6.1903

The Stack, The Heap, Other Things, Review

Fall 2026

Administrative

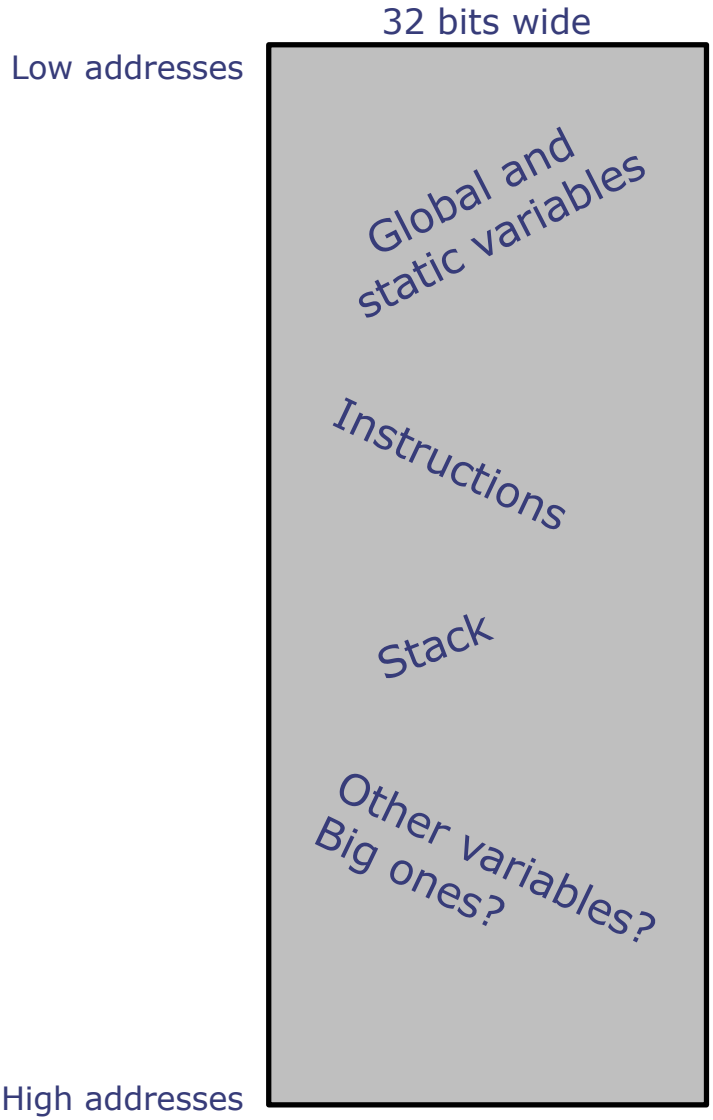
- Recitation 6 is tomorrow!
- Exercise 6 this week!
- No lab this week.
- Next Monday (11/16): Third and final quiz
- Wednesday (11/18): Lab 6 (final lab).
- Then...Exam 2 on Thursday December 3
- Then done (make sure to return lab kits after)

Review

- We now know that everything lives in memory.
  - Variables
  - Instructions
  - *Every element* of our code
  - We've seen this both in C and with our adventures into Assembly

Memory

- Memory is a large chunk of storage that stores stuff
- We use it to store everything
- ....not exactly the whole story...how else can information be stored??? You've seen it!!!



Registers *and* Memory

- A good program will utilize both forms of information storage

Registers:

Very few of them

(32 of them in RV32I)

Memory words:

Very Many of them

100's of thousands in our ESP32 but literally can have billions of them in a computer

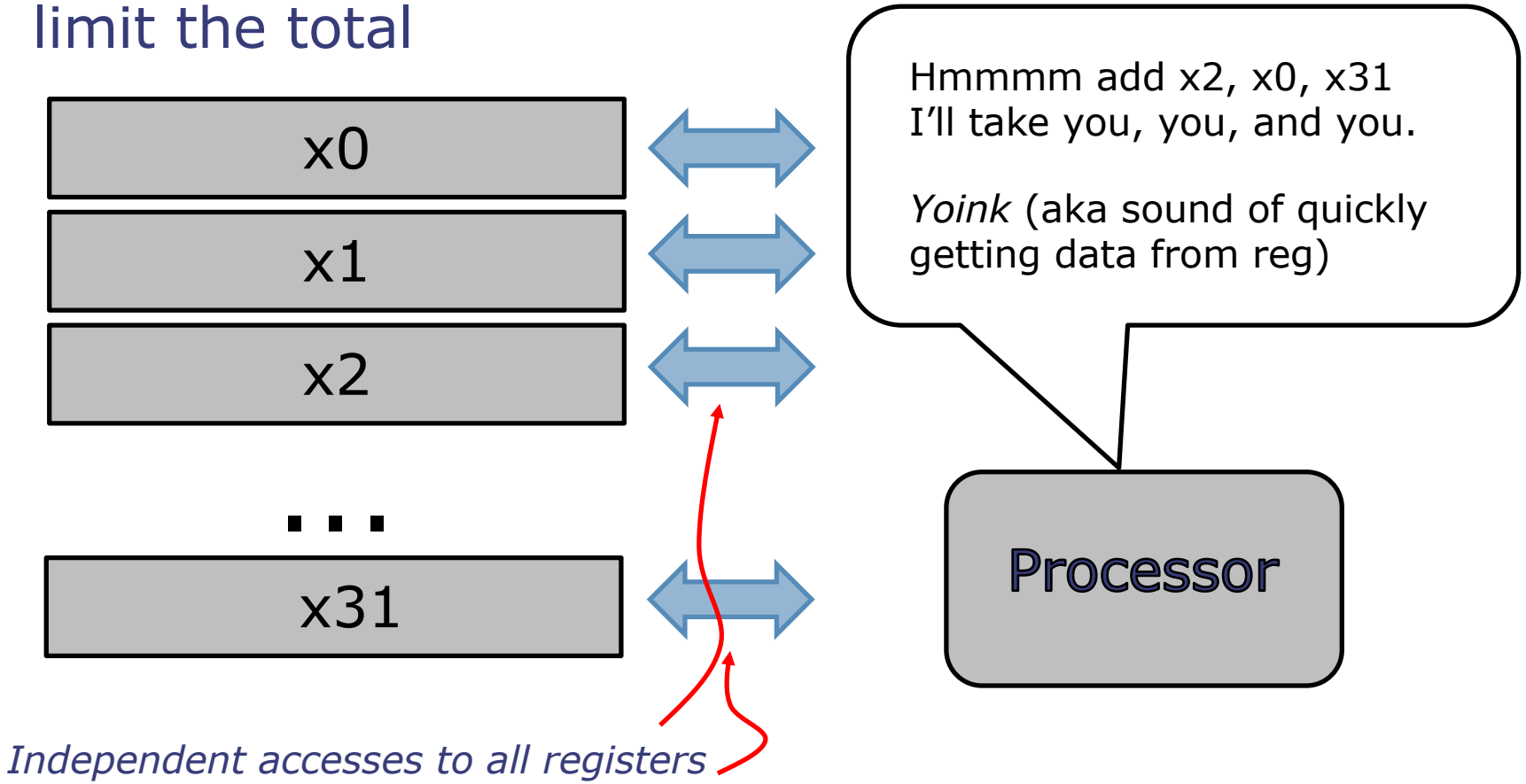
Why So Few Registers and So Many Memory?

- Registers are physically very large and are placed right next to (aka in) the CPU so they can be quickly accessed. There's only so much room in the CPU!

- Memory cells are physically much smaller:
  - SRAM (for cache) might be 10-20 times physically smaller than a register
  - DRAM (for general memory) might be over 100 times physically smaller than a register

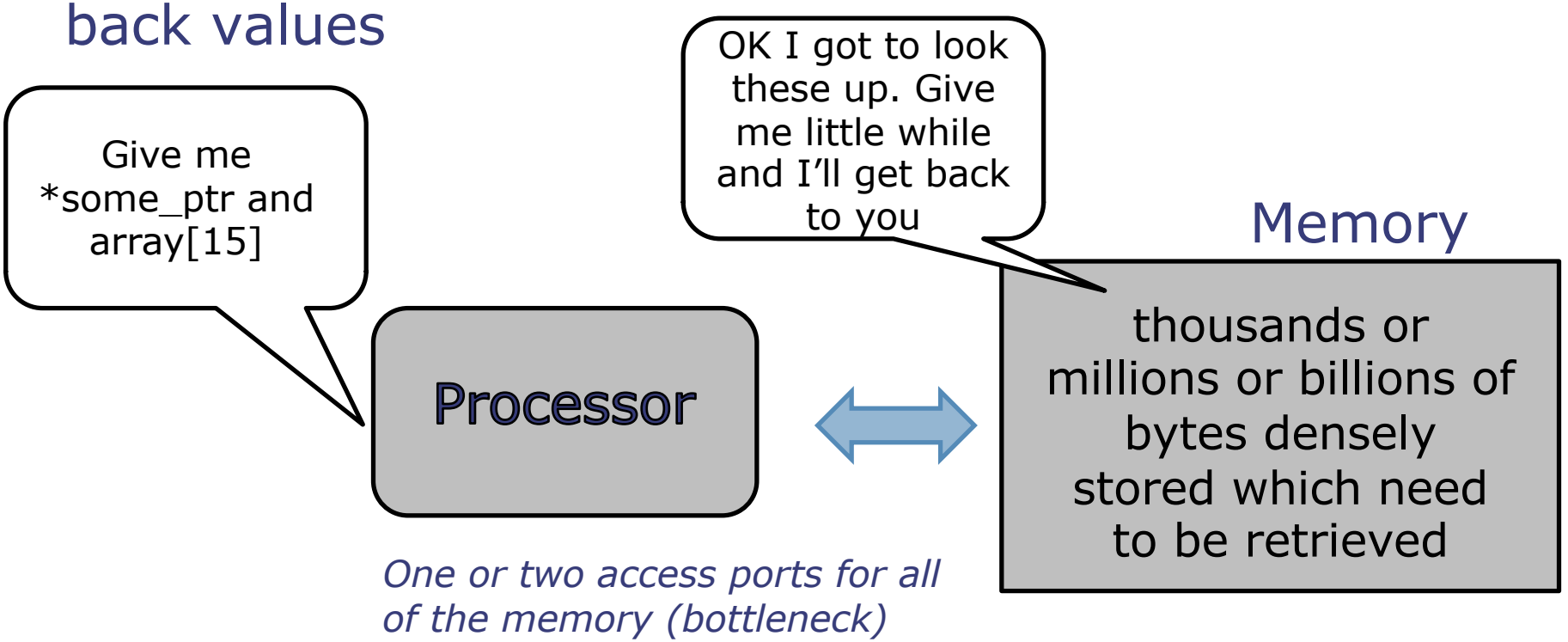
Registers can be accessed whenever you want

- Registers are each easily accessible at any time, but this **takes up a lot of space** so you have to limit the total



Memory is packed tight to get a lot of it into a small space

- UPSIDE: Lots of Memory
- DOWNSIDE: It is packed so tight in fact that you can't access the bytes easily.
- You have to send in commands to request and then get back values



Registers *and* Memory

Registers:

Very fast...can get stuff out of them and put stuff back in a single instruction

Memory words:

Can take from the equivalent of several instructions...up to hundreds of instructions to get into/out of memory

This means...

Code Like This:

```
mv a1, a0
mv a0, t0
addi t1, t2, -4
```

Will run fast because it uses only registers and immediates!

Code Like This:

```
sw s0, 8(sp)
lw s2, 24(sp)
```

Might run fast or might run slower...depends where in memory you're trying to get and other things...

Registers *and* Memory

Registers:

Very fast...can get stuff out of them and put stuff back in a single instruction

Memory words:

Can take the equivalent of ***several instructions...up to hundreds of instructions*** to get into/out of memory



DRAM is Particularly Egregious

- When people say their computer has "16 GB of RAM" they're referring to their computer's memory and in modern systems that memory is comprised of DRAM.
- Crazy high densities (100 billion bits of storage in a few teeny tiny chips)
- But.....

IS43/46TR16640C, IS43/46TR16640CL
IS43/46TR81280C, IS43/46TR81280CL

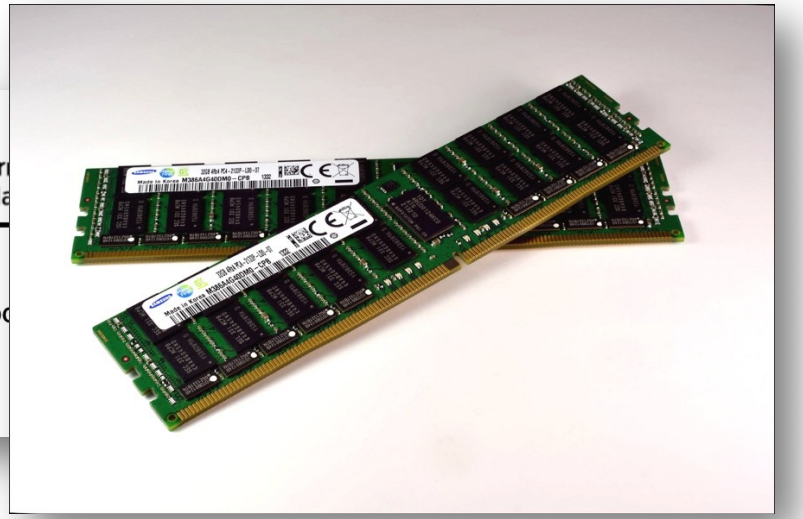
ISSI Long-ter
World Cla

128MX8, 64MX16 1Gb DDR3 SDRAM

OC

FEATURES

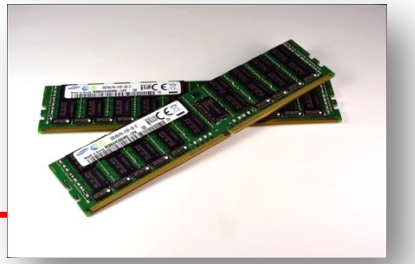
- Standard Voltage: V_{DD} and V_{DDQ} = 1.5V ± 0.075V
- Low Voltage (L): V_{DD} and V_{DDQ} = 1.35V + 0.1V, -0.067V
  - Backward compatible to 1.5V
- Partial Array Self Refresh
- Asynchronous RESET pin



The individual Bits in DRAM are SO TINY

- How tiny are they?
- …
- They’re stored using tiny capacitors that have values of a few femtoFarads.
- This is so tiny that they leak from material imperfections and can’t hold their binary value (1 or 0) for more than ~100 ms at room temperature.
- Also they’re so tiny and use so few electrons to store the bit (like 100,000 electrons) that we can’t read them without destroying the bit in the process...

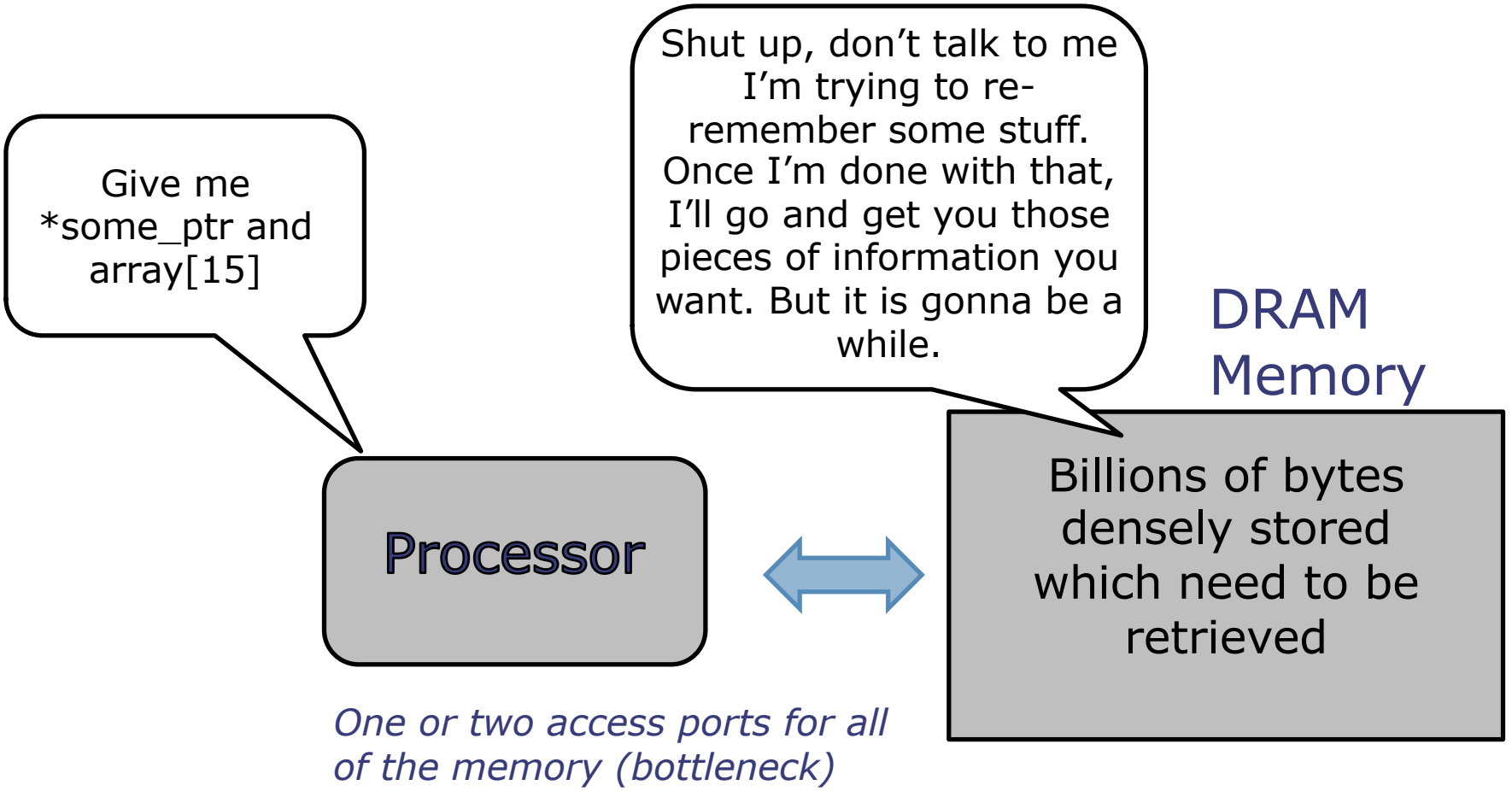
The Burden of DRAM



- Every single bit of the billions stored in DRAM is periodically read and rewritten about every 64 ms on average at room temperature. Call this a "refresh"
- Seriously. The chip can't remember for more than 100 ms so it keeps re-remembering quickly enough that it can overcome forgetting in the long-term.
- This keeps it quite busy.
- And this is a huge problem!

