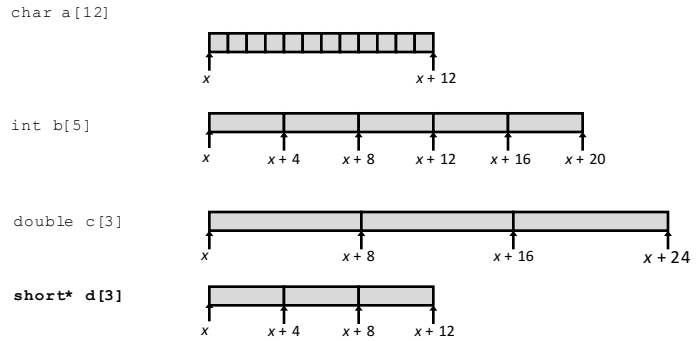
More data layout!

Array Layout

Array of data type T and length N

Contiguously allocated memory region of N * (size of T in bytes)



Two-Dimensional Arrays

Declaration

2D array of data type T

R rows, C columns

Type T element requires K bytes

Array layout and size?

```
[ A[0][0]    . . .   A[0][C-1]
    .                  .
    .                  .
    .                  .
  A[R-1][0]  . . .  A[R-1][C-1] ]
```


Nested Arrays (C)

Declaration

2D array of data type T

R rows, C columns

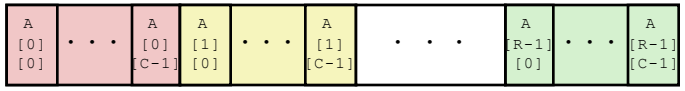
Type T element requires K bytes

Layout

Row-major ordering

```
[ A[0][0]    . . .   A[0][C-1]
    .                  .
    .                  .
    .                  .
  A[R-1][0]  . . .  A[R-1][C-1] ]
```

```
int A[R][C];
```



Uh, raise awareness day!

Seat = 4 bytes

Array tag!

Call out 2D index.

Stand up if that's you.

Nested arrays: 4 rows x 8 columns. Element size: 4 bytes.

Nested Array Access (C)

Each row is an array.

T A[R][C]: A[i] is array of C elements

Each element of type T requires K bytes

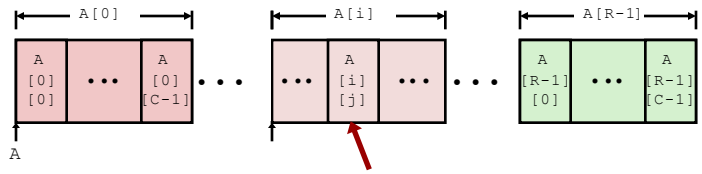
Starting address: _____

Array Elements

A[i][j] is element of type T, which requires K bytes

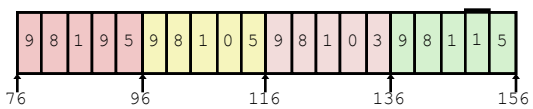
Address of A[i][j]: _____

```
int A[R][C];
```



Strange Referencing Examples (C)

```
int sea[4][5];
```



Reference	Address	Value	Guaranteed?
sea[3][3]	76+20*3+4*3 = 148	1	Yes
sea[2][5]	76+20*2+4*5 = 136	9	Yes
sea[2][-1]	76+20*2+4*-1 = 112	5	Yes
sea[4][-1]	76+20*4+4*-1 = 152	5	Yes
sea[0][19]	76+20*0+4*19 = 152	5	Yes
sea[0][-1]	76+20*0+4*-1 = 72	??	No

C does not do any bounds checking.

Row-major array layout is guaranteed.

