BASIC ASSEMBLY

Memory Structures

Assembly language programming
By xorpd

xorpd.net

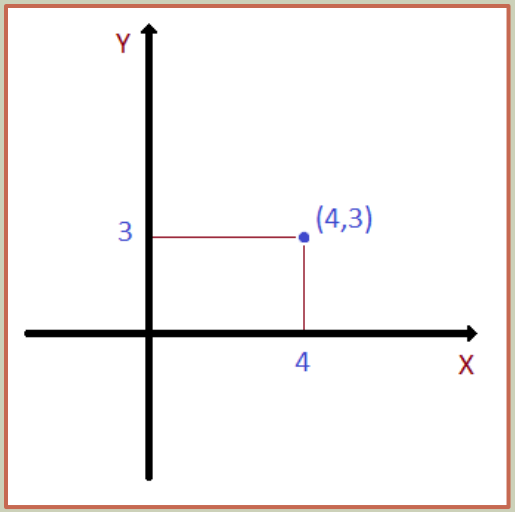
OBJECTIVES

- We will study structures.
- We will learn how to organize our data and our programs using structures.
- We will understand the idea of unions, and how to use them.

GIVING MEANING TO YOUR DATA

- The processor cares about **bytes**.
- You care about the **meaning of your data**.

- Example: How to store a point in memory?
  - A two dimensional point has two coordinates.
  - We could use two consecutive DWORDs.



401000	401001	401002	401003	401004	401005	401006	401007
04	00	00	00	03	00	00	00

X Y

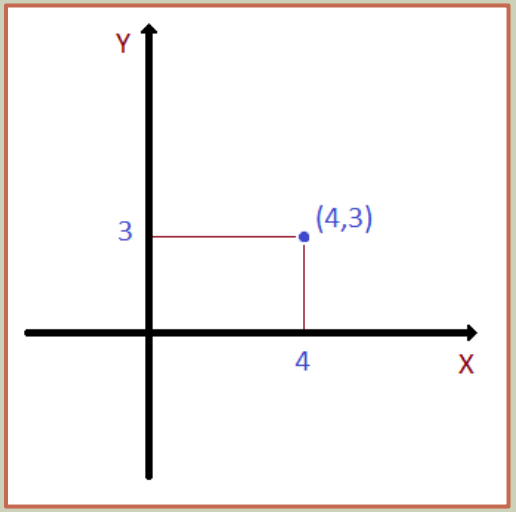
POINTS DECLARATION

- Declaring one point (First round):

```
section '.bss' readable writeable
          ; Declare a point:
          pnt       dd        ?
                    dd        ?

section '.text' code readable executable
start:

          mov       dword [pnt],4
          mov       dword [pnt + 4],3
```



- This code is a bit hard to understand.

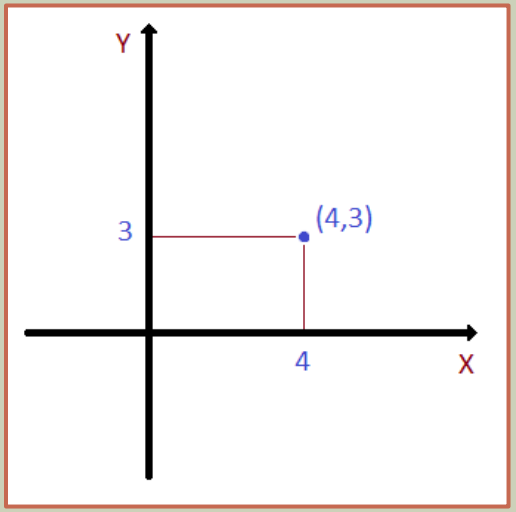
POINTS DECLARATION (CONT.)

- Declaring one point (Second round):

```
section '.bss' readable writeable
          ; Declare a point:
          px        dd        ?
          py        dd        ?

section '.text' code readable executable
start:

          mov       dword [px],4
          mov       dword [py],3
```



- A bit easier to read.

