CS 354 - Machine Organization & Programming
Tuesday April 11, Thursday April 13, 2023

Exam Results expected by Friday April 14

Homework hw5DUE Monday 4/10**Homework hw6:** DUE on or before Monday Apr 17

Homework hw7: DUE on or before Monday Apr 24

Project p5: DUE on or before Friday April 21

Last Week

Instructions - SET
Instructions - Jumps
Encoding Targets
Converting Loops

The Stack from a Programmer's Perspective
The Stack and Stack Frames
Instructions - Transferring Control
Register Usage Conventions
Function Call-Return Example

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
... rest was next week f22
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

Copyright © 2016-2023 Jim Skrentny

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;//BASE CASE
   else        result = n * fact(n - 1);//REC CASE
   return result;
}
```

direct recursion when a function calls itself

recursive case when calls recursive function

base case stops recursion

"infinite" recursion similar to infinite loop, occurs when we don't reach a base case (eventually causes stack overflow, has to end)

%eax ~~1~~ ~~2~~ ~~1~~ ~~1~~ ~~1~~ ~~2~~ 6

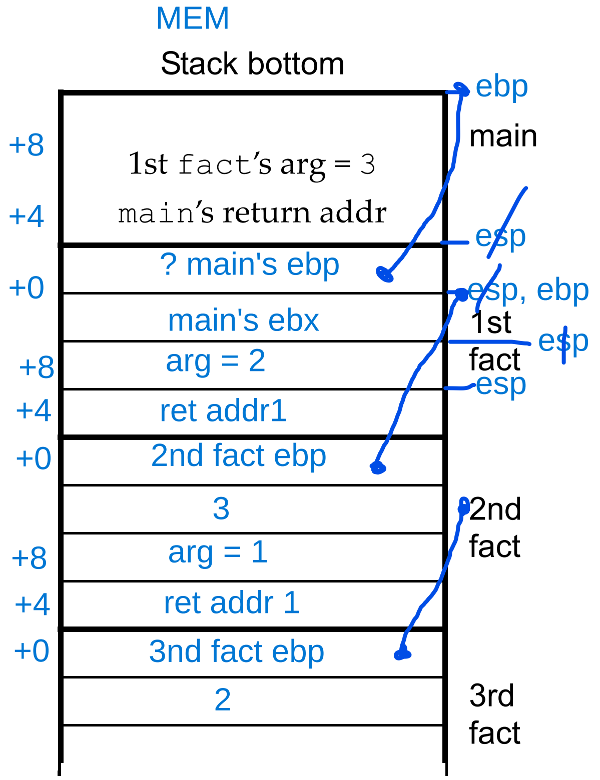
%ebx ~~?~~ ~~3~~ ~~2~~ ~~1~~ ~~2~~ ~~3~~ ?

Assembly Trace

```
fact:
   pushl %ebp
   movl %esp, %ebp        S.F.
   pushl %ebx             setup
   subl $4,%esp

   movl 8(%ebp),%ebx- 1st arg
   movl $1,%eax       - 1
        S2   S1
   cmpl $1,%ebx     ebx -1, ZF - 0, SF = 0
   jle .L1

