Today

- Pointers (Function)
- More struct alignment
- Pointers in Assembly
- Big procedures and pointer example!

Function pointer

points to a function.

```
int sayHi(int x)
{
    ______
}
```

1: `int (*fp)(int);`
- int → return value
- (int) → arguments

2: `fp = sayHi;`

3: `int x = fp(5);`

Ex: 2

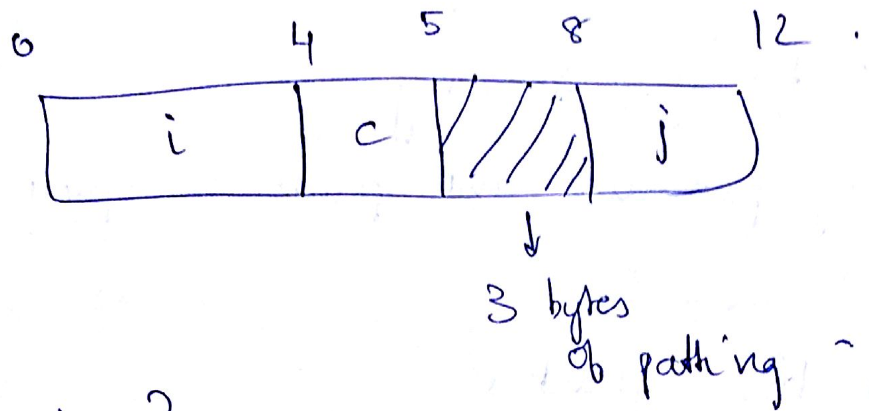
`int* func(int x, int y);`

`int* (*fp)(int, int);`

Scanned by CamScanner

Data Alignment & Structures

```
struct s1
{
    int i;
    char c;
    int j;
};
```



Why? → To make int j start at a memory location that is a multiple of 4.

Basic idea

char will be 1 byte aligned.

short ⟶ 2 - byte aligned

int ⟶ 4 - byte aligned

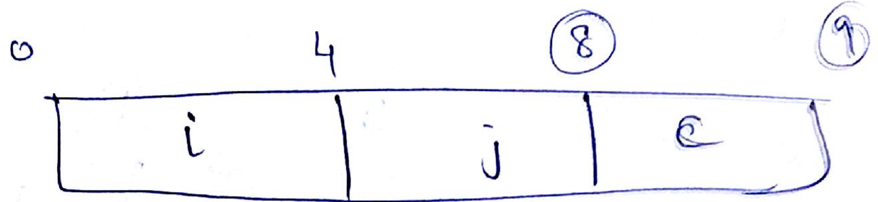
(float, pointer)

double ⟶ 8 - byte aligned.