POINTS DECLARATION (CONT.)

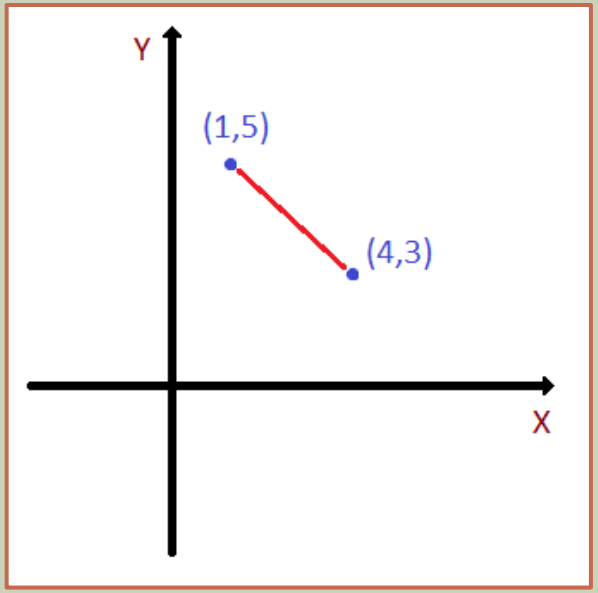
- Declaring a line (two points):

```
section '.bss' readable writeable
          ; Declare a line:
          ; First point:
          p1x       dd        ?
          p1y       dd        ?

          ; Second point:
          p2x       dd        ?
          p2y       dd        ?

section '.text' code readable executable
start:
          mov       dword [p1x],4
          mov       dword [p1y],3

          mov       dword [p2x],5
          mov       dword [p2y],1
```



- Not so fun :(

POINTS DECLARATION (CONT.)

- The problems with the line example:

```
                ; Declare a line:
                ; First point:
                p1x             dd              ?
                p1y             dd              ?
                ; Second point:
                p2x             dd              ?
                p2y             dd              ?
...
```

  - We have to write 4 lines of code to declare one line structure!
  - It is hard for the reader to understand the bigger **line** concept:
    - He only sees two points, or 4 dwords.
  - We might later want to create a triangle, for example.
    - We might want to reuse the point concept, but we have to write the same lines of code over and over again.
  - What if we wanted to change our point to be 3 dimensional?
    - We will have to change every line definition.

STRUCT

- The assembler can help us define our data structures.
- The **struct** assembly directive allows to define data structures:

```
struct PNT
    x dd ?
    y dd ?
ends

section '.bss' readable writeable
          ; Declare a point:
          my_pnt    PNT       ?

section '.text' code readable executable
start:
          mov       dword [my_pnt.x],4
          mov       dword [my_pnt.y],3
```

STRUCT (CONT.)

- Struct is just a directive, not an instruction.
- Your structs will not show up in the final binary.
  - They just help you to write your code.
  - The assembler does the dirty work for you.

```
struct PNT
    x dd ?
    y dd ?
ends

section '.bss' readable writeable
          ; Declare a point:
          my_pnt    PNT       ?

section '.text' code readable executable
start:
          mov       dword [my_pnt.x],4
          mov       dword [my_pnt.y],3
```

```
section '.bss' readable writeable
          ; Declare a point:
          pnt       dd        ?
                    dd        ?
section '.text' code readable executable
start:
          mov       dword [pnt],4
          mov       dword [pnt + 4],3
```

DEFINING STRUCTURES

- Defining structures.
  - Begins with the struct directive.
  - Ends with the ends directive.
  - Everything in the middle is considered to be data inside the structure.

```
struct PNT
    x dd ?
    y dd ?
ends
```

- Structs are usually defined before any section definitions.
  - But could be defined anywhere in your source file.