DRAM is Dense and Always Doing Something

- Memory accesses can be highly variable!



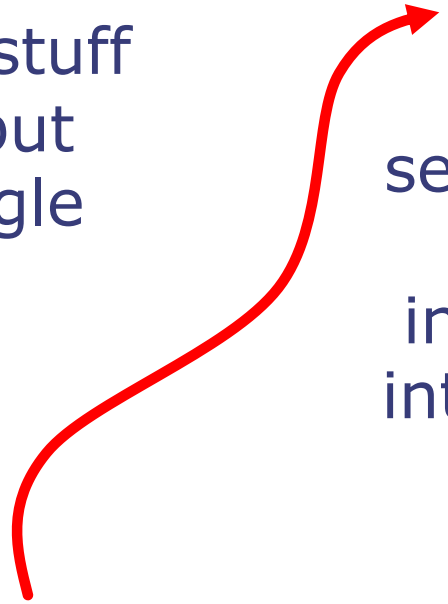
Registers *and* Memory

Registers:

Very fast...can get stuff out of them and put stuff back in a single instruction

Memory words:

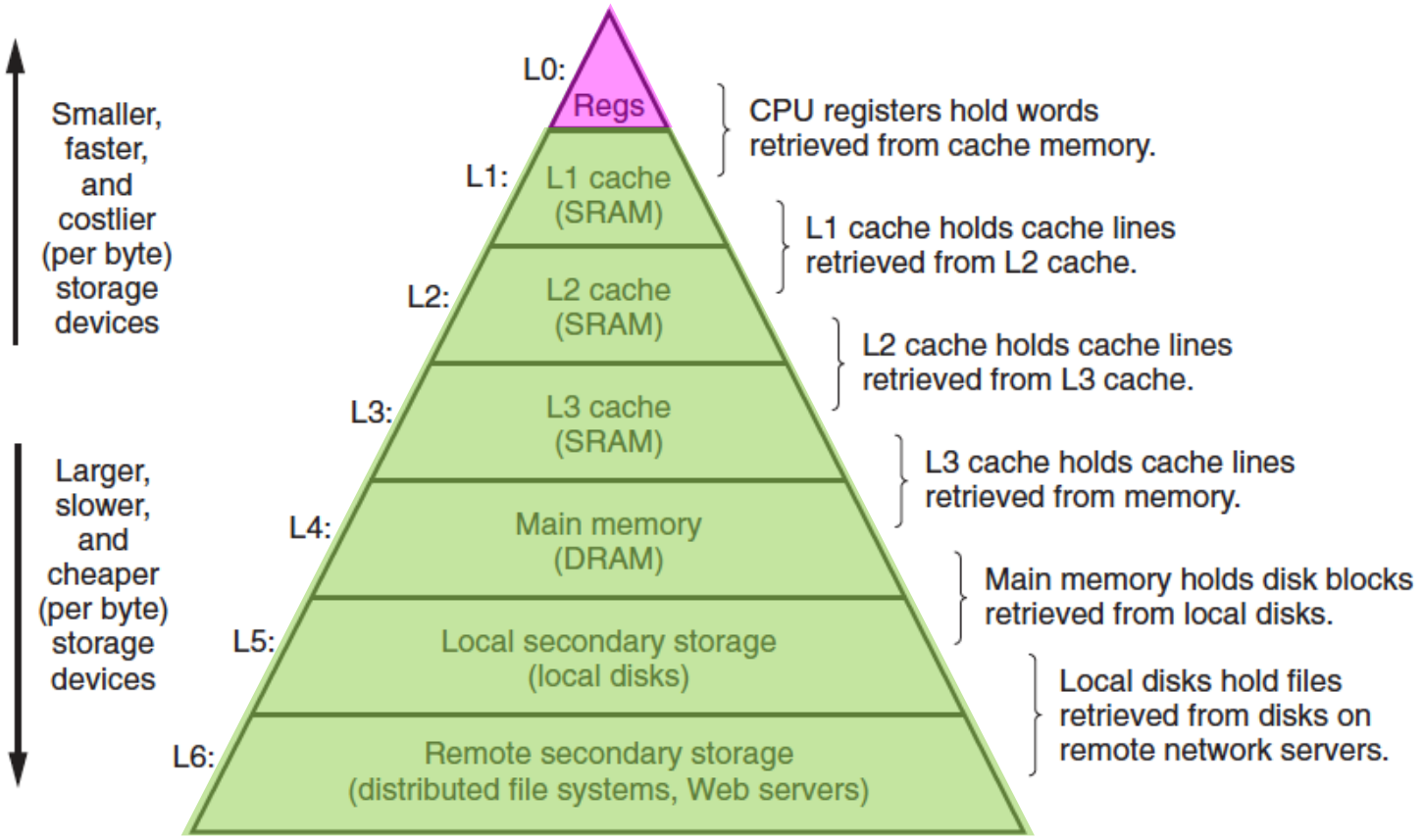
Can take the equivalent of several instructions to hundreds of instructions to get into/out of memory



A lot of different types of memory are used in an effort to balance characteristics

There's a whole hierarchy

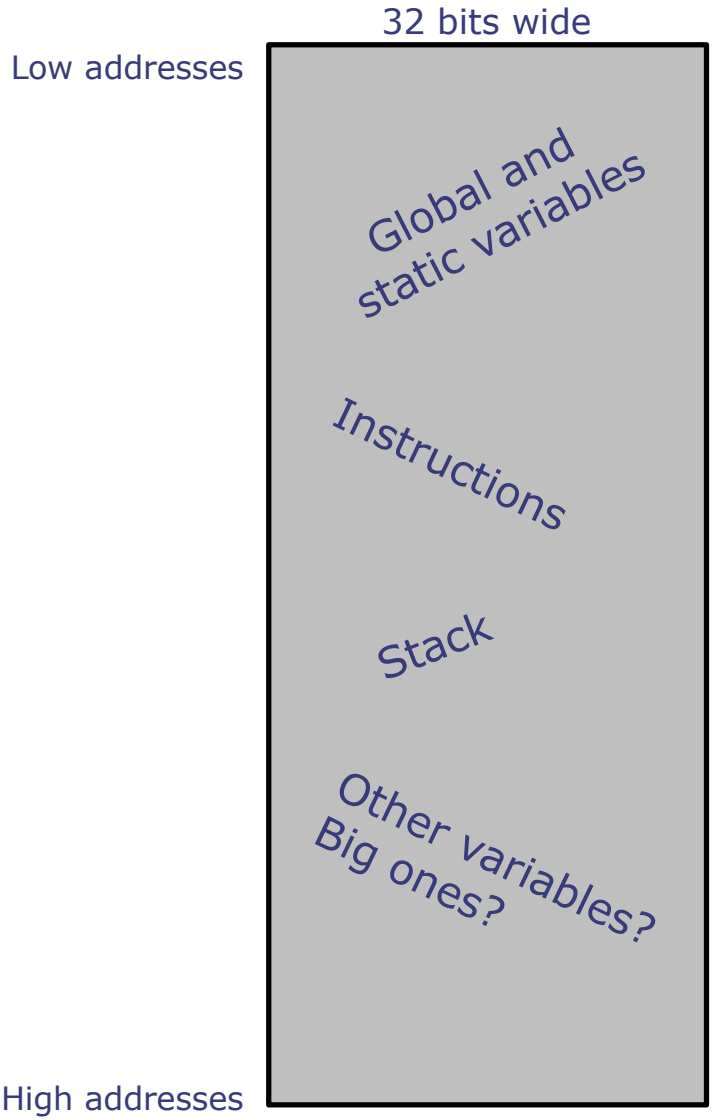
- Data storage is organized into a hierarchy



https://3000rain.medium.com/memory-hierarchy-afb83b61558c

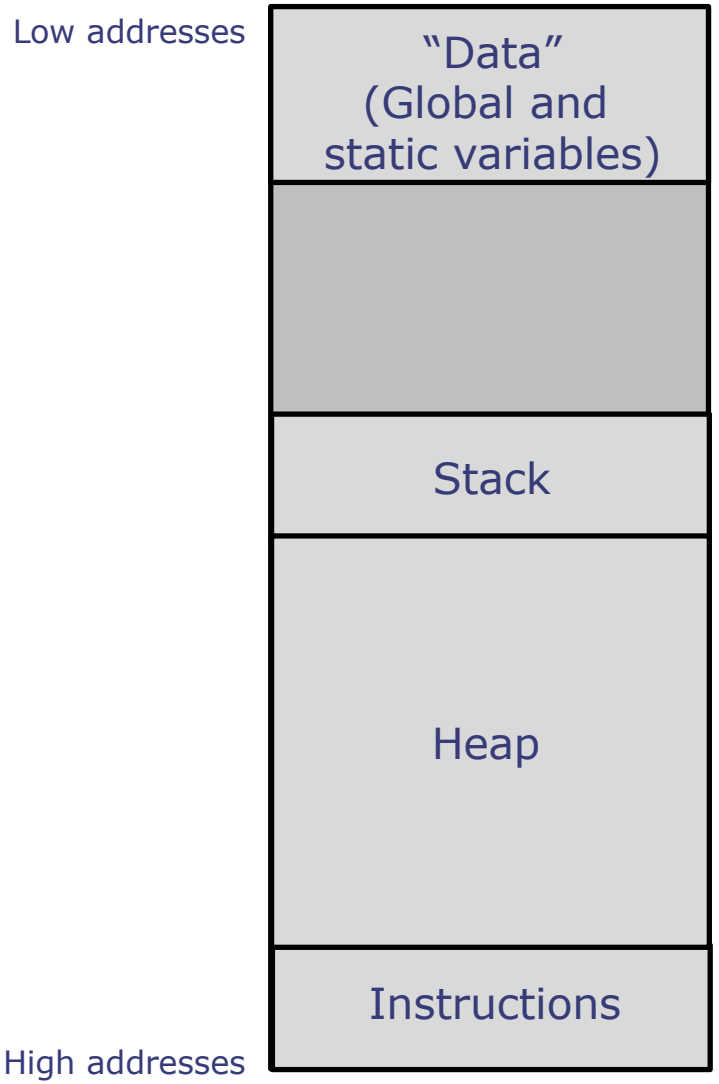
All Together that Hierarchy gives us the Memory

- Memory is a large chunk of storage that stores stuff
- We use it to store everything
- But in order to really be efficient, we need some rules



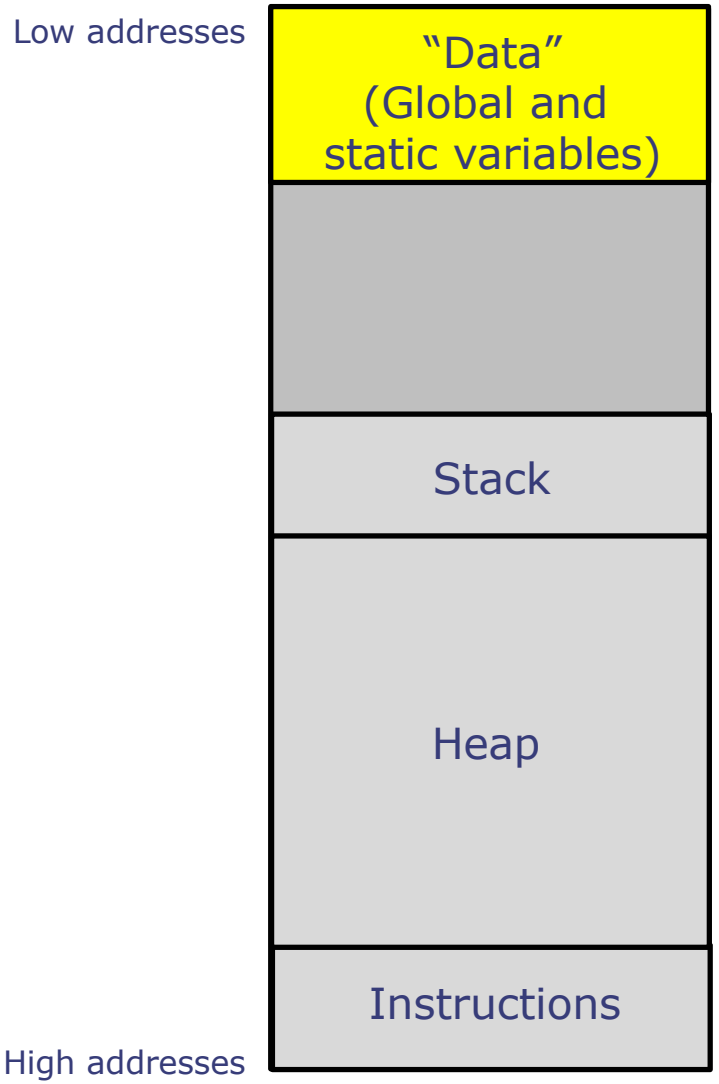
Memory

- In deployment we do NOT just have a homogenous block of stuff where we store stuff wherever
- **It is HIGHLY ORDERED in order to make sure we use it efficiently.**
- The memory of a program is broken into a number of regions for different purposes!



Global and Static Variables

- Variables that live the life of a program are put put in their own memory location.
- Constants and non-constant globals live in different regions here
- Often called the "Data" region

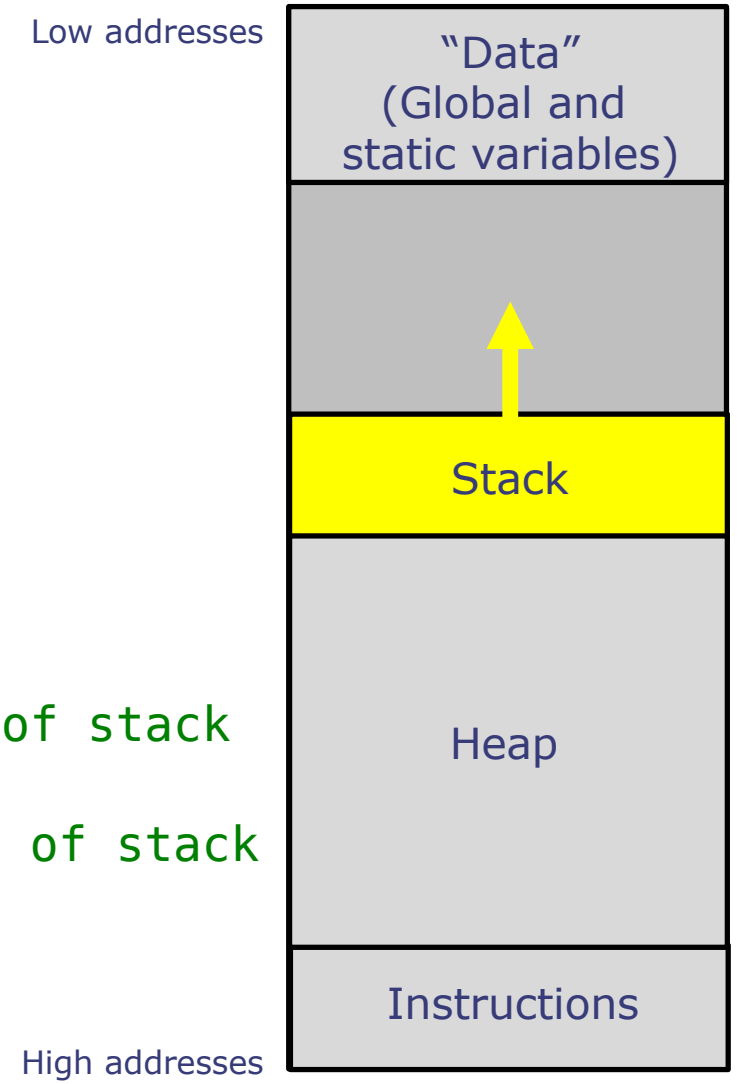


Stack

- Where local variables and local data live
- Accessed and tracked by the stack pointer! (sp)
- The stack grows downward in memory address space
- Seen this in assembly?

