CS 354 - Machine Organization & Programming
Thursday, November 14, 2019

Project p5 (4.5%): DUE at 10 pm on Wednesday, December 2nd
Project p6 (4.5%): Assigned on Tuesday, November 26th

Homework hw6 (1.5%): DUE at 10 pm on Wednesday, November 20th

Last Time

Register Usage Conventions
Function Call-Return Example
Recursion
Stack Allocated Arrays in C
Stack Allocated Arrays in Assembly

Today

Stack Allocated Arrays in Assembly (from last time)
Stack Allocated Multidimensional Arrays
Stack Allocated Structs
Alignment
Alignment Practice
Unions
Exams Returned

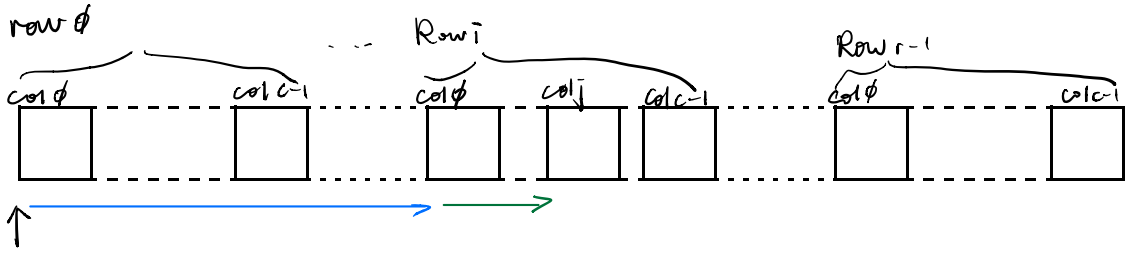
Next Time

Pointers in Assembly, and Stack Smashing
Read: B&O 3.10, 3.12

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];
```

Accessing 2D Arrays in Assembly

`&A[i][j]` ≡ start address + 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
sall $2, %edx
addl %eax, %edx
movl (%edx, %ecx, 4), %eax
```

i 2i+i = 3i
j (shift)

M[(x_A) + 4*3*i + 4j]

note this avoids costly imull

%edx.

Compiler Optimizations

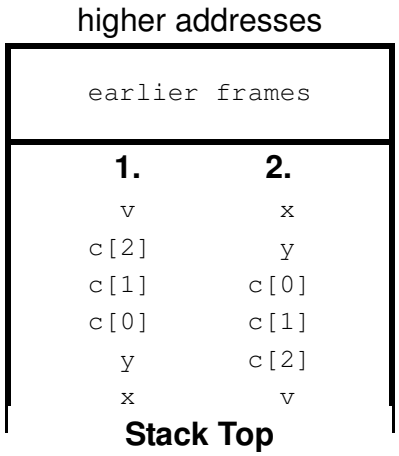
- If only accessing part of array (E.g. one row) compiler makes a pointer to that part and offset from the pointer

- If taking a fixed stride through the array (e.g. look at array third element). Compiler must uses the stride * L as a step size to access

Stack Allocated Structures

Structures on the Stack

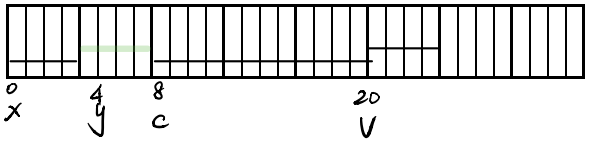
```
struct iCell {
   int x;
   int y;
   int c[3];
   int *v;
};
```



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

The compiler

- associates data member names with their offset from the start address of the structure.
- uses address arithmetic with offset to access data members



❋ *The first data member of a structure* is 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]`	
3. `ip->x`	movl (%edx), %eax
4. `ip->y`	
5. `&ip->c[i]` (array)	leal 8(%edx,%esi,4), %eax ("c")

❋ *Assembly code to access a structure*
is typically a register used as a base address with a varying offsets

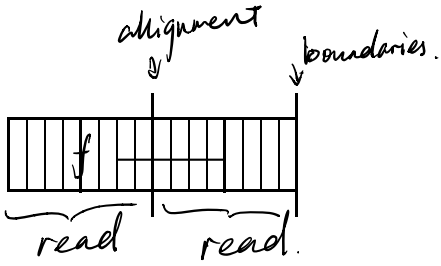
Alignment

What? most computer system restrict the address where primitive data can be store

Why? better memory performance

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

requires 2 reads, extraction and combining parts.