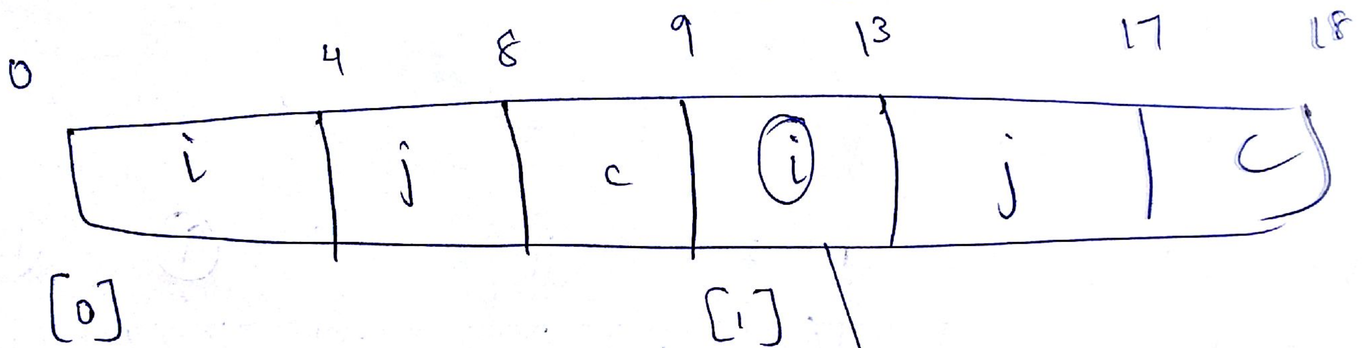
Scanned by CamScanner

Example 2

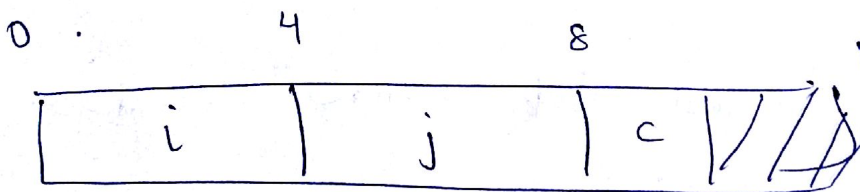
```
struct s2
{
  int i;
  int j;
  char c;
};
```



struct S2 arr [2]



9 is not a multiple of 4 !!!



→ So we add 3 bytes of padding

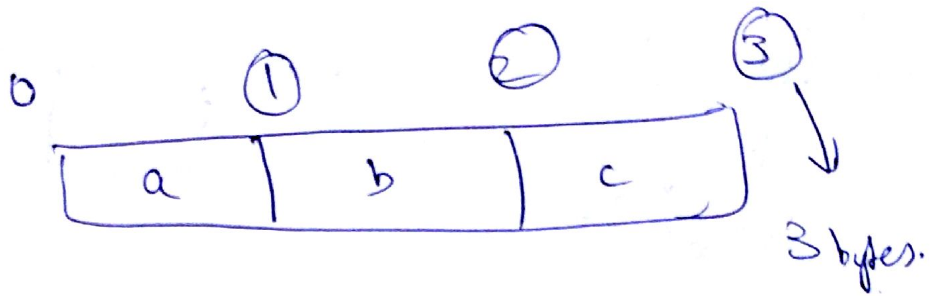
General rule of thumb

⇒ Total size of the struct is a multiple of the largest data member's size //

Scanned by CamScanner

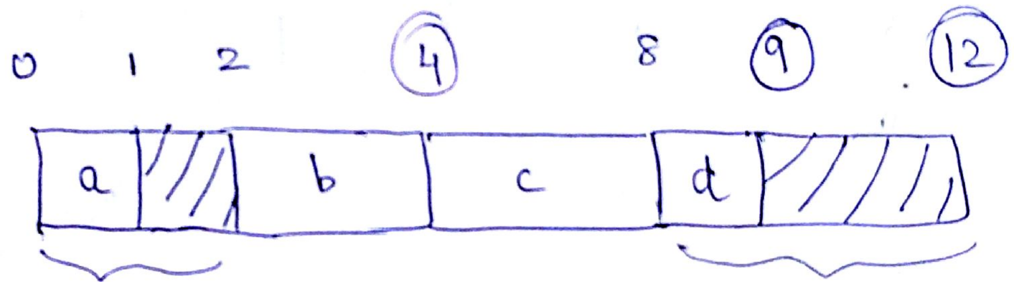
Ex : 1

```
struct s3
{
    char a;
    char b;
    char c;
}
```



