Introduction to binary formats

It is all about the bits

At its most fundamental, a data structure is simply how we interpret a sequence of bits.

The world is full of bits. Consider this photograph: [1] :

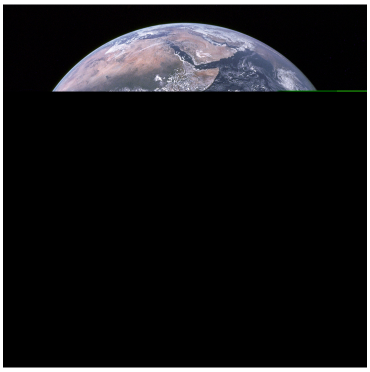

Figure 1.1: The Earth as seen from Apollo 17

Now reconsider it as a monochrome picture:

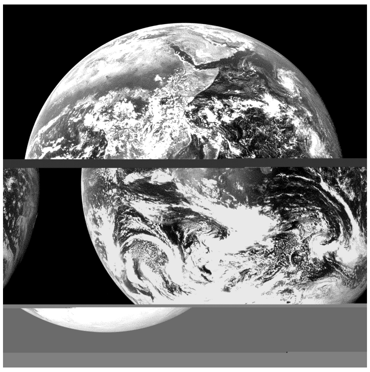

Figure 1.2: The Earth in monochrome

[1]Via Wikipedia's article on Earth. This photograph was taken by the Apollo 17 crew. It is thought that either Harrison Schmitt or Ron Evans took this photograph.

Now map these black and white dots to bits; here are a selection from middle of the image, broken as a consecutive sequence of 7 bits:

```
0000001 0011010 0010010 0100111 1010110 1111010 1011110
1111111 1111000 0111101 0101111 0100100 1010100 0100000
1000000 0001100 0000000 0111101 0111011 1111100 0000000
0000110 1010011 0000000 0000000 0000001 0000000 1110101
1110101 1111011 1110110 1010110 1011101 0110110 1011011
```

Or, converting our groups of 7 bits as base 16

```
01 1A 12 27 56 7A 5E
7F 78 3D 2F 24 54 20
40 0C 00 3D 3B 7C 00
06 53 00 00 01 00 75
75 7B 76 56 5D 36 5B
```

Or rendered as (somewhat) more familiar ASCII:

```
SOH SUB DC2  '   V   z   ^
DEL  x   =   /   $   T  SPC
 @  FF  NUL  =   ;   |  NUL
ACK  S  NUL NUL SOH NUL  u
 u   {   v   V   ]   6   [
```

But you can as easily (and more naturally) view these self-same bits also as a series of 32bit unsigned integers. Here are same bits broken into groups of 8 bits:

```
00000010 01101000 10010010 01111010 11011110 10101111
01111111 11110000 11110101 01111010 01001010 10001000
00100000 00001100 00000000 11110101 11011111 11000000
00000001 10101001 10000000 00000000 00000100 00000111
01011110 10111110 11111011 01010110 [...]
```

While this is a more natural view from the viewpoint of a computer, we might want to view it as base 10:

```
   40,407,674  3,736,043,504  4,118,432,392
  537,657,589  3,753,902,505  2,147,484,679
1,589,574,486
```

From a "bottom-up" perspective, how we understand bits is the heart of structuring data. Data structures are fundamentally an understanding of bits.

Take the previous item; we can regard that series of seven 32 bit unsigned integers as a traditional C array named "`v0`":

```c
#include <stdint.h>

uint32_t v0[7];
v0[0] =   40407674;
v0[1] = 3736043504;
v0[2] = 4118432392;
v0[3] =  537657589;
v0[4] = 3753902505;
v0[5] = 2147484679;
v0[6] = 1589574486;
```

