CS 354 - Machine Organization & Programming
Tuesday Nov 14, Thursday Nov 16, 2023

Exam Results expected by Friday Nov 17

Homework hw5DUE Monday 11/13**Homework hw6:** DUE on or before Monday 11/20

Homework hw7: DUE on or before Monday 11/27

Project p5: DUE on or before Friday Nov 24

Learning Objectives

able to trace function call and its stack frame
able to access parameters and local variables based on location from %ebp and %esp
able to trace recursive function calls through their stack frame
identify and describe effects of ASM **call**, **ret**, **leave** instructions
able to access 1D array element using ASM instructions and memory operand types
able to access multidimensional array via ASM instructions and memory operand types
describe, compute, and use alignment requirements of elements in structs and unions
understand the difference and use of structs and unions in C.

This Week

Function Call-Return Example (L20 p7) Recursion Stack Allocated Arrays in C Stack Allocated Arrays in Assembly Stack Allocated Multidimensional Arrays	Stack Allocated Structs Alignment Alignment Practice Unions
Next Week: Pointers in Assembly, Stack Smashing, and Exceptions B&O 3.10 Putting it Together: Understanding Pointers 3.12 Out-of-Bounds Memory References and Buffer Overflow 8.1 Exceptions 8.2 Processes 8.3 System Call Error Handling 8.4 Process Control through p719	

Recursion

Use a stack trace to determine the result of the call `fact(3)`:

```
int fact(int n) {
   int result;
   if (n <= 1) result = 1;
   else        result = n * fact(n - 1);
   return result;
}
```

direct recursion when func calls itself

recursive case calls recursive func

base case stops recursive

"infinite" recursion similar to infinite loop

eax | ~~-42~~ ~~1~~ ~~2~~ ~~1~~ ~~1~~ ~~1~~ ~~2~~ 6

ebx | ~~42~~ ~~3~~ ~~2~~ ~~1~~ ~~2~~ ~~1~~ ~~3~~ 42

rand

Assembly Trace

```
fact:
   pushl %ebp
   movl %esp, %ebp
   pushl %ebx
   subl $4,%esp

   movl 8(%ebp),%ebx
   movl $1,%eax

   cmpl $1,%ebx
   jle .L1

   leal -1(%ebx),%eax
   movl %eax,(%esp)
   call fact
   imull %ebx,%eax

.L1:
   addl $4,%esp
   popl %ebx
   popl %ebp
   ret
```