- The definition creates a set of labels:
  - `PNT.x` = 0
  - `PNT.y` = 4
  - `sizeof.PNT` = 8

- You may pick default values for the fields of the struct:

```
struct PNT
    x dd 3
    y dd 5
ends
```

STRUCTURES DECLARATIONS

- Given a structure definition, we can declare data:

```
section '.data' data readable writeable
          ; Declare point with initial values:
          my_pnt1   PNT       3,4

section '.bss' readable writeable
          ; Declare point:
          my_pnt2   PNT       ?
```

```
struct PNT
    x dd ?
    y dd ?
ends
```

- The declaration creates a set of labels:
  - my_pnt2
  - my_pnt2.x
  - my_pnt2.y

STRUCTURES DECLARATIONS

- Given a structure definition, we can declare data:

```
section '.data' data readable writeable
          ; Declare point with initial values:
          my_pnt1   PNT       3,4

section '.bss' readable writeable
          ; Declare point:
          my_pnt2   PNT       ?
```

```
struct PNT
    x dd ?
    y dd ?
ends
```

- The declaration creates a set of labels:
  - my_pnt2 = 0x402000
  - my_pnt2.x = 0x402000
  - my_pnt2.y = 0x402004

USING STRUCTURES

- Accessing one field:

```
struct PNT
    x dd ?
    y dd ?
ends

section '.data' data readable writeable
          ; Declare a point:
          my_pnt    PNT       3,4

section '.text' code readable executable
start:
          mov       eax,dword [my_pnt.y]
          call      print_eax

          mov       eax,dword [my_pnt + PNT.y]
          call      print_eax

          mov       eax,dword [my_pnt + 4]
          call      print_eax
```

- The result is 4.

USING STRUCTURES (CONT.)

- Getting the size of the structure.

```
struct PNT3
    x dd ?
    y dd ?
    z dd ?
ends

section '.data' data readable writeable
          ; Declare a three dimensional point:
          my_pnt    PNT3      5,6,7
          end_pnt:

section '.text' code readable executable
start:
          mov       eax,sizeof.PNT3
          call      print_eax

          mov       eax,end_pnt - my_pnt
          call      print_eax
```

- Here the size is 0xC.

NESTING STRUCTURES

- You can define structures using other structures:

```
struct PNT
    x dd ?
    y dd ?
ends

struct CLINE
    color   dd  ?
    p_start PNT ?
    p_end   PNT ?
ends
```

```
section '.data' data readable writeable
        ; Declare a colored line:
        my_line   CLINE     0,<3,4>,<1,5>

section '.text' code readable executable
start:
        mov       eax,dword [my_line.color]
        call      print_eax ; 0

        mov       eax,dword [my_line.p_start.x]
        call      print_eax ; 3

        mov       eax,dword [my_line.p_end.y]
        call      print_eax ; 5
```

CLINE																			
color				p_start								p_end							
color				x				y				x				y			
0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	10	11	12	13

NESTING STRUCTURES (CONT.)

- You may also nest anonymous structures (Without a name).

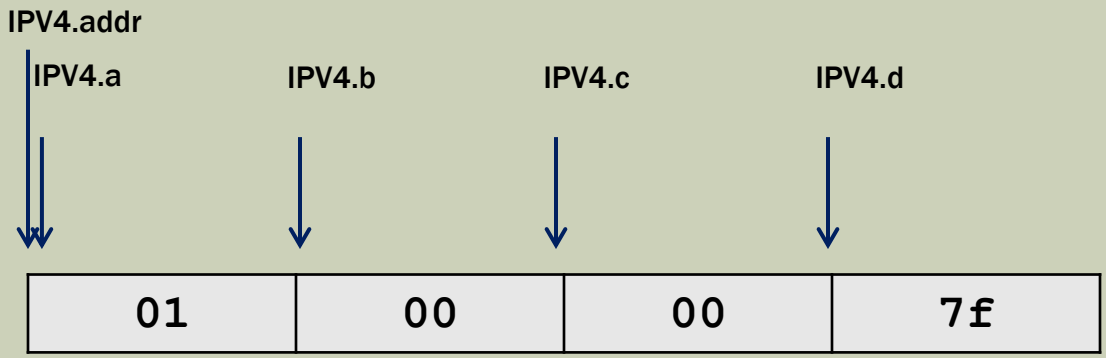
```
struct DLINE
    struct          ; (Anonymous)
        red   db ?
        green db ?
        blue  db ?
              db ?  ; (placeholder)
    ends
    p_start PNT ?
    p_end   PNT ?
ends
```

