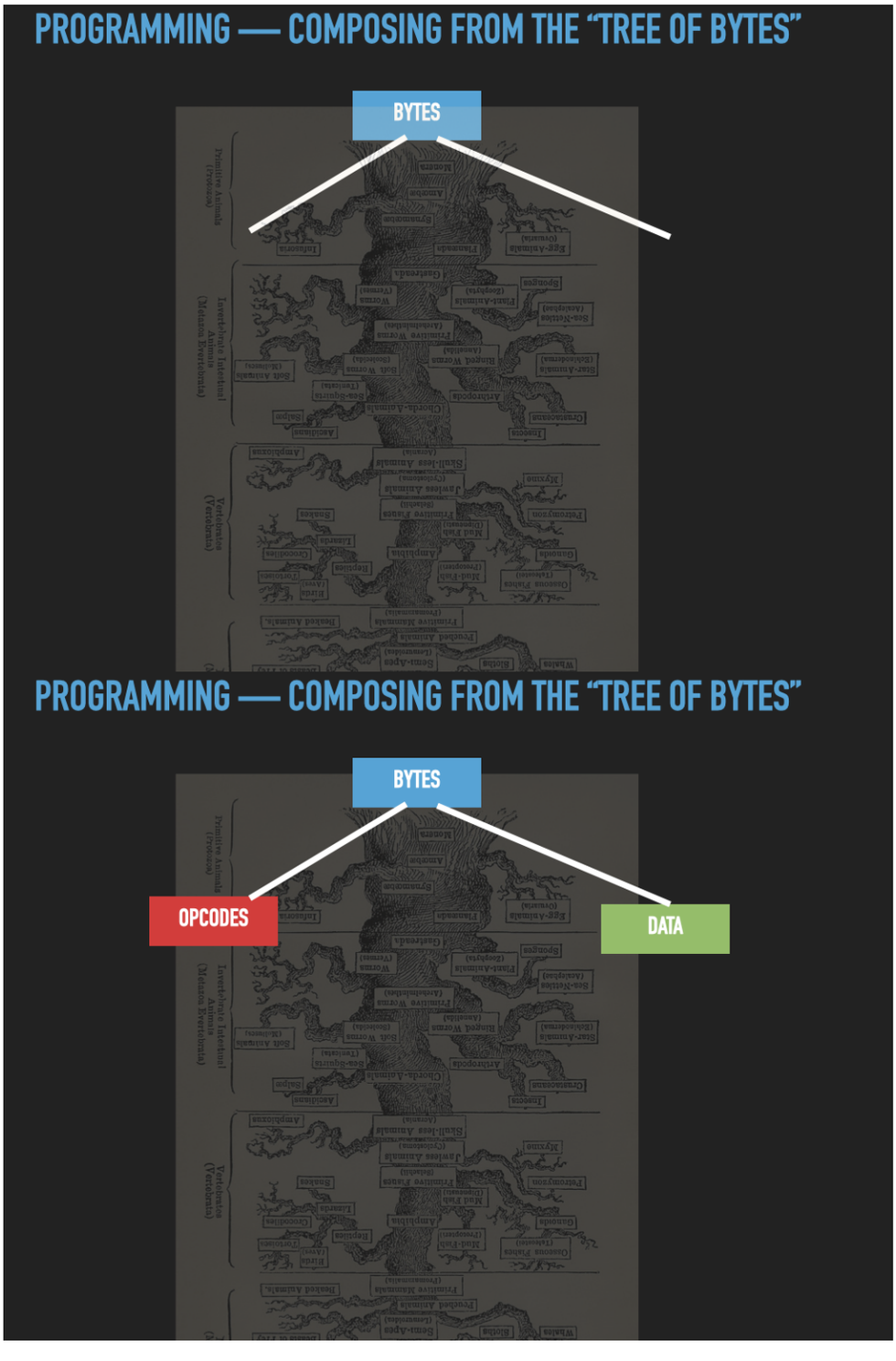
SLS Lecture 13 : Program Anatomy IV: The Tree of Bytes and Data Structures

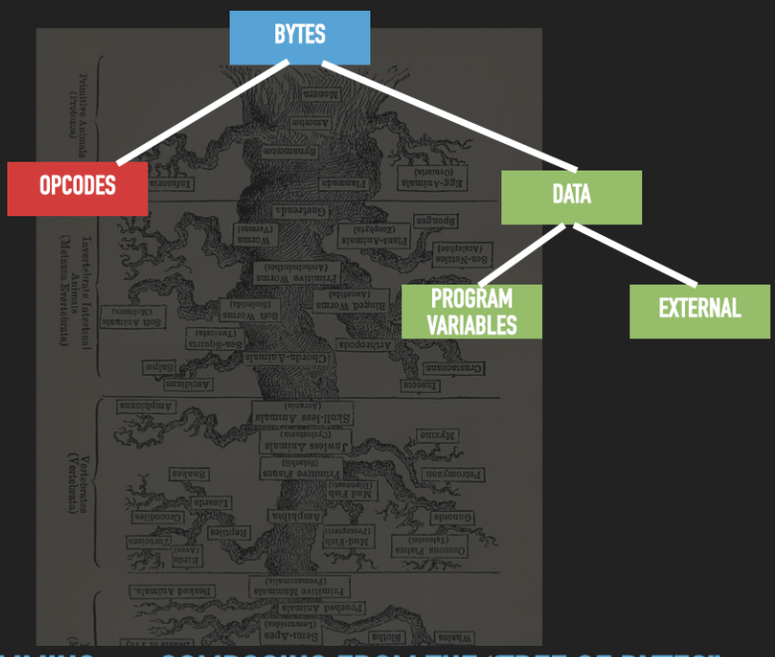
Contents

- create a directory `mkdir datastructs; cd datastructs`
- copy examples
- add a `Makefile` to automate assembling and linking
  - we are going run the commands by hand this time to highlight the details
- add our `setup.gdb` and `tree.gdb` to make working in gdb easier
- normally you would want to track everything in git

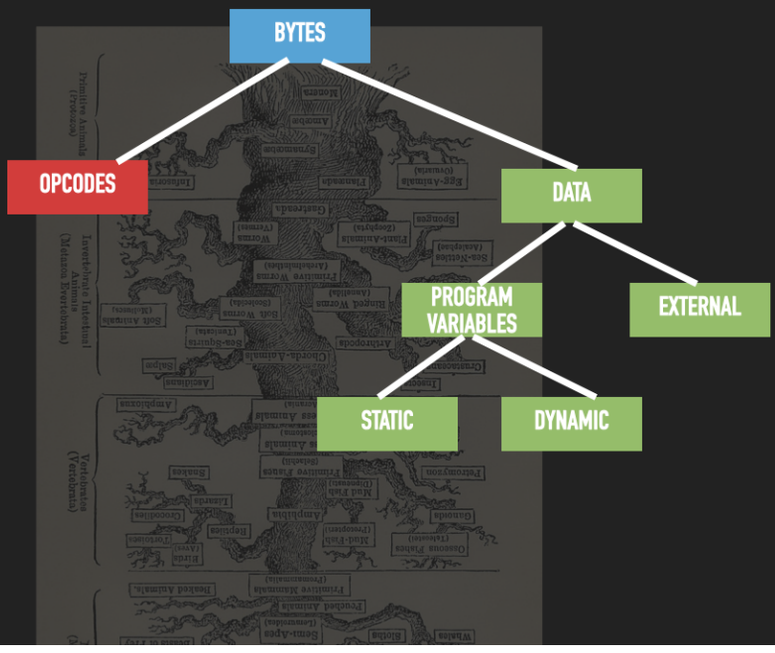
```
$ ls /home/jovyan/datastructs
Makefile  findplayer.S  playertest.S  setup.gdb  tree.S  tree.gdb  tree_bb.S
$
```