Multi-Level Arrays (Java, C)

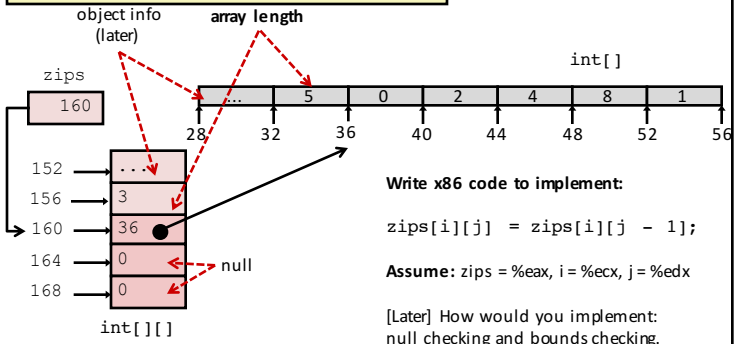
pointers/references

```
int[][] zips = new int[3][];
zips[0] = new int[5] {0, 2, 4, 8, 1};
```

Java

```
int* zips[3];
zips[0] = (int*)malloc(sizeof(int)*5);
...
```

C



Write x86 code to implement:

```
zips[i][j] = zips[i][j - 1];
```

Assume: zips = %eax, i = %ecx, j = %edx

[Later] How would you implement: null checking and bounds checking.

Uh, raise awareness day!

Seat = 4 bytes

Array tag!

- **Call out 2D index.**
- **Stand up if that's you.**

Nested arrays: 4 rows x 8 columns.

Arrays of addresses of arrays:

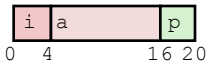
- **4 rows x 8 columns**
- **Variable width**

Please return the addresses at the end.

Structures in C

```
struct rec {
  int i;
  int a[3];
  int* p;
};
```

Memory Layout



Characteristics

- Contiguously-allocated region of memory
- Refer to members within structure by names
- Members may be of different types

Like primitive version of Java objects

17

Structures in C

```
struct rec {
  int i;
  int a[3];
  int* p;
};
```

Accessing Structure Member

Given a struct, use the `.` operator, just like Java:

```
struct rec r1;
r1.i = val;
```

Given a *pointer* to a struct, use `*` and `.` operators or `->` shorthand:

```
struct rec* r = &r1;
(*r).i = val;
r->i = val;
```

} equivalent

```
void set_i(struct rec* r, int val) {
    r->i = val;
}
```

18

struct assignment

```
struct rec a;
struct rec b;
a.i = 1;
a.a[0] = 2;
a.p = &(a.i);
b = a;
```

Copies entire contents of a into b.

```
struct rec* c;
c = &b;
(*c).i++;
```

No copying

What is b.i? a.i?

typedef

struct rec { … };

struct rec a;

typedef struct list_node {
…
} list_node;

Structures in C

```
struct rec {
  int i;
  int a[3];
  int* p;
};
```

Accessing Structure Member

Given an instance of the struct, we can use the `.` operator, just like Java:

`struct rec r1;  r1.i = val;`

What if we have a *pointer* to a struct: `struct rec* r = &r1;`

Using `*` and `.` operators: `(*r).i = val;`

Or, use `->` operator for short: `r->i = val;`

Pointer indicates first byte of structure; access members with offsets

```
void
set_i(struct rec* r,
      int val)
{
    r->i = val;
}
```

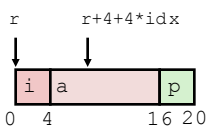
IA32 Assembly

```
# %eax = val
# %edx = r
movl %eax,0(%edx)   # Mem[r+0] = val
```

21

Generating Pointer to Structure Member

```
struct rec {
  int i;
  int a[3];
  int* p;
};
```



Offset of each structure member determined at compile time

```
int* find_address_of_elem
 (struct rec* r, int idx)
{
   return &r->a[idx];
}
```

`&(r->a[idx])`

```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax  # r+4*idx+4
```

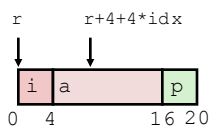
OR

```
leal 4(%edx,%ecx,4),%eax   # r+4*idx+4
```

22

Generating Pointer to Structure Member

```
struct rec {
  int i;
  int a[3];
  int* p;
};
```