   leal -1(%ebx),%eax
   movl %eax,(%esp)- set arg
   call fact        push ret addr
                    jmp fact
ret addr 1imull %ebx,%eax
                       eax <- eax * ebx
.L1:
   addl $4,%esp
   popl %ebx
   popl %ebp
   ret       popl %eip
```



- *"Infinite" recursion causes*

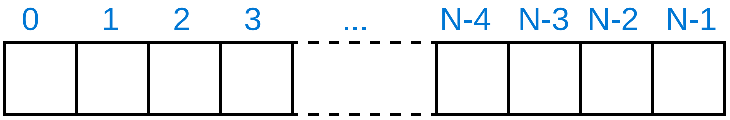
- *When tracing functions in assembly code*

Copyright © 2016-2023 Jim Skrentny

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



1. contiguous region of stack mem $\overset{\text{sizeof(T)}}{L} * N$

2. identifies assoc. with start addr of array

✵ *The elements of* `A` are accessed using address arithmetic

Recall Array Indexing and Address Arithmetic

`&A[i]` $\equiv A + i \equiv x_A + L * i$

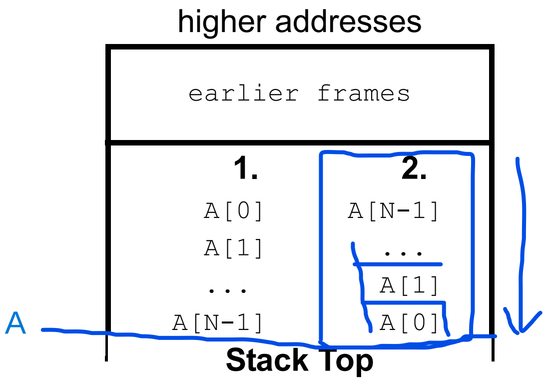
index
element size
start addr of A

→ 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	$x_I + i * 4$	44
2.	`char C[7]`			
3.	`double D[11]`			
*4.	`short S[42]`	2	$x_S + 2 * i$	84
5.	`char *C[13]`			
*6.	`int **I[11]`	4	$x_I + 4 * i$	44
7.	`double *D[7]`			

Copyright © 2016-2023 Jim Skrentny

Stack Allocated Arrays in Assembly



Arrays on the Stack

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

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

is closest 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` $\equiv m[x_A + 4 * i] \equiv *(A+i) \equiv A[i]$

x_A (under %edx), i (under %ecx), L (size of elt) (under 4)

→ 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_I	movl %edx, %eax
*2.	`I[0]`	int	$m[x_I]$	movl (%edx), %eax
3.	`*I`			
4.	`I[i]`			
*5.	`&I[2]`	int*	$x_I + \overset{i}{2} * \overset{L}{4}$	leal (%edx, %ecx, 4), %eax
*6.	`I+i-1`	int*	$x_I + (i - 1)4$	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	$m[x_S + (4 * i + l)2]$	movswl 2(%edx, %ecx, 8), %eax
12.	`S+i-5`			

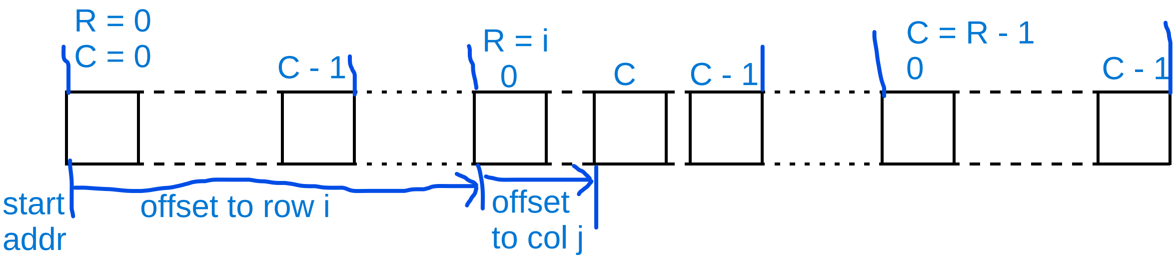
***valid scale factors include 1, 2, 4, and 8

Copyright © 2016-2023 Jim Skrentny

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



✵ *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];`	row_t is 10 array of 3 int 5 row_t array elements

Accessing 2D Arrays in Assembly

`&A[i][j]` $\equiv$ Start addr t offset to row i + offset to col j
$X_A + (L * C * i) + L * j$

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`	i i i + 2i = 3i
`sall $2, %edx`	4 * j
`addl %eax, %edx`	x_A + 4j
`movl (%edx, %ecx, 4), %eax`	x_A + 3i * 4 + 4j

$x_A + 12\,i + 4j$

Compiler Optimizations

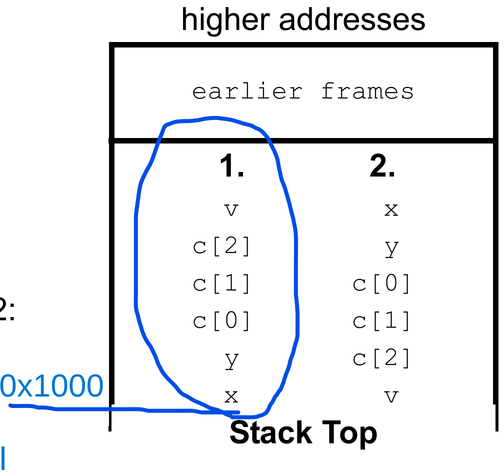
- If only accessing part of array then compiler makes a pointer to that part of array, and then compute offset from there

- If taking a fixed stride through the array the compiler uses stride * L as offset

Copyright © 2016-2023 Jim Skrentny

Stack Allocated Structures

Structures on the Stack

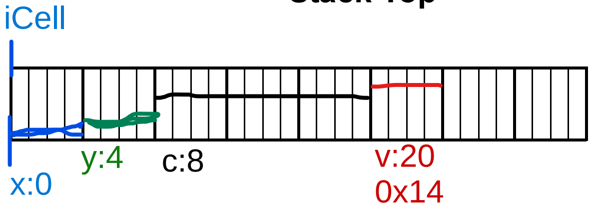