Ex : 2

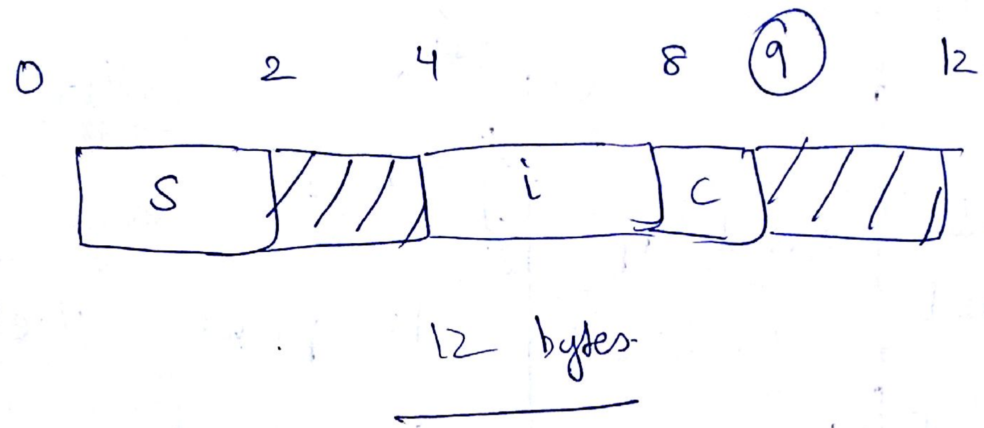
```
struct S4
{
    char a;
    short b;
    int c;
    char d;
}
```



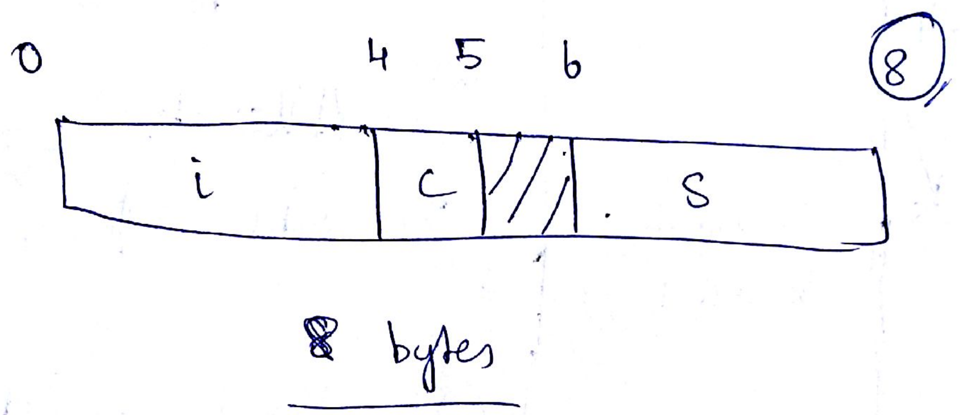
Largest data member => integer

Scanned by CamScanner

```
struct S5
{
    short s;
    int i;
    char c;
};
```




```
struct S6
{
    int i;
    char c;
    short s;
};
```



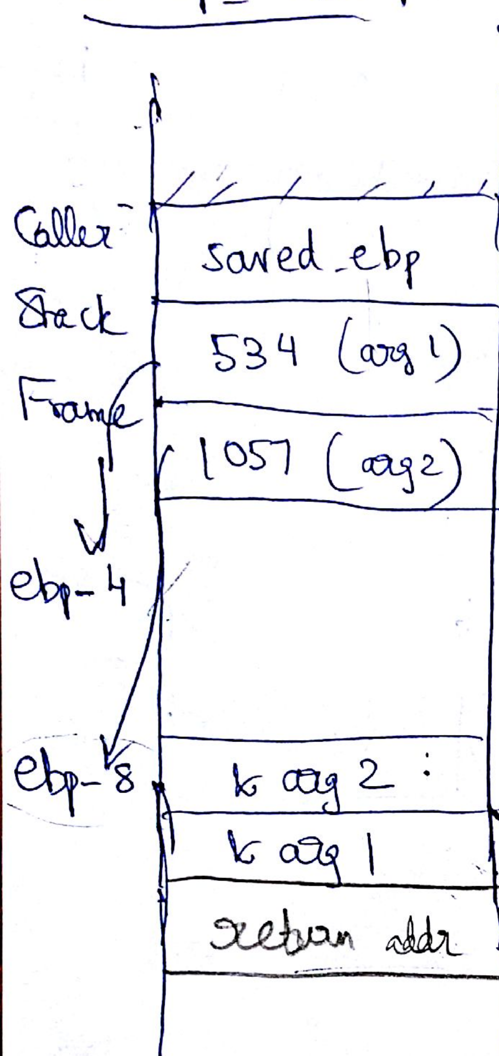
Pointers in Assembly. (Show slides)

