Structs

Structs

- A struct is a compound object

```
struct Customer {
    int  id;
    char name[71];
    char address[71];
    int  balance;
};
```

- How big is this structure?
- There is the easy, but wrong answer
- and the somewhat more complex answer

Structs

- Let us focus on emulating a structure in assembler, without any consideration for C interfacing.
- Easy mode enabled.

Structs

- Let us focus on emulating a structure in assembler, without any consideration for C interfacing.
- Easy mode enabled.
- An `int` is 4 bytes. We have 2 of these in our structure, so that's 8 bytes

Structs

- Let us focus on emulating a structure in assembler, without any consideration for C interfacing.
- Easy mode enabled.
- An `int` is 4 bytes. We have 2 of these in our structure, so that's 8 bytes
- A `char` is 1 byte and we have 2 arrays of 71 characters each, so this is another 142 bytes

Structs

- Let us focus on emulating a structure in assembler, without any consideration for C interfacing.
- Easy mode enabled.
- An `int` is 4 bytes. We have 2 of these in our structure, so that's 8 bytes
- A `char` is 1 byte and we have 2 arrays of 71 characters each, so this is another 142 bytes
- So 150 in total.

Structs

If we want to allocate space for our struct, a simple call to `malloc` will suffice.

```
mov     rdi, 150          ; size of a Customer
call    malloc
mov     [c], rax          ; save the address
```

So `c` now holds a pointer to our allocated structure.

- But how do we use the struct?

Structs

If we want to allocate space for our struct, a simple call to `malloc` will suffice.

```
mov     rdi, 150          ; size of a Customer
call    malloc
mov     [c], rax          ; save the address
```

So `c` now holds a pointer to our allocated structure.

- But how do we use the struct?
- By using offsets.

```
struct Customer {
   int  id;                    [rax]------->id
   char name[71];              [rax+4]----->name
   char address[71];           [rax+75]---->address
   int  balance;               [rax+146]--->balance
};
```

Filling in a C struct

```
char * strcpy ( char * destination, const char * source );
```

```
   segment .data
name    db "Bob",0
address db "22 Duncun street",0
balance dd 123
.....................
        mov     [rax], dword 7  ; set the id
        lea     rdi, [rax+4]    ; name field
        lea     rsi, [name]     ; name to copy to struct
        call    strcpy
        mov     rax, [c]
        lea     rdi, [rax+75]   ; address field
        lea     rsi, [address]  ; address to copy
        call    strcpy
        mov     rax, [c]
        mov     edx, [balance]
        mov     [rax+146], edx
```

Assembly struct

- Using the `yasm` `struc` pseudo-op we can define a Customer

```
        struc   Customer
id      resd    1
name    resb    71
address resb    71
balance resd    1
        endstruc
```

- `id`, `name`, `address` and `balance` are globals
- It's almost the same as doing 4 equates
- The size is `Customer_size`

Assembly struct

- Using the `yasm` `struc` pseudo-op we can define a Customer

```
        struc   Customer
id      resd    1
name    resb    71
address resb    71
balance resd    1
        endstruc
```

- `id`, `name`, `address` and `balance` are globals
- It's almost the same as doing 4 equates
- The size is `Customer_size`
- But, you could not have `id` in 2 structs

Assembly struct

- One alternative is to prefix field names with dots

```
         struc   Customer
.id      resd    1
.name    resb    71
.address resb    71
.balance resd    1
         endstruc
```

- Then you would have to use `Customer.id`
- Another alternative is to use an abbreviated prefix

```
        struc   Customer
c_id      resd    1
c_name    resb    71
c_address resb    71
c_balance resd    1
        endstruc
```

Program to allocate and fill a struct - data segment

```
        segment .data
name    db      "Calvin", 0
address db      "12 Mockingbird Lane",0
balance dd      12500
        struc   Customer
c_id      resd    1
c_name    resb    71
c_address resb    71
c_balance resd    1
        endstruc
c       dq      0       ; to hold a Customer pointer
```

Program to allocate and fill a struct - part of text segment

```
mov     rdi, Customer_size
call    malloc
mov     [c], rax    ; save the pointer
mov     [rax+c_id], dword 7
lea     rdi, [rax+c_name]
lea     rsi, [name]
call    strcpy
mov     rax, [c]    ; restore the pointer
lea     rdi, [rax+c_address]
lea     rsi, [address]
call    strcpy
mov     rax, [c]    ; restore the pointer
mov     edx, [balance]
mov     [rax+c_balance], edx
```

Size Discrepancy

- Now the hard question. How big would the same C++ struct be?