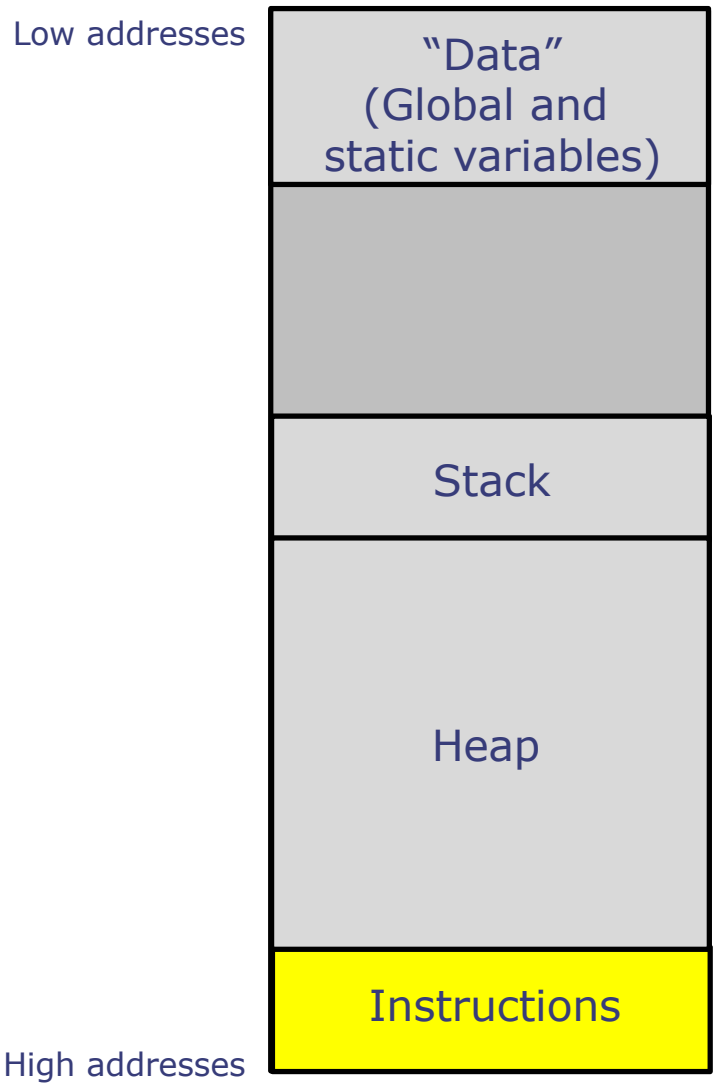
```
addi sp, sp, -8 # allocate 8 bytes of stack

addi sp, sp, 8 # deallocate 8 bytes of stack
```



Instructions

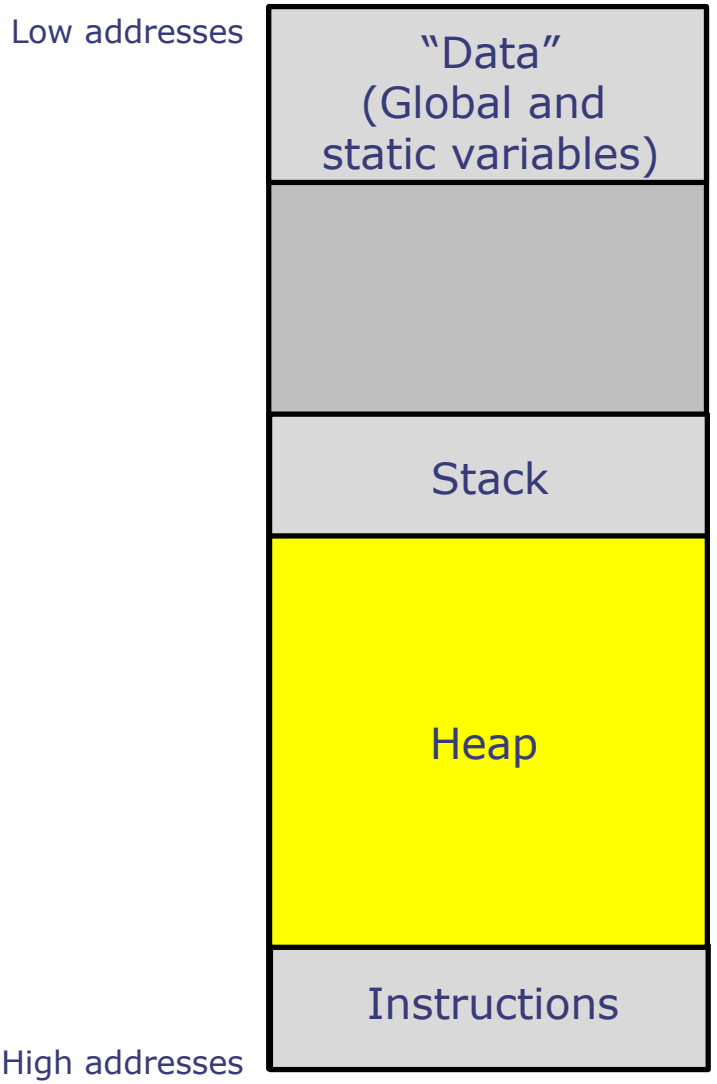
- The actual machine code that gets created from the assembly that gets created from the C code (or written raw) generally lives in one area
- Sometimes this memory is actually a different type of memory than the rest of memory
  - On ESP32-C3, instructions by default put into "IRAM", which can only be 32-bit addressed
  - Rest of memory can be byte-addressable
  - Faster-to-read memory (why is this desirable?)



Other things (end of lecture)

- Bigger things,
- Things with sizes known only at run-time,
- These things all go into the memory "heap"

- Talk about this last



Let's Investigate some RISCV memory

- Let's write a function that if given a pointer will print the values at that pointer for **len** words after:

main.C

```
//print len words at address
void memoryPrint(const int* a, int len){
  printf("Address: \t\tValue:\n"); //pretty
  for(int i=0; i<len; i++){
    printf("%p:\t\t0x%08x\n",(a+i), *(a+i));
  }
}
```

Print as pointer
Print as8 digit hex number

Some tabs for formatting prettily

The value of `a+i`
An Address

The contents of `a+i` interpreted as an integer

Let's Look at Global Variables?

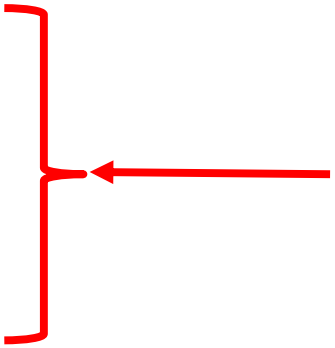
- Let's investigate further:

```
int x[10] = {0,1,2,3,4,5,6,7,8,9};

void app_main(){
  printf("Printing at location of x array");
  memoryPrint(x,20);
}
```

Print the memory where x is:

```
Address:            Value:
0x3fc8ef04:         0x00000000
0x3fc8ef08:         0x00000001
0x3fc8ef0c:         0x00000002
0x3fc8ef10:         0x00000003
0x3fc8ef14:         0x00000004
0x3fc8ef18:         0x00000005
0x3fc8ef1c:         0x00000006
0x3fc8ef20:         0x00000007
0x3fc8ef24:         0x00000008
0x3fc8ef28:         0x00000009
0x3fc8ef2c:         0x00000000
0x3fc8ef30:         0xffffffff
…
```



Is this what we would expect to live in memory?

Let’s Look at Global Variables?

- Investigate further:

```
char x[] = "MIT babyyyy.";

void app_main(){
  printf(”Printing at location of x array");
  memoryPrint(x,20);
}
```

Print the memory where x is:

```
Address:            Value:
0x3fc8ef04:         0x2054494d
0x3fc8ef08:         0x79626162
0x3fc8ef0c:         0x2e797979
0x3fc8ef10:         0x00000000
0x3fc8ef14:         0x00000000
0x3fc8ef18:         0xffffffff
0x3fc8ef1c:         0xffffffff
0x3fc8ef20:         0xffffffff
0x3fc8ef24:         0xffffffff
0x3fc8ef28:         0x40387b2c
0x3fc8ef2c:         0x4038cc10
…
```

ASCII

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

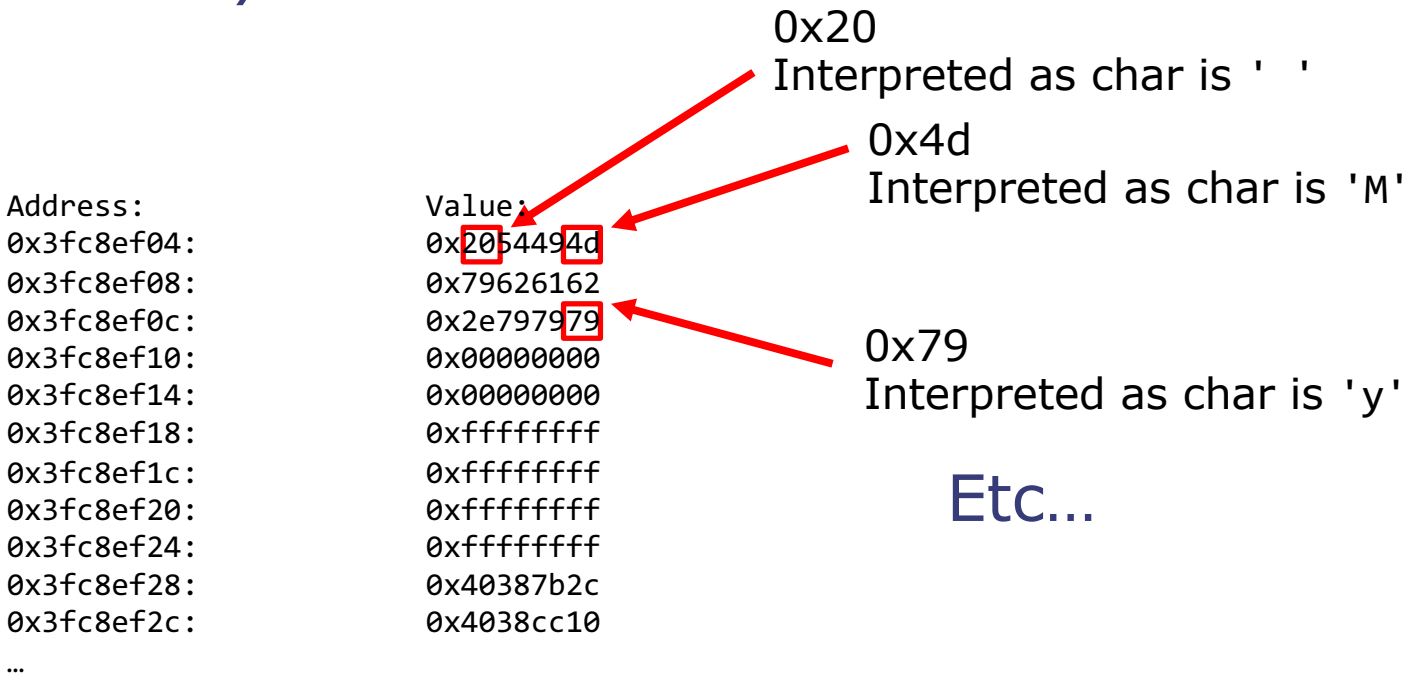
Let's Look at Global Variables?

- Investigate further:

```
char x[] = "MIT babyyyy.";

void app_main(){
  printf("Printing at location of x array");
  memoryPrint(x,20);
}
```

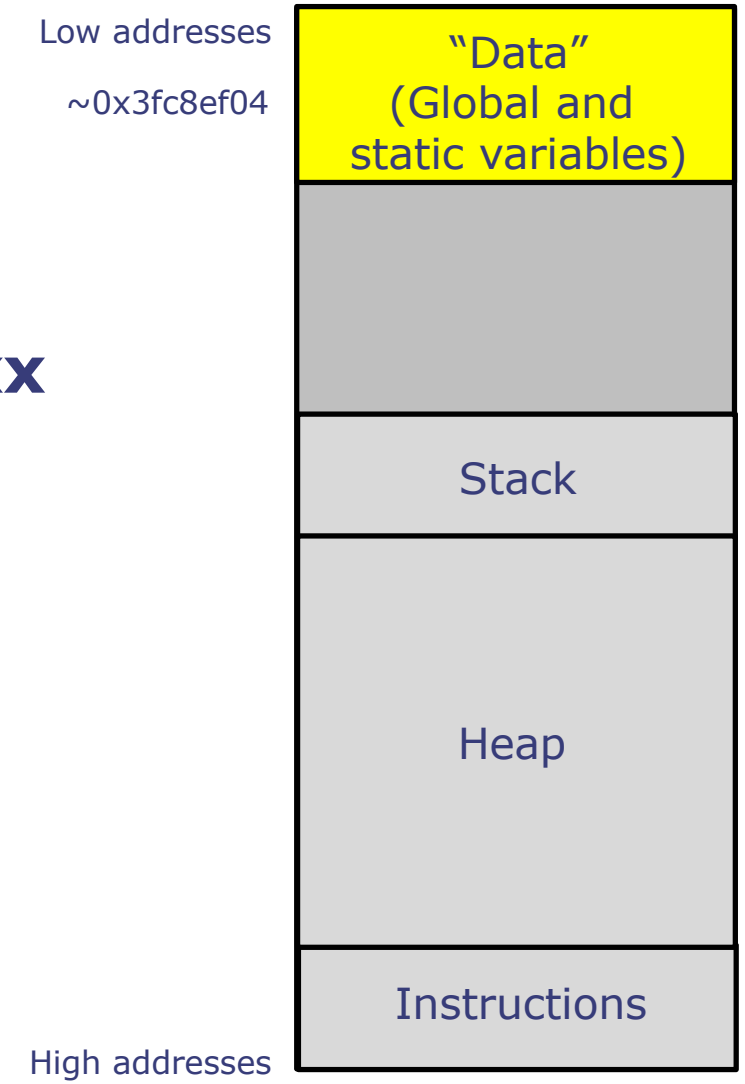
Print the memory where x is:



Global Conclusion

- Regardless of what we're putting in as globals, they seem to be existing in a particular memory space.
- For our board **~0x3fc8xxxx**

```
Address:                Value:
0x3fc8ef04:             0x2054494d
0x3fc8ef08:             0x79626162
0x3fc8ef0c:             0x2e797979
0x3fc8ef10:             0x00000000
0x3fc8ef14:             0x00000000
0x3fc8ef18:             0xffffffff
0x3fc8ef1c:             0xffffffff
0x3fc8ef20:             0xffffffff
0x3fc8ef24:             0xffffffff
0x3fc8ef28:             0x40387b2c
0x3fc8ef2c:             0x4038cc10
…
```

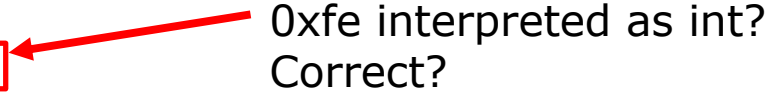


What about a different variable?

```
void app_main(){
  int* ptr_x = x;
  printf("Printing at location of x array:\n");
  memoryPrint(ptr_x,20);
  int qq = 254;
  int* ptr_qq = &qq;
  printf("Printing at location of qq:\n");
  memoryPrint(ptr_qq,20);
}
```

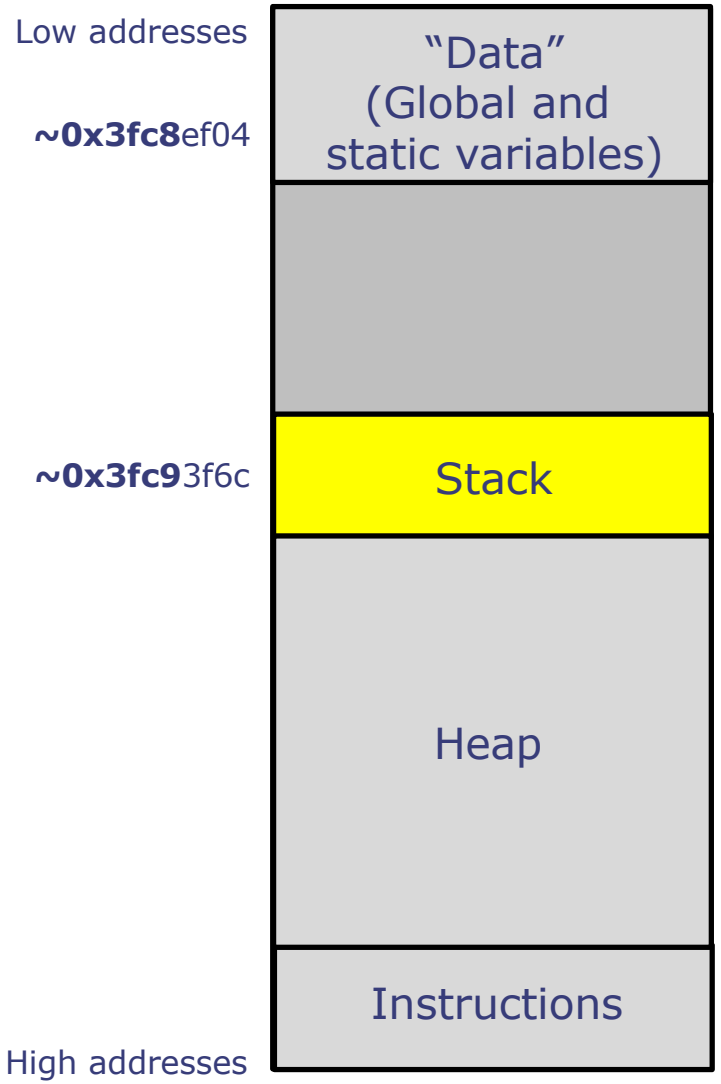
- Make `qq` inside of `app_main` function. What prints?

```
Address:        Value:
0x3fc93f6c:     0x000000fe
0x3fc93f70:     0x00000000
0x3fc93f74:     0x00000000
0x3fc93f78:     0x00000000
0x3fc93f7c:     0x42016518
0x3fc93f80:     0x00000000
0x3fc93f84:     0x00000000
0x3fc93f88:     0x00000000
0x3fc93f8c:     0x40388980
…
```



Stack

- Anything created inside a function (even `app_main`) is created *on the stack*
- Based on location of qq on previous page and in code demo, the stack seems to be in area of `0x3fc93xxx` in memory
- How many bytes separate stack from globals?

