CS354 - Machine Organization & Programming
April 11 – April 18, 2023

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

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 a function calls itself

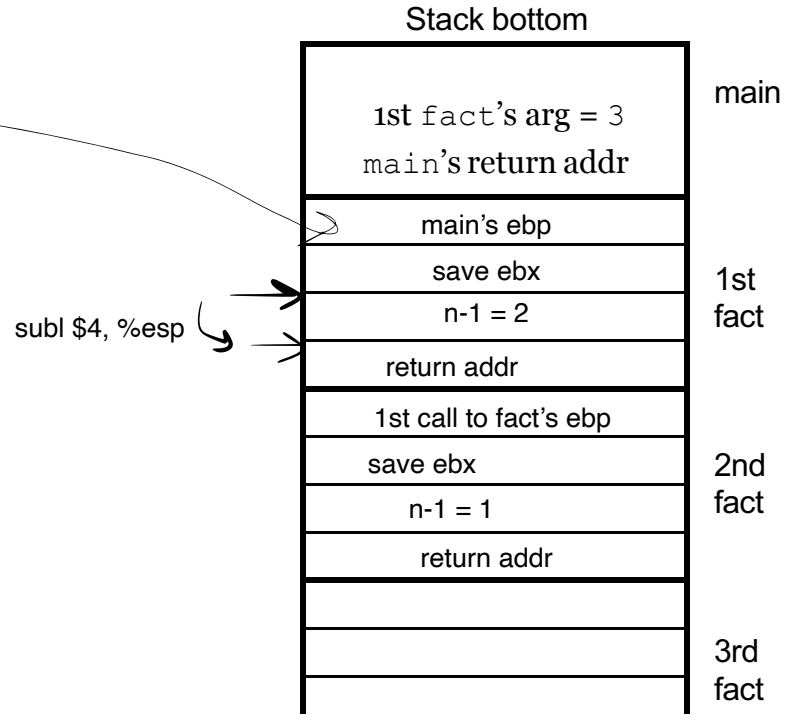
recursive case when it calls itself

base case when it exits

"infinite" recursion

Assembly Trace

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

   movl 8(%ebp),%ebx  n -> ebx
   movl $1,%eax

   cmpl $1,%ebx
   jle .L1

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

   imull %ebx,%eax

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



- *"Infinite" recursion causes* stack overflow

- *When tracing functions in assembly code*

Stack Allocated Arrays in C

Recall Array Basics Statically Allocated

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

exrtra room on the stack frame will be created with this much space -> sizeof(T) * N

1. contiguous allocation in the stack frame

2. A is an identifier A is more like an name/identifier since there is no pointer to this

✵ *The elements of* `A` are accessed with indexing eg A[3] (A + i) does not work because A is not a pointe only a pointer when u do malloc

Recall Array Indexing and Address Arithmetic

`&A[i]` like doing A + i

assembler will know where A is sitting on the stack frame,
eg A will be at -100(%ebp) for the +i, it will do something like -100(%ebp,%ecx,4)

→ 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() + 4i	44 bytes
2.	`char C[7]`	1	X() + i	7 bytes but allocator will allocate 8 bytes
3.	`double D[11]`			22 bytes
4.	`short S[42]`			
5.	`char *C[13]`			
6.	`int **I[11]`	4	X() + 4i	44 bytes
7.	`double *D[7]`			

Stack Allocated Arrays in Assembly

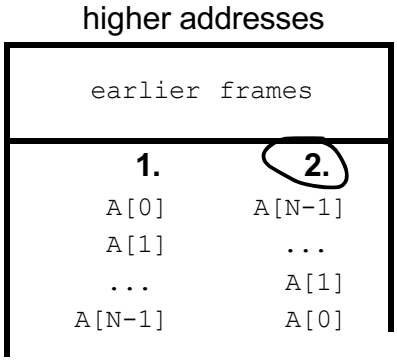
Arrays on the Stack

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

Option 2

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

Is closest to the top of the stack (stacks grow down)



Accessing 1D Arrays in Assembly

leal - for efficient access to stack array elements

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

`movl (%edx, %ecx, 4), %eax` will get ecx-th element in the array

movl -100(%ebp, %ecx, 4), %eax — if looking from the stack frame

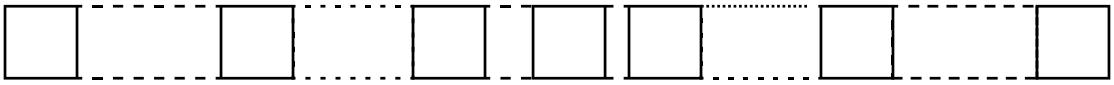
→ Assume `I` is an `int` array, `S` is a `short int` array, for both the array's start address is in %edx, and the index `i` is in %ecx. Determine the element type and instruction for each:

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



✵ *Recall that 2D arrays are stored on the stack*

#rows * #columns * L = 5 * 3 * 4 = 60 bytes

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

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

Accessing 2D Arrays in Assembly

`&A[i][j]` if X_a is address to first element/start of array, then to access i,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:

ecx + ecx * 2 -> ecx so ecx will have 3 times original value
3 because array has 3 columns, gets to i

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

shift left 2 is multiply by 4 into edx, gets to j

avoids imull

X_a + j * 4 -> edx

X_a + 4*j + 4 * 3 * i
edx ecx * 4

Compiler Optimizations

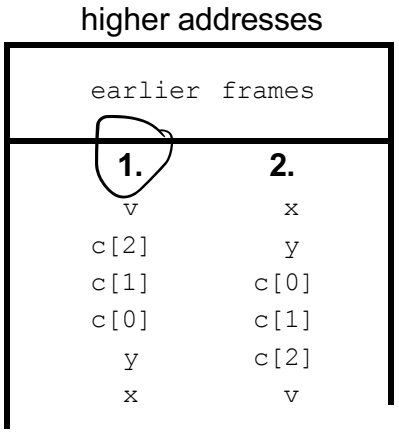
- If only accessing part of array
  compiler might make a pointer to start of row