And, finally, completing the circle, we can write a small C program to display the bits:

```c
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

//
// The following code is courtesy of https://stackoverflow.com/
    questions/111928,
// from the posting by "ideasman42".
//

/* --- PRINTF_BYTE_TO_BINARY macro's --- */
#define PRINTF_BINARY_PATTERN_INT8 "%c%c%c%c%c%c%c%c "
#define PRINTF_BYTE_TO_BINARY_INT8(i)    \
    (((i) & 0x80ll) ? '1' : '0'), \
    (((i) & 0x40ll) ? '1' : '0'), \
    (((i) & 0x20ll) ? '1' : '0'), \
```

```
    (((i) & 0x10ll) ? '1' : '0'), \
    (((i) & 0x08ll) ? '1' : '0'), \
    (((i) & 0x04ll) ? '1' : '0'), \
    (((i) & 0x02ll) ? '1' : '0'), \
    (((i) & 0x01ll) ? '1' : '0')

#define PRINTF_BINARY_PATTERN_INT16 \
    PRINTF_BINARY_PATTERN_INT8              PRINTF_BINARY_PATTERN_INT8
#define PRINTF_BYTE_TO_BINARY_INT16(i) \
    PRINTF_BYTE_TO_BINARY_INT8((i) >> 8),   PRINTF_BYTE_TO_BINARY_INT8(i)
#define PRINTF_BINARY_PATTERN_INT32 \
    PRINTF_BINARY_PATTERN_INT16             PRINTF_BINARY_PATTERN_INT16
#define PRINTF_BYTE_TO_BINARY_INT32(i) \
    PRINTF_BYTE_TO_BINARY_INT16((i) >> 16), PRINTF_BYTE_TO_BINARY_INT16(i)
#define PRINTF_BINARY_PATTERN_INT64    \
    PRINTF_BINARY_PATTERN_INT32             PRINTF_BINARY_PATTERN_INT32
#define PRINTF_BYTE_TO_BINARY_INT64(i) \
    PRINTF_BYTE_TO_BINARY_INT32((i) >> 32), PRINTF_BYTE_TO_BINARY_INT32(i)
/* --- end macros --- */

// End of code from stackoverflow.

int main()
{
  uint32_t v0[7];
  v0[0] =   40407674;
  v0[1] = 3736043504;
  v0[2] = 4118432392;
  v0[3] =  537657589;
  v0[4] = 3753902505;
  v0[5] = 2147484679;
  v0[6] = 1589574486;

  printf("The base address for v0 is %p\n",v0);
  for(int i=0; i<7; i++)
    {
      printf("at address %p, we have v0[%d] = "
          PRINTF_BINARY_PATTERN_INT32 "\n",&v0[i],i,
          PRINTF_BYTE_TO_BINARY_INT32(v0[i]));
    }
  uint64_t *v1 = (uint64_t *)v0;

  printf("\n");

  printf("The base address for v1 is %p\n",v1);
  for(int i=0; i<3; i++)
    {
      printf("at address %p, we have v1[%d] = "
```

```
            PRINTF_BINARY_PATTERN_INT64 "\n",&v1[i],i,
            PRINTF_BYTE_TO_BINARY_INT64(v1[i]));
        }

}
```

And here's what we see when we run our program:

```
$ ./example1
The base address for v0 is 0x7fff140be7f0
at address 0x7fff140be7f0, we have v0[0] = 00000010 01101000 10010010
    01111010
at address 0x7fff140be7f4, we have v0[1] = 11011110 10101111 01111111
    11110000
at address 0x7fff140be7f8, we have v0[2] = 11110101 01111010 01001010
    10001000
at address 0x7fff140be7fc, we have v0[3] = 00100000 00001100 00000000
    11110101
at address 0x7fff140be800, we have v0[4] = 11011111 11000000 00000001
    10101001
at address 0x7fff140be804, we have v0[5] = 10000000 00000000 00000100
    00000111
at address 0x7fff140be808, we have v0[6] = 01011110 10111110 11111011
    01010110

The base address for v1 is 0x7fff140be7f0
at address 0x7fff140be7f0, we have v1[0] = 11011110 10101111 01111111
    11110000 00000010 01101000 10010010 01111010
at address 0x7fff140be7f8, we have v1[1] = 00100000 00001100 00000000
    11110101 11110101 01111010 01001010 10001000
at address 0x7fff140be800, we have v1[2] = 10000000 00000000 00000100
    00000111 11011111 11000000 00000001 10101001
```

We can regard those individual bits as a sequence of 7 bit ASCII, or 8 bit bytes, or 32 bit unsigned integers, or 64 bit unsigned integers: it's not the bits that make it information, it's our interpretation of these bits.

Likewise, a data structure is our agreement as to the meaning of a given arrangement of bits.

A data structure can be a contiguous, like all of our data bits from the monochromatic rendition of Earth:

XXXXXXX	XXXXXXX	XXXXXXX	XXXXXXX	XXXXXXX	XXXXXXX	XXXXXXX
0000001	0011010	0010010	0100111	1010110	1111010	1011110
1111111	1111000	0111101	0101111	0100100	1010100	0100000
1000000	0001100	0000000	0111101	0111011	1111100	0000000
0000110	1010011	0000000	0000000	0000001	0000000	1110101
1110101	1111011	1110110	1010110	1011101	0110110	1011011
XXXXXXX	XXXXXXX	XXXXXXX	XXXXXXX	XXXXXXX	XXXXXXX	XXXXXXX

How do we understand common data structures?