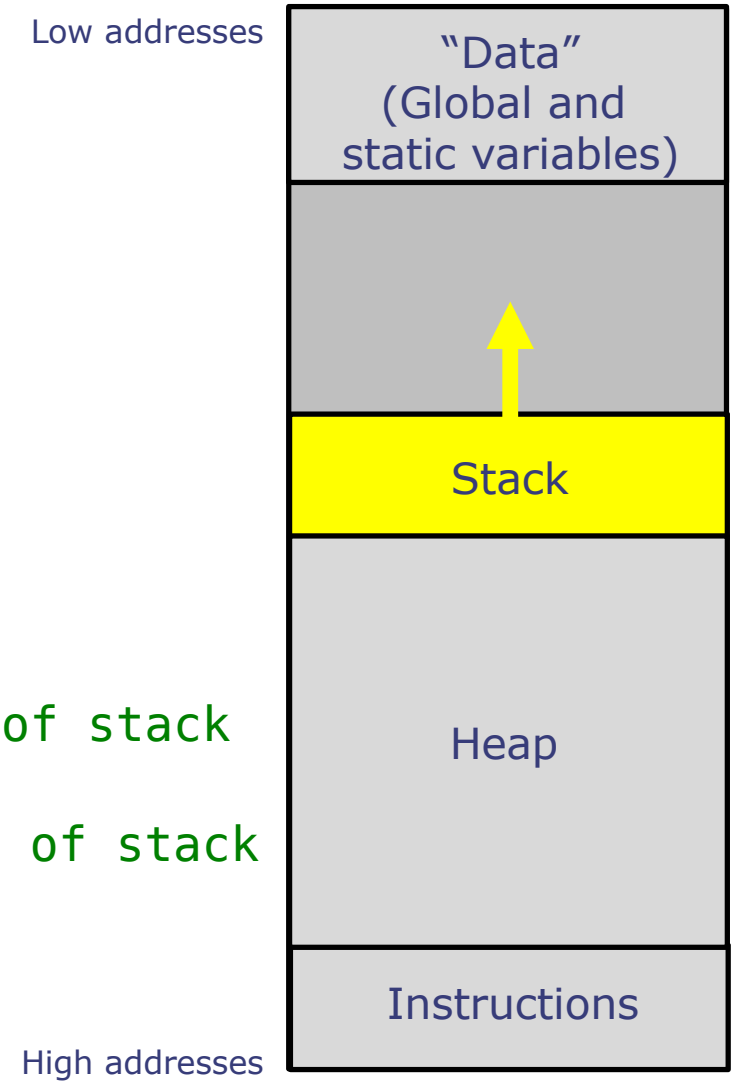


Stack

- Where local variables and data live
- Accessed and tracked by the stack pointer! (sp)
- The stack grows downward in memory address space
- Remember assembly?

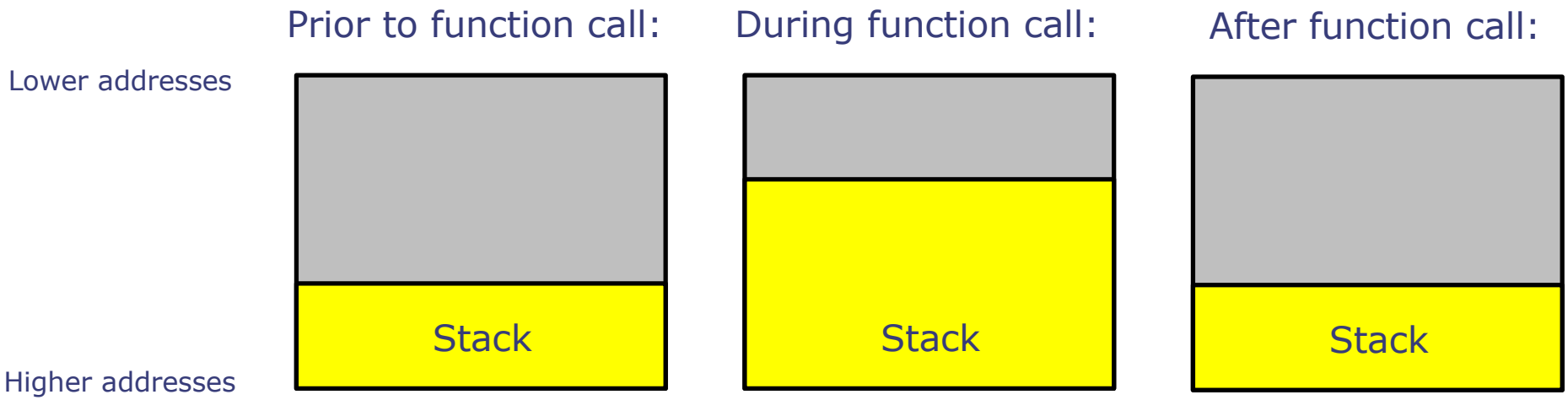
```
addi sp, sp, -8 # allocate 8 bytes of stack

addi sp, sp, 8 # deallocate 8 bytes of stack
```



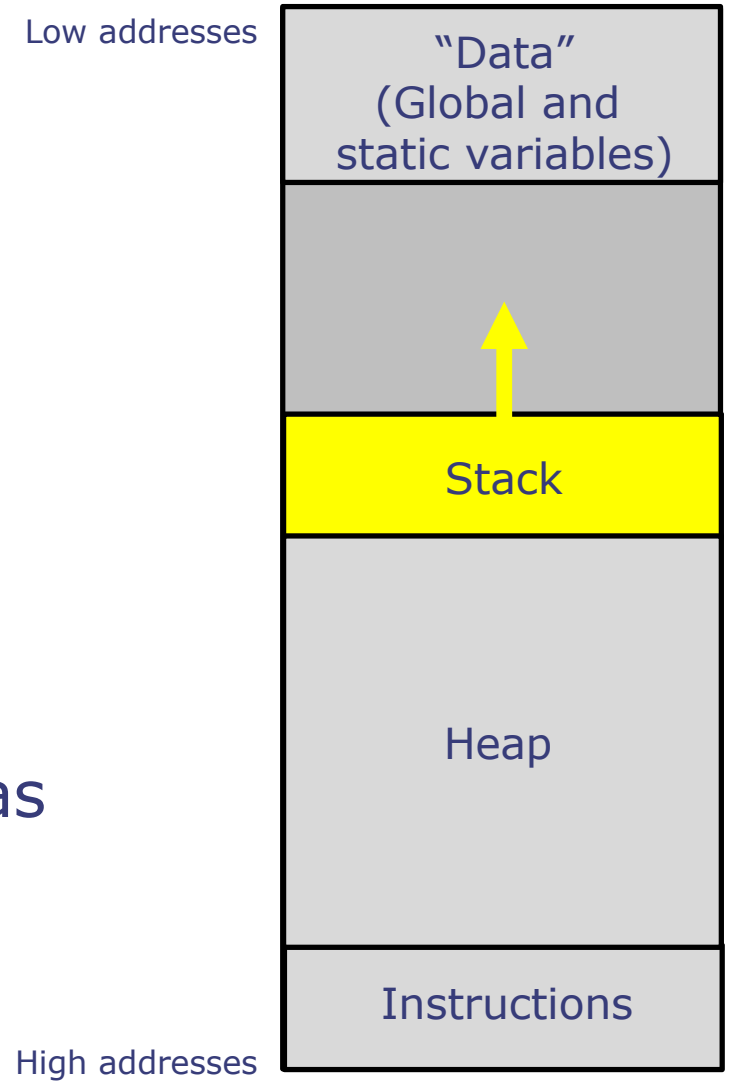
Stack is LIFO (Last-In-First-Out)

- Stack grows on every function call before shrinking at exit of function
- On recursive calls, the stack winds up…and then winds back down (stack detective problems!)



Stack

- Everything that goes on the stack should have a *size known* at compile time!
- The maximum stack size is often pretty small (on the order of a few kilobytes on many systems)
- Stack grows and contracts as used



Why must we know size?

- Things going on the stack need* to know what the size of everything is at compile time.
- This is saying that something like this shouldn't be allowed:

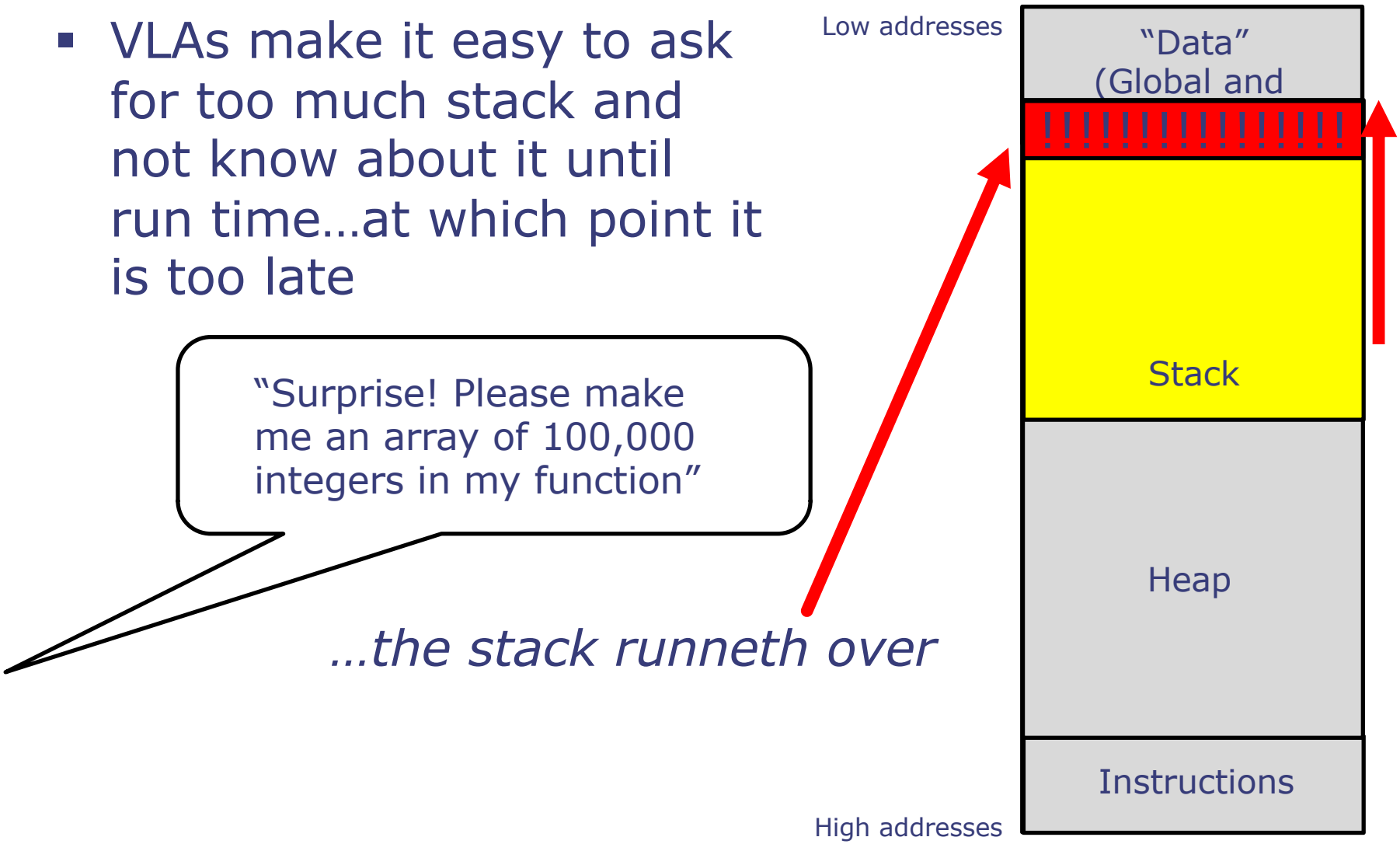
```
void func(int x){
  //VARIABLE LENGTH ARRAY (VLA):
  int y[x]; //length of array based on run-time argument
  for(int i=0; i<x; i++){
    y[i] = i;
    printf("%d\n",y[i]);
  }
}
```

- WHY Though?
  - It seems easy enough to just move the stack pointer a variable amount, right?

*C99 and beyond actually *do* allow variable-length arrays, but this is still frowned upon by many and not part of C++11 and things…

Careful of Stack Growth

- VLAs make it easy to ask for too much stack and not know about it until run time…at which point it is too late



Variable-Length Arrays (VLAs)



"…USING VLA'S IS ACTIVELY STUPID! It generates much more code, and much _slower_ code (and more fragile code), than just using a fixed key size would have done."

-Linus Torvalds
March 7th, 2018

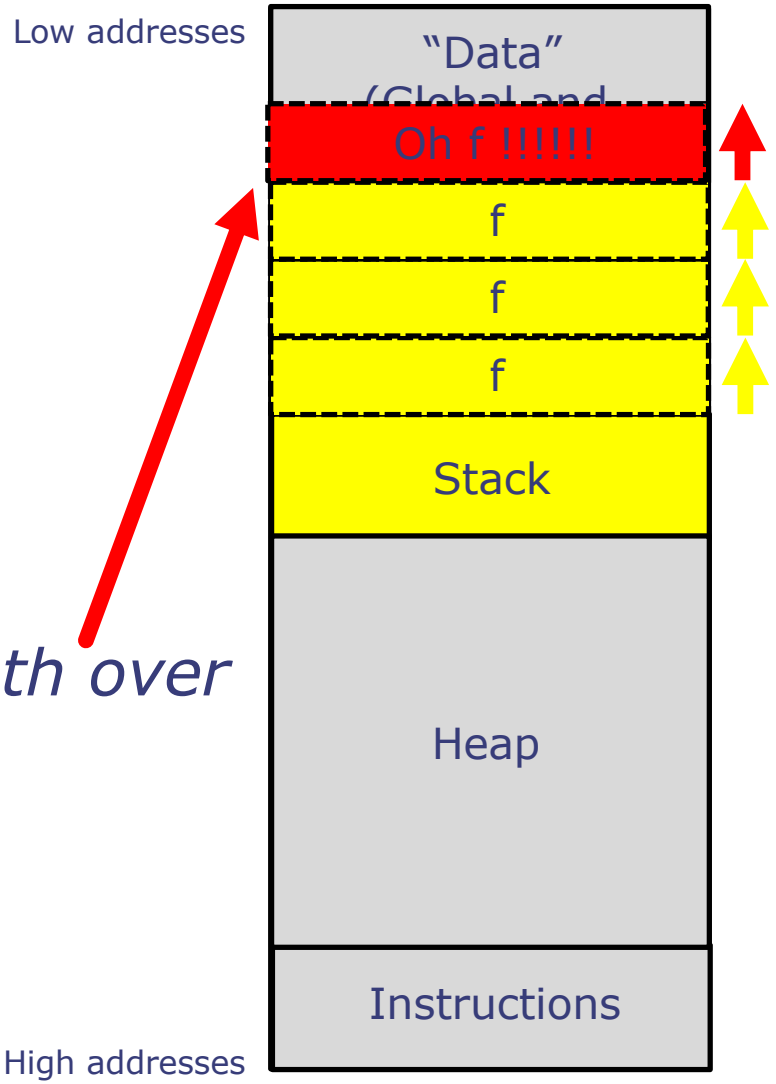
https://lkml.org/lkml/2018/3/7/621

Careful of Stack Growth

- Even ignoring VLAs, other patterns are prone to stack limitations too and hard to detect at compile time!