- If taking a fixed stride through the array
  computer base pointer + stride

Stack Allocated Structures

Structures on the Stack

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



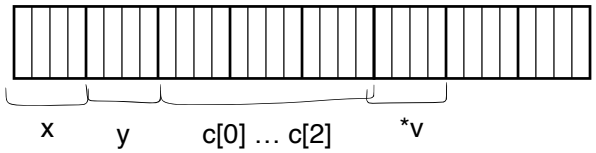
→ How is a structure laid out on the stack? Option 1 or 2:

Option 1

The compiler

- Just keeps track of offsets

when allocate iCell, compiler will know it is at some offset relative to ebp, will know y is 4 bytes up from that offset



- pointer arithmetic to access arrays

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

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]`	movl 8(%esp, %esi, 4), %eax
3.	`ip->x`	movl (%edx), %eax
4.	`ip->y`	movl 4(%edx), %eax
5.	`&ip->c[i]`	leal 12(%edx, %esi, 4), %eax

✵ *Assembly code to access a structure*

names are lost, everything is done by offset into the struct

Alignment

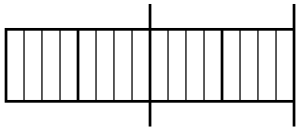
What? Lots of computers put restrictions on where data can be stored
intel machines are generous (can store at any address eg …F7)
whats bad is that if its end of cache line only get a few bytes of whole data

Why?

better performance if aligned

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

require 2 memory access, 1 to get first part,
another to get second part, then combine after



Restrictions

Linux: short 2 byte alignment — 1 least significant bit must be equal to 0
int, float, pointer, double 4 byte alignment — 2 least significant bit must be equal to 0

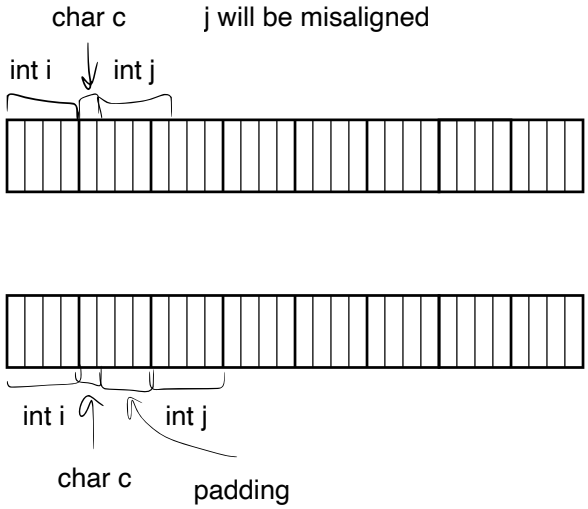
Windows: same as Linux except
double 8 byte alignment — 3 least significant bit must be equal to 0

Implications

when creating structures, we'll need to pad the structures

Structure Example

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



- *The total size of a structure*

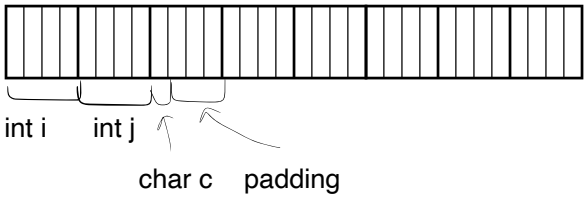
sizeof(s1) == 12

size of struct will be sth divisible by biggest item in struct

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;
   };
```

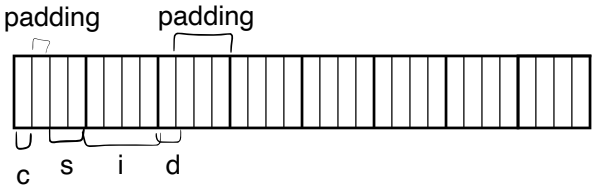


sizeof(sA) == 12

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

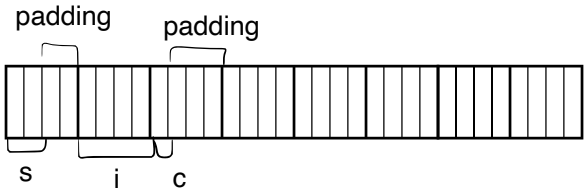
sizeof(sB) == 3

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



sizeof(sC) == 12

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



sizeof(sD) == 12

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

int i;
char a;
char b;
short s;

sizeof(sE) == 8

❋ *The order that a structure's data members are listed*

can affect the size of the overall struct

Unions

What? A *union* is

- like a struct when defining,
  but instead of each member in struct having memory storage, they all share memory

- allocated large enough to hold the largest value

Why?

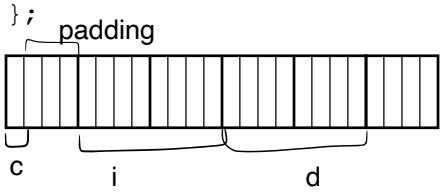
- can access data as different types

- access hardware

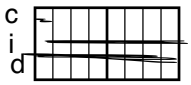
- implementing polymorphism

How?

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




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



size of largest field

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;
```

defining bits

bit 0
bit 1
bit 2
bit 3
bit 4
bit 5-7

sizeof(struct bits) = 1
everything stored in single byte

size of this union is 1 byte

```
CntrlrReg cr;
cr.cntrlrByte = readCrtrlr ()
if (cr.bits.fre1butn == 1) do sth…
```