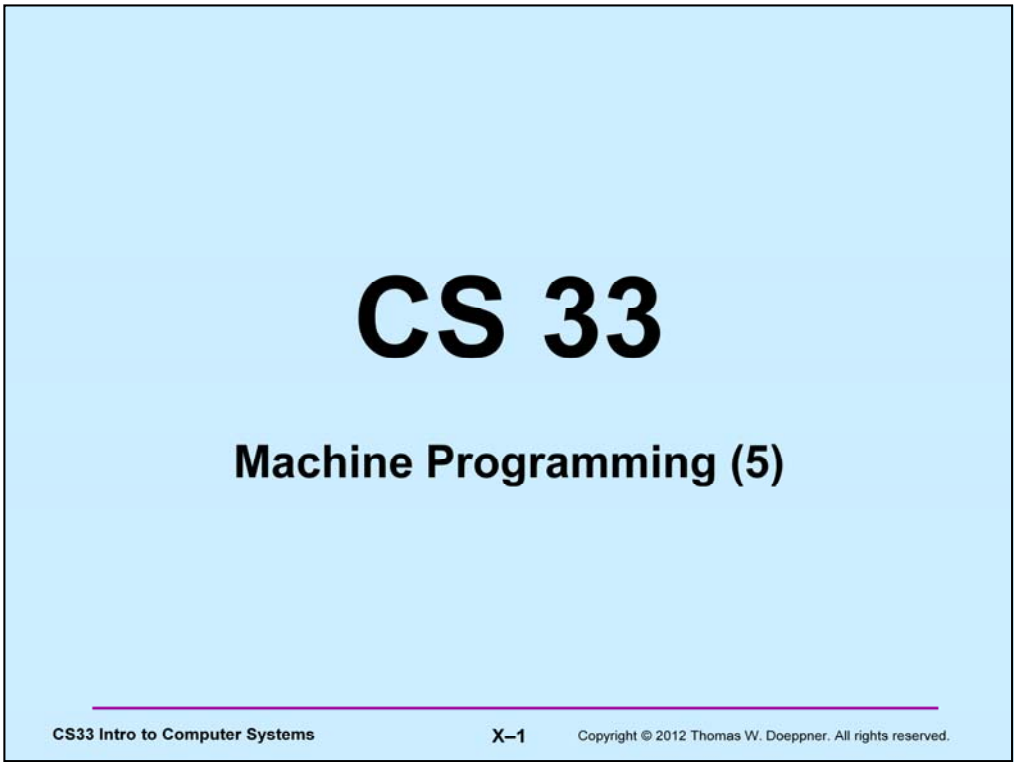
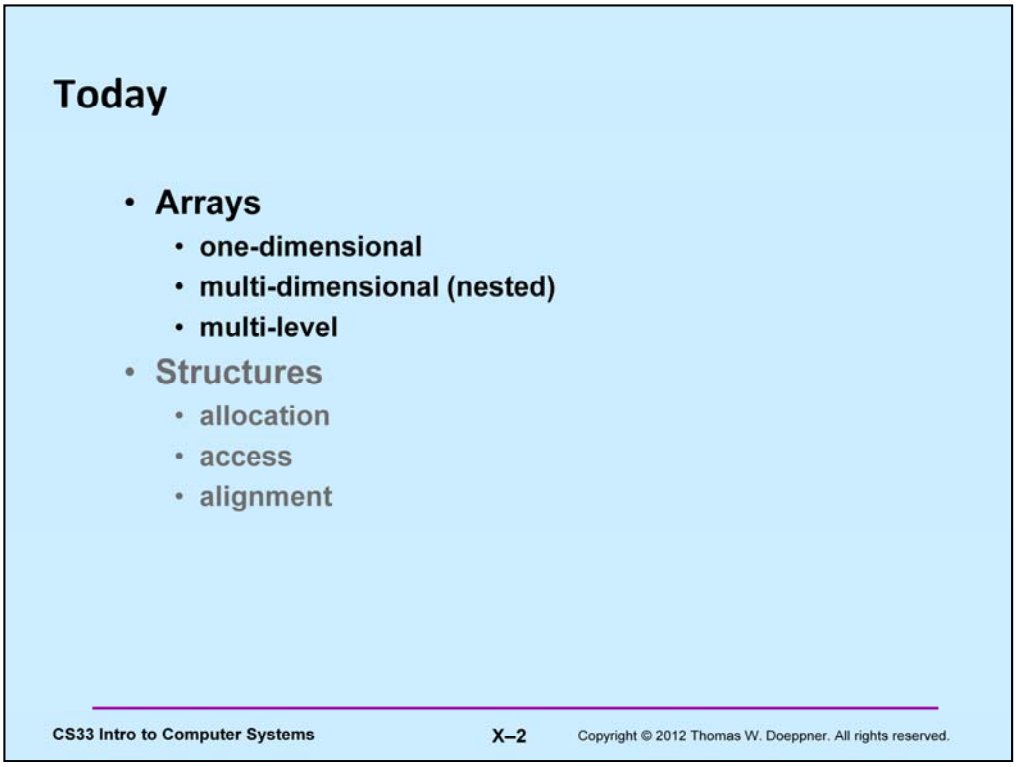


