LEARN

C

FROM THE TRENCHES

Carsten Haitzler
<raster@rasterman.com>
http://www.rasterman.com
http://www.enlightenment.org

How will we do this?

- Interactivity is key
  - Ask questions
  - Don’t stay confused or unsure
    - Ask until you get it
  - Ask stuff
  - Ask
  - Try things
    - Get your laptop out and ready to do things
    - If things don’t work – ASK
  - Make comments
    - Leads to feedback, questions, answers or corrections

The Open Source

http://www.rasterman.com/files/fossasia-2018-c-from-the-trenches-source.tar.gz

Why C?

- One of the most common languages for system software
- Old & a basis for many languages to build on
- Linux Kernel
- Open source middleware and libraries
- Embedded
- Teaches you how a machine works
- It can be a fun challenge
- It does not mean everything has to be done in C
  - But anything can pretty much be done in it
  - No limits on peformance or compactness


You are very un-hipster
DEATH
TO
HIPSTERS

Why listen to me?

- Well over 20 years of full-time C experience
  - Mostly userspace
  - Almost all on Linux
  - Written something on the order of a million lines of C
- ~6-7 years of assembly
  - M68k though…. some x86 MMX/SSE and ARM

Why listen to me?

- Well over 20 years of full-time C experience
  - Mostly userspace
  - Almost all on Linux
  - Written something on the order of a million lines of C
- ~6-7 years of assembly
  - M68k though…. some x86 MMX/SSE and ARM
- Old and grumpy

Why listen to me?

- Well over 20 years of full-time C experience
  - Mostly userspace
  - Almost all on Linux
  - Written something on the order of a million lines of C
- ~6-7 years of assembly
  - M68k though…. some x86 MMX/SSE and ARM
- Old and ~~grumpy~~ wise

What you need today

- A terminal/shell
- A text editor
  - Install emacs | emacs-nox | vi | vim | nano | jed | eclipse |…
- A compiler
  - Install gcc | clang

What you need today

- A terminal/shell
- A text editor
  - Install emacs | emacs-nox | vi | vim | nano | jed | eclipse |…
- A compiler
  - Install gcc | clang
- Your tap-dancing shoes

Let’s get cracking

```
#include <stdio.h>
int main() {
   printf("Hello World\n");
}

$ cc source.c -o source && ./source
Hello World
$
```

Let’s get cracking

```
#include <stdio.h>
int main(int argc, char **argv) {
   printf("Hello %s\n", argv[1]);
}

$ cc source.c -o source && ./source Bob
Hello Bob
$
```

Let’s get cracking

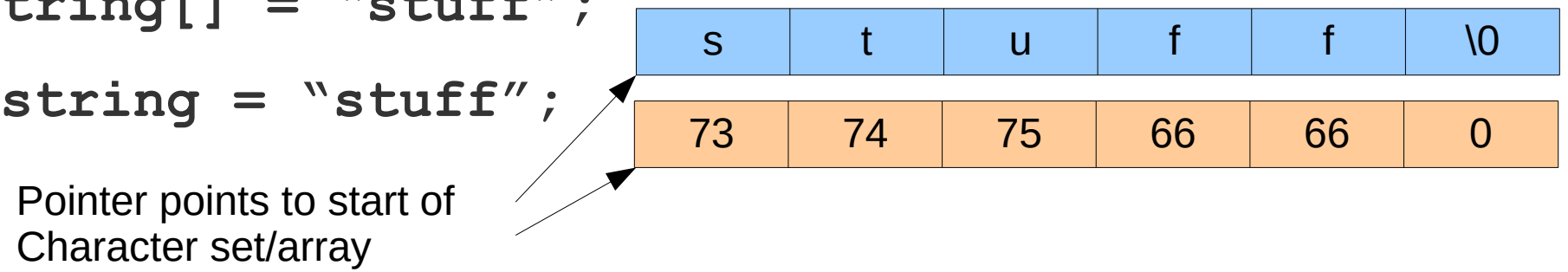
```
#include <stdio.h>
int main(int argc, char **argv) {
   printf("Hello %s. %i: ", argv[0], argc);
   for (int i = 1; i < argc; i++) {
      printf("%s ", argv[i]);
   }
   printf("\n");
}
```

```
$ cc source.c -o source && ./source Bob Eliza Jill
Hello ./source. 4: Bob Eliza Jill
$
```

Types

- **`int`** – An integer (normally 32bits -2147483648 – 2147483647)
  - If unsigned 0 – 4294967295, or 0x00000000 - 0xffffffff
- **`char`** – A character (normally 8bits -128 – 127)
  - If unsigned 0 – 255, or 0x00 – 0xff
  - Often used to store string characters (a-z, A-Z, 0-9, etc.)
    - **`man ascii`**
- **`char * / char []`** - a pointer to chars / an array or chars

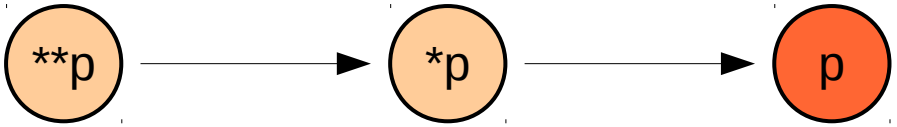
```
char string[6] = { 's', 't', 'u', 'f', 'f', '\0' };
char string[] = { 's', 't', 'u', 'f', 'f', 0 };
char string[] = "stuff";
char *string = "stuff";
```



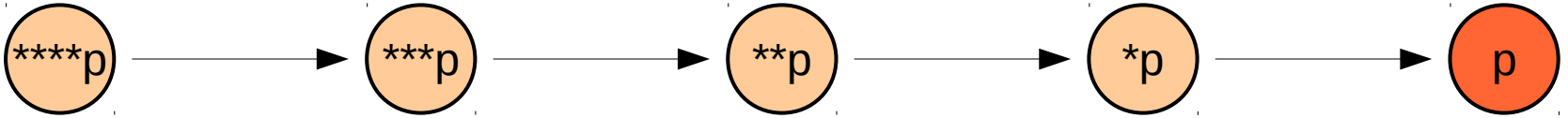
Pointers

- * - pointer to something

- ** - pointer to pointer



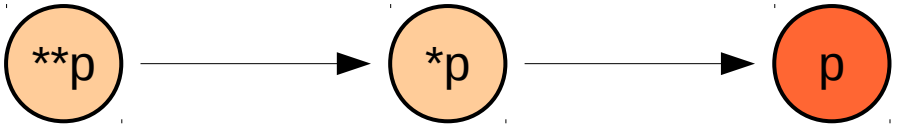
- …



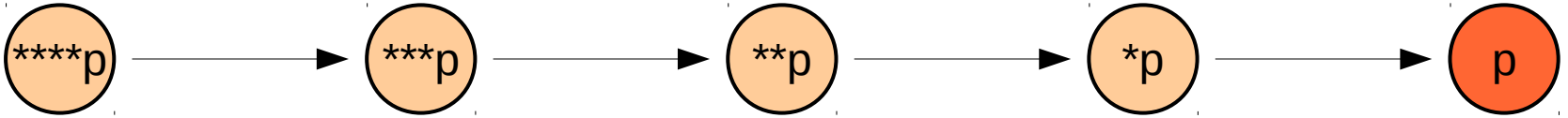
Pointers

- * - pointer to something

- ** - pointer to pointer



- …

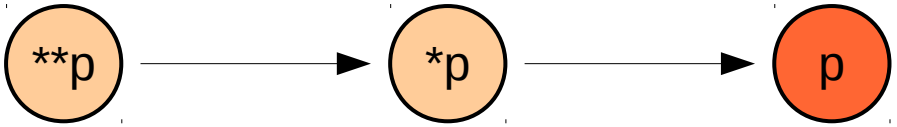


- It never ends!