Why 24 bytes? (From next page)

Whenever you allocate for the stack, you want the stack size to be a multiple 16. (For data alignment)

Scanned by CamScanner

Swap_add. 3.7.4



Caller

① Stack setup

1: pushl %ebp

2: movl %esp, %ebp

ebp

② Allocate 24 bytes

3: subl $24, %esp

③ Local variables)

4: movl $534, -4 (%ebp)

5: movl $1057, -8 (%ebp)

④ Set up the arguments

set up &arg2

6: leal -8 (%ebp), %eax

7: movl %eax, 4 (%esp)

Sets up &arg 1

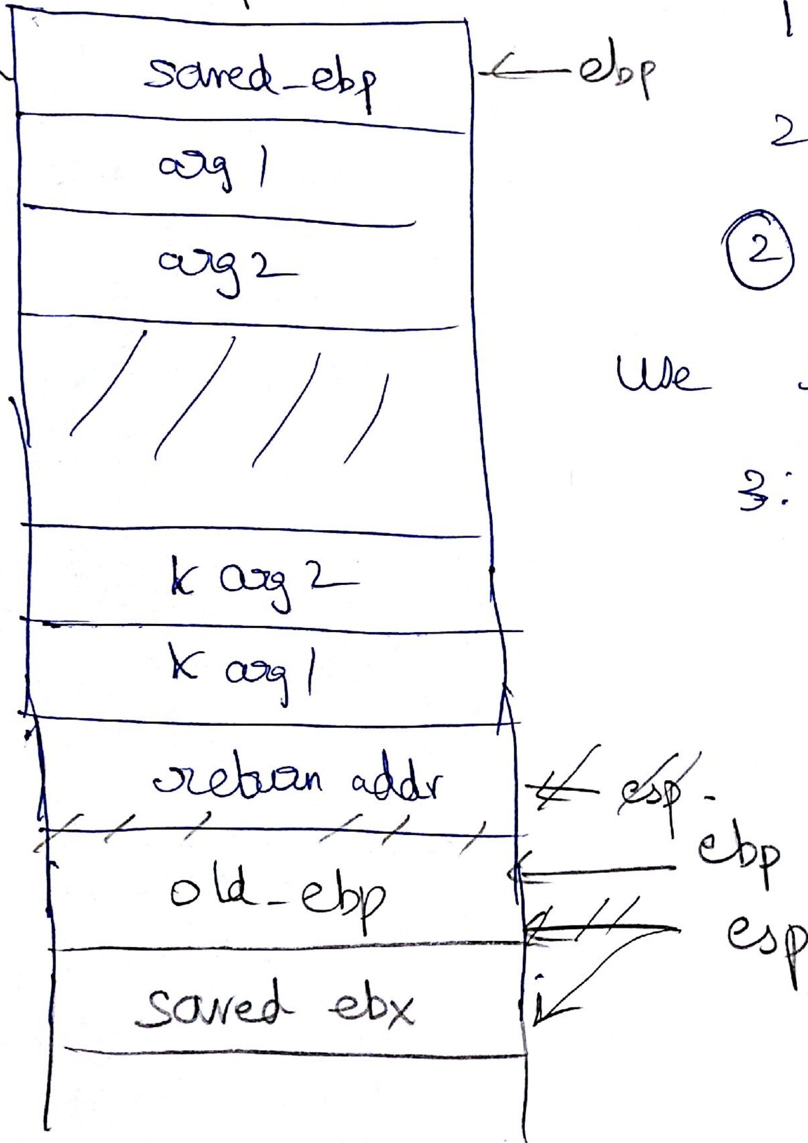
8: leal -4 (%ebp), %eax

9: movl %eax, (%esp)

Scanned by CamScanner

⑤ 10: call swap_add

11: ______________



① Stack setup.