- In this example:
  - DLINE.red = 0x0
  - DLINE.green = 0x1
  - DLINE.blue = 0x2
  - DLINE.p_start = 0x4
  - DLINE.p_end = 0xC
  - sizeof.DLINE = 0x14

UNIONS

- We sometimes want to think about the same chunk of data in two (or more) different ways.
  - This is what **unions** are for.

- Example: We want to store IP address (IPv4) as a dword, but also be able to access each byte separately.




```
struct IPV4
    union
        struct
            a db ?
            b db ?
            c db ?
            d db ?
        ends
        addr dd ?
    ends
ends
```

- Unions basically create more labels with the same values.

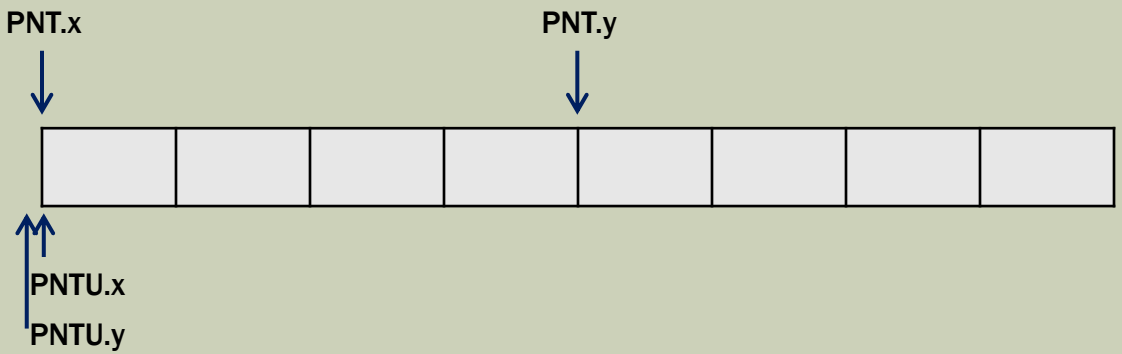
UNIONS (CONT.)

- Defining unions:
  - Unions are defined inside structs definitions.
  - They are anonymous (Have no name).
  - The definition begins with the union directive, and ends with ends.
- Inside unions, the offset does not increase.

```
struct PNT
    x dd ?
    y dd ?
ends
```

```
struct PNTU
    union
        x dd ?
        y dd ?
    ends
ends
```

- Example:
  - PNT.x = 0
  - PNT.y = 4
  - PNTU.x = 0
  - PNTU.y = 0



UNIONS IPV4 EXAMPLE

```
section '.data' data readable writeable
        ; localhost:
        lhost     IPV4      <127,0,0,1>

section '.text' code readable executable
start:
        mov       eax,dword [lhost.addr]
        ; eax == 0x0100007f

        mov       eax,dword [lhost]
        ; eax == 0x0100007f


        mov       bl, byte [lhost.d]
        ; bl == 1

        mov       bl, byte [lhost + 3]
        ; bl == 1
```

```
struct IPV4
    union
        struct
            a db ?
            b db ?
            c db ?
            d db ?
        ends
        addr dd ?
     ends
ends
```

SUMMARY

- Structures help us to declare meaningful objects in memory.
- In assembly language, Structures are just a smart way to define labels.
  - They only help you to write your program.
  - Can not be seen in the resulting binary.
- Unions allow us to deal with the same memory location in more than one way.

EXERCISES

- Read code.
- Write code.
- Have fun :)