Size Discrepancy

- Now the hard question. How big would the same C++ struct be?
- 152 **bytes**

C Alignment, and Struct padding

- This happens because C/C++ enforces primitives to have specific alignment based on its size.
- In effect C/C++ is padding the struct to achieve this alignment.

C Alignment, and Struct padding

- This happens because C/C++ enforces primitives to have specific alignment based on its size.
- In effect C/C++ is padding the struct to achieve this alignment.
- Certain data types have specific alignment requirements.
- The ones relevant to us in 64-bit linux are:
  - chars(1 byte) have no alignment requirement.
  - shorts(2 bytes) must start on an even address (multiple of 2).
  - int,float(4 bytes) must start on an multiple of 4
  - long,double(8 bytes) must start on a multiple of 8
  - pointer must start on a multiple of 8
- Furthermore alignment must still be preserved across struct elements in an array.

C Alignment, and Struct padding

- For example the struct

```
struct example
{
   char *p;       // 8 bytes
   char  c;       // 1 byte
   int   x;       // 4 bytes
}
```

C Alignment, and Struct padding

- For example the struct

```
struct example
{
   char *p;       // 8 bytes
   char  c;       // 1 byte
   int   x;       // 4 bytes
}
```

- Will actually be stored as

```
struct example
{
   char *p;       // 8 bytes
   char  c;       // 1 byte
   char  pad[3]; // 3 bytes
   int   x;       // 4 bytes
}
```

C Alignment, and Struct padding

- For example the struct

```
struct example
{
   char  c;      // 1 byte
   char *p;      // 8 bytes
}
```

C Alignment, and Struct padding

- For example the struct

```
struct example
{
   char  c;      // 1 byte
   char *p;      // 8 bytes
}
```

- Will actually be stored as

```
struct example
{
   char  c;      // 1 byte
   char  pad[7];// 7 bytes
   char *p;      // 8 bytes
}
```

C Alignment, and Struct padding

- The padding also has to be applicable for aligning multiples of the same struct (arrays)

```
struct example
{
    int  e;  //4 bytes
    char c;  //1 byte
}
```

C Alignment, and Struct padding

- The padding also has to be applicable for aligning multiples of the same struct (arrays)

```
struct example
{
   int  e;  //4 bytes
   char c;  //1 byte
}
```

- Will actually be stored as

```
struct example
{
   int  e;        //4 bytes
   char c;        //1 byte
   char pad[3];   //3 bytes
}
```

C Alignment, and Struct padding

- ```
  struct example
  {
     long  e;  //8 bytes
     char  c;  //1 byte
  }
  ```
  ```

C Alignment, and Struct padding

- ```
  struct example
  {
     long  e;  //8 bytes
     char  c;  //1 byte
  }
  ```
- Will actually be stored as

```
struct example
{
   long  e;        //8 bytes
   char c;         //1 byte
   char pad[7];    //7 bytes
}
```
```

C Alignment, and Struct padding

- ```
  struct example
  {
      int a;  //4 bytes
      long b; //8 bytes
      char c; //1 byte
  }
  ```
  ```

C Alignment, and Struct padding

- ```
  struct example
  {
      int a;  //4 bytes
      long b; //8 bytes
      char c; //1 byte
  }
  ```
- Will actually be stored as

  ```
  struct example
  {
      int a;         //4 bytes
      char pad[4]; //4 bytes
      long b;        //8 bytes
      char c;        //1 byte
      char pad2[7];//7 bytes
  }
  ```

  Why 7 and not 3?
  ```

Allocating a slightly more complex array of customers

```
            segment .data
            struc   Customer
c_id        resd    1       ; 4 bytes in total
c_name      resb    65      ; 69 bytes in total
c_address   resb    65      ; 134 bytes in total
            align   4       ; aligns to 136
c_balance   resd    1       ; 140 bytes in total
c_rank      resb    1       ; 141 bytes in total
            align   4       ; aligns to 144
            endstruc
customers   dq      0
            segment .text
            mov   rdi, 100 ; for 100 structs
            imul   rdi, Customer_size
            call  malloc
            mov   [customers], rax
```

Printing an array of customers

```
            segment .data
format      db    "%s %s %d",0x0a,0
            segment .text
            push rbp
            mov rbp, rsp
            push  r15
            push  r14
            mov   r15, 100          ; counter saved through calls
            mov   r14, [customers]; pointer saved through calls
more        lea   rdi, [format]
            lea   rsi, [r14+c_name]
            lea   rdx, [r14+c_address]
            mov   ecx, [r14+c_balance]
            mov   rax, 0
            call  printf
            add   r14, Customer_size
            dec   r15
            jnz   more
            pop   r14
            pop   r15
            leave
```