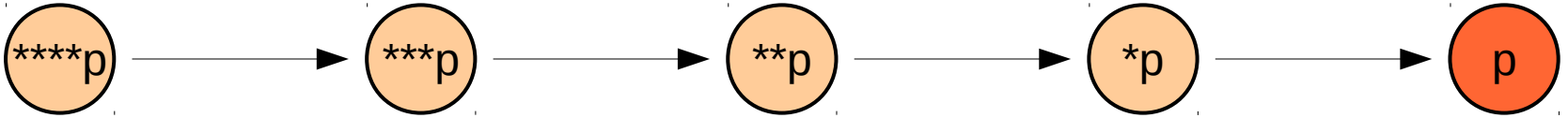
Pointers

- * - pointer to something

- ** - pointer to pointer



- …



- It never ends!
- Be aware that every pointer de-reference needs to access memory
  - Must read pointer value to use it

printf

- Print to standard output (stdout)
- “%XXX” - replaced by the matching argument
- “%i” - integer
- “%s” - string (char * / char[])
- “\n” - newline

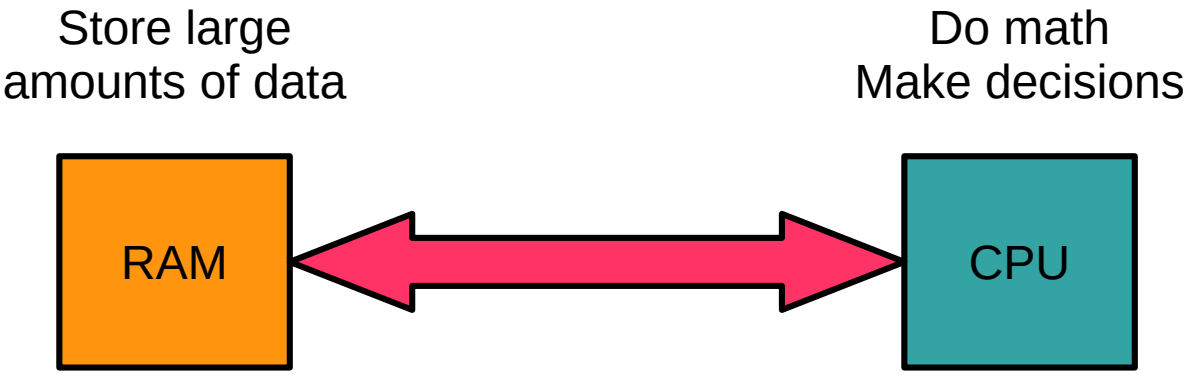
For loops

- Iterate over a set of things
  - Can use any number of start states
  - Can use many “still loop” conditions
  - Can use many iteration steps

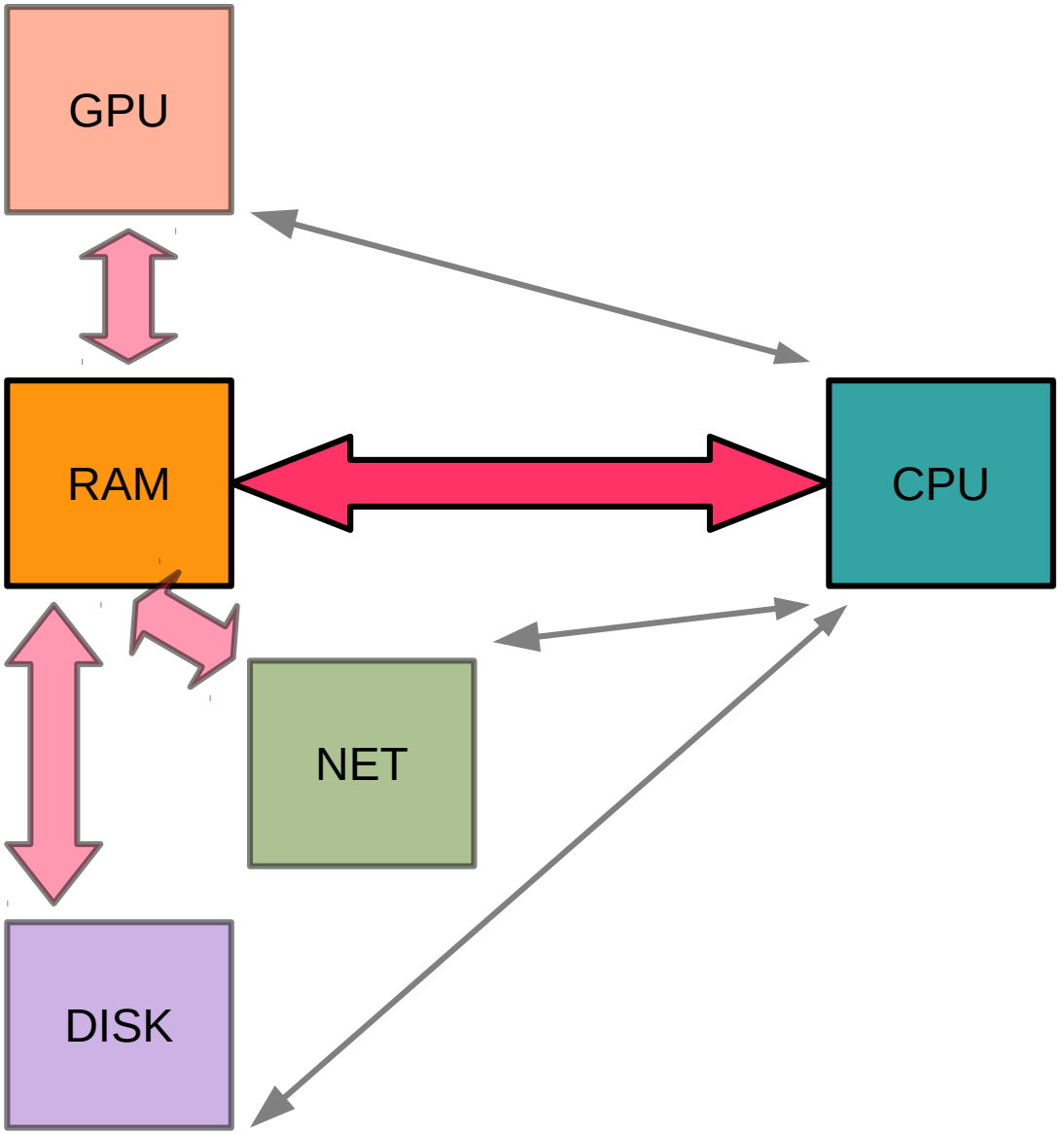
```
for (i = 0; i < 10; i++) {
   printf(“%i\n”, i);
}
```

```
for (i = 0, j = 10; (i < 10) && (i > 5); i++, j--) {
   printf(“i=%i j=%i\n”, i, j);
}
```