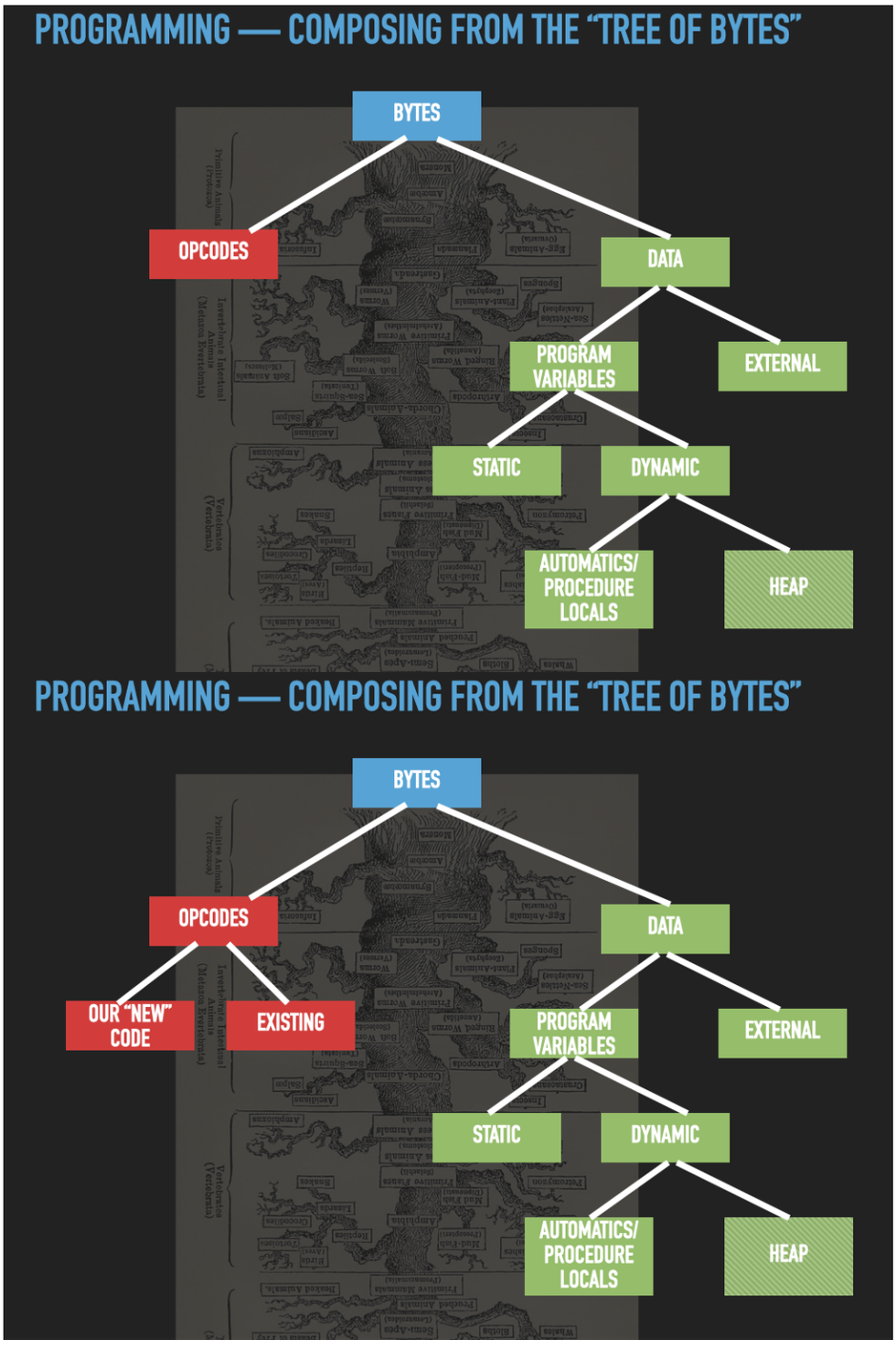


PROGRAMMING — COMPOSING FROM THE "TREE OF BYTES"



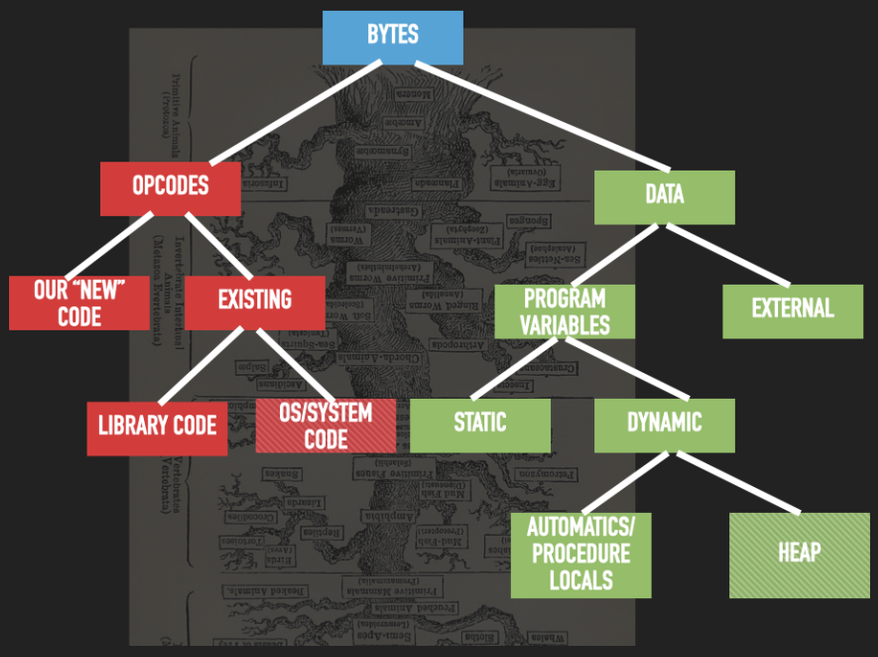
PROGRAMMING — COMPOSING FROM THE "TREE OF BYTES"



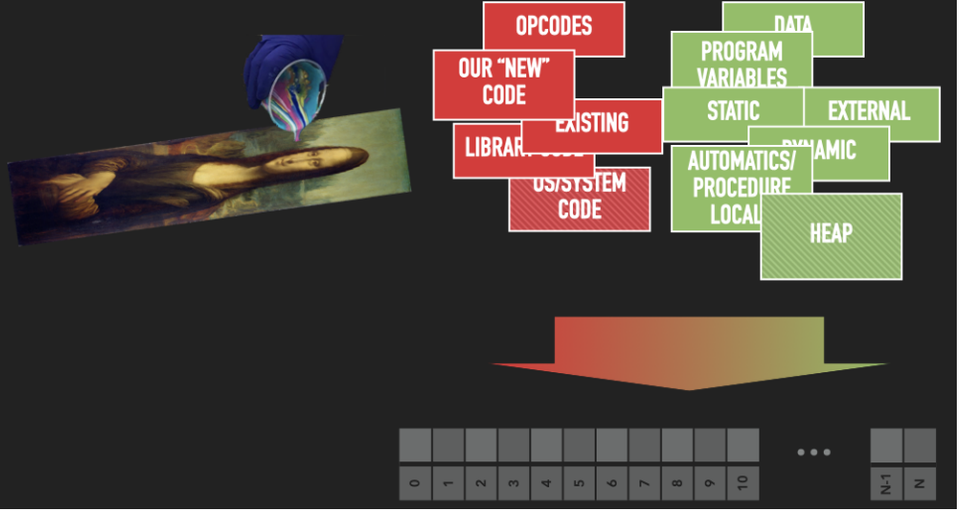

PROGRAMMING — COMPOSING FROM THE "TREE OF BYTES"
BYTES
OPCODES
DATA
PROGRAM VARIABLES
EXTERNAL
STATIC
DYNAMIC
AUTOMATICS/ PROCEDURE LOCALS
HEAP

