.

This is the **instruction pointer (EIP)** - it points to the address of the next instruction to be executed.

These values indicate the **addresses** of the SFP and RIP, respectively. We often use the address of the RIP to do our buffer overflow math.

This is the **value** of the **RIP of the current frame**. When this function completes, the CPU will jump to the instruction at stored at this address.