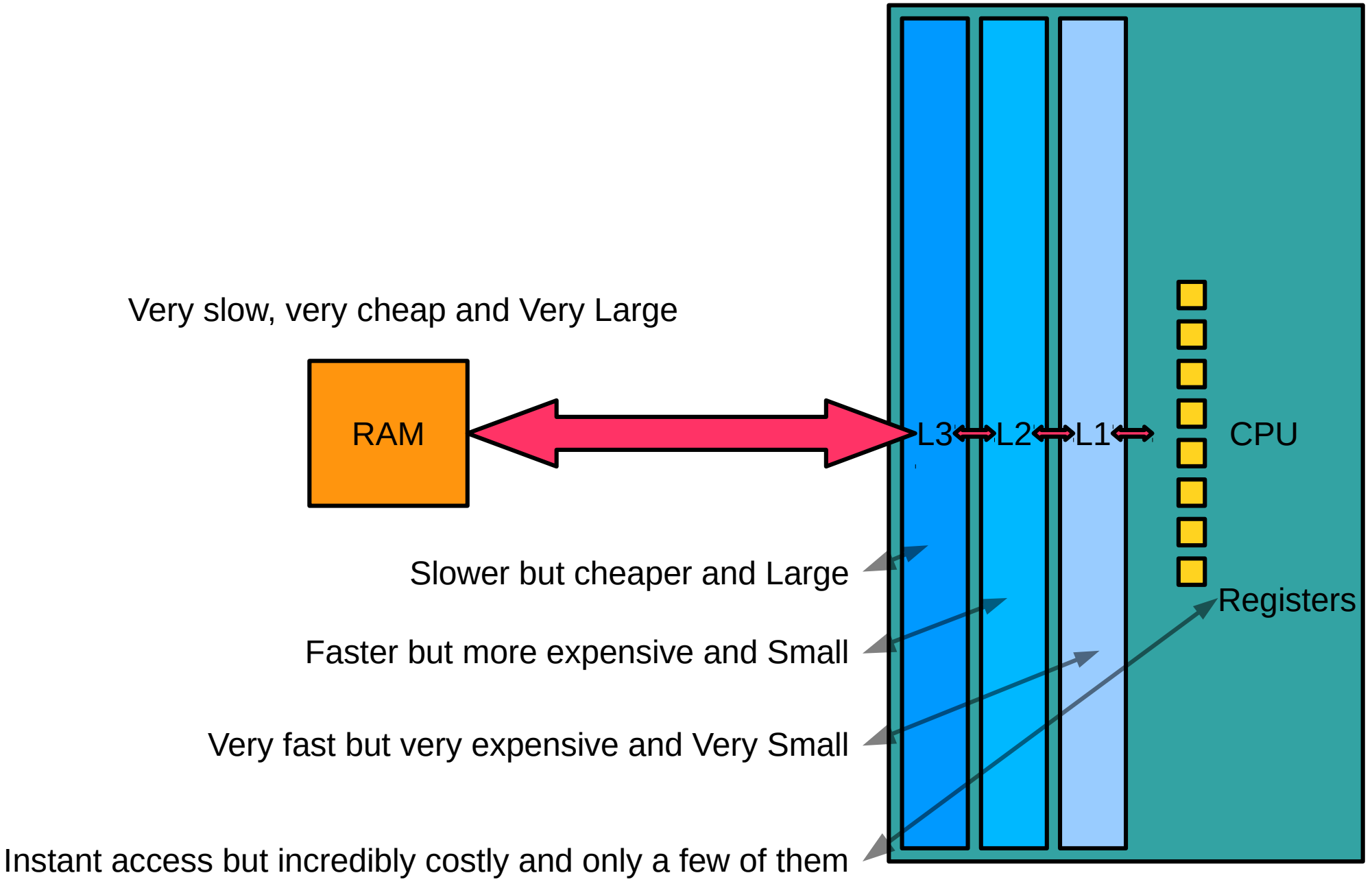
Machine concepts



Machine concepts



Machine concepts



Key core concepts

- Data lives in RAM
  - Moves from RAM to CPU registers and back as needed
    - CPU generates it
    - Disk
    - Other system devices (GPU, Network etc.)
  - Ways of speeding up RAM access
    - L1, L2, L3 Caches (invisible)
- Processor
  - Has a few registers for workspace
  - CPU reads and writes data
    - Instructions are just data
  - Does math
  - Makes logical decisions

What is data?

NUMBERS

Where is data?

NUMBERS

Numbers in the digital world

There is only 0 & 1

Numbers in the digital world

```
  0 = 0
  1 = 1

 00 = 0
 01 = 1
 10 = 2
 11 = 3

000 = 0
001 = 1
010 = 2
011 = 3
100 = 4
101 = 5
110 = 6
111 = 7
```

Hex – more compact

```
 0 = 0
 1 = 1
 2 = 2
 3 = 3
 4 = 4
 5 = 5
 6 = 6
 7 = 7
 8 = 8
 9 = 9
10 = a
11 = b
12 = c
13 = d
14 = e
15 = f
```

```
da8f

   d-   a-   8-   f
1110-1100-1000-1111

  (15(f)              ) +
  ( 8(8)*16           ) +
  (10(a)*16*16        ) +
  (13(d)*16*16*16     )

= 55951
```

What is Memory?

A lot of boxes with numbers in them

1c	01	24	1f	d0	d7	00	12	8f	ff	f0	00	00	01	8d	f3
ff	ff	31	07	00	00	00	01	00	00	00	01	32	37	3a	6c
ff	64	61	5f	63	2e	64	00	72	65	73	69	65	78	64	8f
6e	69	00	74	5f	5f	5e	47	40	66	65	5f	4f	46	41	4e

1 register = 4 or 8 bytes (SIMD 8, 16, … maybe 64)
(CPU has ~4, 8, 16, maybe 8 or 16 SIMD registers)