set up

SUB S,D = D - S = ebx - 1

// true/fall thru if ebx > 1

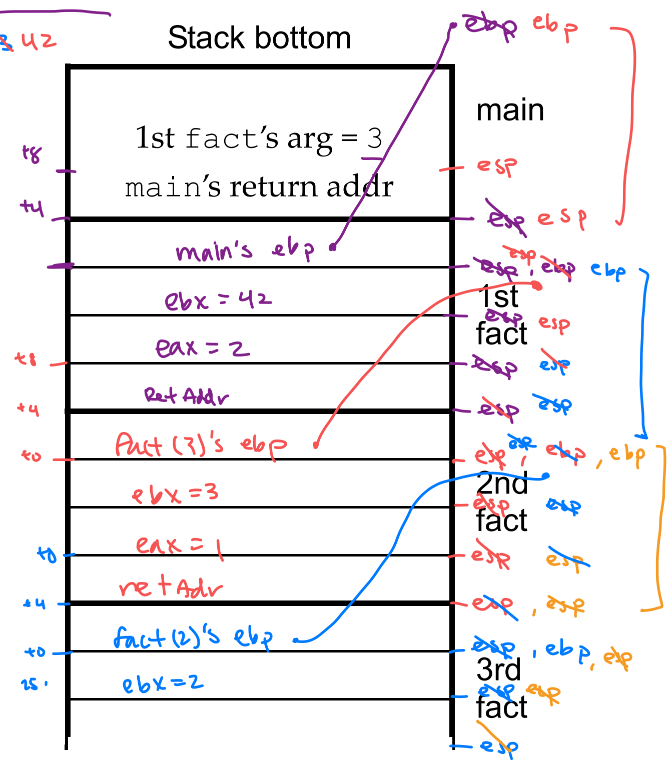
// eax --

// push ret, jmp

Ret A

= eax * ebx

pop top of stack to eip



✵ *"Infinite" recursion causes*

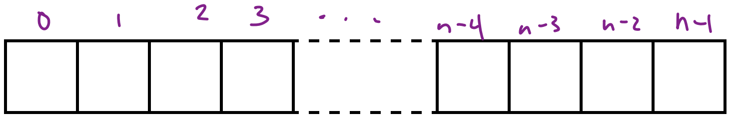
✵ *When tracing functions in assembly code*

Stack Allocated Arrays in C

Recall Array Basics

T `A[N];` where T is the element datatype of size L bytes and N is the number of elements

L = size of(T)



A:

1. contiguous region of stack L * N bytes

2. identifier is associated w/ start addr of array

❋ *The elements of* `A` are accessed using arithmetic

expressed as mem type operands in ASM

Recall Array Indexing and Address Arithmetic

`&A[i]` ≡ A + i ≡ X_A + L * i

starting addr   elt size   index

→ For each array declarations below, what is L (element size), the address arithmetic for the ith element, and the total size of the array?

	C code	L	address of ith element	total array size
	1. `int I[11]`	4 bytes	$X_I + (4 * i)$	4 * 11 = 44 bytes
	2. `char C[7]`			
	3. `double D[11]`			
*	4. `short S[42]`	2 bytes	$X_S + (2 * i)$	84 bytes
	5. `char *C[13]`			
*	6. `int **I[11]`	4 bytes	$X_I + (4 * i)$	44 bytes
	7. `double *D[7]`			

Stack Allocated Arrays in Assembly

Arrays on the Stack

→ How is an array laid out on the stack? Option 1 or 2:

higher addresses

earlier frames	
1.	**2.**
`A[0]`	`A[N-1]`
`A[1]`	`...`
`...`	`A[1]`
`A[N-1]`	`A[0]`

Stack Top — lowest addr

❋ *The first element (index 0) of an array*

is the closest elt to "top" of stack

Accessing 1D Arrays in Assembly

IA-32 Addr modes are designed to simplify array access

Assume array's start address in %edx and index is in %ecx

```
movl (%edx, %ecx, 4), %eax
```

X_A i L

$\equiv M[X_A + 4*i] \equiv *(A+i) \equiv A[i]$ in C

→ Assume `I` is an `int` array, `S` is a `short int` array, for both, the array's start address is in %edx, and the index `i` is in %ecx. Determine the element type and instruction for each:

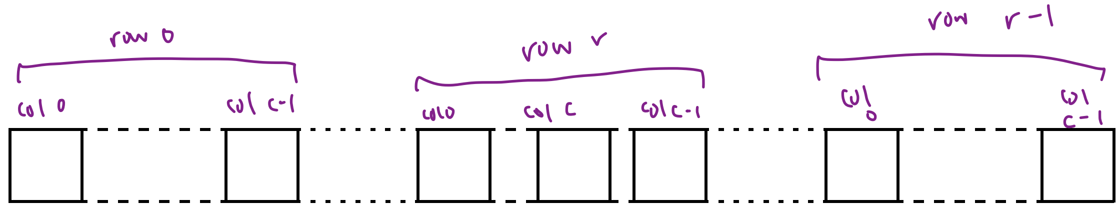
	C code	type	assembly instruction to move C code's value into %eax	
	1. `I`	int *	X_A	movl %edx, %eax
	2. `I[0]`	int	$M[X_A]$	movl (%edx), %eax
	3. `*I`			
	4. `I[i]`			
	5. `&I[2]`			
*	6. `I+i-1`	int *	$X_I + (4*i) - (4*1)$	leal -4(%edx, %ecx, 4), %eax
	7. `*(I+i-3)`			
	8. `S[3]`			
	9. `S+1`			
	10. `&S[i]`			
*	11. `S[4*i+1]`	short	$M[X_S + 2*(4*i+1)]$ $8i + 2$	movw 2(%edx, %ecx, 8), %ax
	12. `S+i-5`			movswl 2(%edx, %ecx, 8), %eax

