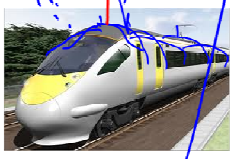
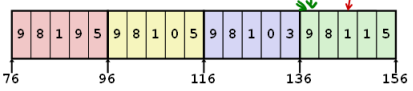
Course speed

- 65%!



Nested Array Example

```
#define PCOUNT 4
zip_dig sea[PCOUNT] =
  { { 9, 8, 1, 9, 5 },
    { 9, 8, 1, 0, 5 },
    { 9, 8, 1, 0, 3 },
    { 9, 8, 1, 1, 5 } };
```

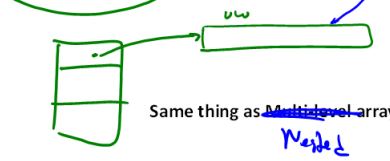


- "row-major" ordering of all elements
- Guaranteed?

Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw  = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

#define UCOUNT 3
int *univ[UCOUNT] = {uw, cmu, ucb};
```



Same thing as ~~Multidevel~~ array?

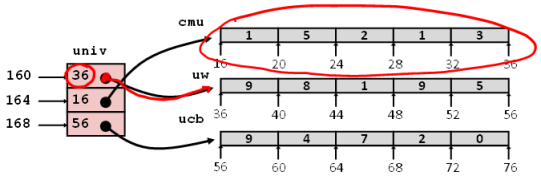
Nested

Multi-Level Array Example

```
zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig uw  = { 9, 8, 1, 9, 5 };
zip_dig ucb = { 9, 4, 7, 2, 0 };

#define UCOUNT 3
int *univ[UCOUNT] = {uw, cmu, ucb};
```

- Variable `univ` denotes array of 3 elements
- Each element is a pointer
  - 4 bytes
- Each pointer points to array of `int`s



2-level vs nested arrays

int B[10];

- int A[10][10] = {{1,…,10},…{20,…,29}};
  - What do you get when you write: A[5]

B

- int *pA[2]; pA[0] = {1, 2, 3}; pA[1] = {5, 6, 7};

pA[1][0] → *(pA[1])

- How many memory accesses in:
  - A[1][1]
  - pA[1][1]

pA

- Btw, what is: int **pptr;

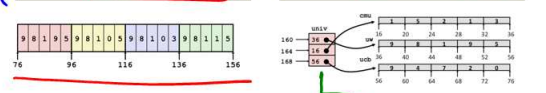
Array Element Accesses

Nested array

```
int get_sea_digit
  (int index, int dig)
{
  return sea[index][dig];
}
```

Multi-level array

```
int get_univ_digit
  (int index, int dig)
{
  return univ[index][dig];
}
```

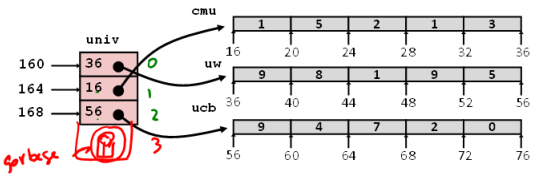


Access looks similar, but it isn't:

```
Mem[sea+20*index+4*dig]
```

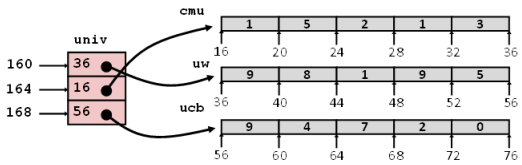
```
Mem[Mem[univ+4*index]+4*dig]
```

Strange Referencing Examples



Reference	Address	Value	Guaranteed?
univ[2][3]	56+3×4=68	2	Yes
univ[1][5]	16+20=36	9	No
univ[2][-1]	What values go here?		
univ[3][-1]			
univ[1][12]			

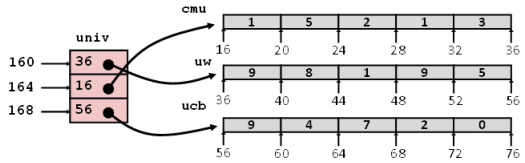
Strange Referencing Examples



Reference	Address	Value	Guaranteed?
univ[2][3]	56+4*3 = 68	2	
univ[1][5]	16+4*5 = 36	9	
univ[2][-1]	56+4*-1 = 52	5	
univ[3][-1]	??	??	
univ[1][12]	16+4*12 = 64	7	

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed

Strange Referencing Examples

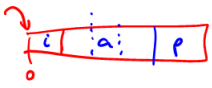


Reference	Address	Value	Guaranteed?
univ[2][3]	56+4*3 = 68	2	Yes
univ[1][5]	16+4*5 = 36	9	No
univ[2][-1]	56+4*-1 = 52	5	No
univ[3][-1]	??	??	No
univ[1][12]	16+4*12 = 64	7	No

- Code does not do any bounds checking
- Ordering of elements in different arrays not guaranteed

Structures

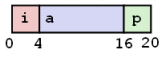
```
struct rec {
  int i;
  int a[3];
  int *p;
};
```



Structures

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

Memory Layout



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

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

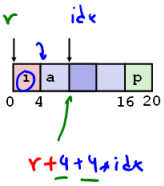
In java: `r.i = val;`

IA32 Assembly

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

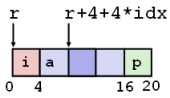
Generating Pointer to Structure Member

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



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

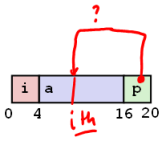
```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax  # r+4*idx+4
```

Structure Referencing (Cont.)

- C Code

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