1K = 1024 bytes
1M = 1,048,576 bytes
1G = 1,073,741,824 bytes

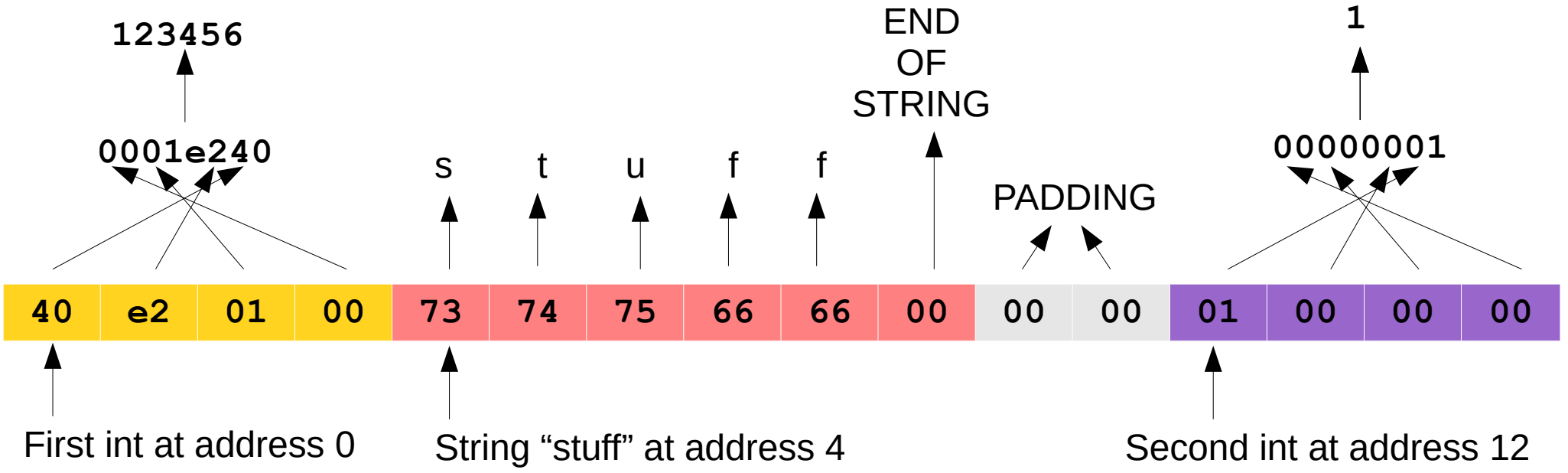
Powerful Server = 137,438,953,472 bytes
Powerful PC (16GB) = 17,179,869,184 bytes
Powerful Phone (4GB) = 4,294,967,296 bytes

Data types (generally)

char	1 byte
short	2 bytes
int	4 bytes
long	4 or 8 bytes
long long	8 bytes
float	4 bytes
double	8 bytes
XXX * (pointer)	4 or 8 bytes

Pointers

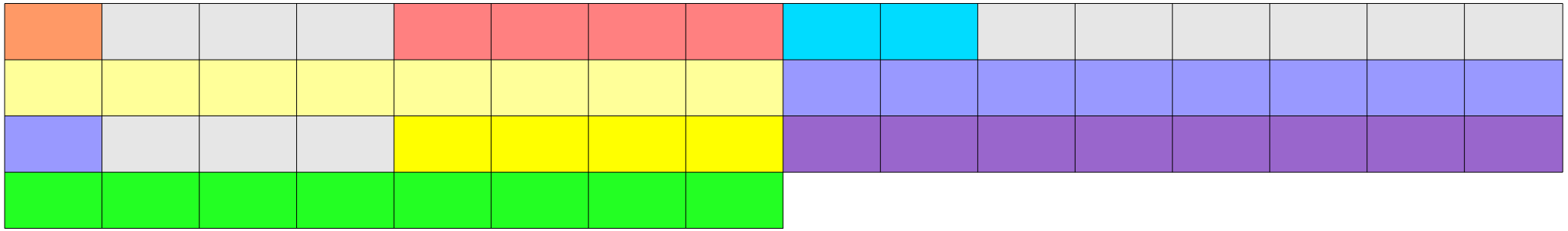
- Are just a number
- Indicate the starting location for the data
- Order stored depends on endianess
  - Big endian → MSB stored before LSB (MIPS, Sparc, PPC, m68k)
  - Little endian → LSB stored before MSB (x86, ARM)



Structs

- Define memory layout with standard types

```
struct mydata {
    char val0;
    int val1;
    short val2;
    char *val3;
    char val4[9];
    int val5;
    long long val6;
    struct mydata *val7;
};
```



Code

```
#include <stdio.h>
struct mydata {
   char val0;
   int val1;
   short val2;
   char *val3;
   char val4[9];
   int val5;
   long long val6;
   struct mydata *val7;
};
int main(int argc, char **argv) {
   struct mydata data = {
      123, 99, 101, argv[0], "stuff it", 131313, 88, &data
   };
   unsigned char *ptr, *endptr;
   for (ptr = (unsigned char *)&data, endptr = ptr + sizeof(data);
        ptr < endptr; ptr++) {
      printf("%02x ", *ptr);
   }
   printf("\n");
}
```

Data Structs

```
$ cc s4.c -o s4
$ ./s4
```

Result

```
7b 00 00 00 63 00 00 00 65 00 94 70 70 55 00 00

be 9f 21 e4 fd 7f 00 00 73 74 75 66 66 20 69 74

00 47 94 70 f1 00 02 00 58 00 00 00 00 00 00 00

10 97 21 e4 fd 7f 00 00
```

Result

```
7b 00 00 00 63 00 00 00 65 00 94 70 70 55 00 00

be 9f 21 e4 fd 7f 00 00 73 74 75 66 66 20 69 74

00 47 94 70 f1 00 02 00 58 00 00 00 00 00 00 00

10 97 21 e4 fd 7f 00 00
```

Result

```
7b 00 00 00 63 00 00 00 65 00 94 70 70 55 00 00
123         99          101
be 9f 21 e4 fd 7f 00 00 73 74 75 66 66 20 69 74
Mem addr of argv[1]     s  t  u  f  f     i  t
00 47 94 70 f1 00 02 00 58 00 00 00 00 00 00 00
            131313      87
10 97 21 e4 fd 7f 00 00
Mem addr of this data
```

Memory locations

- Stack
  - When a function/scope is “called”, local variables pushed
  - When function/scope exits, that same region popped off stack
  - The more data on stack, the more space stack needs
  - The more functions that call functions, the more space needed
  - Can allocate dynamically at runtime with alloca()
- Heap
  - Everything else but stack
  - Normally gain memory via malloc() and return via free()
    - Also can use calloc(), realloc() and mmap()
  - Have to explicitly request and release
    - Forget to release and your process will leak
      - Leaks may eventually result in running out of memory

Code

```
#include <stdio.h>
#include <unistd.h>
struct mydata {
   char val0;
   int val1;
   short val2;
   char *val3;
   char val4[9];
   int val5;
   long long val6;
   struct mydata *val7;
};
int main(int argc, char **argv) {
   struct mydata data = {
      123, 99, 101, argv[0], "stuff it", 131313, 88, &data
   };
   unsigned char *ptr, *endptr;
   for (ptr = (unsigned char *)&data, endptr = ptr + sizeof(data);
        ptr < endptr; ptr++) {
      printf("%02x ", *ptr);
   }
   printf("\n");
   pause();
}
```