PROGRAMMING — COMPOSING FROM THE "TREE OF BYTES"
BYTES
OPCODES
OUR "NEW" CODE
EXISTING
DATA
PROGRAM VARIABLES
EXTERNAL
STATIC
DYNAMIC
AUTOMATICS/ PROCEDURE LOCALS
HEAP

PROGRAMMING — COMPOSING FROM THE "TREE OF BYTES"

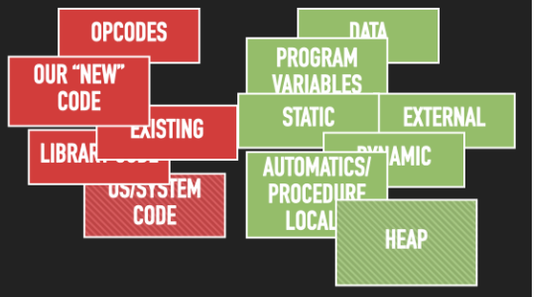


POURING BYTES INTO MEMORY : TO FORM A "PICTURE"



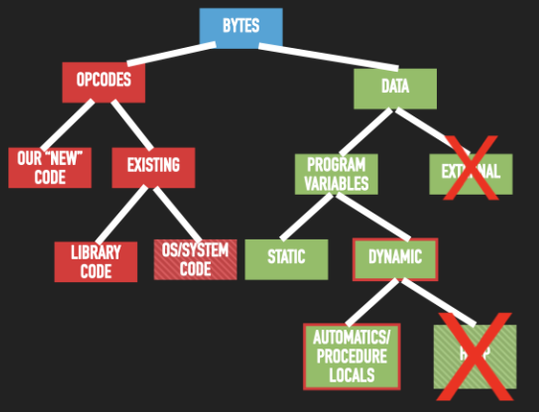
POURING BYTES INTO MEMORY : TO FORM A "PICTURE"

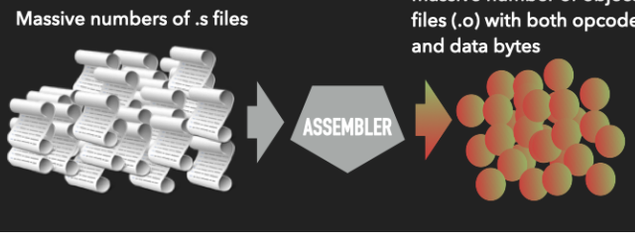
1. Generate chucks of bytes using our tools
2. Organize and ensure they all connect up correctly
3. load them into memory

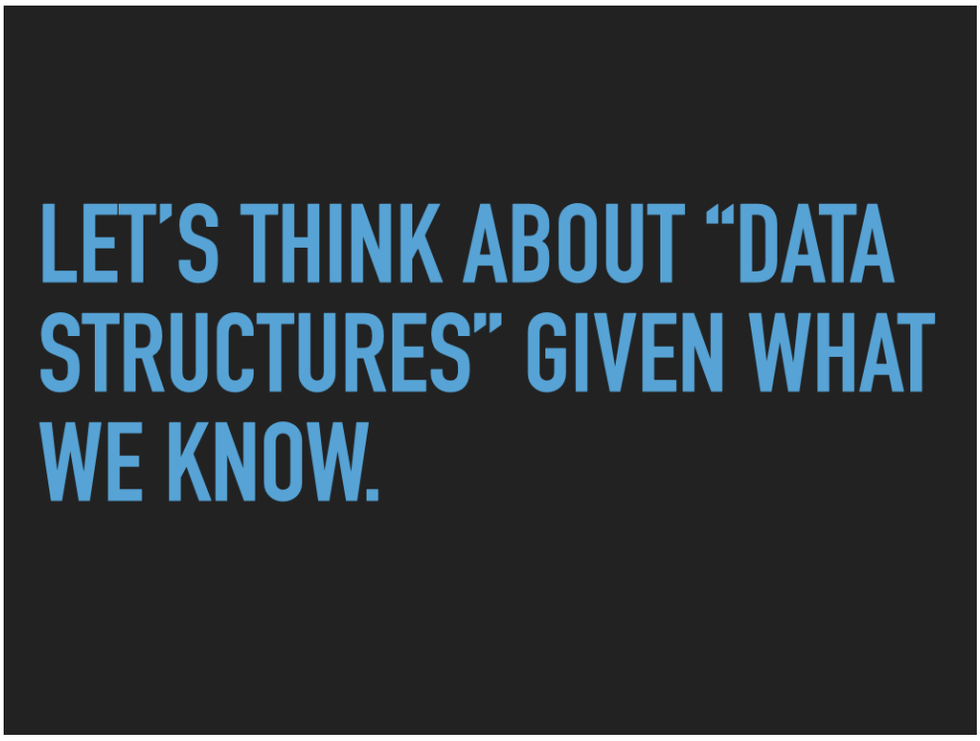


POURING BYTES INTO MEMORY : TO FORM A "PICTURE"

1. Generate chucks of bytes using our tools
    1. Assembler gives us a "human" language to describe
        1. OPCODES
        2. and DATA



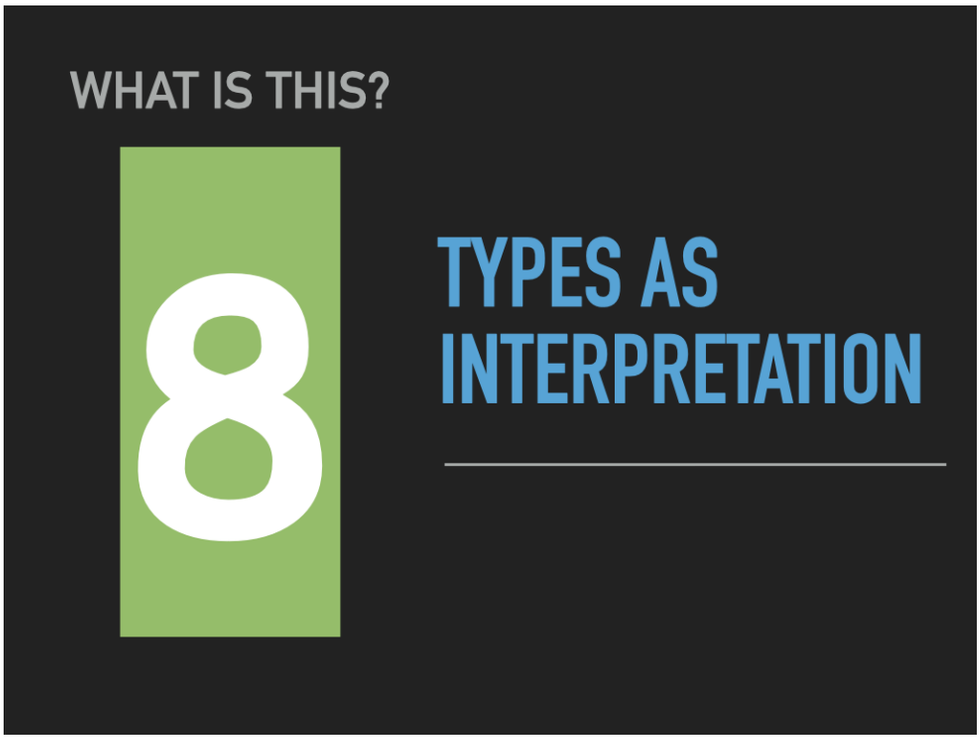




13.1. Overview

REMEMBER A DATA STRUCTURE ENDS UP AS ...