"please run f(f(f(f(f(f(f(f(x)))))))))"

…the stack runneth over



- *Always be aware of your recursion limits/worst case in your algorithms*

Stack Overflow



- If at run-time your stack goes too far you'll get a stack overflow.
- Hopefully this is noticed while happening and the system can fail-out gracefully and tell you.

- Even worse is to have a stack overflow and not be aware of it. (could corrupt data in nearby frames)

- Often place "stack canaries" or "watermarks" or do a thing called "stack painting" at boundaries and monitor their modification to determine stack misuse.

Let's Investigate the Stack More

- Already have a function `memoryPrint` that if given a pointer will print the values at that pointer for **len** words.

- Would like to print the "top" of the stack (where sp is pointing)…so we need to get the stack pointer…hard to actually do in pure C or really any language...why?

Let’s Investigate some ESP32 memory

- I want a function that can access the stack pointer
- Can do that with Assembly!

  debug.S

What would be a good function to investigate??

- A recursive function to find the $\log_2$ of an integer:

Let's print the stack In the function:

And

Right before we call it:

```
int ilog2(uint32_t x) {
  uint32_t y;
  printf("calculating ilog2 for %d:\n",x);
  getStack(20);
  if (x == 1) return 0;
  else {
    /* Shifts x right by 1 bit. */
    y = x >> 1;
    return ilog2(y) + 1;
  }
}
```

```
void app_main(){
  // other code here...
  printf(" Before calling ilog2:\n");
  getStack(20);
  int z = ilog2(18);
  printf("log_2(18) = %d",z);
}
```

ilog2?...algorithm aside…

- Base 2 logs are easy to calculate in a computer!
- Why?

- Well consider base 10 logs with base 10 numbers…what is:

$\text{int}(\log_{10}(1023)) \rightarrow 3$

$\text{int}(\log_{10}(80001)) \rightarrow 4$

$\text{int}(\log_{10}(79000023)) \rightarrow 7$

- Just find the most significant place with non-zero stuff in it

ilog2?...

- Now consider these base-2 numbers. What is:

$\text{int}(\log_2(0\text{b}10110)) \rightarrow 4$

$\text{int}(\log_2(0\text{b}00011010_01011001)) \rightarrow 12$

$\text{int}(\log_2(43000)) \rightarrow 15$

- Just find the most significant place with non-zero stuff in it!!!
  - **Start with lsb and scan upwards (by right shifting):**
    - As a human our eyes scan automatically…can't do that with program...must think like a computer

So let's investigate `ilog2`

- A recursive function to find the $\log_2$ of an integer:

```
int ilog2(uint32_t x) {
  uint32_t y;
  printf("calculating ilog2 for %d:\n",x);
  getStack(20);
  if (x == 1) return 0; // found the highest 1 place!
  else {
    // Shifts x right by 1 bit. (scan upwards by 1)
    y = x >> 1;
    return ilog2(y) + 1;
  }
}
```

What should this need to put on the Stack?????

```
int ilog2(uint32_t x) {
  uint32_t y;
  printf("calculating ilog2 for %d:\n",x);
  getStack(20);
  if (x == 1) return 0; // found the highest 1 place!
  else {
    // Shifts x right by 1 bit. (scan upwards by 1)
    y = x >> 1;
    return ilog2(y) + 1;
  }
}
```

- Thoughts???

So let's investigate `ilog2`

- A recursive function to find the $\log_2$ of an integer:

Let's print the stack In the function:

And

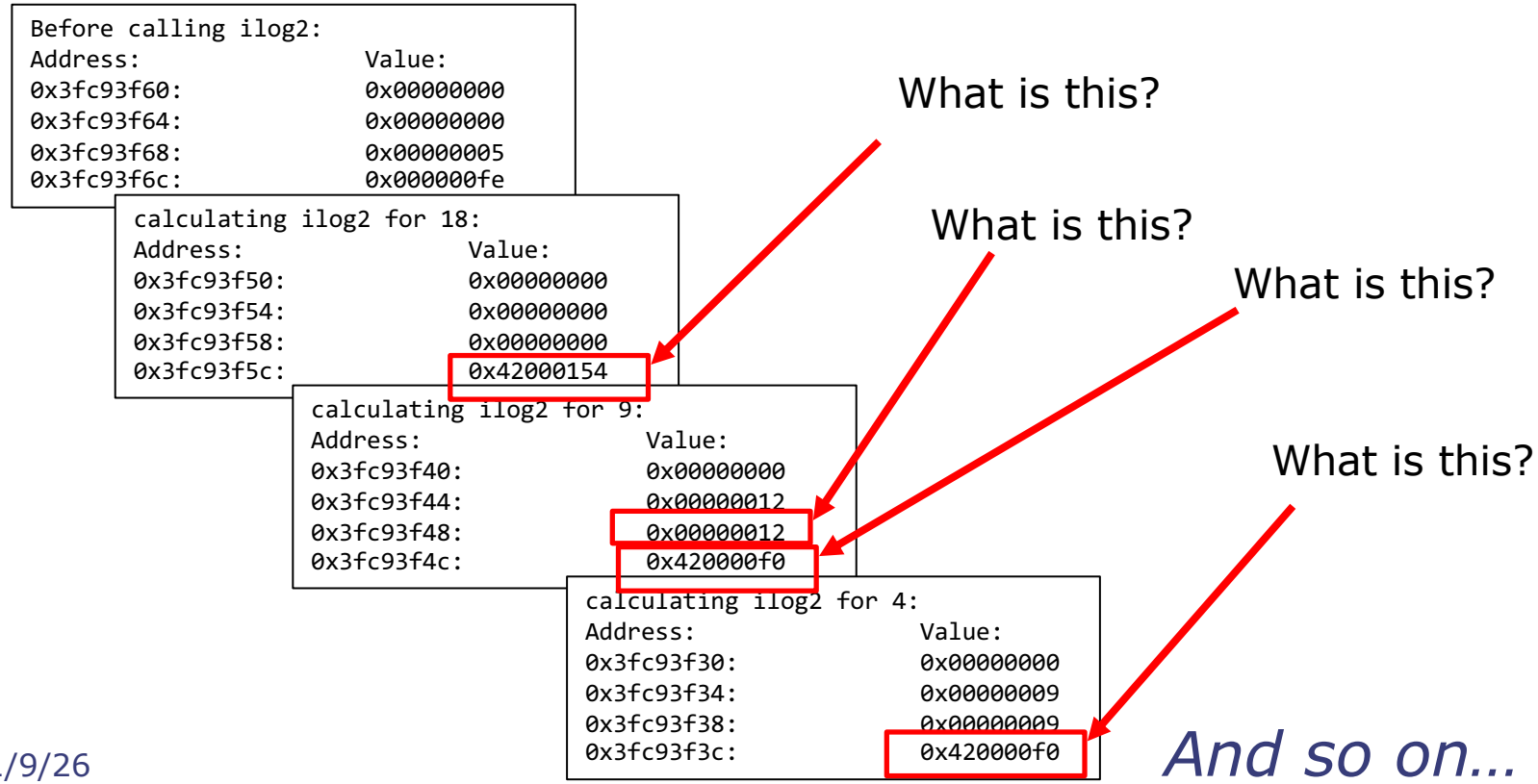
Right before we call it:

```
int ilog2(uint32_t x) {
  uint32_t y;
  printf("calculating ilog2 for %d:\n",x);
  getStack(20);
  if (x == 1) return 0; // only 1 in lsb!
  else {
    /* Shifts x right by 1 bit. */
    y = x >> 1;
    return ilog2(y) + 1;
  }
}
```

```
void app_main(){
  // other code here...
  printf(" Before calling ilog2:\n");
  getStack(20);
  int z = ilog2(18);
  printf("log_2(18) = %d",z);
}
```

What do we see?

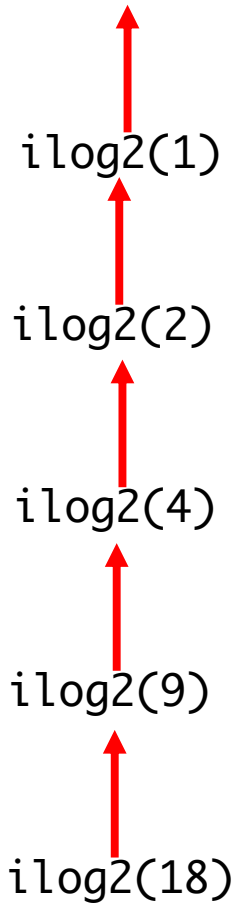
- Each recursive call to ilog2 seems to decrement the stack pointer by 0x10 (16) bytes
- Some content that we would expect seems to show up (repeated halvings of the input value)



And so on…

Final Stack Grab

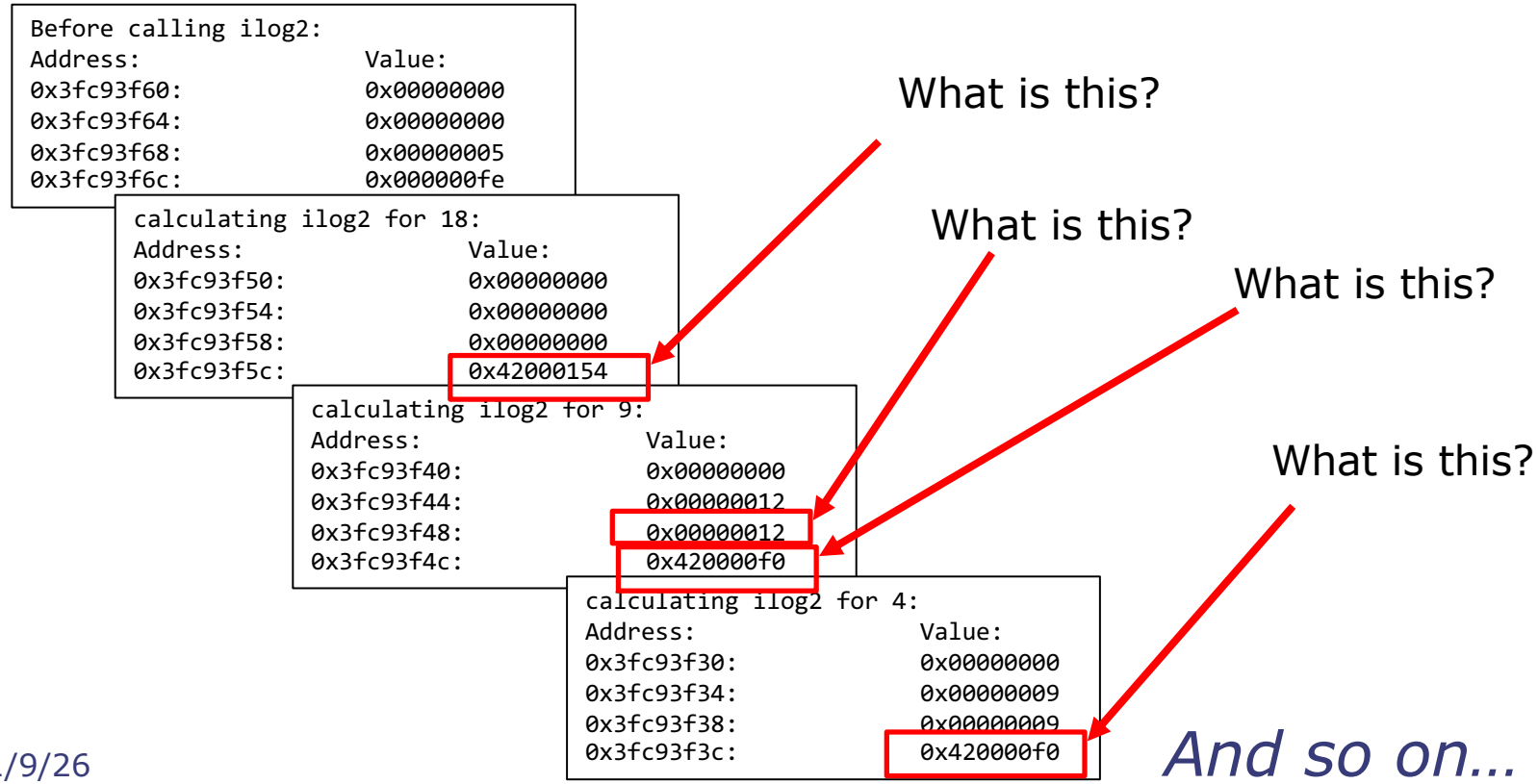
- On the last stack trace printed you can see the entire history of the algorithm running




```
calculating ilog2 for 1:
Address:                    Value:
0x3fc94790:                 0x00000000
0x3fc94794:                 0x00000002
0x3fc94798:                 0x00000002
0x3fc9479c:                 0x420000f0
0x3fc947a0:                 0x00000000
0x3fc947a4:                 0x00000004
0x3fc947a8:                 0x00000004
0x3fc947ac:                 0x420000f0
0x3fc947b0:                 0x00000000
0x3fc947b4:                 0x00000009
0x3fc947b8:                 0x00000009
0x3fc947bc:                 0x420000f0
0x3fc947c0:                 0x00000000
0x3fc947c4:                 0x00000012
0x3fc947c8:                 0x00000012
0x3fc947cc:                 0x420000f0
0x3fc947d0:                 0x00000000
0x3fc947d4:                 0x00000000
0x3fc947d8:                 0x00000000
0x3fc947dc:                 0x42000154
```

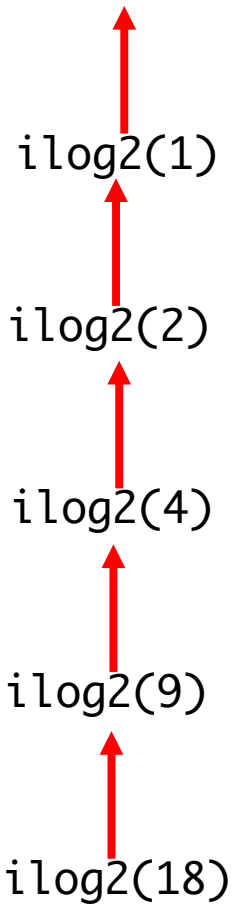
What do we see?

- Each recursive call to ilog2 seems to decrement the stack pointer by 0x10 (16) bytes
- Some content that we would expect seems to show up (repeated halvings of the input value)



And so on…

Why 16 bytes per call? (aside)




```
calculating ilog2 for 1:
Address:                  Value:
0x3fc94790:               0x00000000
0x3fc94794:               0x00000002
0x3fc94798:               0x00000002
0x3fc9479c:               0x420000f0
0x3fc947a0:               0x00000000
0x3fc947a4:               0x00000004
0x3fc947a8:               0x00000004
0x3fc947ac:               0x420000f0
0x3fc947b0:               0x00000000
0x3fc947b4:               0x00000009
0x3fc947b8:               0x00000009
0x3fc947bc:               0x420000f0
0x3fc947c0:               0x00000000
0x3fc947c4:               0x00000012
0x3fc947c8:               0x00000012
0x3fc947cc:               0x420000f0
0x3fc947d0:               0x00000000
0x3fc947d4:               0x00000000
0x3fc947d8:               0x00000000
0x3fc947dc:               0x42000154
```

- Don’t actually need it here BUT...
- If you read the *actual* RISC-V calling convention, you’ll see that stack moves must be 16-byte aligned for compatibility with RV64 and because the largest possible data primitive is the long double (128 bits)

Conclusions Part 1

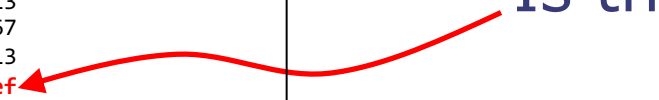
- The value stored on the stack repetitively during recursive calls are in a different spot:
  - Somewhere in the 0x42000000 neighborhood
- Silly, ESP32. You've revealed to us where your instructions possibly live so we should go look:
- That value is the return address of the recursive call to ilog2!
- That means -4 bytes before that number *should* be a **RISCV jal instruction!!!**
- Is it???

Instruction Location

- Let's print some memory right around where the ilog2 recursive return address is:

```
int* ins = 0x420000f0-40;
// Go back by ten words then we'll print 20 words
printf("instructions (hopefully)\n");
memoryPrint(ins,20);
```

```
instructions (hopefully):
Address:                Value:
0x420000c8:             0xf59ff0ef
0x420000cc:             0x00100793
0x420000d0:             0x00f41c63
0x420000d4:             0x00000513
0x420000d8:             0x00c12083
0x420000dc:             0x00812403
0x420000e0:             0x01010113
0x420000e4:             0x00008067
0x420000e8:             0x00145513
0x420000ec:             0xfb9ff0ef
0x420000f0:             0x00150513
0x420000f4:             0xfe5ff06f
0x420000f8:             0xfe010113
0x420000fc:             0x00112e23
0x42000100:             0x3c020537
0x42000104:             0x12050513
0x42000108:             0x733070ef
0x4200010c:             0x01400593
0x42000110:             0x90418513
0x42000114:             0xf29ff0ef
```



Is that a **jal**?