Mappings

```
$ cc s5.c -o s5
$ ./s5
```

Memory layout

```
$ pmap `pidof s5`
30228:   ./s5
000055a9d908b000      4K r-x-- s5
000055a9d928b000      4K r---- s5
000055a9d928c000      4K rw--- s5
000055a9da0d1000    132K rw---   [ anon ]
00007f66a648b000   1720K r-x-- libc-2.26.so
00007f66a6639000   2044K ----- libc-2.26.so
00007f66a6838000     16K r---- libc-2.26.so
00007f66a683c000      8K rw--- libc-2.26.so
00007f66a683e000     16K rw---   [ anon ]
00007f66a6842000    148K r-x-- ld-2.26.so
00007f66a6a64000      8K rw---   [ anon ]
00007f66a6a66000      4K r---- ld-2.26.so
00007f66a6a67000      4K rw--- ld-2.26.so
00007f66a6a68000      4K rw---   [ anon ]
00007ffed78aa000    136K rw---   [ stack ]
00007ffed7936000     12K r----   [ anon ]
00007ffed7939000      8K r-x--   [ anon ]
 total             4272K
```

Stack grows in one direction

- May gow up or down depending on platform
  - Call function → **`sp = sp + 16; (sp += 16)`**
  - End function → **`sp = sp - 16; (sp -= 16)`**
  - 16 (size) may vary per function
    - Includes parameters passed to function
    - Includes space for return
    - Includes space for local variables

Code

```
#include <stdio.h>
void dumpmem(unsigned char *from, unsigned char *to) {
   unsigned char *ptr = from, *endptr = to;
   int i;
   printf("start=%p end=%p\n", from, to);
   for (i = 0; ptr < endptr; ptr++, i++) {
      printf("%02x ", *ptr);
      if (i == 15) {
         i = -1;
         printf("\n");
      }
   }
   if (i != 0) printf("\n");
}
int myfunc(int param, int count, unsigned char *top) {
   int localdata = 0x77777777;
   if (count == 0) {
      dumpmem((unsigned char *)&localdata, top);
      return 0x88888888;
   }
   return myfunc(param + 1, count - 1, top);
}
int main(int argc, char **argv) {
   int topdata = 0x99999999;
   int ret;
   ret = myfunc(0x11111100, 3, (unsigned char *)&topdata);
   printf("ret = %x\n", ret);
}
```

Stack

```
$ cc s6.c -o s6
$ ./s6
```

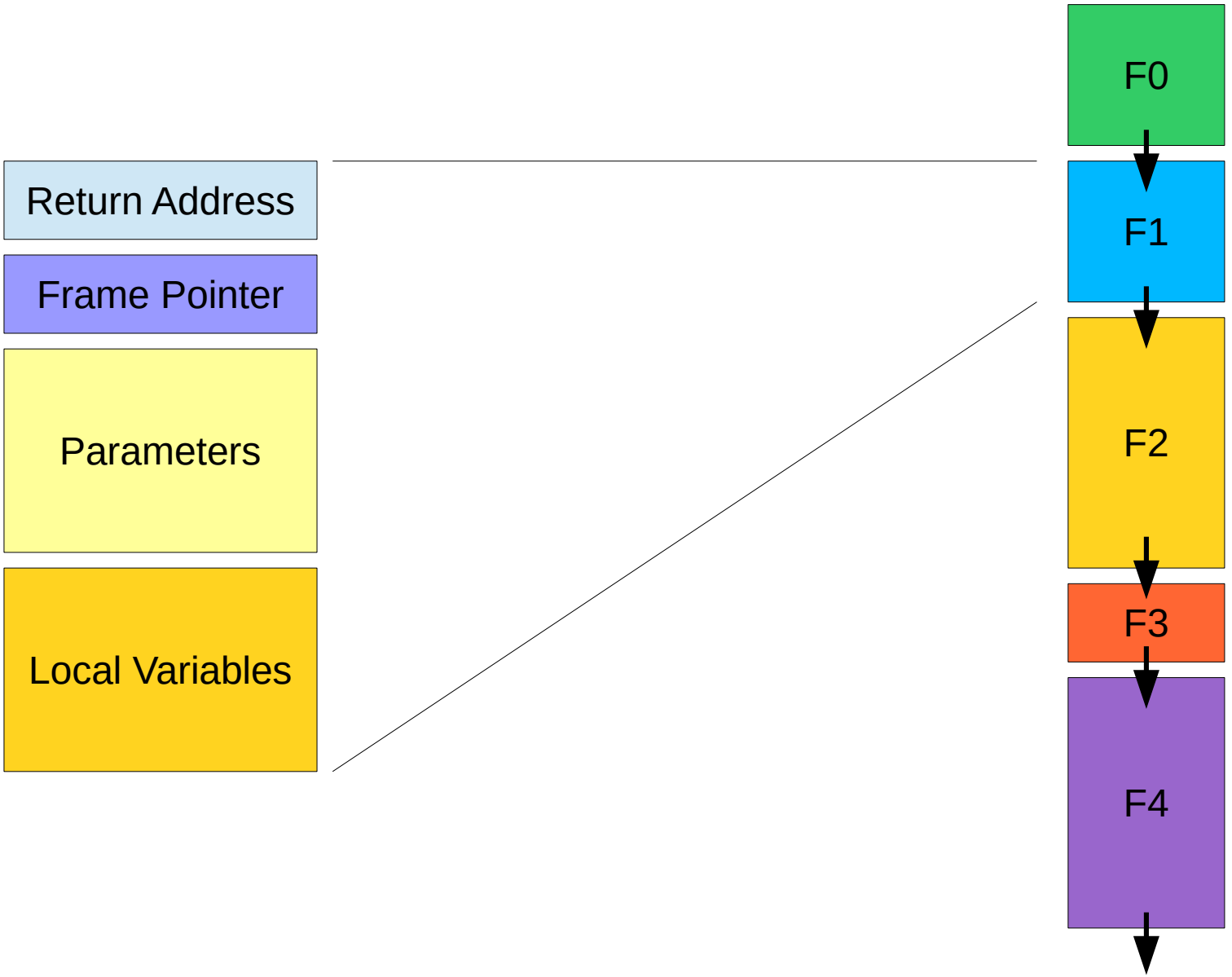
Output

```
start=0x7fffd3bb3794 end=0x7fffd3bb3850
77 77 77 77 00 02 c8 96 ab 44 52 7d d0 37 bb d3
ff 7f 00 00 0b e8 91 db 5c 55 00 00 50 38 bb d3
ff 7f 00 00 01 00 00 00 02 11 11 11 00 00 00 00
77 77 77 77 00 02 c8 96 ab 44 52 7d 00 38 bb d3
ff 7f 00 00 0b e8 91 db 5c 55 00 00 50 38 bb d3
ff 7f 00 00 02 00 00 00 01 11 11 11 00 00 00 00
77 77 77 77 00 02 c8 96 ab 44 52 7d 30 38 bb d3
ff 7f 00 00 0b e8 91 db 5c 55 00 00 50 38 bb d3
ff 7f 00 00 03 00 00 00 00 11 11 11 01 00 00 00
77 77 77 77 00 02 c8 96 ab 44 52 7d 60 38 bb d3
ff 7f 00 00 5c e8 91 db 5c 55 00 00 48 39 bb d3
ff 7f 00 00 00 e6 91 db 01 00 00 00
ret = 88888888
```