- Bytes in memory
  - At particular locations – Memory Address
  - taking up some number of **contiguous** bytes
- Intrinsic types : groups of bytes that the processor has built in "interpretations" for
  - bit vectors of various lengths (1,2,4,8,16,...)
  - signed and unsigned integers (1,2,4,8,16,...)
  - floating point number (IEEE 754)
- Complex data structures often are broken into pieces
  - each piece connected to others
  - connections formed via Addresses
  - pieces record the address of the pieces it connects to



TEXT

TYPES A MATTER OF INTERPRETATION

```
(gdb) x/16bx 0x402008
0x402008: 0x08  0x00  0x00  0x00  0x00  0x00  0x00  0x00
0x402010: 0x20  0x20  0x40  0x00  0x00  0x00  0x00  0x00
(gdb) x/8xh 0x402008
0x402008: 0x0008  0x0000  0x0000  0x0000  0x2020  0x0040  0x0000  0x0000
(gdb) x/4xw 0x402008
0x402008: 0x00000008 0x00000000 0x00402020 0x00000000
(gdb) x/2xg 0x402008
0x402008: 0x0000000000000008  0x0000000000402020
(gdb)
```

- Our Basic types
  - 1,2,4 or 8 byte Signed and unsigned integers
- How do we know which one?
- And what do they mean?
- Why?

TEXT

ALL DEPENDS ON THE CODE THAT "USES" THE MEMORY

```
(gdb) x/9i _start
   0x401000:  mov    rax,QWORD PTR ds:0x402000
   0x401008:  mov    rbx,QWORD PTR [rax+0x8]
   0x40100c:  mov    rdx,QWORD PTR [rax+0x10]
   0x401010:  cmp    QWORD PTR [rax],0x0
   0x401014:  cmovl  rax,rbx
   0x401018:  cmovge rax,rdx
   0x40101c:  cmp    rax,0x0
   0x401020:  jne    0x401008
   0x401022:  int3
```

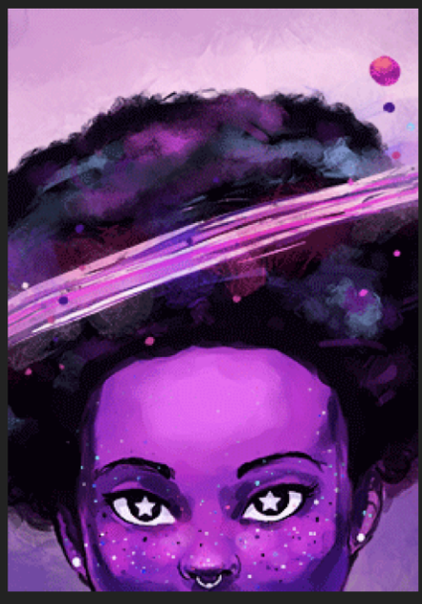
- The instructions of a program load values from memory and do things with those values – Interprets the memory
- To both write good code and to understand how programs works we must understand this relationship

ADVANCED CODE MIGHT INTERPRET THE SAME MEMORY IN DIFFERENT WAYS

TEXT

ONCE WE GET IT

- Memory and code become our paints
- Use our creativity to map our ideas into code and memory structures
- A computer is a powerful tool in the hands of a creative programmer
  - After all every app and “device” you use is the product of this creativity
  - The machine is just a cpu and memory – ideas express as programs convert it into all that you love (and hate)



TEXT

MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

- Lets start with a quick examination of the basic behavior
  - What do we see?

```
(gdb) x/9i _start
   0x401000:  mov    rax,QWORD PTR ds:0x402000
   0x401008:  mov    rbx,QWORD PTR [rax+0x8]
   0x40100c:  mov    rdx,QWORD PTR [rax+0x10]
   0x401010:  cmp    QWORD PTR [rax],0x0
   0x401014:  cmovl  rax,rbx
   0x401018:  cmovge rax,rdx
   0x40101c:  cmp    rax,0x0
   0x401020:  jne    0x401008
   0x401022:  int3
```

TEXT

MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

- Lets start with a quick examination of the basic behavior
  - What do we see?

```
(gdb) x/9i _start
   0x401000:  mov    rax,QWORD PTR ds:0x402000
   0x401008:  mov    rbx,QWORD PTR [rax+0x8]
   0x40100c:  mov    rdx,QWORD PTR [rax+0x10]
   0x401010:  cmp    QWORD PTR [rax],0x0
   0x401014:  cmovl  rax,rbx
   0x401018:  cmovge rax,rdx
   0x40101c:  cmp    rax,0x0
   0x401020:  jne    0x401008
   0x401022:  int3
```

A Loop : until rax == 0

TEXT

MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

- Lets start with a quick examination of the basic behavior
  - What do we see?

"Blind" writes to rax, rbx, rdx

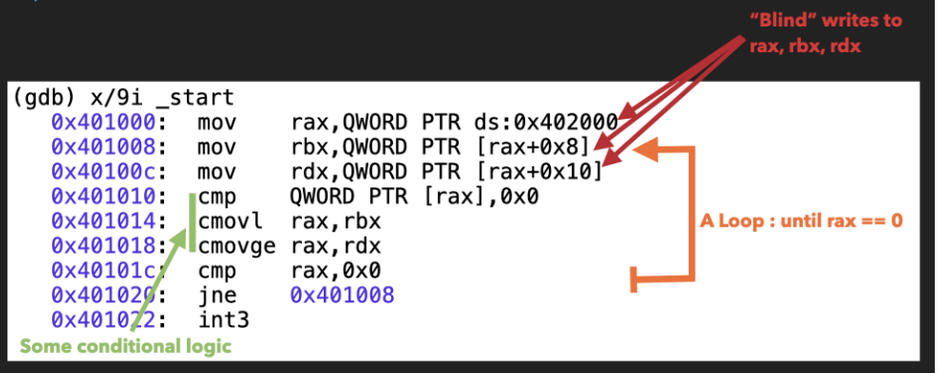
(gdb) x/9i _start
   0x401000: mov   rax,QWORD PTR ds:0x402000
   0x401008: mov   rbx,QWORD PTR [rax+0x8]
   0x40100c: mov   rdx,QWORD PTR [rax+0x10]
   0x401010: cmp   QWORD PTR [rax],0x0
   0x401014: cmovl rax,rbx
   0x401018: cmovge rax,rdx
   0x40101c: cmp   rax,0x0
   0x401020: jne   0x401008
   0x401022: int3

A Loop : until rax == 0

TEXT

MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

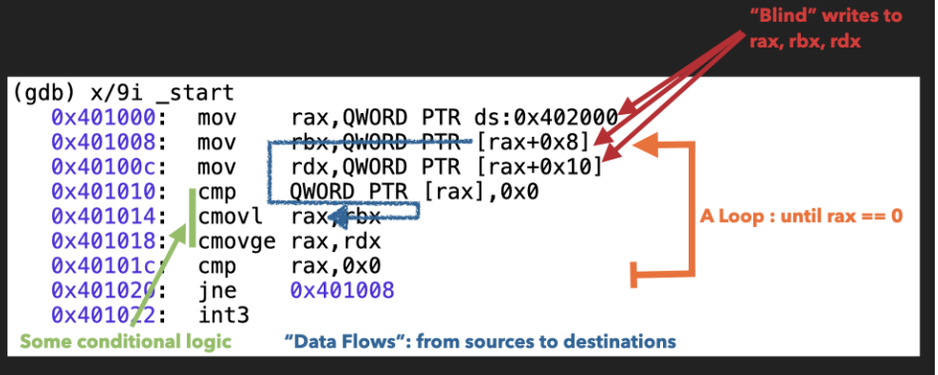
- Lets start with a quick examination of the basic behavior
  - What do we see?