```
                    (hex)
struct iCell { #bytes  offset
   int x;        4      0x0
   int y;        4      0x4
   int c[3];     12     0x8
   int *v;       4      0x14
};
```

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

The compiler

- associates data member names with their offset from start of struct



- uses addr arith. with offsets to access data members

✵ *The first data member of a structure* is closest to top of stack

Accessing Structures in Assembly

Given:

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

void function(iCell *ip) {
```

%esp

→ 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]`	movl	4(%esp, %esi, 4), %eax
3. `ip->x`	movl	(%edx), %eax
4. `ip->y`	movl	4(%edx),%eax
5. `&ip->c[i]`	leal	8(%edx, %esi, 4), %eax

✵ *Assembly code to access a structure* does not have data member names & types

Copyright © 2016-2023 Jim Skrentny

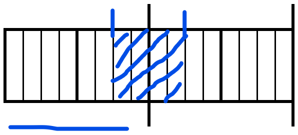
Alignment

What? Most computer systems restrict addresses where primitive data can be stored

Why? Better memory performance

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

Slower: requires two reads, extract bytes
recombine into single float value



Restrictions

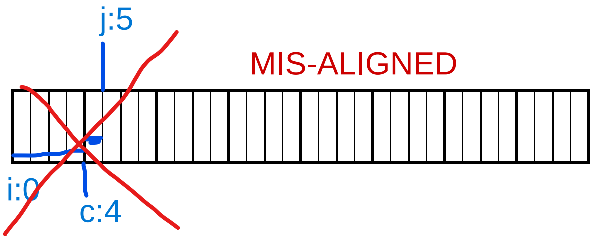
IA-32 has NO alignment restriction

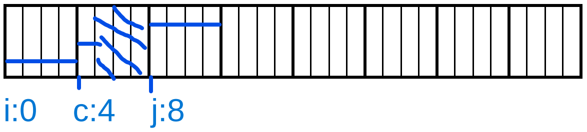
Linux:	short	addr must be mult of 2
	int, float, pointer, double	addr must be mult of 4
Windows:	same as Linux except	
	double	must be mult of 8
Implications	padding might be inserted by compiler into structs to keep data members aligned	

Structure Example

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

~~Offset~~	Offset
~~:0~~	:0
~~:4~~	:4
~~:5~~	:8



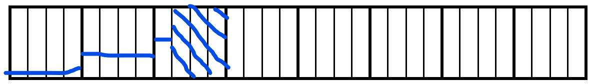


✵ *The total size of a structure* is typically a multiple of size of longest data member

Alignment Practice

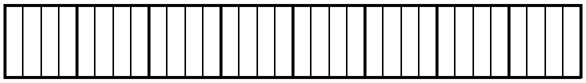
→ For each structure below, complete the memory layout and determine the total bytes allocated.