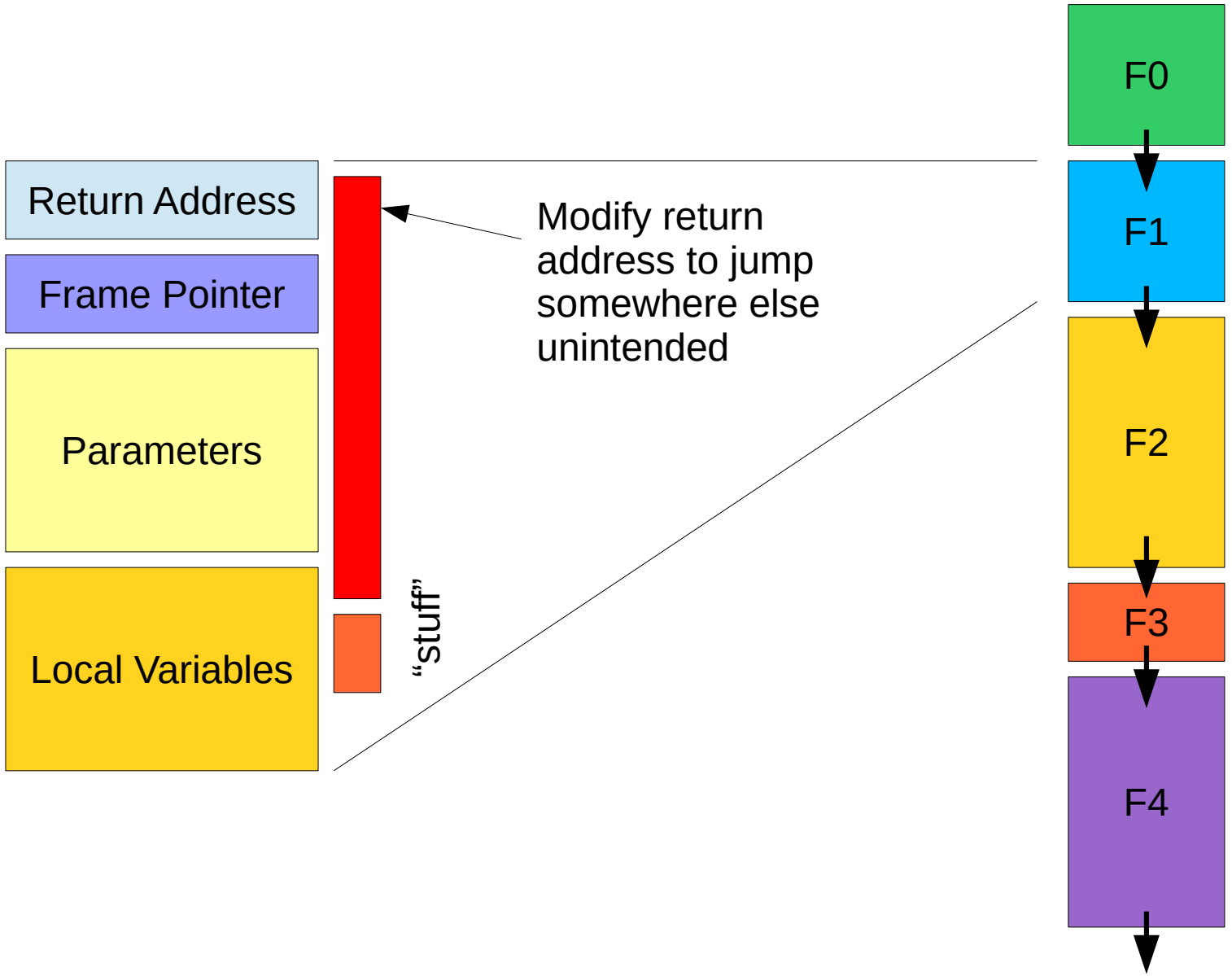
Output (Fixed alignment)

```
start=0x7fffd3bb3794 end=0x7fffd3bb3850
                77 77 77 77 00 02 c8 96 ab 44 52 7d
d0 37 bb d3 ff 7f 00 00 0b e8 91 db 5c 55 00 00
50 38 bb d3 ff 7f 00 00 01 00 00 00 02 11 11 11
00 00 00 00 77 77 77 77 00 02 c8 96 ab 44 52 7d
00 38 bb d3 ff 7f 00 00 0b e8 91 db 5c 55 00 00
50 38 bb d3 ff 7f 00 00 02 00 00 00 01 11 11 11
00 00 00 00 77 77 77 77 00 02 c8 96 ab 44 52 7d
30 38 bb d3 ff 7f 00 00 0b e8 91 db 5c 55 00 00
50 38 bb d3 ff 7f 00 00 03 00 00 00 00 11 11 11
01 00 00 00 77 77 77 77 00 02 c8 96 ab 44 52 7d
60 38 bb d3 ff 7f 00 00 5c e8 91 db 5c 55 00 00
48 39 bb d3 ff 7f 00 00 00 e6 91 db 01 00 00 00
ret = 88888888
```