Instruction Location

- Let’s break the binary down:

```
instructions (hopefully):
Address:                Value:
0x420000c8:             0xf59ff0ef
0x420000cc:             0x00100793
0x420000d0:             0x00f41c63
0x420000d4:             0x00000513
0x420000d8:             0x00c12083
0x420000dc:             0x00812403
0x420000e0:             0x01010113
0x420000e4:             0x00008067
0x420000e8:             0x00145513
0x420000ec:             0xfb9ff0ef
0x420000f0:             0x00150513
0x420000f4:             0xfe5ff06f
0x420000f8:             0xfe010113
0x420000fc:             0x00112e23
0x42000100:             0x3c020537
0x42000104:             0x12050513
0x42000108:             0x733070ef
0x4200010c:             0x01400593
0x42000110:             0x90418513
0x42000114:             0xf29ff0ef
```

Binary?

`0b11111011100111111111000011101111`

Let’s break the binary down:

Use the encoding instruction set table

- Use the opcodes (highlighted) to first figure out what instruction

RV32I Base Instruction Set (MIT 6.191 (6.004) subset)

imm[31:12]				rd	0110111	LUI
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

- For JAL and branch instructions (BEQ, BNE, BLT, BGE, BLTU, BGEU), the immediate encodes the target address as an offset from the current pc (i.e., $pc + imm = label$).
- Not all immediate bits are encoded. Missing lower bits are filled with zeros and missing upper bits are sign-extended.

Instruction Decoding

`0b11111011100111111111000011101111`

Check lower 7:

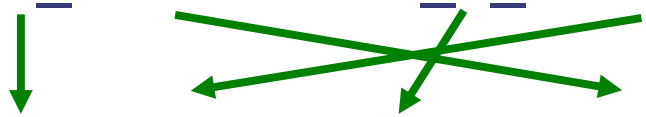
`0b11111011100111111111100001_1101111`

OOOH! That's a JAL!!!

imm[20|10:1|11|19:12]			rd	1101111	JAL
imm[11:0]	rs1	000	rd	1100111	JALR

Split up/interpret rest of data!

`0b1_1111011100_1_11111111_00001_1101111`



x1(ra) JAL

`1_11111111_1_1111011100`

Jump by? `1_11111111_1_11110111000` bytes?

Add that extra 0 for 0th bit

How to interpret that jump amount?

- That `1_11111111_1_11110111000` needs to be read in two's complement
- How do I figure out what its value is?
- Let's negate it and find what its negative is since that's easier: (remember –A = ~A+1):

```
0b11111111111110111000
~0b11111111111110111000 -> 0b00000000000001000111
                                          +       1
                           0b00000000000001001000
                          =64+8 = 72 bytes
                                = 18 words
```

So says jump 18 instructions *back* !

How to interpret that jump amount?

- That `1_11111111_1_11110111000` needs to be read in two's complement
- How do I figure out what its value is?
- Let's negate it and find what its negative is since that's easier: (remember –A = ~A+1):

```
0b11111111111110111000
~0b11111111111110111000 -> 0b00000000000001000111
                                        +          1
                           0b00000000000001001000
                          =64+8 = 72 bytes
                                = 18 words
```

So `0b11111111111110111000` is -72 corresponding to -18 instructions (18 instructions back!!!)

Decoding Instructions

- Can do this for all instructions!

```
Address:                Value:
0x420000c8:             0xf59ff0ef
0x420000cc:             0x00100793
0x420000d0:             0x00f41c63
0x420000d4:             0x00000513
0x420000d8:             0x00c12083
0x420000dc:             0x00812403
0x420000e0:             0x01010113
0x420000e4:             0x00008067
0x420000e8:             0x00145513              srli    a0, s0, 1
0x420000ec:             0xfb9ff0ef              jal     ra, -72
0x420000f0:             0x00150513              addi    a0, a0, 1
0x420000f4:             0xfe5ff06f
0x420000f8:             0xfe010113
0x420000fc:             0x00112e23
0x42000100:             0x3c020537
0x42000104:             0x12050513
0x42000108:             0x733070ef
0x4200010c:             0x01400593
0x42000110:             0x90418513
0x42000114:             0xf29ff0ef
```

~~Kinda~~ ~~Very~~ *Disgustingly* Cool

- Original C:

```c
int ilog2(uint32_t x) {
  uint32_t y;
  printf("calculating ilog2 for %d:\n",x);
  getStack(20);
  if (x == 1) return 0;
  else {
    /* Shifts x right by 1 bit. */
    y = x >> 1;
    return ilog2(y) + 1;
  }
}
```

- Leads to this: Kinda Makes sense!

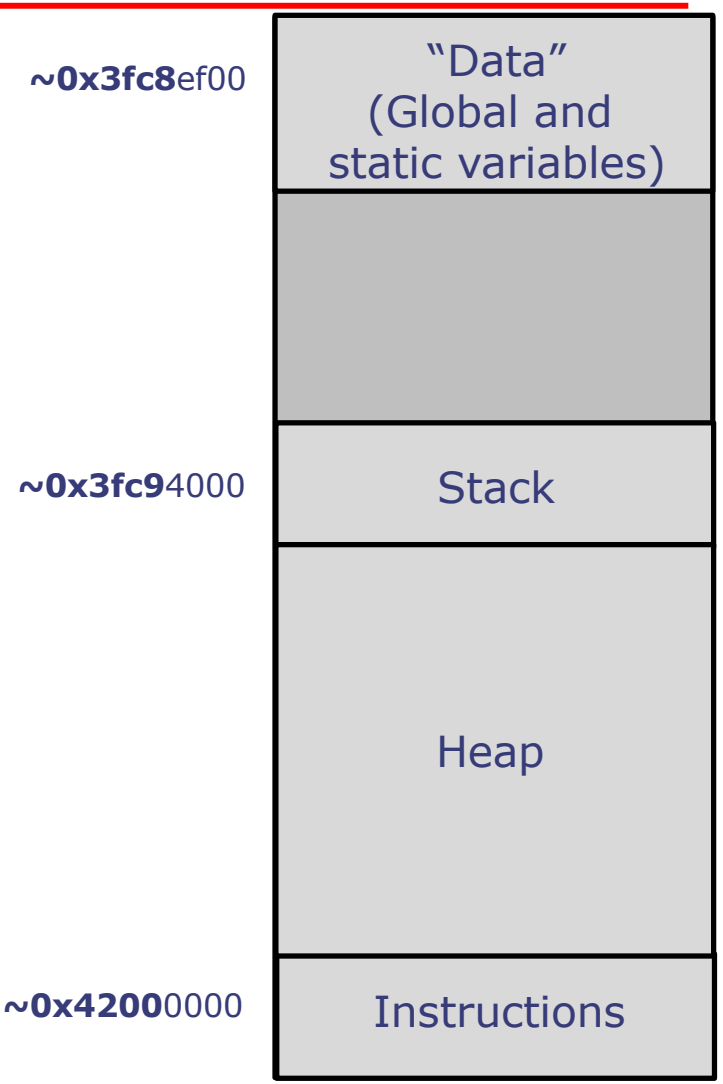
```
Address:                Value:
...
...
0x420000e8:             0x00145513          srli   a0, a0, 1
0x420000ec:             0xfb9ff0ef          jal    ra, -72
0x420000f0:             0x00150513          addi   a0, a0, 1
...
```

Use a Disassembler

- The entire program can be dis-assembled from the byte code back into the assembly using a disassembler.
- There are some online

Conclusion

- Interesting So to review:
- Stack is ~ 0x3fc94000
- Const, globals ~ 0x3fc8ef00
- Globals ~ 0x3fc8ef00
- Instructions far away in another chunk of memory: ~0x42000000



Limitations...what about...

- Flexible-length data structures?...we shouldn't do "VLA's" but I still want that functionality...

```
void func(int x){
  //VARIABLE LENGTH ARRAY (VLA):
  int y[x]; //length of array based on run-time argument
  for(int i=0; i<x; i++){
    y[i] = i;
    printf("%d\n",y[i]);
  }
}
```

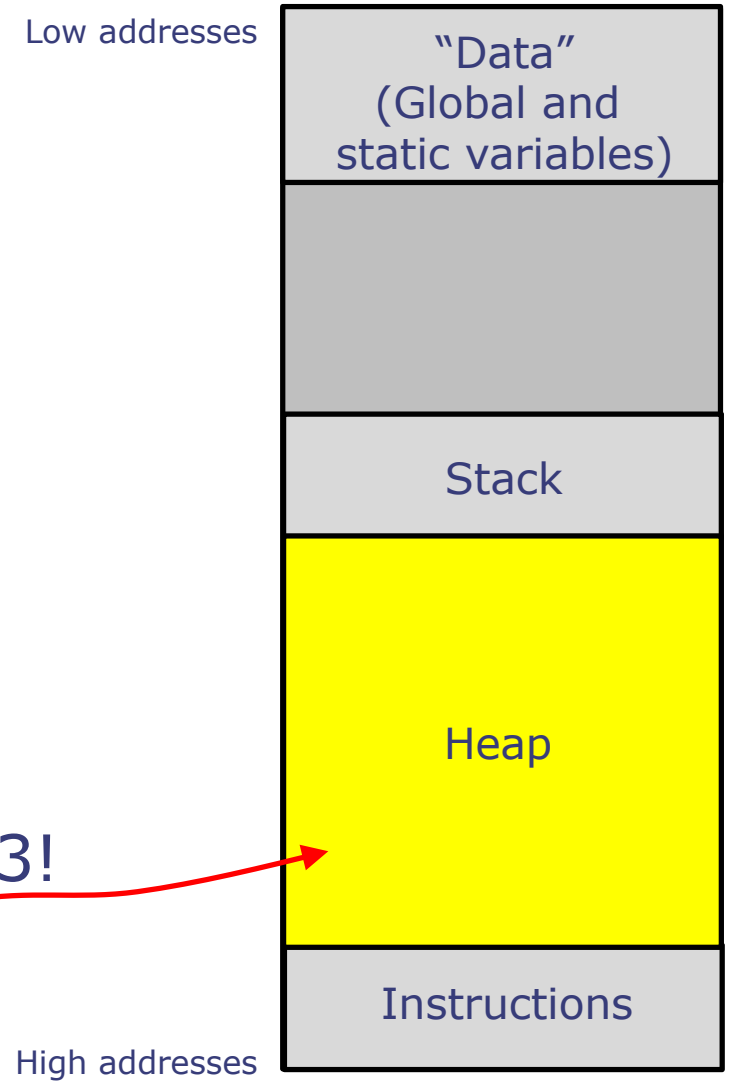
- Deal with unknown-size objects at run-time?

- This is where the heap comes in!

Heap

- There is also a big pile of memory that can be used on-demand at run-time to help store data that isn't known ahead of time!
- This is the Heap and is usually shown in between global/static stuff and stack*

*not the case on ESP32-C3! The heap is here:



The Heap

- Need a string to keep growing?
  - Heap
- Need a dictionary that keeps getting bigger?
  - Heap
- Need 10000's of bytes to manipulate data with?
  - Heap
- Don't know at compile time how much memory you'll need?
  - Heap

Four Classic C functions to deal with Heap

- malloc
- calloc
- realloc
- free

malloc

"**m**emory **alloc**ation"

```
void* malloc( uint32_t size );
```

- Ask for `size` number of bytes and return a pointer to it
- *If size is not achievable return a pointer value of* NULL
  - Why is NULL safe?
  - We'll never have a legit pointer at NULL so it is a good "flag" value

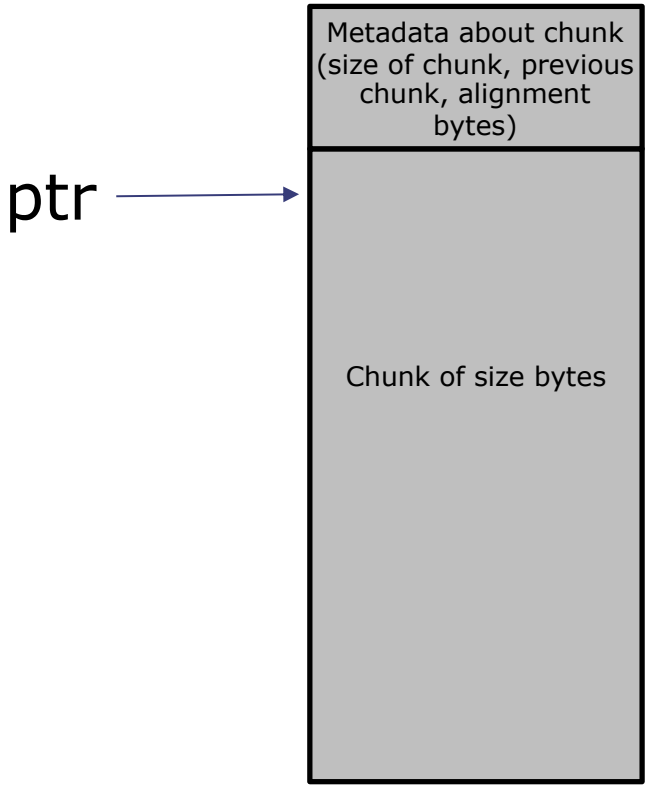
void* ???

- What is it?
- A pointer of no specific type
- If you want to use it, you'll need to cast it to another pointer type (you might get a warning or you might get an error…depends on compiler/OS settings)

What does malloc "make"?

```
void* malloc( uint32_t size );
```

- malloc returns a pointer to a chunk in memory from the heap
- Generally in front of the pointer are a few bytes (16) with information about the chunk. (important soon)



"**m**emory **alloc**ation"

calloc

```
void* calloc( uint32_t num, uint32_t size );
```

- Allocates enough memory for an array of `num` objects that are each `size` bytes.
- Also initializes all bytes in the memory block to zero.
- Pros and cons vs. malloc?
  - `malloc` takes the memory as it is (Faster)
  - `calloc` has to write to each spot (Slower)
  - With `malloc` don't know what starting with (danger!)
  - With `calloc` do know what starting with (less danger)

"**c**ontinguous **alloc**ation"

realloc

```
void* realloc( void *ptr, uint32_t new_size );
```

- Either:
  - Expand or contract memory chunk attached to ptr based on new_size
  - Generate new block and return that pointer
- Return NULL if can't be allocated
- If `ptr` is NULL this acts just like malloc and just searches and gives you a new memory

"**re-** **alloc**ation"

free

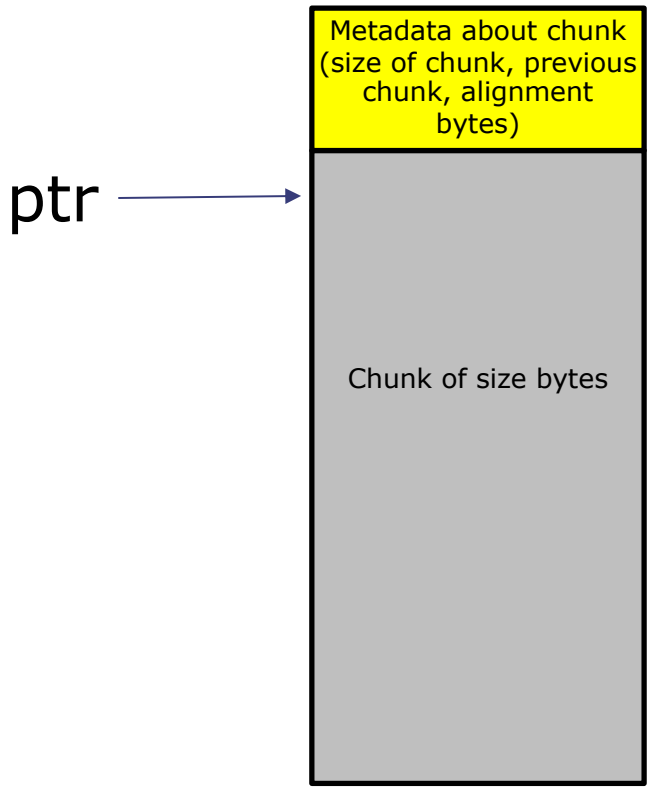
```
void free( void* ptr );
```

- Calling free on a pointer to a heap memory chunk will release it back for general use
- What does “release” here mean?
- It unfortunately doesn’t necessarily mean that it will be merged into the larger pool of available memory because of fragmentation!

How does free work?

- Free knows to look ahead of where the pointer is for the metadata about the chunk
- That's how it knows how much to free!

```
void free( void* ptr );
```



Malloc-free-malloc-free-etc…

- Behind the scenes is a complicated accounting program that keeps track of the memory
- Keeps a list of available spots based on what were previously freed
- When new `malloc` calls arrive, it will try to use previously used segments where possible



General heap rules

- Always make sure to check that you've gotten the memory you asked for!
- AND....
- Always make sure to free memory when done!

- ...seems so simple and easy. Heap usage must obviously not be the source of many bugs in the world. Right?