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

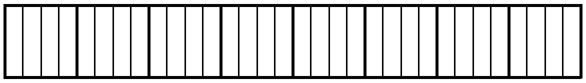


i:0 j:4 c:8 total size = 12

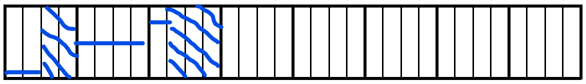
```
2) struct sB {
      char a;
      char b;
      char c;
   };
```



```
3) struct sC {
      char c;
      short s;
      int i;
      char d;
   };
```

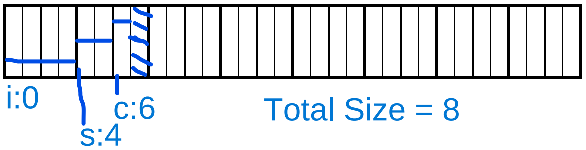


```
4) struct sD {
      short s;
      int i;
      char c;
   };
```



s:0 i:4 c:8 Total Size = 12 bytes

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



i:0 s:4 c:6 Total Size = 8

✵ *The order that a structure's data members are listed* can affect mem. utilization because of alignment padding

Copyright © 2016-2023 Jim Skrentny

Unions

What? A *union* is

- like a struct, except fields share same memory
  bypassing C's type checking

- allocates only enough memory for the largest field

Why?

- allows data to be accessed as different types

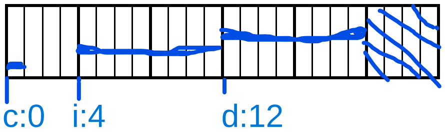
- used to access hardware

- low-level polymorphism

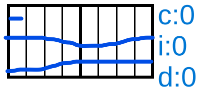
How?

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

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



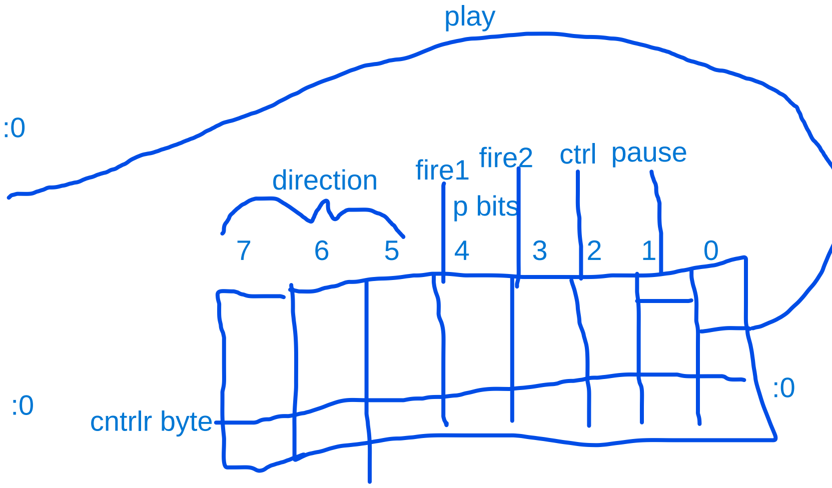
Total Size = 24 bytes



Total Size = 8 bytes

Example

```
typedef union {
   unsigned char cntrlrByte;
   struct {
      unsigned char playbutn   : 1;
      unsigned char pausebutn  : 1;
      unsigned char ctrlbutn   : 1;
      unsigned char fire1butn  : 1;
      unsigned char fire2butn  : 1;
      unsigned char direction  : 3;
   } bits;
} CntrlrReg;
```



Cntrlr Reg r1; //1 byte
readCntrlrReg(r1.CntrlrByte)
if (r1.bits.fire1butn) { .... fire shot}

Copyright © 2016-2023 Jim Skrentny