Arrays of native types are generally implemented in C and C++ in this fashion, as you can see from running example1. C strings are also done in the fashion, but a new issue starts to creep in, alignment. ASCII is a 7 bit standard, but we store it 8 bit bytes. [2]

Structs and classes are also laid in this fashion, but, like with ASCII, you start to run into issues such as alignment; for instance, look at this code:

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

struct s0
{
  int8_t f0;
  int32_t f1;
  int8_t f2;
  int16_t f3;
  int8_t f4;
};

int main()
{
  struct s0 s;


  printf("field f0 is at address %p\n",&s.f0);
  printf("field f1 is at address %p\n",&s.f1);
  printf("field f2 is at address %p\n",&s.f2);
```

[2] This actually turns out to be a benefit; UTF-8 encoding of Unicode actually turns out to embed 7bit ASCII by the convention that if the high bit is zero, then it uses the 7-bit ASCII value, and if the top bit is instead set, then it maps this byte and succeeding bytes to a Unicode character.

```
  printf("field f3 is at address %p\n",&s.f3);
  printf("field f4 is at address %p\n",&s.f4);

}
```

When you compile this code with clang and run it, you get:

```
clang -o example2 example2.c
langley@localhost ~/mounts/www/public_html/COP4530/Lectures $ ./
    example2
field f0 is at address 0x7ffee5d8b280
field f1 is at address 0x7ffee5d8b284
field f2 is at address 0x7ffee5d8b288
field f3 is at address 0x7ffee5d8b28a
field f4 is at address 0x7ffee5d8b28c
```

But f0 is only 8bits, so you might have expected f1 to start at 0x7ffee5d8b281 rather than at 0x7ffee5d8b284. But compilers tend to optimize for speed rather than memory efficiency. While the x86_64 family of processors can read f1 at either address (not a given with all processors), such unaligned access does have a significant runtime penalty for the x86_64 architecture.

If you add a *pragma*, you can ask the compiler to do such packing for you:

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

struct s0
{
  int8_t f0;
  int32_t f1;
  int8_t f2;
  int16_t f3;
  int8_t f4;
} __attribute__((packed));

int main()
{
  struct s0 s;


  printf("field f0 is at address %p\n",&s.f0);
  printf("field f1 is at address %p\n",&s.f1);
```

```
  printf("field f2 is at address %p\n",&s.f2);
  printf("field f3 is at address %p\n",&s.f3);
  printf("field f4 is at address %p\n",&s.f4);

}
```

Now when you run the code, you get these in truly consecutive order:

```
./example2
field f0 is at address 0x7ffc3756f540
field f1 is at address 0x7ffc3756f541
field f2 is at address 0x7ffc3756f545
field f3 is at address 0x7ffc3756f546
field f4 is at address 0x7ffc3756f548
```

Data structures are composed of elements, such as integers, and the relationships among those elements.

There are two ways of thinking of data structures: an "abstract" data structure, and a "realized" data structure. An abstract data structure only specifies that some sort of relationship exists between elements; a realized data structure specifies the actual relationships and the actual elements.

A realized data structure extends this idea from merely adjacent bits as a basic type, such as an integer, to multiple elements. The elements follow an agreed pattern; the agreement can be based on simple adjacency (i.e., adjacent elements have (effectively) a zero distance between them (subject to alignment issues)), or it can be based on internal components that specify the location of other elements, such as pointers.

Arrays are usually implemented by agreement; the most common agreement is that 1) each element of the array is of uniform type, and 2) that all elements are laid consecutively. For example, let's declare an array `arr` of type `uint32_t` with 20 elements, and look at how it's laid out in memory:

```
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>


int main()
{
  uint32_t arr[20];

  printf("Array 'arr' begins at address %p and is %lu bytes in size.\n"
      ,arr,sizeof(arr));
  for(int i = 0; i<20; i++)
    {
```

```

        printf("\t element %02d starts at %p and is %lu bytes in size.\n"
            ,i,&arr[i],sizeof(arr[i]));
    }
}
```