TEXT

MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

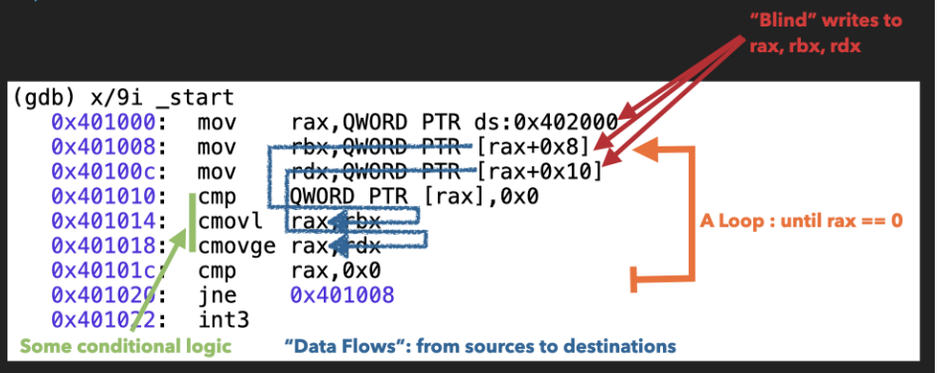
- Lets start with a quick examination of the basic behavior
  - What do we see?



TEXT

MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

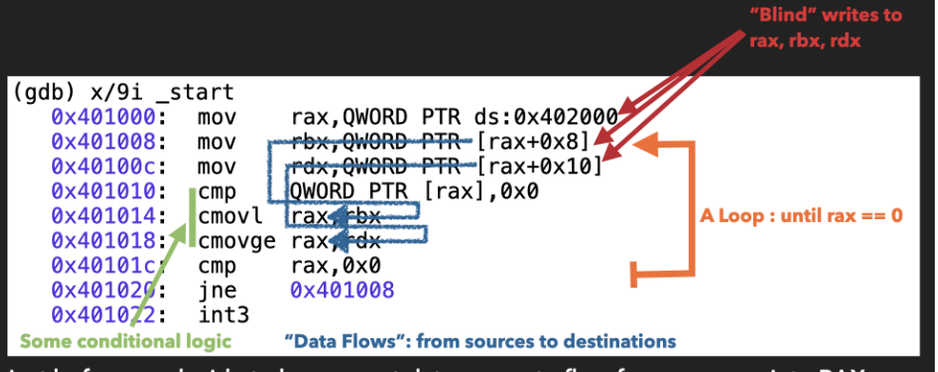
- Lets start with a quick examination of the basic behavior
  - What do we see?



TEXT

MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

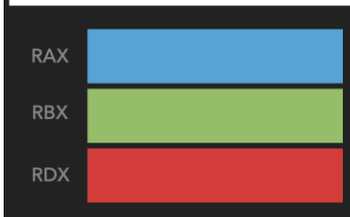
- Lets start with a quick examination of the basic behavior
  - What do we see?



1) just before we decide to loop or not data seems to flow from memory into RAX
2) 8 byte value at 0x1023 seems to kick everything off rax=M[0x1023]
3) rax, rbx, rdx all get values from memory

MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

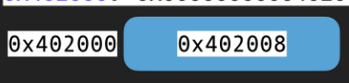
```
0x401000:
  mov     rax,QWORD PTR ds:0x402000
  mov     rbx,QWORD PTR [rax+0x8]
  mov     rdx,QWORD PTR [rax+0x10]
  cmp     QWORD PTR [rax],0x0
  cmovl   rax,rbx
  cmovge  rax,rdx
  cmp     rax,0x0
  jne     0x401008
  int3
```

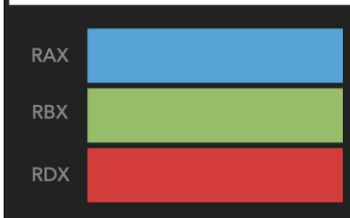


MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

```
0x401000:
  mov     rax,QWORD PTR ds:0x402000
  mov     rbx,QWORD PTR [rax+0x8]
  mov     rdx,QWORD PTR [rax+0x10]
  cmp     QWORD PTR [rax],0x0
  cmovl   rax,rbx
  cmovge  rax,rdx
  cmp     rax,0x0
  jne     0x401008
  int3
```

```
(gdb) x/1xg 0x402000
0x402000: 0x0000000000402008
```

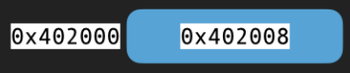


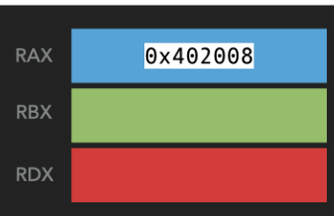


MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

```
0x401000:
  mov     rax,QWORD PTR ds:0x402000
  mov     rbx,QWORD PTR [rax+0x8]
  mov     rdx,QWORD PTR [rax+0x10]
  cmp     QWORD PTR [rax],0x0
  cmovl   rax,rbx
  cmovge  rax,rdx
  cmp     rax,0x0
  jne     0x401008
  int3
```

```
(gdb) x/1xg 0x402000
0x402000: 0x0000000000402008
```



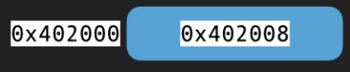


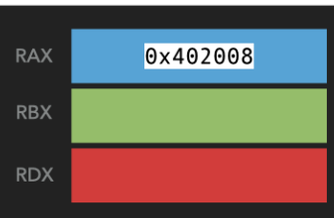
the value from 0x402000 is now in RAX – how is it used? – that will tell us what the location 0x402000 is being used for

MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

```
0x401000:
  mov     rax,QWORD PTR ds:0x402000
  mov     rbx,QWORD PTR [rax+0x8]
  mov     rdx,QWORD PTR [rax+0x10]
  cmp     QWORD PTR [rax],0x0
  cmovl   rax,rbx
  cmovge  rax,rdx
  cmp     rax,0x0
  jne     0x401008
  int3
```

```
(gdb) x/1xg 0x402000
0x402000: 0x0000000000402008
```



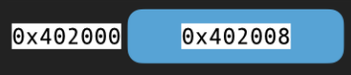


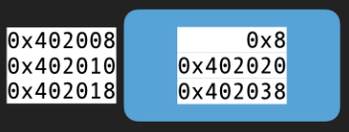
What we can see is that the value is used as an address – we are loading values and comparing values located relative to this address specifically 8,16 and 0 bytes from it

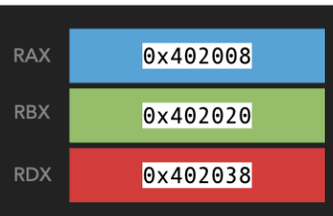
MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

```
0x401000:
  mov     rax,QWORD PTR ds:0x402000
  mov     rbx,QWORD PTR [rax+0x8]
  mov     rdx,QWORD PTR [rax+0x10]
  cmp     QWORD PTR [rax],0x0
  cmovl   rax,rbx
  cmovge  rax,rdx
  cmp     rax,0x0
  jne     0x401008
  int3
```

```
(gdb) x/1xg 0x402000
0x402000: 0x0000000000402008
```





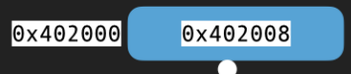


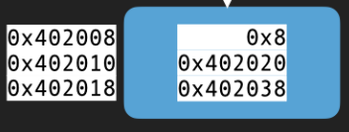

```
(gdb) x/1xg 0x402008 + 0x8
0x402010: 0x0000000000402020
(gdb) x/1xg 0x402008 + 0x10
0x402018: 0x0000000000402038
(gdb) x/1xg 0x402008 + 0x0
0x402008: 0x0000000000000008
```