Example

- Remember our struct Subject from Lecture 03*:

```
struct Subject{
  int dept;
  int units;
  char name[500];
  char description[5000];
  int num_students;
};
```

  - Let's create structs on the heap as needed!

*slightly modified

Example

- Just repeatedly create (malloc) and delete (free) an instance of a `Subject` struct

```
void app_main() {
  printf("Starting Heap: %d\n",esp_get_free_heap_size());
  for (int i=0; i<1000; i++){
    struct Subject* x; //pointer of type struct Subject
    x = malloc(sizeof(struct Subject)); //allocate memory for a Subject struct!
    printf("Value at 0x%08x:\t %d\n",x,x[0]);
    printf("Heap with Struct: %d\n",esp_get_free_heap_size());
    free(x);
    printf("Ending Heap: %d\n",esp_get_free_heap_size());
  }
}
```

But what if…

- Just repeatedly create (malloc) and delete (free) an instance of a `Subject` struct

```
void app_main() {
  printf("Starting Heap: %d\n",esp_get_free_heap_size());
  for (int i=0; i<1000; i++){
    struct Subject* x; //pointer of type struct Subject
    x = malloc(sizeof(struct Subject)); //allocate memory for a Subject struct!
    printf("Value at 0x%08x:\t %d\n",x,x->num_students); //"use" x
    printf("Heap with Struct: %d\n",esp_get_free_heap_size());
    free(x); //forget to free the memory?
    printf("Ending Heap: %d\n",esp_get_free_heap_size());
  }
}
```

Catastrophe!


REMEMBER TO CALL
FREE AFTER MALLOC
THAT'S EASY, RIGHT?
THAT'S EASY, RIGHT?

One type of Error: Memory Leak

- Forgetting or failing to free memory after usage
- Can eventually run out of memory!
- Seconds, minutes, days later your system will go to allocate a certain amount of bytes…and none will be there…and you're busted.

Another Type of Error: Allocation Failure

- You accidentally assume that you always get a good pointer back no matter what (forget to check that it is not NULL!)

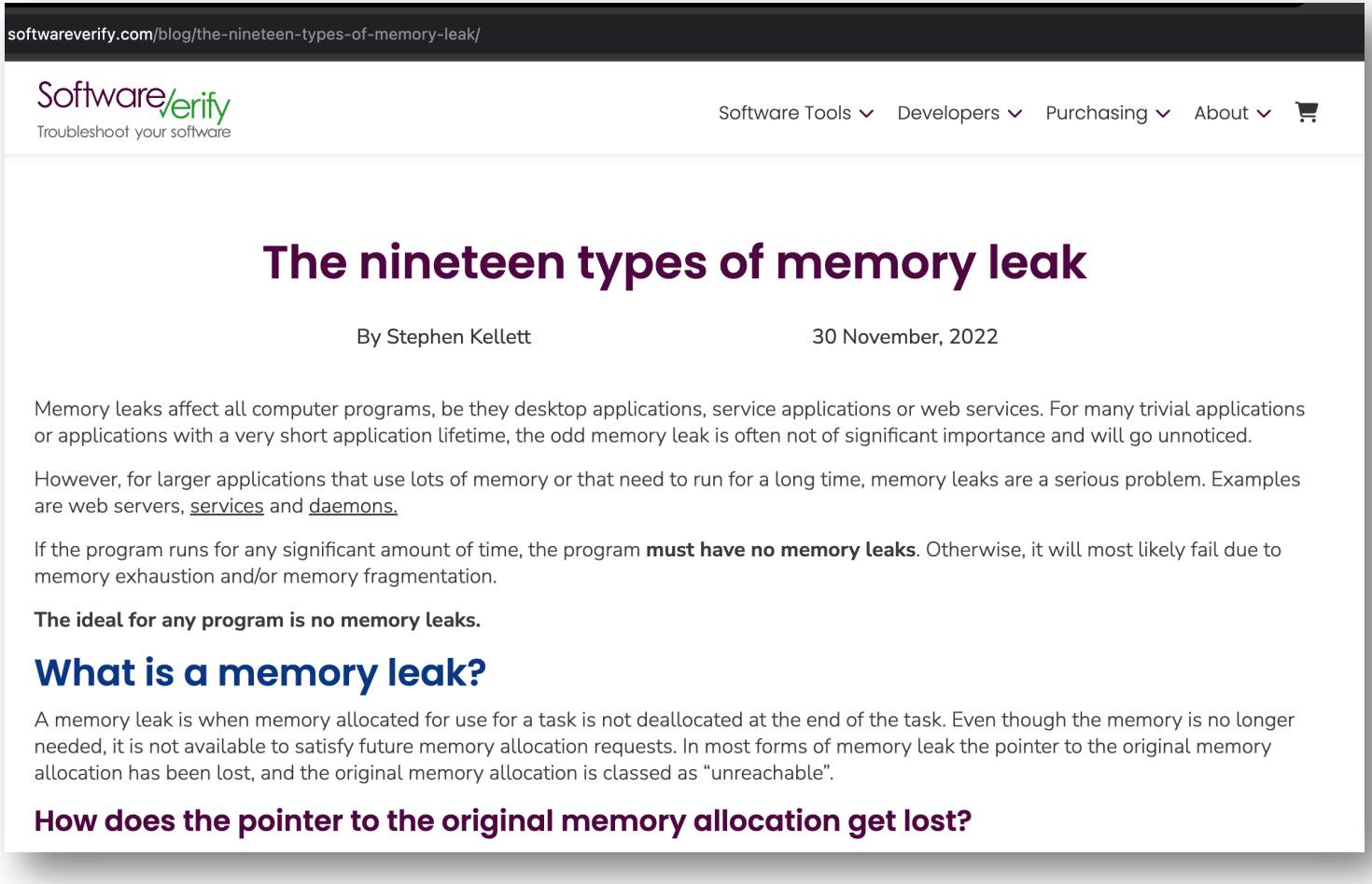
- Writing to NULL will cause a crash

Other Errors

- **Use-After-Free:** You free a block of memory, but then still use it (no guarantee what is in it anymore) *("dangling" pointer*)
- **Use-Before-Alloc:** Using pointer that doesn't point to valid allocated memory *("wild" pointer*)
- **Double-Free:** freeing a memory block…then freeing it again

- A lot of these bugs might not fail right away

Common Topic in EECS



softwareverify.com/blog/the-nineteen-types-of-memory-leak/

Software verify
Troubleshoot your software

Software Tools | Developers | Purchasing | About

The nineteen types of memory leak

By Stephen Kellett 30 November, 2022

Memory leaks affect all computer programs, be they desktop applications, service applications or web services. For many trivial applications or applications with a very short application lifetime, the odd memory leak is often not of significant importance and will go unnoticed.

However, for larger applications that use lots of memory or that need to run for a long time, memory leaks are a serious problem. Examples are web servers, services and daemons.

If the program runs for any significant amount of time, the program **must have no memory leaks**. Otherwise, it will most likely fail due to memory exhaustion and/or memory fragmentation.

The ideal for any program is no memory leaks.

What is a memory leak?

A memory leak is when memory allocated for use for a task is not deallocated at the end of the task. Even though the memory is no longer needed, it is not available to satisfy future memory allocation requests. In most forms of memory leak the pointer to the original memory allocation has been lost, and the original memory allocation is classed as “unreachable”.

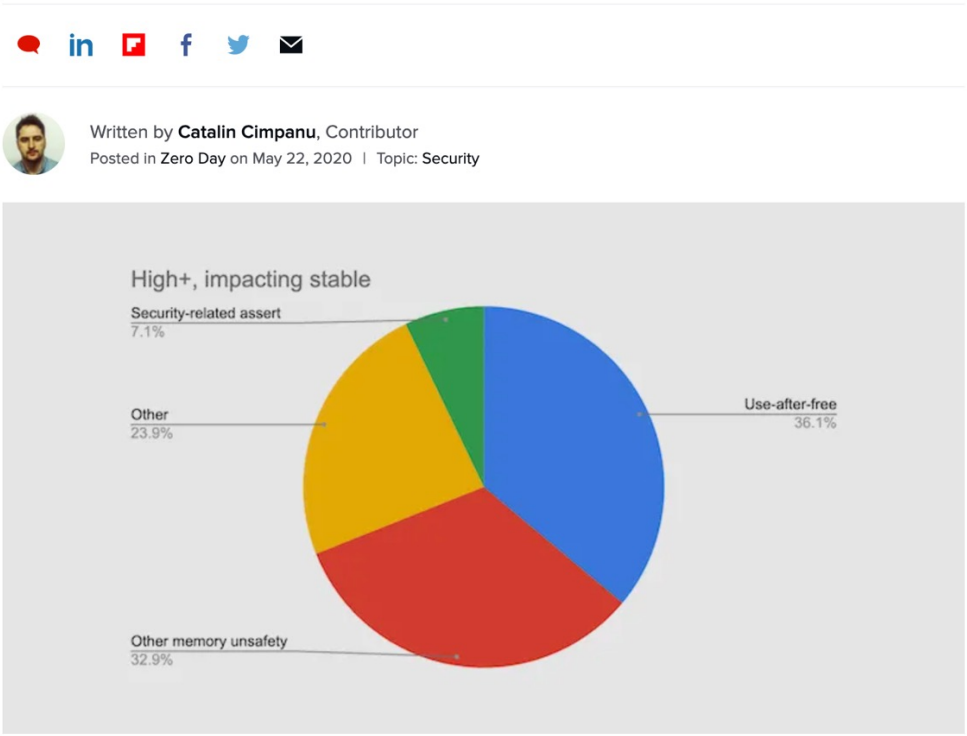
How does the pointer to the original memory allocation get lost?

https://www.softwareverify.com/blog/the-nineteen-types-of-memory-leak/

Heap usage is Very Important and relevant!

Chrome: 70% of all security bugs are memory safety issues

Google software engineers are looking into ways of eliminating memory management-related bugs from Chrome.



Written by **Catalin Cimpanu**, Contributor
Posted in **Zero Day** on May 22, 2020 | Topic: **Security**

RELATED


The best remote desktop software: Work from anywhere


The best security keys: Protect your online accounts

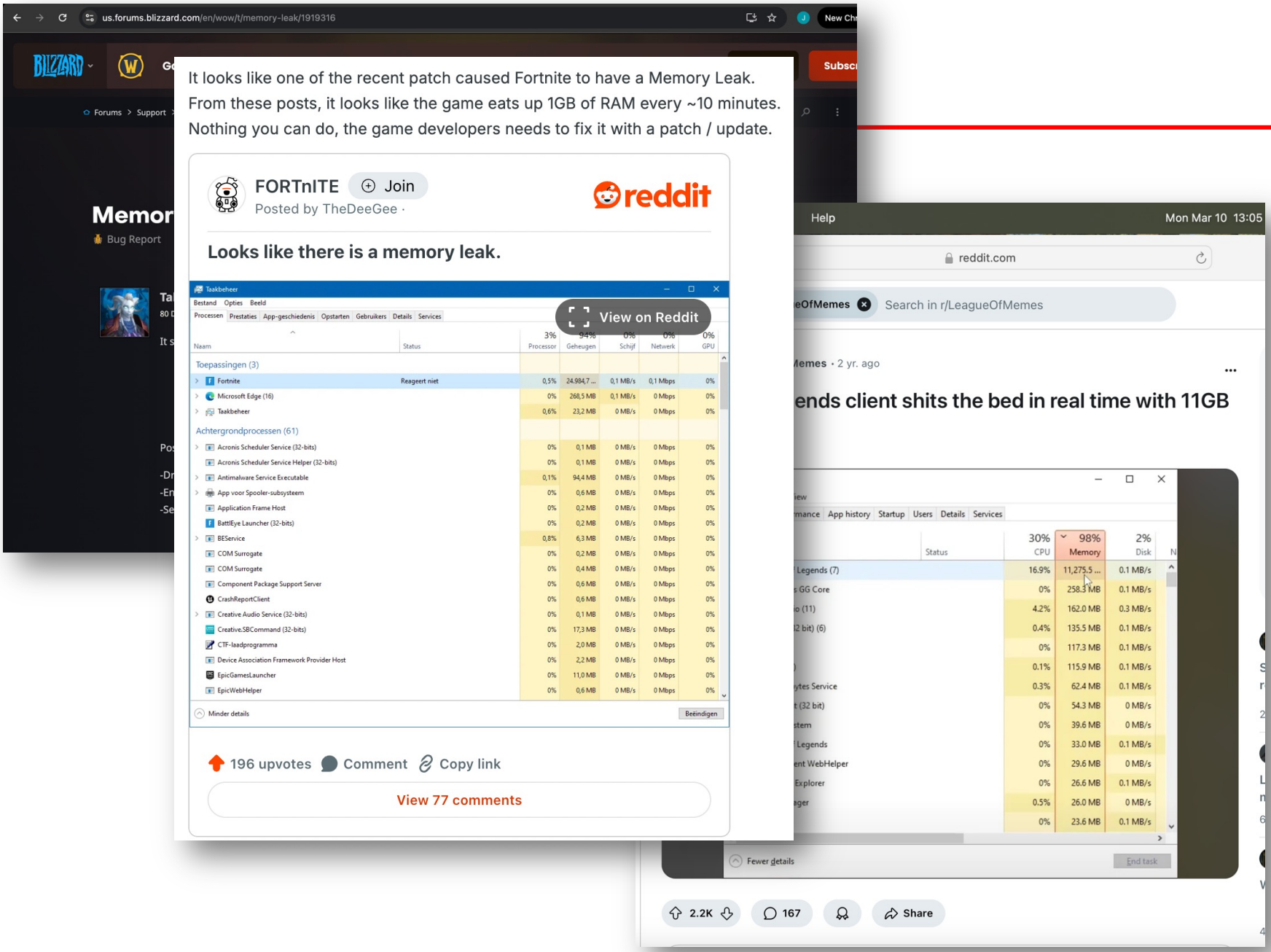

Best e-signature software 2022: Box, Docusign, and more

NEWSLETTERS

ZDNet Security

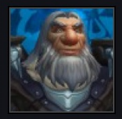
Your weekly update on security around the globe, featuring research, threats, and more.

Your email address SUBSCRIBE


us.forums.blizzard.com/en/wow/t/memory-leak/1919316
Forums > Support
Memor
Bug Report
It looks like one of the recent patch caused Fortnite to have a Memory Leak.
From these posts, it looks like the game eats up 1GB of RAM every ~10 minutes.
Nothing you can do, the game developers needs to fix it with a patch / update.
FORTnITE
Join
Posted by TheDeeGee ·
reddit
Looks like there is a memory leak.
View on Reddit
196 upvotes
Comment
Copy link
View 77 comments
Help
Mon Mar 10 13:05
reddit.com
Search in r/LeagueOfMemes
2 yr. ago
ends client shits the bed in real time with 11GB
2.2K
167
Share

Memory leak getting worse every patch!

Bug Report



Cannir <Survivors of Caerbannog> 22 posts

80 Dwarf Shaman 21170

Mar 2

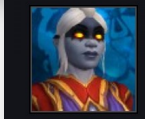
I have figured out how to easily get the game to use up all my RAM:

There is a world quest near dornagal that requires you to kill a bunch of nerubians and collect some tools, and another near the weavers lair that requires you to kill a bunch of nerubians. Both reward a little more than 800 gold.

1. Fly to the one with the tools and do it first
2. Fly back to doragol and use the portal to get to the weavers lair, then do the quest near her lair.
3. log out of the character and into another one that has not done the quests.
4. fly to the quest with the tools and do it.
5. fly back to dornagol and use the portal.

As soon as you try to use the portal on the second character the game will lock up and eat up all the RAM.

I did the above six times and it always ran out of RAM at the same point. I watch the game in task manage and as soon as the portal was entered on the second character the RAM it used started increasing very quickly.



Pawgwalker 1847 posts

10 Dark Iron Dwarf Mage 30

Mar 20

Sounds like either an addon(I'd assume the client has a max addon memory usage cap?), GPU driver issue or a non-Windows Defender security suite messing with Windows' ability to manage the games memory correctly.

If this game were really having a memory leak issue, there'd be hundred post threads left and right. There was one around the DF prepatch that involved the appearances collection, but they fixed it pretty quickly. Oh and there was one that involved crafting that would hit people that were crafting for hours and hours on end (I want to say the datatables were storing uncompressed icons or something along those lines?). Pretty sure it either got fixed or mitgated to where it wouldn't affect the 99.9% of players that don't sit around crafting thousands of items for hours on end without ever triggering a loading screen to clear the history cache.

https://us.forums.blizzard.com/en/wow/t/memory-leak-getting-worse-every-patch/2068483

How to protect?

- Be responsible with your memory usage!
  - Watch your usage of [m,c,re]alloc and free
- Use the stack when possible OR avoid using the heap!
- Create a higher level and/or alt language:
  - Approach in C++: new, delete keywords/operations
  - Approach in Java, Python, Golang, etc…: only let users make things through higher-level abstractions and use a ***garbage collector***…which prowls around and watches your heap usage and deletes unused stuff…
    - however this takes up resources!
  - Rust: Bakes "ownership" of memory into the language itself so that it is nearly impossible to lose track of heap:
    - Makes language "weird"
  - Zig: Has things like Allocators to help with memory safety

Why have stack and heap?