When you run this code, you see that the array is made of 20 uniformly 4 byte integers packed side-by-side:

```
./example3
Array 'arr' begins at address 0x7ffde225d760 and is 80 bytes in size.
     element 00 starts at 0x7ffde225d760 and is 4 bytes in size.
     element 01 starts at 0x7ffde225d764 and is 4 bytes in size.
     element 02 starts at 0x7ffde225d768 and is 4 bytes in size.
     element 03 starts at 0x7ffde225d76c and is 4 bytes in size.
     element 04 starts at 0x7ffde225d770 and is 4 bytes in size.
     element 05 starts at 0x7ffde225d774 and is 4 bytes in size.
     element 06 starts at 0x7ffde225d778 and is 4 bytes in size.
     element 07 starts at 0x7ffde225d77c and is 4 bytes in size.
     element 08 starts at 0x7ffde225d780 and is 4 bytes in size.
     element 09 starts at 0x7ffde225d784 and is 4 bytes in size.
     element 10 starts at 0x7ffde225d788 and is 4 bytes in size.
     element 11 starts at 0x7ffde225d78c and is 4 bytes in size.
     element 12 starts at 0x7ffde225d790 and is 4 bytes in size.
     element 13 starts at 0x7ffde225d794 and is 4 bytes in size.
     element 14 starts at 0x7ffde225d798 and is 4 bytes in size.
     element 15 starts at 0x7ffde225d79c and is 4 bytes in size.
     element 16 starts at 0x7ffde225d7a0 and is 4 bytes in size.
     element 17 starts at 0x7ffde225d7a4 and is 4 bytes in size.
     element 18 starts at 0x7ffde225d7a8 and is 4 bytes in size.
     element 19 starts at 0x7ffde225d7ac and is 4 bytes in size.
```

Now we consider the case where we have an indicator from one element to another element; this particular indicator is a "pointer", which is just a variable that has memory address in it. Here is some code that implements a simple list of integers using pointers:

```

#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>
```

```c

struct st
{
  int val;
  struct st *next;
};

int main()
{
  struct st *struct0;
  struct st *struct1;
  struct st *struct2;

  struct0 = malloc(sizeof(struct st));
  printf("For struct0, we allocated %lu bytes at memory location %p.\n"
      ,sizeof(struct st),struct0);
  struct1 = malloc(sizeof(struct st));
  printf("For struct1, we allocated %lu bytes at memory location %p (
      distance struct1-struct0 is %ld bytes).\n",sizeof(struct st),
      struct1,(void *)struct1-(void *)struct0);
  struct2 = malloc(sizeof(struct st));
  printf("For struct2, we allocated %lu bytes at memory location %p (
      distance struct2-struct1 is %ld bytes).\n",sizeof(struct st),
      struct2,(void*)struct2-(void *)struct1);

  struct0->val = 1;
  struct0->next = struct1;

  struct1->val = 2;
  struct1->next = struct2;

  struct2->val = 3;
  struct2->next = NULL;

  struct st *s = struct0;
  while(s)
    {
      printf("This element is at memory location %p; it has value %d,
          and a pointer %p to a next element.\n",
          s,
          s->val,
          s->next);
      s=s->next;
    }
}
```

If we were to run this code, we can see now that while the pointers are linearly increasing (the heap grows up), these are not contiguous:

```
$ clang -g -o example4 example4.c
fsucs@localhost ~/mounts/www/public_html/COP4530/Lectures $ ./example4
For struct0, we allocated 16 bytes at memory location 0x2349260.
For struct1, we allocated 16 bytes at memory location 0x2349690 (
    distance struct1-struct0 is 1072 bytes).
For struct2, we allocated 16 bytes at memory location 0x23496b0 (
    distance struct2-struct1 is 32 bytes).
This element is at memory location 0x2349260; it has value 1, and a
    pointer 0x2349690 to a next element.
This element is at memory location 0x2349690; it has value 2, and a
    pointer 0x23496b0 to a next element.
This element is at memory location 0x23496b0; it has value 3, and a
    pointer (nil) to a next element.
```

Finally, we will look at an actual data structure being created in x86_64 assembly language. This data structure is called an “ELF header”. It’s what is used, for instance, as the header for every binary executable on an x86_64 Linux computer.