Stack Allocated Multidimensional Arrays

Recall 2D Array Basics

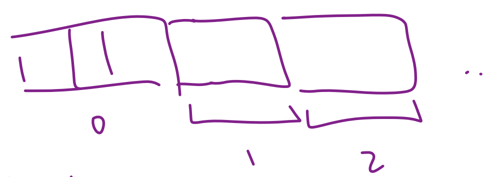
T `A[`*R*`][`*C*`];` where *T* is the element datatype of size *L* bytes,
R is the number of rows and *C* is the number of columns

identifiers



❋ *Recall that 2D arrays are stored on the stack* in row major order

```
int A[5][3];        typedef int row_t[3];
                    row_t A[5];
```



Accessing 2D Arrays in Assembly

`&A[i][j]` ≡ start addr + offset to row i + offset to col j

$x_A + (L * C * i) + (L * j)$

cols

Given array `A` as declared above, if x_A in %eax, i in %ecx, j in %edx
then `A[i][j]` in assembly is:

```
leal (%ecx, %ecx, 2), %ecx
sall $2, %edx
addl %eax, %edx
movl (%edx, %ecx, 4), %eax
```

$3i \rightarrow$ %ecx

$4j \rightarrow$ %edx

$x_A + 4j \rightarrow$ %edx

$M[x_A + 4j + 3i * 4] \rightarrow$ %eax

Compiler Optimizations

- If only accessing part of array compiler makes a ptr (addr) to that part of array

- If taking a fixed stride through the array then compiler uses stride * elt size as offset

Stack Allocated Structures

Structures on the Stack

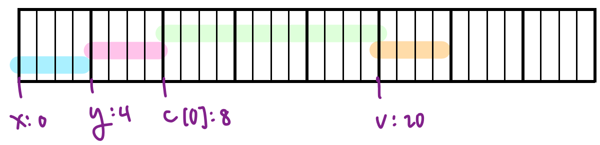
```
struct iCell {
   int x;        :0
   int y;        :4
   int c[3];     :8
   int *v;
};
```

→ How is a structure laid out on the stack? Option 1 or 2:

higher addresses

1.	2.
earlier frames	
v	x
c[2]	y
c[1]	c[0]
c[0]	c[1]
y	c[2]
x	v

Stack Top



The compiler

- assoc. data member name w/ its offset from start of struct
- uses addr arith. w/ offset to access data member

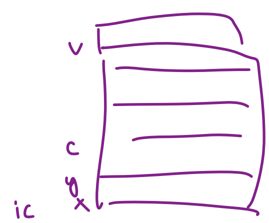
❋ *The first data member of a structure* is closest to "top" of stack (%esp)

Accessing Structures in Assembly

Given:

```
struct iCell ic = //assume ic is initialized

void function(iCell *ip) {
```



→ Assume `ic` is at the top of the stack, %edx stores `ip` and %esi stores `i`.
Determine for each the assembly instruction to move the C code's value into %eax:

C code	assembly
* 1. `ic.v`	movl 20(%esp), %eax
2. `ic.c[i]`	
* 3. `ip->x`	movl (%edx), %eax
4. `ip->y`	
* 5. `&ip->c[i]`	leal 8(%edx, %esi, 4), %eax

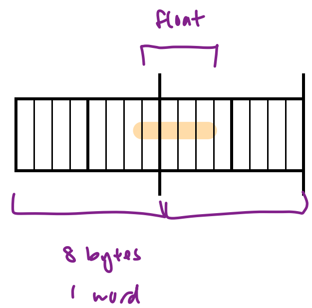
❋ *Assembly code to access a structure* does not have data member names + types

Alignment

What? most computers restrict addr values where prim can be stored

Why? better memory performance

Example: Assume cpu reads 8 byte words
f is a misaligned float



- slow : requires 2 reads , extract bits, recombine

Restrictions

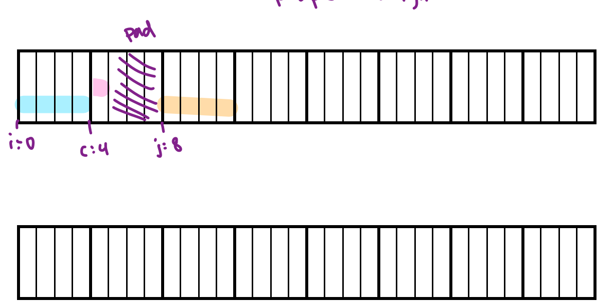
IA-32 : has no alignment req.