Stack growing down



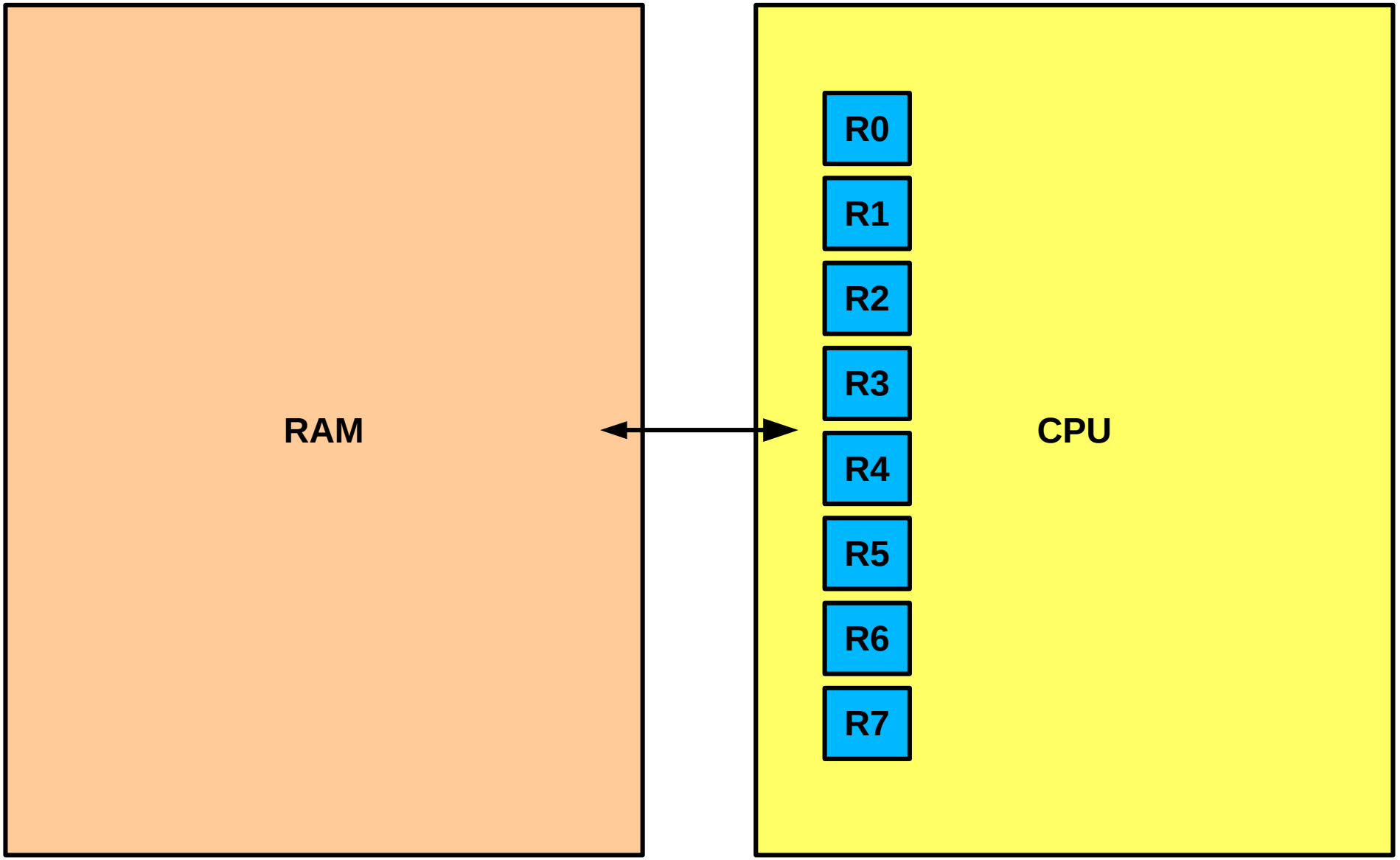
Aside: Buffer Overflows



Lies lies lies

- Not all parameters passed on stack
  - Passed in registers to save modifying/using stack
  - Registers may have to be saved to stack before calling new func
  - Return value may be passed in registers too

Registers



Registers?

- Locate “variables” inside the CPU
  - Often 32 or 64bits in size (SIMD may be 64, 128, 256 or 512)
  - Often only 4, 8 or 16 on a CPU
  - Instant access with no waiting
  - Generally data has to move from RAM to register to work on
    - Some instructions allow immediate constants
    - Some can work from a memory address to fetch data for op
- Best to think that all data has to be loaded from RAM
  - Then worked on in register
  - Then stored back to RAM
- Data may be entirely generated by CPU (so only stored to RAM)

Variables and Registers

- Logically variables and parameters live on the stack
  - In real life the compiler ASSIGNS a register to them
    - If it runs out of registers, it has to generate instructions to move data between stack (RAM) and register
    - This is invisible to you (compile takes are of it)
      - Compilers are quite smart and won't move data around if it is not absolutely needed
- Below will only use 1 register for i, j and k combined.

```
int i, j, k;

for (i = 0; i < 10; i++) printf("%i\n", i);

for (j = 0; j < 10; j++) printf("%i\n", j);

for (k = 0; k < 10; k++) printf("%i\n", k);
```

Heap Memory

- To store more “permanent” things
- Has no structure (just is a “heap” of memory)
- Every allocation needs to be tracked and freed when not needed
- Core C library (LibC) provides implementation and policy
  - **`malloc()`**
    - Allocate memory (content on allocation undefined)
  - **`calloc()`**
    - Allocate memory and guarantee it is all zero
  - **`realloc()`**
    - Resize an allocation retaining content
  - **`free()`**
    - Release memory previously allocated or resized by the above

malloc

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
   int i;
   char *content = "agEr5m3!e-";
   char *data = malloc(1024 * 1024);
   if (!data) {
      printf("ERROR: Cannot allocate memory.\n");
      abort();
   }
   for (i = 0; i < ((1024 * 1024) - 1); i++) {
      data[i] = content[i % 10];
   }
   data[i] = 0;
   printf("%s\n", data);
   free(data);
}
```

realloc

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    char *data = malloc(1024 * 1024);
    if (!data) {
        printf("ERROR: Cannot allocate memory.\n");
        abort();
    }
    char *bigger = realloc(data, 1024 * 1024 * 2);
    if (!bigger) {
        printf("ERROR: Cannot make chunk bigger.\n");
        abort();
    }
    free(bigger);
}
```

calloc

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char **argv) {
    char *data = calloc(1, 1024 * 1024);
    if (!data) {
        printf("ERROR: Cannot allocate memory.\n");
        abort();
    }
    free(data);
}
```

calloc vs. realloc/malloc

- Calloc guarantees content is 0
  - Not necessary if you are about to overwite all the memory
  - Very important for data structures to have guaranteed state
  - Suggest using calloc for all structure allocation
    - Unless you totally guarantee every member is set
  - Can optimize and not zero memory if it knows it already is zero

LibC

- Linked by default
- The most basic C library of function calls
  - Basic I/O (printf, scanf, fopen, fclose, fread, fwrite, etc.)
  - Memory management (malloc, calloc, realloc, free, etc.)
  - String handling (strcpy, strcat, strstr, strchr, strrchr, strlen, etc.)
  - POSIX system calls
    - Generally lower level than the above
      - Used inside above functions
    - open, close, read, write, fsync, kill, fcntl, ioctl, mmap. etc.
      - Almost all map to a real system call
      - Systems calls are expensive
        - Interrupt kernel, switch into kernel mode, switch back

1000’s of other C libraries

- EFL (data structures, I/O, GUI, GUI, rendering etc.)
- Glib (data structures, I/O)
- GTK+ (GUI)
- libJPEG (JPEG decoding + encoding)
- libPNG (PNG decoding + encoding)
- Zlib (compression, decompression)
- OpenSSL (encryption, decryption)
- Gstreamer (media playback, encoding)
- LibX11 (Raw access to X server protocol)
- libwayland-client/server (Raw access to wayland protocol)
- OpenGL (hardware 3D rendering)
- Freetype (Truetype font loading, rendering and querying)

File I/O

```
#include <stdio.h>
int main(int argc, char **argv) {
   char buf[1024];
   size_t count;
   FILE *infile = fopen(argv[1], "r");
   FILE *outfile = fopen(argv[2], "w");
   while (1) {
      count = fread(buf, 1, sizeof(buf), infile);
      if (count > 0) {
         printf("writing %i bytes...\n", (int)count);
         fwrite(buf, 1, count, outfile);
      } else {
         printf("end of file\n");
         break;
      }
   }
   fclose(infile);
   fclose(outfile);
}
```

File I/O