1: push %ebp

2: movl %esp, %ebp

② Swap_add is going to use register ebx.

3: push %ebx

Scanned by CamScanner

③

4: movl 8(%ebp), edx (gets xp)

5: movl 12(%ebp), ecx (gets yp)

Now, edx → xp ecx → yp.

④ Get the value of x & y

6: movl (%edx), %ebx (get x)

7: movl (%ecx), %eax (get y).

Now ebx → x eax → y.

---

⑤ Swap

8: movl %eax, (%edx) (Store y at xp)

9: movl %ebx, (%ecx) (Store x at yp)

⑥ Return x + y

10: addl %ebx, %eax

return values are stored in %eax

Scanned by CamScanner

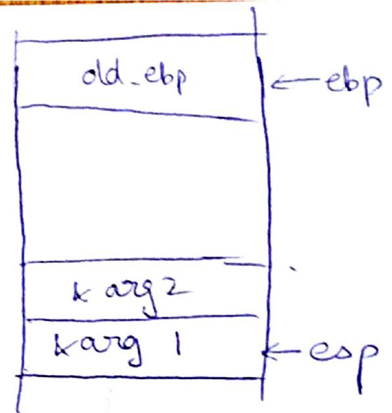
⑦ Restore %ebx

11: popl %ebx

⑧ Stack Teardown

12: popl %ebp

13: ret



(Restore %ebp to ~~point to~~ its old value)
i.e make it point to the caller's base address.

Control now gets transferred to the instruction after the call instruction in caller

~~Finish reading in 3.7.4~~

<Caller>

:

11: movl -4(%ebp), %edx (Move arg1 to edx)

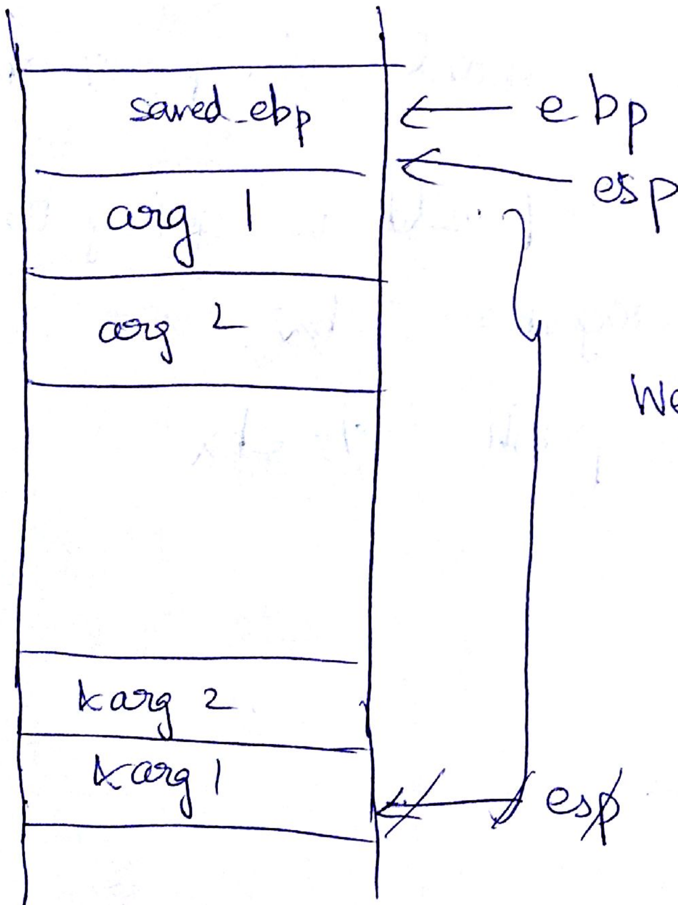
12: subl -8(%ebp), %edx (arg1 - arg2)

13: imull %edx, %eax ⇒ return value

Stack teardown of caller

14: Leave ⇒ movl %ebp , %esp
popl %ebp

15: ret



We "deallocate" all this space on the stack frame

Scanned by CamScanner