- The LIFO nature of the stack forces its usage to be smaller scale. It is also quick to access because it grows and contracts throughout the program.
- A stack way of structuring memory also forces things that get used together to be near each other in memory...this has great benefits since memory accesses are expensive!
- Larger data that you need to keep around outside function calls (beyond local scope) should go on the heap:
  - Create when you want!
  - Delete when you want!

Benefit of Stack

- Generally variables in a function are associated with each other...the stack keeps these things localized near each other in memory.
- The closer things are in memory, the (in general) faster they are to access.
  - This is because most modern memory takes a variable amount of time to access, but you can often access a large chunk at once easily (so some of the delay gets canceled out by getting a lot at once)
  - This is only beneficial if grabbing a continuous chunk of memory at once gets a lot of relevant data (and it does with a stack)
- Very cache friendly

What is “better”? Stack or Heap?

- To allocate stack space, how many instructions are needed?
  - Just one: `addi sp, sp, -8 # allocate 8 bytes of stack`
- Heap allocation takes a lot longer:
  - Has to search for available spots
  - Then determine if spots are available
  - Then get it…etc…
  - This takes time!
- Heap usage is also much more dangerous and prone to bugs!

THE END
(of content)

What is Next after 6.1903?

Where to go now (Classes)?

- Ok, you've messed with C and assembly for six weeks…where do you go from here?
- **6.191(6.004):** Actually go in and build a device that implements the instructions that we've been studying…
- **6.1060(6.172):** Dive even deeper into C and Assembly to make super efficient code
- **6.192(6.175):** go even further with computer architecture
- **6.205(6.111):** Develop algorithms in hardware on Field Programmable Gate Arrays (FGPAs)…also doing 6.S965 again this fall that's a follow-on to 6.205…goes even further into FPGAs/SOCs
- **6.206(6.115):** Do embedded microcontroller work
- **6.110 (6.035):** Computer Language Engineering…make compilers and related things
- **6.5950/1:** Secure Hardware Design...security weaknesses in architecture
- **6.S894:** Accelerated Computing: Programming with GPU cores
- Many more! Email or ask!!!

Where to go now (Hobbies)?

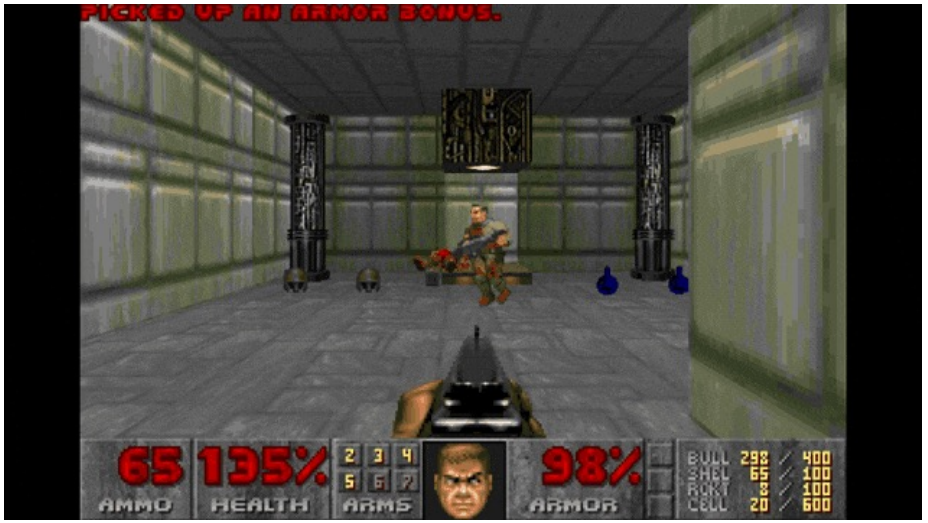
- Wanna mess with C in general?:
  - Build games with raylib (super nice basic graphics package with very nice C bindings)
- Wanna mess with RISC-V?:
  - Get an ESP32-C3 ($10)+ toolchain is free!
  - Use a RISC-V emulator!
- Wanna do more embedded stuff?:
  - Get an ESP32-C3 ($10)
- Other fun things (other languages!)?:
  - Look into the Rust Programing Language:
    - Tries to fix a lot of issues with C, particularly with memory
    - Weird(er) syntax
  - Zig programming language
  - Web Assembly also very cool...assembly language...for the web

Exciting Time in Hardware

- The last few years have been really exciting times for low-level stuff and hardware design!
- There's no clear roadmap about what is next and a lot of current technologies that got us where we are have maxed out so the field is searching in other directions.
- What a "computer" is and the scale of the computer has also changed and continues to change
- A lot of "this-is-how-it-is-done" stuff is now being questioned:
  - Traditional memory approaches
  - Digital vs. analog

More For Funs

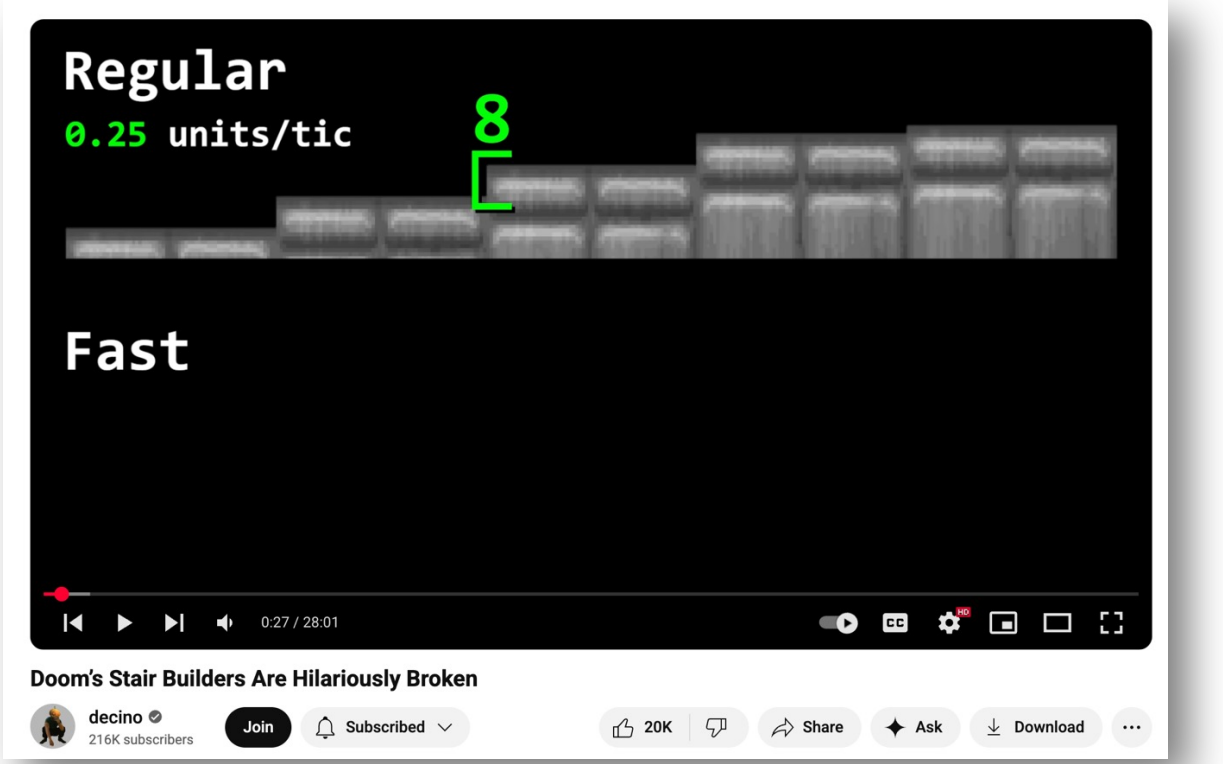
- The famous game Doom/Doom 2 was written all in C and all of its source code was publicly released:
  - https://github.com/id-Software/DOOM

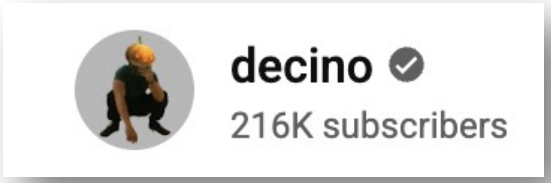


- At this point, you basically have enough knowledge to go through it and understand it*

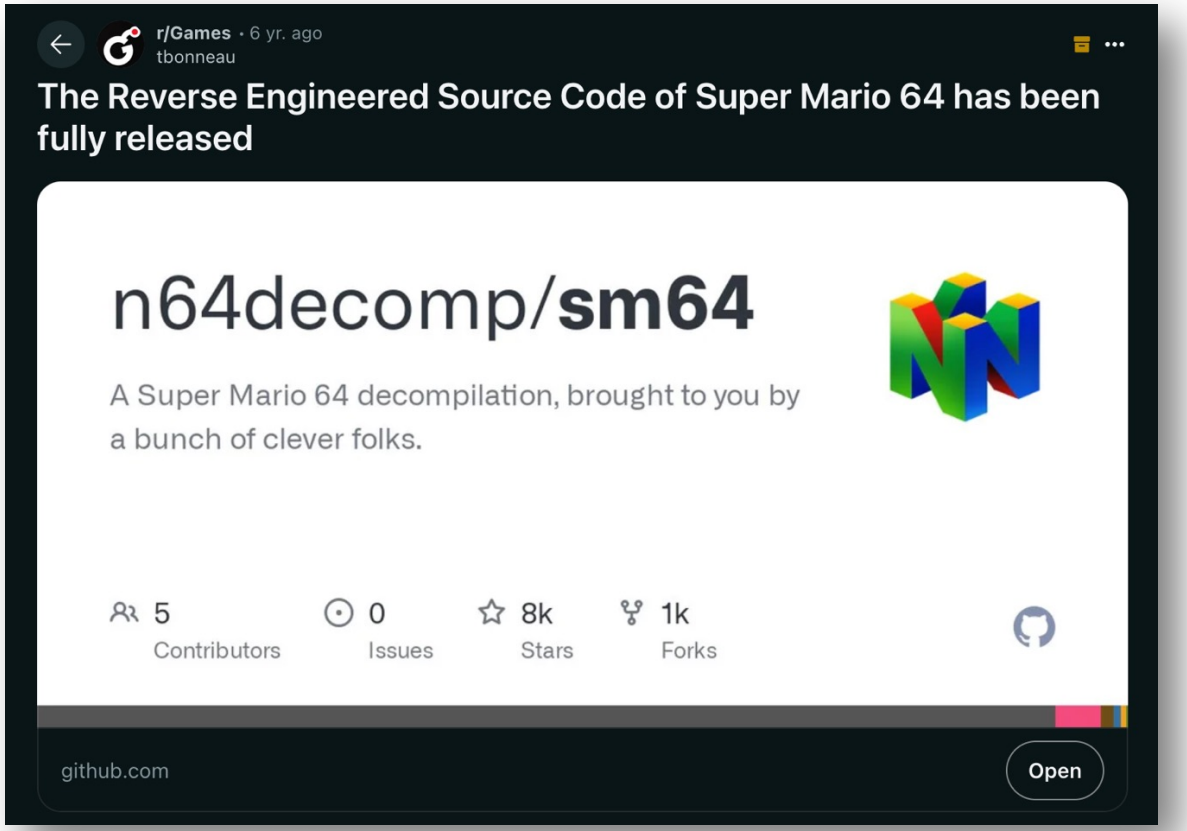
Lots of Great Analyses of the Code

- People go deeeeeeep on youtube et al. about how the game works
- This is also how speed runners find bugs





SuperMario64 Source Code

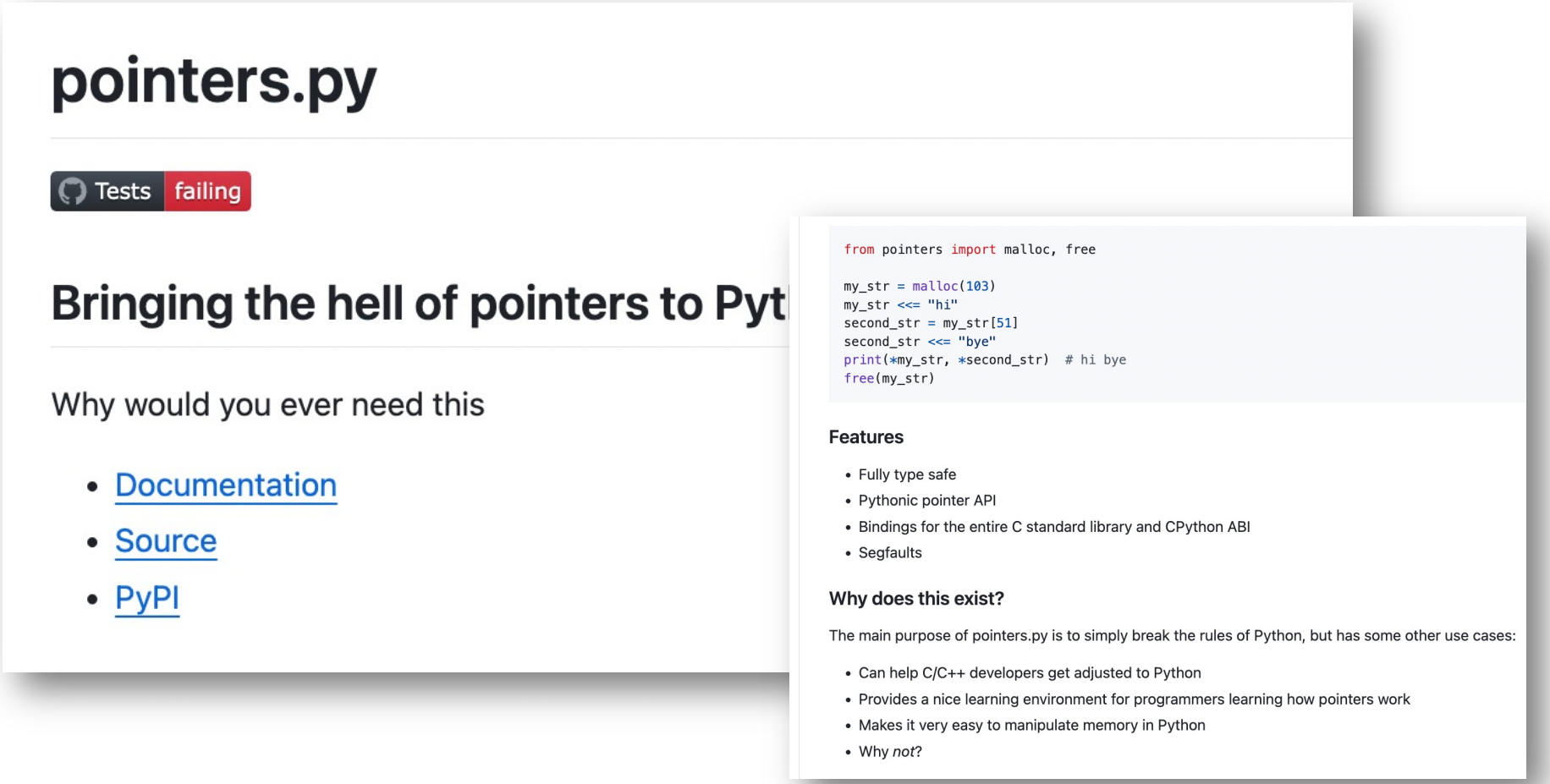


- Pretty Cool
- https://github.com/n64decomp/sm64/tree/master


Pannenkoek2012
Exploiting glitches, providing guides, and doing the impossible. With style.
pannenkoek2012
@pannenkoek2012 · 308K subscribers · 430 videos
I make videos about Super Mario 64. ...more
Subscribed
Join
Every Goomba Glitch in Super Mario 64
0:27 / 2:24:12 • Movement Decisions
Every Goomba Glitch in Super Mario 64

No real purpose other than learning/fun:

https://github.com/ZeroIntensity/pointers.py



pointers.py

Tests failing

Bringing the hell of pointers to Pyt

Why would you ever need this

- Documentation
- Source
- PyPI

```
from pointers import malloc, free

my_str = malloc(103)
my_str <<= "hi"
second_str = my_str[51]
second_str <<= "bye"
print(*my_str, *second_str)  # hi bye
free(my_str)
```

Features

- Fully type safe
- Pythonic pointer API
- Bindings for the entire C standard library and CPython ABI
- Segfaults

Why does this exist?

The main purpose of pointers.py is to simply break the rules of Python, but has some other use cases:

- Can help C/C++ developers get adjusted to Python
- Provides a nice learning environment for programmers learning how pointers work
- Makes it very easy to manipulate memory in Python
- Why *not*?

Administrative

- Recitation 6 is tomorrow!
- Exercise 6 this week!
- No lab this week.
- Next Monday (11/16): Third and final quiz
- Wednesday (11/18): Lab 6 (final lab).
- Then...Exam 2 on Thursday December 3
- Then done (make sure to return lab kits after)