```
$ cc s7.c -o s7
$ ./s7 s7 newfile
writing 1024 bytes...
writing 1024 bytes...
writing 1024 bytes...
writing 1024 bytes...
writing 1024 bytes...
writing 1024 bytes...
writing 1024 bytes...
writing 1024 bytes...
writing 1024 bytes...
writing 1024 bytes...
writing 768 bytes...
end of file
```

Using another library to compress I/O

```
#include <stdio.h>
#include <stdlib.h>
#include <zlib.h>
int main(int argc, char **argv) {
   char buf[1024];
   size_t compress_bufsize = compressBound(sizeof(buf));
   char *compressed = malloc(compress_bufsize);
   uLongf destsize;
   size_t count;
   FILE *infile = fopen(argv[1], "r");
   FILE *outfile = fopen(argv[2], "w");
   while (1) {
      count = fread(buf, 1, sizeof(buf), infile);
      if (count > 0) {
         destsize = compress_bufsize;
         compress(compressed, &destsize, buf, count);
         fwrite(&destsize, 1, sizeof(destsize), outfile);
         printf("writing %i bytes...\n", (int)destsize);
         fwrite(compressed, 1, destsize, outfile);
      } else {
         printf("end of file\n");
         break;
      }
   }
   fclose(infile);
   fclose(outfile);
   free(compressed);
}
```

Compressed File I/O

```
$ cc s8.c -o s8 -lz
$ ./s8 s7 newfile
writing 313 bytes...
writing 544 bytes...
writing 585 bytes...
writing 164 bytes...
writing 460 bytes...
writing 367 bytes...
writing 335 bytes...
writing 471 bytes...
writing 251 bytes...
writing 195 bytes...
end of file
```

What is this black magic?

- Have a look at **`/usr/include/zlib.h`**
  - Defines functions functions you can use
  - Includes documentation
  - Is fairly simple
- Make the function definitions etc. available with
  - **`#include <zlib.h>`**
- Tell the compiler to link the library when producing a binary
  - **`cc file.c -o file -lz`**
  - The -l option(s) say to link that library (can provide many -l opts)
    - -lz = link libz.so
    - -lm = link libm.so
    - -leina = link libeina.so
    - -lsmellyskunk = link libsmellyskunk.so

Exercise

- Modify the previous example to DECOMPRESS the same file we just compressed
  - Copy s8.c to s9.c
  - Compile a binary s9 from the s9.c source
  - Use zlib
  - Arguments should be
    - **`argv[1]`** == compressed input file
    - **`argv[2]`** == uncompressed output file
  - Use command-line tool: diff to see if uncompressed output differs from original file
    - **`./s9 newfile uncompressed`**
    - **`diff s7 uncompressed`**
      - “Binary files s7 and uncompressed differ”
      - Or no output if they are the same

Sample

```
#include <stdio.h>
#include <stdlib.h>
#include <zlib.h>
int main(int argc, char **argv) {
   char buf[1024];
   char *compressed;
   uLongf destsize;
   size_t count;
   FILE *infile = fopen(argv[1], "r");
   FILE *outfile = fopen(argv[2], "w");
   while (1) {
      count = fread(&destsize, 1, sizeof(destsize), infile);
      if (count > 0) {
         printf("reading %i bytes...\n", (int)destsize);
         compressed = malloc(destsize);
         count = fread(compressed, 1, destsize, infile);
         destsize = sizeof(buf);
         uncompress(buf, &destsize, compressed, count);
         free(compressed);
         printf("writing %i bytes...\n", (int)destsize);
         fwrite(buf, 1, destsize, outfile);
      } else {
         printf("end of file\n");
         break;
      }
   }
   fclose(infile);
   fclose(outfile);
}
```

So what is wrong with this code

- Don’t check error returns from fopen()
  - Returns NULL if open failed
- Writing compressed chunk sizes using system dependent size
  - Ulong → unsigned long → 4 OR 8 bytes (32/64bit)
  - Don’t define byte order (endianess) on write (always native)
  - Ulong is too long for out chunks. Short (16bit) would do
- When reading with fread() don’t look for short reads on decompress
- Don’t check value of chunk size
  - Assuming endianess is fixed, need to check sanity of value
- Don’t check malloc return values (malloc failed?)
- Don’t check short read on payload read
- Don’t check error returns from compress() or uncompress()
- Would be better to use “rb” and “wb” than “r” and “w”

More Advanced topics

Functional C

- Functions can be addressed by pointer
  - Like almost anything that actually exists in memory
  - Can be used to indirect calling of a function
  - Can be used to store a function to be called later
  - Very common in event systems
    - Provide functions to be called in future when event happens
      - Referred to as “callbacks”
        - To call you back later on (like a phone call)

Function Pointers

```
#include <stdio.h>
#include <stdlib.h>
void func1(void) {
   printf("func1\n");
}
void func2(void) {
   printf("func2\n");
}
void func3(void) {
   printf("func3\n");
}
int main(int argc, char **argv) {
   typedef void (*myfunc_t) (void);
   myfunc_t array[3] = {
      func1, func2, func3
   };
   int n = atoi(argv[1]);
   array[n]();
}
```

Even more Advanced

OO in C

- Not hard
  - Combine several of the prior concepts
    - Structs
    - Typedefs
    - Heap allocation
    - Function pointers
- Many ways to do this
  - Linux kernel uses this method to do OO
  - GTK+ uses a more complex method to do OO
  - EFL uses an even more elaborate way to do OO
  - The following is just one possible one

OO

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* public header */
typedef struct obj_t obj_t;
typedef struct pub_t pub_t;
struct pub_t {
   pub_t *(*create)   (void);
   void   (*destroy)  (pub_t *obj);
   void   (*set_text) (pub_t *obj, char
*text);
};
struct obj_t {
   pub_t pub;
   char *text;
};

/* private implementation */
pub_t *obj_create(void);
void   obj_destroy(pub_t *obj);
void   obj_set_text(pub_t *obj, char *text);
pub_t klass = {
   obj_create,
   obj_destroy,
   obj_set_text
};
```

```
pub_t *obj_create(void) {
   obj_t *o = calloc(1, sizeof(obj_t));
   o->pub = klass;
   return &(o->pub);
}
void obj_destroy(pub_t *obj) {
   obj_t *o = (obj_t *)obj;
   free(o->text);
   free(o);
}
void obj_set_text(pub_t *obj, char *text) {
   obj_t *o = (obj_t *)obj;
   free(o->text);
   o->text = strdup(text);
}

/* in use */
int main(int argc, char **argv) {
   pub_t *obj = klass.create();
   obj->set_text(obj, "hello");
   obj->destroy(obj);
}
```

Not covered

- C pre-processor & macros
- Deeper examples and topics
  - Only scratched the surface
  - A lot of C is just putting together basic tools and ideas
- Debugging
  - GDB
  - Valgrind
    - Memcheck
    - Massif
    - Callgrind
  - Memory debugging libraries
    - Electric fence