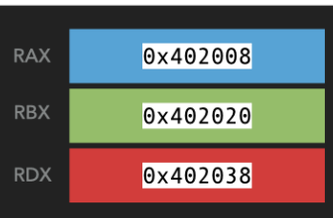

MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

```
0x401000:
  mov     rax,QWORD PTR ds:0x402000
  mov     rbx,QWORD PTR [rax+0x8]
  mov     rdx,QWORD PTR [rax+0x10]
  cmp     QWORD PTR [rax],0x0
  cmovl   rax,rbx
  cmovge  rax,rdx
  cmp     rax,0x0
  jne     0x401008
  int3
```

```
(gdb) x/1xg 0x402000
0x402000: 0x0000000000402008
```





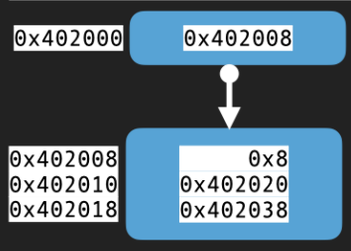

```
(gdb) x/1xg 0x402008 + 0x8
0x402010: 0x0000000000402020
(gdb) x/1xg 0x402008 + 0x10
0x402018: 0x0000000000402038
(gdb) x/1xg 0x402008 + 0x0
0x402008: 0x0000000000000008
```

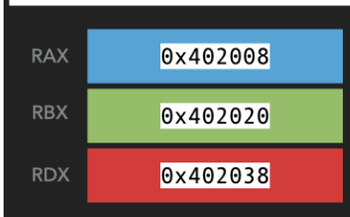

So 8, 3 byte quantities are worked with [rax +8], [rax + 16] and [rax + 0]. We think of 0x402000 as pointing to another location and visualize it with an arrow

MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

```
0x401000:
  mov     rax,QWORD PTR ds:0x402000
  mov     rbx,QWORD PTR [rax+0x8]
  mov     rdx,QWORD PTR [rax+0x10]
  cmp     QWORD PTR [rax],0x0
  cmovl   rax,rbx
  cmovge  rax,rdx
  cmp     rax,0x0
  jne     0x401008
  int3
```

```
(gdb) x/1xg 0x402000
0x402000: 0x0000000000402008
```



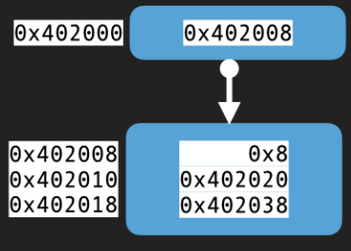


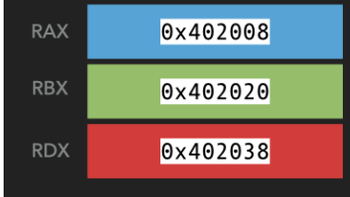
Now how are these locations used? Lets examine one by one

MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

```
0x401000:
  mov     rax,QWORD PTR ds:0x402000
  mov     rbx,QWORD PTR [rax+0x8]
  mov     rdx,QWORD PTR [rax+0x10]
  cmp     QWORD PTR [rax],0x0
  cmovl   rax,rbx
  cmovge  rax,rdx
  cmp     rax,0x0
  jne     0x401008
  int3
```

```
(gdb) x/1xg 0x402000
0x402000: 0x0000000000402008
```



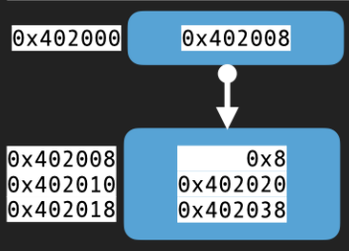


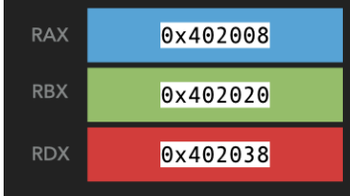
The first value located at [rax + 0] is used in there cmp with 0 then the two conditional moves are based on the value being less than or greater and equal (cmovl and cmovge). So the first value is being treated as a signed number.

MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

```
0x401000:
  mov     rax,QWORD PTR ds:0x402000
  mov     rbx,QWORD PTR [rax+0x8]
  mov     rdx,QWORD PTR [rax+0x10]
  cmp     QWORD PTR [rax],0x0
  cmovl   rax,rbx
  cmovge  rax,rdx
  cmp     rax,0x0
  jne     0x401008
  int3
```

```
(gdb) x/1xg 0x402000
0x402000: 0x0000000000402008
```



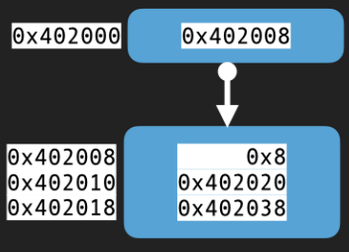


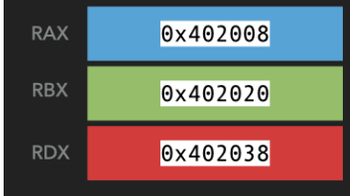
The next two values which are already in RBX and RDX are conditionally moved into RAX. If the value < 0 then the second 8 byte value is moved into RAX others the third is. If chosen value is 0, then the loop is terminated.

MUST READ THE CODE, EXAMINE MEMORY AND DRAW THINGS OUT

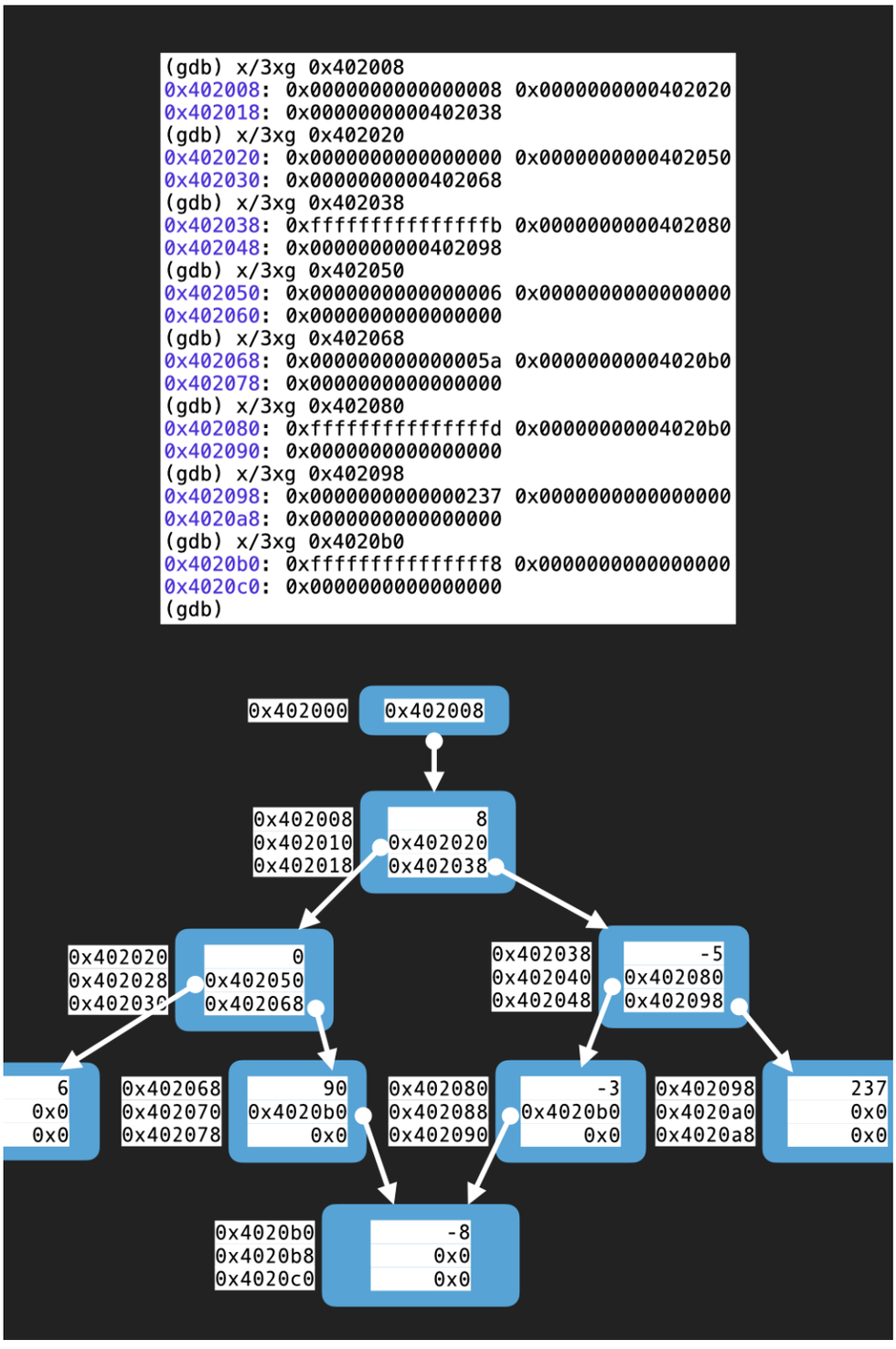
```
0x401000:
  mov     rax,QWORD PTR ds:0x402000
  mov     rbx,QWORD PTR [rax+0x8]
  mov     rdx,QWORD PTR [rax+0x10]
  cmp     QWORD PTR [rax],0x0
  cmovl   rax,rbx
  cmovge  rax,rdx
  cmp     rax,0x0
  jne     0x401008
  int3
```

```
(gdb) x/1xg 0x402000
0x402000: 0x0000000000402008
```