All of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.



Supplied by CMU.

Basic Data Types

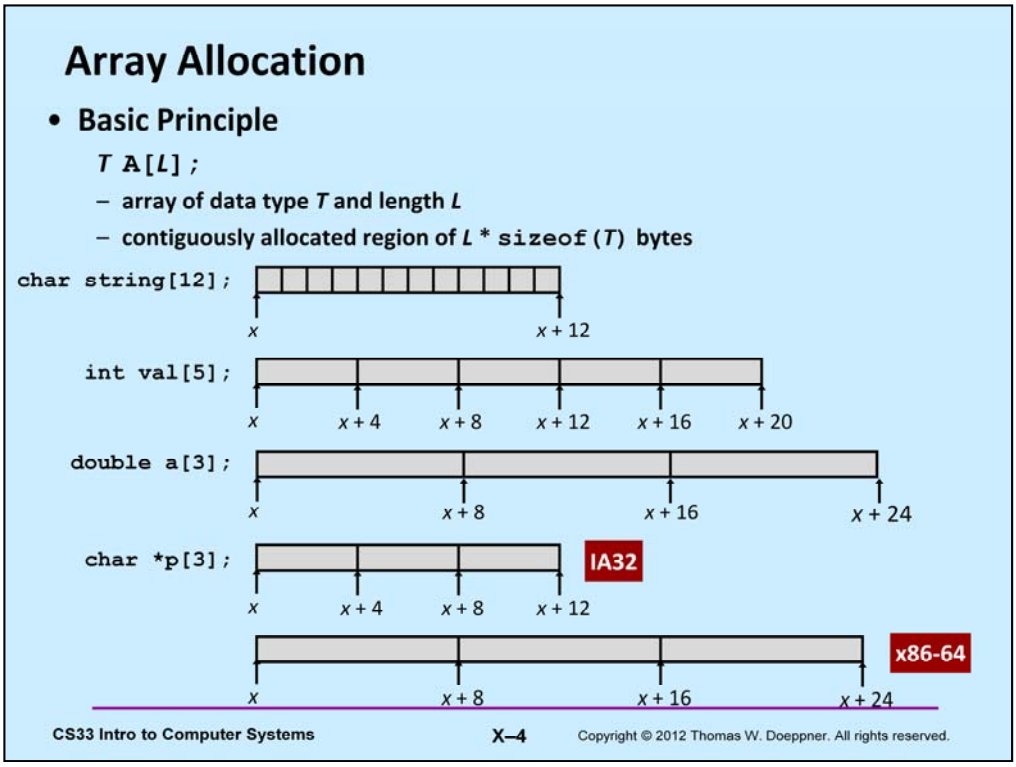
- **Integral**
  - stored & operated on in general (integer) registers
  - signed vs. unsigned depends on instructions used

Intel	ASM	Bytes	C
byte	b	1	[unsigned] char
word	w	2	[unsigned] short
double word	l	4	[unsigned] int
quad word	q	8	[unsigned] long int (x86-64)

- **Floating Point**
  - stored & operated on in floating point registers

Intel	ASM	Bytes	C
single	s	4	float
double	l	8	double
extended	t	10/12/16	long double

Supplied by CMU.