Generating Pointer to Array Element

Offset of each structure member determined at compile time

```
int* find_address_of_elem
 (struct rec* r, int idx)
{
   return &r->a[idx];
}
```

`&(r->a[idx])`

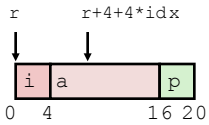
```
# %ecx = idx
# %edx = r
leal 4(%edx,%ecx,4),%eax   # r+4*idx+4
```

OR

23

Accessing Structure Member

```
struct rec {
  int i;
  int a[3];
  int* p;
};
```



Reading Array Element

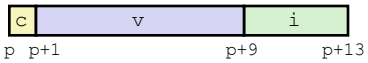
Offset of each structure member *still* determined at compile time

```
int* find_address_of_elem
 (struct rec* r, int idx)
{
   return &r->a[idx];
}
```

```
# %ecx = idx
# %edx = r
movl 4(%edx,%ecx,4),%eax   # Mem[r+4*idx+4]
```


Structures & Alignment

Unaligned Data

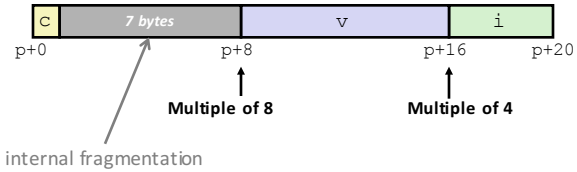



```
struct S1 {
  char c;
  double v;
  int i;
} * p;
```

Aligned Data

Primitive data type requires ***K*** bytes

Address must be multiple of ***K***



Alignment Principles

Aligned Data

Primitive data type requires K bytes

Address must be multiple of K

Aligned data is required on some machines; it is *advised* on IA32

Treated differently by IA32 Linux, x86-64 Linux, Windows, Mac OS X, …

Motivation for Aligning Data

Physical memory is accessed by aligned chunks of 4 or 8 bytes (system-dependent)

Inefficient to load or store datum that spans these boundaries

Also, virtual memory is very tricky when datum spans two pages (later…)

Compiler

Inserts padding in structure to ensure correct alignment of fields

`sizeof()` should be used to get true size of structs

Specific Cases of Alignment (IA32)

1 byte: char, …

no restrictions on address

2 bytes: short, …

lowest 1 bit of address must be 0_2

4 bytes: int, float, char *, …

lowest 2 bits of address must be 00_2

8 bytes: double, …

Windows (and most other OSs & instruction sets): lowest 3 bits 000_2

Linux: lowest 2 bits of address must be 00_2

i.e., treated liked 2 contiguous 4-byte primitive data items

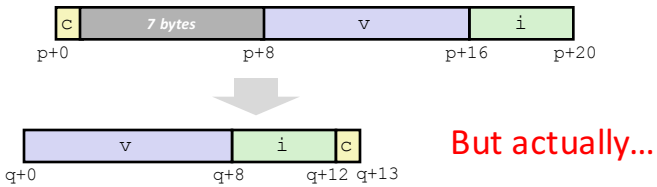
Saving Space

Put large data types first:

```
struct S1 {
  char c;
  double v;
  int i;
} * p;
```

```
struct S2 {
  double v;
  int i;
  char c;
} * q;
```

Effect (example ia32, both have K=8)

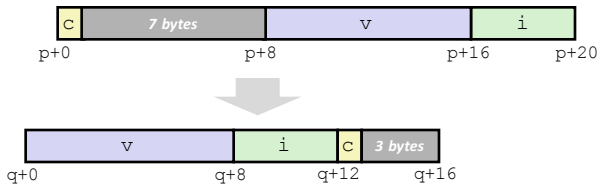


But actually…

Struct Alignment Principles

Size must be a multiple of the largest primitive type inside.

K = 8 so *size* mod 8 = 0



Arrays of Structures

Satisfy alignment requirement for every element

How would accessing an element work?

```
struct S2 {
  double v;
  int i;
  char c;
} a[10];
```