Restrictions

IA-32 has no alignment restrictions

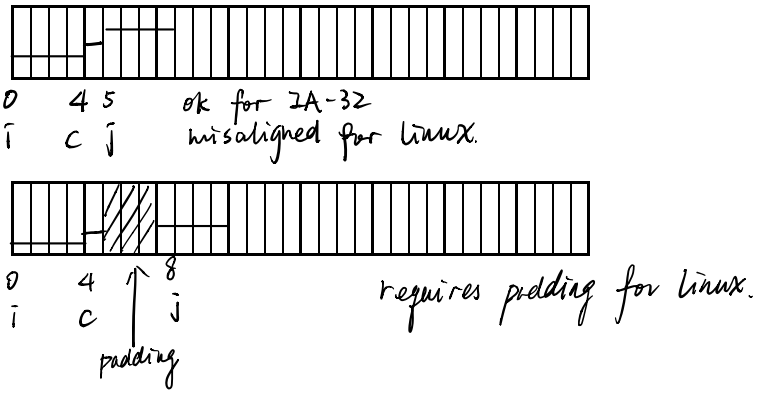
Linux:
- short — address must be a multiple of 2
- int, float, pointer, double — address must be a multiple of 4.

Windows: same as Linux except
- double must be a multiple of 8.

Implications padding might be added.

Structure Example

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

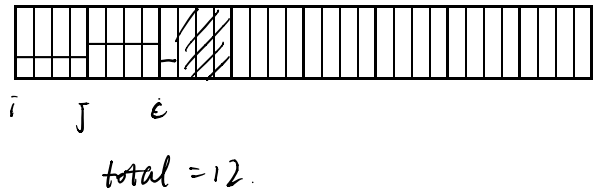


❋ *The total size of a structure*
is typically a multiple of its largest data member size.

Alignment Practice

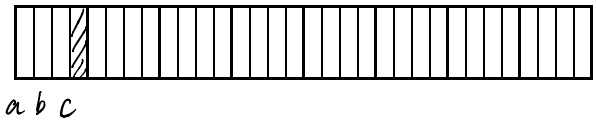
→ For each structure below, complete the memory layout and determine the total bytes allocated.

```
1) struct sA {
      int i;   0
      int j;   4
      char c;  8
   };
```

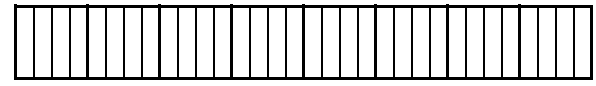


total = 12.

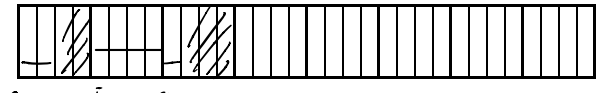
```
2) struct sB {
      char a;
      char b;
      char c;
   };
```




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

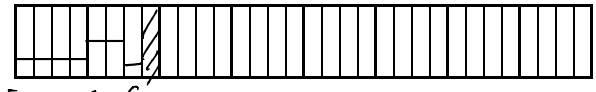


```
4) struct sD {
      short s;  0
      int i;    4
      char c;   8
   };
```



total = 12 bytes.

```
5) struct sE {
      int i;    0
      short s;  4
      char c;   6
   };
```



total = 8 bytes.

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

Unions

What? A *union* is

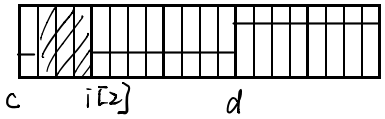
- like a structure except union have fields.
  that store the same memory → bypassing C's type checking.
- allocated only enough memory for its largest field

Why?

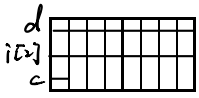
- allows data to be accessed a different types.
- used to access hardware.
- implements polymorphism (poorly).

How?

```
struct s {
   char c;     0
   int i[2];   4
   double d;   12
};
```




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



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;
```

Field 1
Field 2.
← Bit Designation.

```
CntrlrReg C1;
readCntrl (C1.cntrlrByte);
    :
if (C1.bits.fire1butn) . . .
```