Linux:	short	addr must be mult of 2 , lsb = 0
	int, float, pointer, double	" " " " " 4 , ls 2bits = 00
Windows:	same as Linux except double	addr must be mult of 8
Implications	padding might be inserted by compiler into structs to keep data aligned	

Structure Example

```
struct s1 {
    int i;      : 0
    char c;     : 4
    int j;      : 8
};
```

proper align.



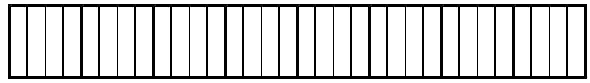
total size: 12 bytes

✵ *The total size of a structure* is typically a multiple of its longest data member size

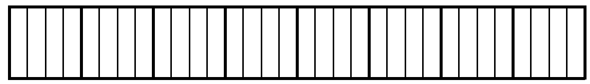
Alignment Practice

→ For each structure below, complete the memory layout and determine the total bytes allocated.

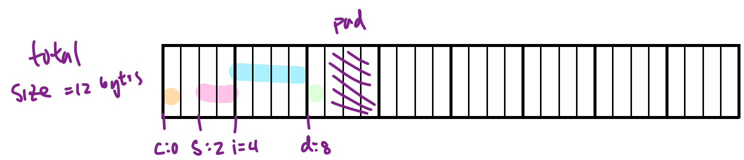
```
1) struct sA {
      int i;
      int j;
      char c;
   };
```



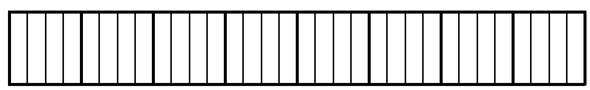
```
2) struct sB {
      char a;
      char b;
      char c;
   };
```



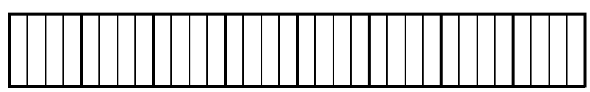
```
3) struct sC {
      char c;       : 0
      short s;      : 2
      int i;        : 4
      char d;       : 8
   };
```




```
4) struct sD {
      short s;
      int i;
      char c;
   };
```



```
5) struct sE {
      int i;
      short s;
      char c;
   };
```



✵ *The order that a structure's data members are listed* can affect mem

until b/c of padding for alignment

Unions

What? A *union* is

- like a struct, except fields share the same memory

  bypassing C's type checking

- allocates only enough mem for the largest fields

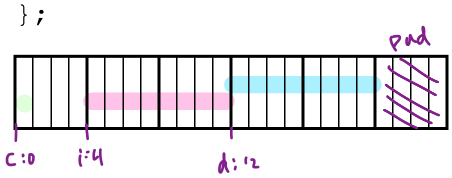
Why?

- allows data to be accessed as different types
- used to access hardware
- low level "polymorphism"

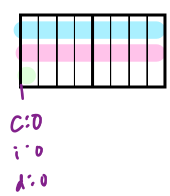
How?

total size: 24

```
struct s {
   char c;
   int i[2];
   double d;
};
```




```
union u {
   char c;      :0
   int i[2];    :0
   double d;
};
```



Example

```
typedef union {
   unsigned char cntrlrByte; :0     // all 8 bits as one char
   struct {
      unsigned char playbutn  : 1;   :0
      unsigned char pausebutn : 1;   :1
      unsigned char ctrlbutn  : 1;   :2
      unsigned char fire1butn : 1;   :3
      unsigned char fire2butn : 1;   :4
      unsigned char direction : 3;   :5-7
   } bits;
} CntrlrReg;
```