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.

```python
>>> hex(0x10 + 0x11)
'A-F'). To add/subtract addresses in hex, just use Python.
```

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. **Code**
2. **Static**
3. **Heap**
4. **Stack**

Each row is typically one word.

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.

```python
>>> hex(0x10 + 0x11)
```

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. **Code**
2. **Static**
3. **Heap**
4. **Stack**

Each row is typically one word.

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.

```python
>>> hex(0x10 + 0x11)
```

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. **Code**
2. **Static**
3. **Heap**
4. **Stack**

Each row is typically one word.

---

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)**

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

**Function Epilogue (Steps 8-10)**

```assembly
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/32x [VAR]**: print the value of a variable
- **x/32wx [VAR]**: print "w" words of memory starting at VAR in hex.
- **info registers**: display current ESP/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.
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.

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.

The "value of the RIP of the current frame" indicates that when this function completes, the CPU will jump to the instruction at the address stored at this address.

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!

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

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

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!