```
void
set_p(struct rec *r)
{
     r->p = &r->a[r->i];
// (*r).p = &(*((*r).a+(*r).i));
}
```

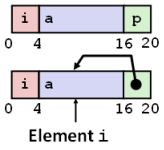


Structure Referencing (Cont.)

- C Code

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

```
void
set_p(struct rec *r)
{
     r->p = &r->a[r->i];
// (*r).p = &(*((*r).a+(*r).i));
}
```



Element i

```
 # %edx = r
 movl  (%edx),%ecx       # r->i
 leal  0(,%ecx,4),%eax   # 4*(r->i)
 leal  4(%edx,%eax),%eax # r+4+4*(r->i)
 movl  %eax,16(%edx)     # Update r->p
```

Alignment

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on IA32
    - treated differently by IA32 Linux, x86-64 Linux, and Windows!
- What is the motivation for alignment?

Alignment

- Aligned Data
  - Primitive data type requires K bytes
  - Address must be multiple of K
  - Required on some machines; advised on IA32
    - treated differently by IA32 Linux, x86-64 Linux, and Windows!
- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system-dependent)
    - Inefficient to load or store datum that spans quad word boundaries
    - Virtual memory very tricky when datum spans two pages (later…)
- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

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 OS's & instruction sets): lowest 3 bits 000_2
  - Linux: lowest 2 bits of address must be 00_2
    - i.e., treated the same as a 4-byte primitive data type
- 12 bytes: `long double`
  - Windows, Linux: (same as Linux double)

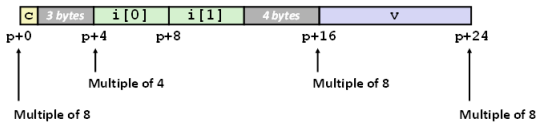
Alignment and structs

- Hmm, how would you satisfy alignments in structs?

Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy element's alignment requirement
- **Overall structure placement**
  - Each structure has alignment requirement K
    - K = Largest alignment of any element
  - Initial address & structure length must be multiples of K
- **Example (under Windows or x86-64):**
  - K = 8, due to `double` element

```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Arrays of structs?

Unions

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

```
union U1 {
  int i;
  int a[3];
  int *p;
} *up;
```

- **Concept**
  - Allow same regions of memory to be referenced as different types
  - Aliases for the same memory location

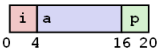
Unions

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

```
union U1 {
  int i;
  int a[3];
  int *p;
} *up;
```

- **Concept**
  - Allow same regions of memory to be referenced as different types
  - Aliases for the same memory location

Structure Layout



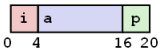
Unions

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

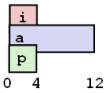
```
union U1 {
  int i;
  int a[3];
  int *p;
} *up;
```

- **Concept**
  - Allow same regions of memory to be referenced as different types
  - Aliases for the same memory location

Structure Layout



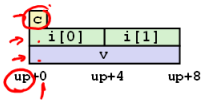
Union Layout



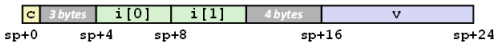
Union Allocation

- Allocate according to largest element
- Can only use one field at a time

```
union U1 {
  char c;
  int i[2];
  double v;
} *up;
```




```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Using Union to Access Bit Patterns

```
typedef union {
  float f;
  unsigned u;
} bit_float_t;
```


```
float bit2float(unsigned u)
{
  bit_float_t arg;
  arg.u = u;
  return arg.f;
}
```

Same as `(float) u` ?

```
unsigned float2bit(float f)
{
  bit_float_t arg;
  arg.f = f;
  return arg.u;
}
```

Same as `(unsigned) f` ?

Summary

- **Arrays in C**
  - Contiguous allocation of memory
  - Aligned to satisfy every element's alignment requirement
  - Pointer to first element
  - No bounds checking
- **Structures**
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- **Unions**
  - Overlay declarations
  - Way to circumvent type system