Supplied by CMU.

Array Access

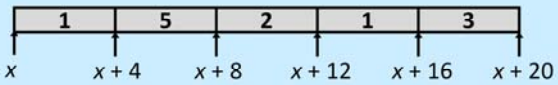
- **Basic Principle**

  T `A[`*L*`];`

  – array of data type *T* and length *L*

  – identifier `A` can be used as a pointer to array element 0: type *T**

`int val[5];`



- **Reference**

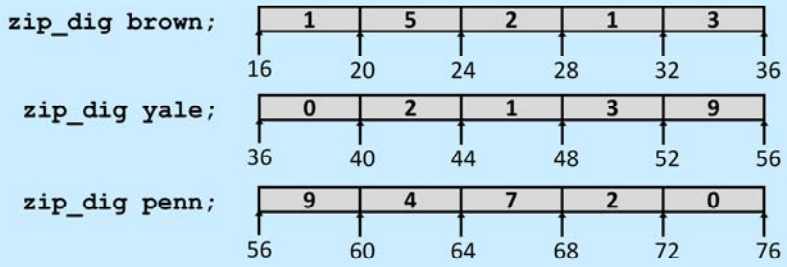
Type	Value
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	*x* + 4*i*

Supplied by CMU.

Array Example

```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig brown = { 1, 5, 2, 1, 3 };
zip_dig yale = { 0, 2, 1, 3, 9 };
zip_dig penn = { 9, 4, 7, 2, 0 };
```

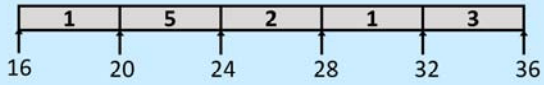


- Declaration "`zip_dig brown`" equivalent to "`int brown[5]`"
- Example arrays were allocated in successive 20-byte blocks
  - not guaranteed to happen in general

Supplied by CMU.

Array Accessing Example

```
zip_dig brown;
```




```
int get_digit
  (zip_dig z, int dig)
{
  return z[dig];
}
```

IA32

```
  # %edx = z
  # %eax = dig
   movl (%edx,%eax,4),%eax  # z[dig]
```

- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at `4*%eax + %edx`
- Use memory reference `(%edx,%eax,4)`

Supplied by CMU.

Array Loop Example (IA32)

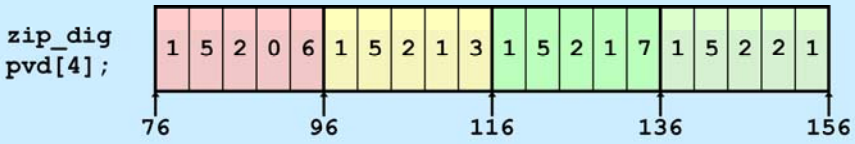
```
void zincr(zip_dig z) {
  int i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # edx = z
  movl  $0, %eax            #   %eax = i
.L4:                        # loop:
  addl  $1, (%edx,%eax,4)   #   z[i]++
  addl  $1, %eax            #   i++
  cmpl  $5, %eax            #   i:5
  jne   .L4                 #   if !=, goto loop
```


Supplied by CMU.

