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	

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

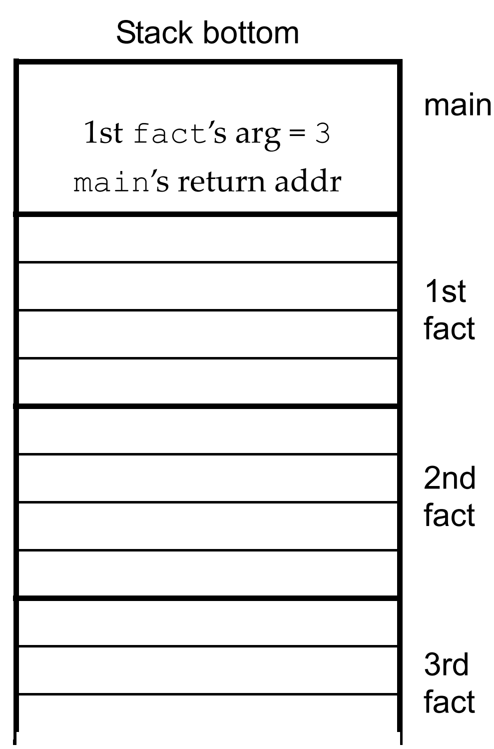
recursive case

base case

"infinite" recursion

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
```



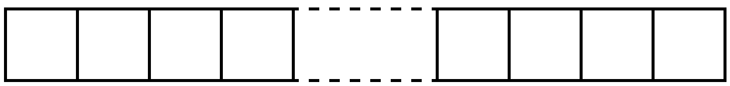
❋ *"Infinite" recursion causes*

❋ *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



1.

2.

❋ *The elements of* `A`

Recall Array Indexing and Address Arithmetic

`&A[i]`

→ 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]`			
2.	`char C[7]`			
3.	`double D[11]`			
4.	`short S[42]`			
5.	`char *C[13]`			
6.	`int **I[11]`			
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

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

Accessing 1D Arrays in Assembly

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

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

→ 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:

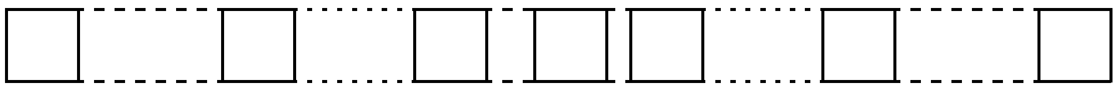
C code　　type　　assembly instruction to move C code's value into %eax

1. `I`
2. `I[0]`
3. `*I`
4. `I[i]`
5. `&I[2]`
6. `I+i-1`
7. `*(I+i-3)`
8. `S[3]`
9. `S+1`
10. `&S[i]`
11. `S[4*i+1]`
12. `S+i-5`

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*

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

Accessing 2D Arrays in Assembly

```
&A[i][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

sall $2, %edx

addl %eax, %edx

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

Compiler Optimizations

- If only accessing part of array

- If taking a fixed stride through the array

Stack Allocated Structures

Structures on the Stack

```
struct iCell {
   int x;
   int y;
   int c[3];
   int *v;
};
```

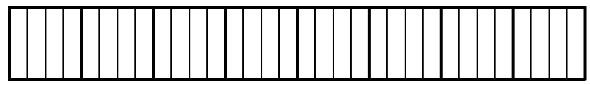
→ How is a structure laid out on the stack? Option 1 or 2:

higher addresses

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

Stack Top

The compiler
-



-

✵ *The first data member of a structure*

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`
2. `ic.c[i]`
3. `ip->x`
4. `ip->y`
5. `&ip->c[i]`

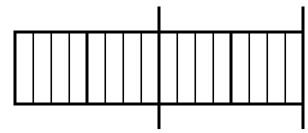
✵ *Assembly code to access a structure*

Alignment

What?

Why?

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



Restrictions

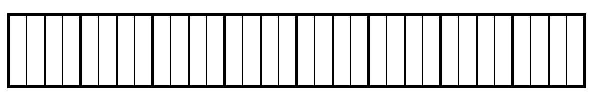
Linux: short
int, float, pointer, double

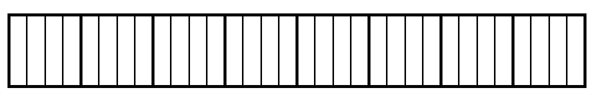
Windows: same as Linux except
double

Implications

Structure Example

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





✵ *The total size of a structure*

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;
      short s;
      int i;
      char d;
   };
```

```
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*

Unions

What? A *union* is

◆

◆

Why?

◆

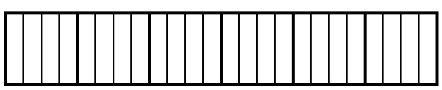
◆

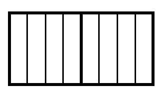
◆

How?

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

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





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;
```