Otherwise we repeat the the process. So the second and third values will be treated as pointers to other chucks of memory that will be examined again in units of 3, 8 byte values. We can use gdb to figure out what's a these locations currently look like



13.2. Worth carefully examining the assembly for this example

Understanding this code is a great test of your knowledge

CODE: asm - tree.S

```
        .intel_syntax noprefix
        # TREE NODE STRUCTURE IN MEMORY
        # A NODE is composed of 3, 8 byte values
        # First 8 bytes is an 8 byte signed value
        # Second 8 bytes is a pointer to the left child (another node)
        # Third 8 bytes is a pointer to the right child (another node)
        # a 0 value pointer means there are no nodes in that direction

        #  node.VAL
        #  node.LEFT offset of left child
        #  node.RIGHT
        .equ VAL, 0       # tree node value 8 bytes (offset 0)
        .equ LEFT, 8      # pointer to left child 8 bytes (offset 8)
        .equ RIGHT, 16    # pointer to right child 8 bytes (offset 16)

        .section .text
        .global _start
_start:
        mov rax, QWORD PTR [ROOT]  # pointer to root node is in a memory location ROOT
loop:
        mov rbx, QWORD PTR [rax + LEFT]  # rbx = left child location
        mov rdx, QWORD PTR [rax + RIGHT] # rdx = right child location
        cmp QWORD PTR [rax + VAL], 0     # compare node's value to zero
        cmovl  rax, rbx                  # if val < 0 then rax = rbx -- left child
        cmovge rax, rdx                  # if val >= 0 then rax = rdx -- right child
        cmp rax, 0                       # if location of next node is 0 we are done
        jne loop                         # otherwise keep walking the tree

        int3

        .section .data
ROOT:
        .quad N0               # ROOT global variable stores address of
        # N0 node &N0 (eg points to N0)

        # A BUNCH OF NODES CONNECTED TO FORM A TREE LIKE STRUCTURE
N0:
        .quad 8                # N0.VAL   = 8
        .quad N1               # N0.LEFT  = &N1
        .quad N2               # N0.RIGHT = &N2

N1:
        .quad 0                # N1.VAL   = 0
        .quad N3               # N1.LEFT  = &N3
        .quad N4               # N1.RIGHT = &N4

N2:
        .quad -5               # N2.VAL   = -5
        .quad N5               # N2.LEFT  = &N5
        .quad N6               # N2.RIGHT = &N6

N3:
        .quad 6                # N3.VAL   = 6
        .quad 0                # N3.LEFT  = 0
        .quad 0                # N3.RIGHT = 0

N4:
        .quad 90               # N4.VAL   = 90
        .quad N7               # N4.LEFT  = &N7
        .quad 0                # N4.RIGHT = 0

N5:
        .quad -3               # N5.VAL   = -3
        .quad N7               # N5.LEFT  = &N7
        .quad 0                # N5.RIGHT = 0

N6:
        .quad 567              # N6.VAL   = 567
        .quad 0                # N6.LEFT  = 0
        .quad 0                # N6.RIGHT = 0

N7:
        .quad -8               # N7.VAL   = -8
        .quad 0                # N7.LEFT  = 0
        .quad 0                # N7.RIGHT = 0
```

13.2.1. To assemble and link

```
as -g -a=tree.o.lst tree.S -o tree.o
ld -g -Map=tree.map tree.o -o tree
```

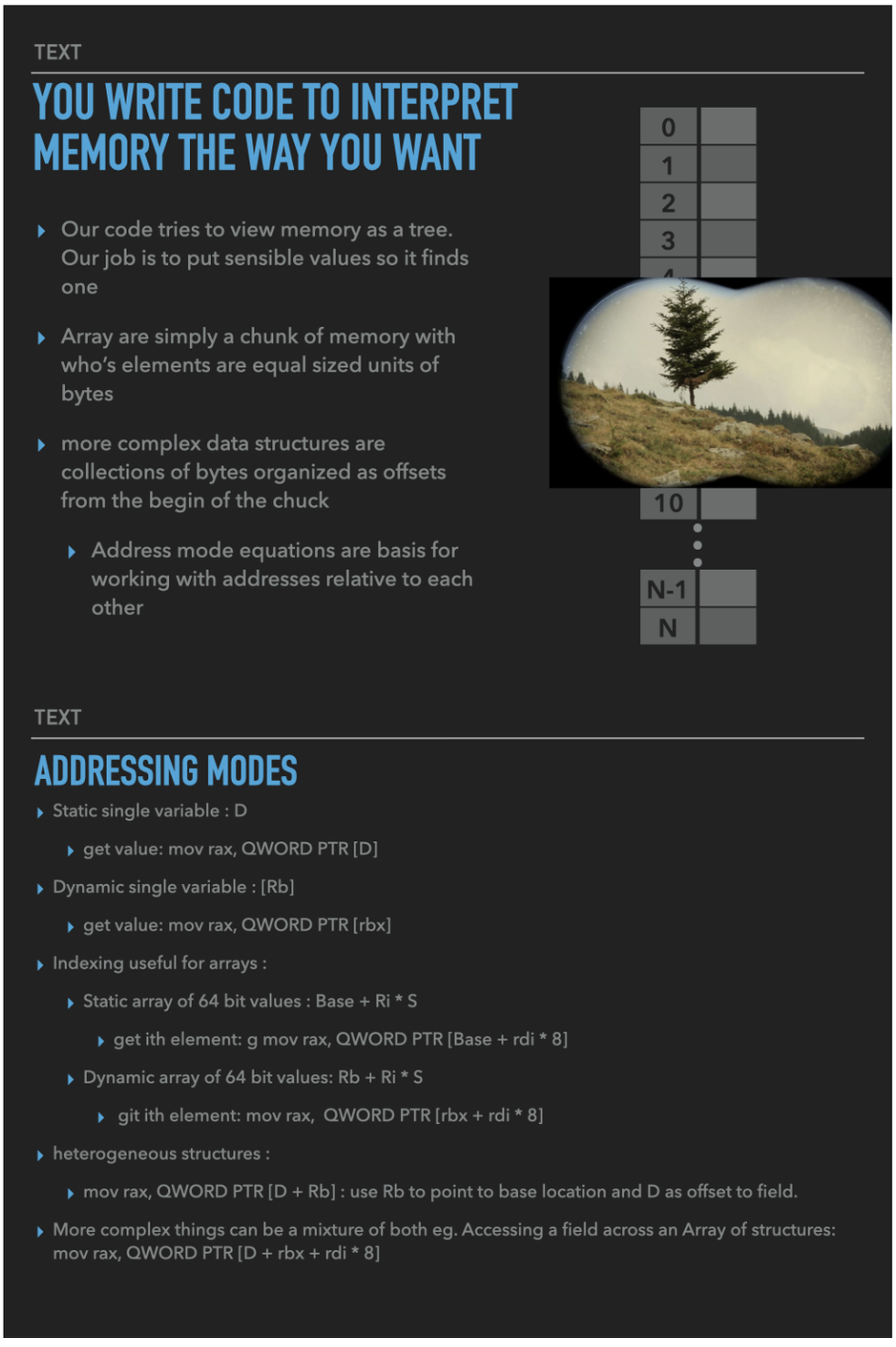
13.2.2. Exploring the tree with gdb

```
set pagination off
set disassembly-flavor intel
x/1gx &ROOT
x/16xb 0x402008
x/8xh 0x402008
x/4xw 0x402008
x/2xg 0x402008

x/9i _start

b _start
run
display /x $rax
display /1dg $rax + 0
display /2gx $rax + 8

b loop
c
c
c
c
c
```