Here’s the NASM code for a trivial program that lays out its own ELF header and then has a trivial body that only exits (with 42, naturally):

```

    ;; inspired by http://www.muppetlabs.com/~breadbox/software/tiny/
        teensy.html
    ;;              https://blog.stalkr.net/2014/10/tiny-elf-3264-with-
        nasm.html
    ;;              ... and others of similar ilk


BITS 64

ORG 0x400000

;;;
;;;  Definitions from "ELF-64 Object File Format" (aka "EOFF document")
    :
;;;
;;;
;;;      Elf64_Addr    8 bytes, aligned on 8 bytes   ; program address
;;;      Elf64_Off     8 bytes, aligned on 8 bytes   ; file offset
```

```
;;;       Elf64_Half    2 bytes, aligned on 2 bytes   ; medium integer
;;;       Elf64_Word    4 bytes, aligned on 4 bytes   ; integer
;;;       Elf64_Sword   4 bytes, aligned on 4 bytes   ; signed integer
;;;       Elf64_Xword   8 bytes, aligned on 8 bytes       ; long integer
;;;       Elf64_Sxword  8 bytes, aligned on 8 bytes       ; signed long integer
;;;       unsigned char 1 byte, aligned on 1 byte     ; small integer


elf64_file_header:       ; This is often just called the elf64 header

    ;; at 0: unsigned char e_ident[16]
    db 127,"ELF"         ; e_ident[0-3]: EI_MAG{0,1,2,3} (aka "magic number")
    db 2             ; e_ident[4]: EI_CLASS; ELFCLASS32=1, ELFCLASS64=2 (aka "File class")
    db 1             ; e_ident[5]: EI_DATA; ELFDATALSB=1, ELFDATAMSB=2 (aka "Data encoding")
    db 1             ; e_ident[6]: EI_VERSION; EV_CURRENT=1 (aka "File version")
    db 0             ; e_ident[7]: EI_OSABI; ELFOSABI_SYSV=0 (aka "OS/ABI identification")
    db 0             ; e_ident[8]: EI_ABI (always zero)
    times 7 db 0         ; e_ident[9-15]: EI_PAD

    ;; at 16: Elf64_Half
    dw 2             ; e_type: 2 = "executable file" (aka "object file type")
    ;; at 18: Elf64_Half
    dw 62            ; e_machine: EM_X86_64 = 62 (aka "machine type")
    ;;           that is found for Linux in "include/uapi/linux/elf-em.h"

    ;; at 20: Elf64_Word
    dd 1             ; e_version: always 1
    ;; at 24: Elf64_Addr
    dq _start        ; e_entry: the address where you want to start running
    ;; at 32: Elf64_Off
    dq elf64_program_header - $$
                ; e_phoff: offset to program header(s) start - required in
    ;;          ;   all executables since they give the actual segments to be
    ;;                      ;   to be laid out in memory
    ;; at 40: Elf64_Off
    dq 0             ; e_shoff: offset to section header(s) start - not required in
    ;;                      ; static executables since sections are only important for relocation
    ;; at 48: Elf64_Word
```

```
    dd 0            ; e_flags: processor-specific flags
    ;;                      ; (where is this documented in the kernel source code?)
    ;;
    ;; at 52: Elf64_Half
    dw elf64_file_header_size ; e_ehsize: elf64 file header size
    ;; at 54: Elf64_Half
    dw elf64_program_header_entry_size
    ;;                      ; e_phentsize: size of one program header entry
    ;;
    ;; at 56: Elf64_Half
    dw 1            ; e_phnum: how many program header entries do we have?
    ;; at 58: Elf64_Half
    dw 0            ; e_shentsize: size of one section header entry
    ;; at 60: Elf64_Half
    dw 0            ; e_shnum: how many section header entries do we have?
    ;; at 62: Elf64_Half
    dw 0            ; e_shstrndx: section name string table index
    ;;
    ;;
elf64_file_header_size equ $ - elf64_file_header
    ;;
    ;;                      ; compute how big the header was (64 bytes!)


elf64_program_header:       ; This is our only program header since we only want one segment

    ;; at 64: Elf64_Word
    dd 1            ; p_type: type of segment, 1 = "loadable segment" (from EOFF document)
    ;; at 68: Elf64_Word
    dd 7            ; p_flags: segment attributes; 0x1 = execute permission
    ;;                      ;                              0x2 = write permission
    ;;                      ;                              0x4 = read permission
    ;;
    ;; at 72: Elf64_Off
    dq 0            ; p_offset: offset in file -- where does this segment start in file?
    ;; at 80: Elf64_Addr
    dq $$           ; p_vaddr: virtual address of the segment in memory
    ;; at 88: Elf64_Addr
    dq $$           ; p_paddr: reserved for systems with physical addressing
```

```
    ;; at 96: Elf64_Xword
    dq total_size       ; p_filesz: size of segment in file
    ;; at 104: Elf64_Xword
    dq total_size       ; p_memsz: size of segment in memory
    ;; at 112: Elf64_Xword
    dq 0x1000       ; p_align: alignment of the segment. p_offset =
       p_vaddr MOD p_align
    ;;
elf64_program_header_entry_size equ $ - elf64_program_header


_start:
    mov rax, 231        ; sys_exit_group
    mov rdi, 42     ; answer to everything
    syscall

total_size equ $ - $$
```