13.3. A more complex example

This example should help to get your creative juices flowing and get a deeper appreciation for how we use the computer to write the kind of code you are used to.

Note I have not tested this much. I encourage you to try the exercises and test the code out.

13.3.1. The Story: An Array of Players

1. **Lets assume in our program we have an array of "Players"**
2. **Our program will have routines that work on the array and on individual players**
3. **In our example we will layout a static version of the array with a few players**

Remember to draw things out to ensure you are understanding things

13.3.1.1. A Player

- **Lets use a chuck of memory to represent a player**
- **Each player has:**
  1. **ID: A binary value that can fit in 8 bytes to uniquely identify a player**
  2. **Name: A "string" : An array of ascii characters with 0 to mark the end of the string**
     - **maximum length of the string array is 80**
  3. **Score: A four byte signed integer value**
  4. **Age: A single byte unsigned integer value**

13.3.1.2. The Array of Players

- **Lets assume there is one global Array for the players**
  - **One symbol `PLAYER_ARRAY` should mark the beginning of the player Array**

- One symbol `PLAYER_ARRAY_END` should mark the end of the player Array

13.3.2. Our “main” program

- The following is our main program that has the “entry point”
  - in this case it will simply call our `find_player`
- It also lays out the memory for the static global player array
  - It initializes the players with some hard code players
- When done exits passing the return value for find player as the process exit code

CODE: asm - playertest.S

```
        .intel_syntax noprefix
        .section .text

        .global _start
_start:
        mov rdi, OFFSET PLAYER_ARRAY
        mov rsi, (PLAYER_ARRAY_END - PLAYER_ARRAY)/93   #ugly magic number

        call find_player

        mov rdi, rax
        mov rax, 60
        syscall

        .data
PLAYER_ARRAY:
        .quad   7            # id
        .string "The Doctor" # name
        .zero   80 - 11      # fill rest of name array with 0
        .int    42           # score
        .byte   255          # age

        .quad   37           # id
        .string "Bugs Bunny" # name
        .zero   80 - 11      # fill rest of name array with 0
        .int    -4           # score
        .byte   9            # age
PLAYER_ARRAY_END:
```

13.3.3. `find_player`

This routine searches an Array of Players:

- **starting from the beginning of array**
- **find the first player with capital ‘B’ in their name**
- **Either returns the index of the found player or -1**

Arguments: Address of the Array and length of the Array

CODE: asm - findplayer.S

```
        .intel_syntax noprefix
        # EXAMPLE ASSEMBLY CODE OF SOMETHING A LITTE MORE REALISTIC
        #   NOTE THIS IS BY NO MEANS MEANT TO BE THE MOST EFFICIENT
        #   OR ADVANCE WAY OF WRITING THIS CODE.  RATHER IT IS MEANT
        #   TO BE SIMPLE AND HOPEFULLY CORRECT

        # Player Structure
        #   id   : 8 byte id
        #   name : 80 byte ascii encoded name
        #   score: 4  byte score
        #   age  : 1  byte age
        #  total number of bytes for a player is 8 + 80 + 4 + 1 = 93
        .equ PLAYER_STRUCT_SIZE, 93   # size of player structure in bytes

        # offsets to start of each field
        .equ PLAYER_ID_OFFSET,0       # offset 8 byte unsigned id
        .equ PLAYER_NAME_OFFSET,8     # offset 80 byte ascii name
        .equ PLAYER_SCORE_OFFSET, 88  # offset 4 byte score
        .equ PLAYER_AGE_OFFSET, 92    # offset 1 Byte unsigned age


        #  Routine to search an array of player structures
        #  to find first player who's name contains a 'B'
        #  We assume the location of the array is passed in %rdi
        #  and %rsi contains the length of the array.
        #  Each element of the array is a player structure
        #  When done the index of the first player found that
        #  has a B in its name shoud be left in %rax.
        #  if not found then %rax should contain -1

        # INPUTS
        # rdi -> array : address of player array
        # rsi -> len   : length of player array
        # OUTPUTS
        # rax -> i : index of player with B in name or -1 if none found

        # REGISTER USED AS TEMPORARIES
        # rdx -> player_ptr : pointer to the ith player structure
        # r8  -> j        : temporary integer used to seach name
        # r9b -> tmpc     : temporary byte used to hold the jth charater of the current
        #                 : player's name
        .global find_player
find_player:
        xor rax, rax                 #  i = 0

        jmp find_player_loop_condition
find_player_loop:
        mov  rdx, rax                 # player_ptr = i
        imul rdx, PLAYER_STRUCT_SIZE  # player_ptr = i * size of player stucture
        add  rdx, rdi                 # player_ptr += array starting address
                                      # rdx now holds the address of the ith player
        xor  r8, r8                   # j=0
name_search_loop:
        # tmpc = player->name[j]
        mov  r9b, BYTE PTR [rdx +  r8 + PLAYER_NAME_OFFSET]

        cmp r9b, 'B'                 # compare tmpc to 'B'
        je  find_player_done         # found a 'B' in the ith player name
        cmp r9b, 0                   # is the current character 0 if so end of name
        je  name_search_loop_end     # done searching this player's name exit name loop
        inc r8                      # j++
        jmp name_search_loop          # goto top of name search loop to examine next byte in name
name_search_loop_end:

        inc rax                     # i++
find_player_loop_condition:
        cmp rax,  rsi                # if i < len
        jl find_player_loop

find_player_notfound:
        mov rax, -1

find_player_done:
        ret
```

13.3.4. Assemble and link

```
as -g -a=playertest.o.lst playertest.S -o playertest.o
as -g -a=findplayer.o.lst findplayer.S -o findplayer.o
ld -g -Map=playertest.map playertest.o findplayer.o -o playertest
```

13.3.5. Exercises

1. **Modify the find routine to taking the search character as a parameter**
2. **Add more players**
3. **Write a routine to update a player's score**
4. **Replace the Array with a list**
    - **convert static array with static list (see tree example for inspiration)**
    - **rewrite `find_player` to search a list**