Nested-Array Example

```
#define PCOUNT 4
zip_dig pvd[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

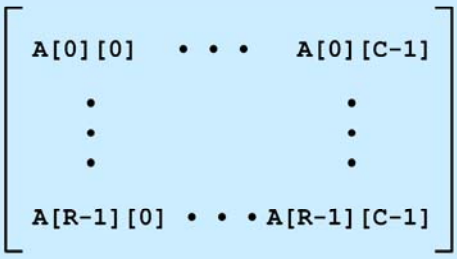


- "`zip_dig pvd[4]`" equivalent to "`int pvd[4][5]`"
  - variable `pvd`: array of 4 elements, allocated contiguously
  - each element is an array of 5 `int`'s, allocated contiguously
- "Row-Major" ordering of all elements guaranteed

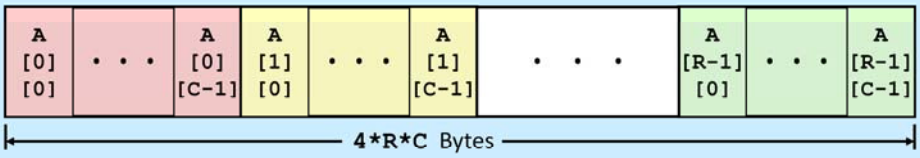
Supplied by CMU.

Multidimensional (Nested) Arrays

- **Declaration**
  - *T* `A[R][C];`
  - 2D array of data type *T*
  - *R* rows, *C* columns
  - type *T* element requires *K* bytes
- **Array Size**
  - *R* * *C* * *K* bytes
- **Arrangement**
  - row-major ordering



`int A[R][C];`

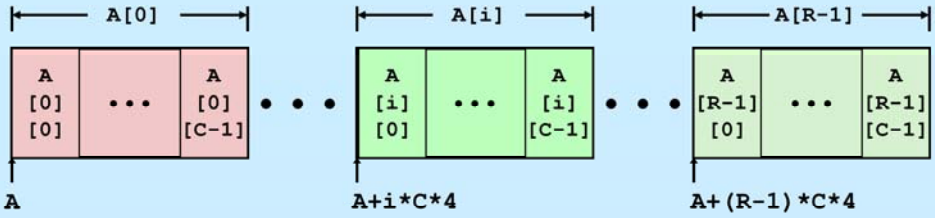


Supplied by CMU.

Nested-Array Row Access

- Row Vectors
  - `A[i]` is array of *C* elements
  - each element of type *T* requires *K* bytes
  - starting address `A` + *i* * (*C* * *K*)

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



Supplied by CMU.

Nested-Array Row-Access Code

```
int *get_pvd_zip(int index)
{
  return pvd[index];
}
```

```
#define PCOUNT 4
zip_dig pvd[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```

```
  # %eax = index
   leal (%eax,%eax,4),%eax # 5 * index
   leal pvd(,%eax,4),%eax  # pvd + (20 * index)
```

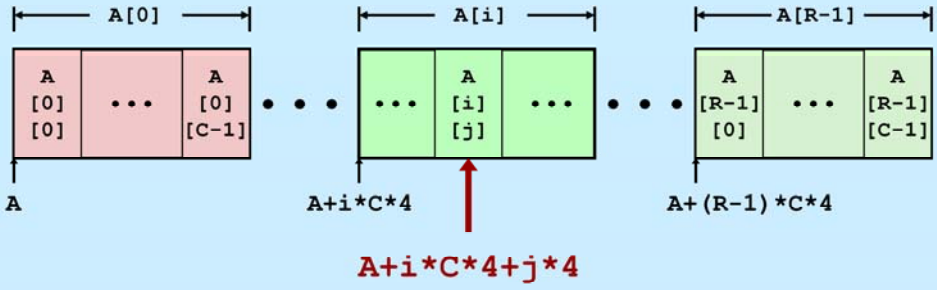
- Row Vector
  - pvd[index] is array of 5 int's
  - starting address pvd+20*index
- IA32 Code
  - computes and returns address
  - compute as pvd + 4*(index+4*index)

Supplied by CMU.

Nested-Array Element Access

- Array Elements
  - `A[i][j]` is element of type *T*, which requires *K* bytes
  - address `A` + *i* * (*C* * *K*) + *j* * *K* = *A* + (*i* * *C* + *j*)* *K*

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



Supplied by CMU.

Nested-Array Element-Access Code

```
int get_pvd_digit
  (int index, int dig)
{
  return pvd[index][dig];
}
```

```
movl   8(%ebp), %eax          # index
leal   (%eax,%eax,4), %eax    # 5*index
addl   12(%ebp), %eax         # 5*index+dig
movl   pvd(,%eax,4), %eax     # offset 4*(5*index+dig)
```

- **Array Elements**
  - **pvd[index][dig] is int**
  - **address: pvd + 20*index + 4*dig**
    - **= pvd + 4*(5*index + dig)**
- **IA32 Code**
  - **computes address pvd + 4*((index+4*index)+dig)**

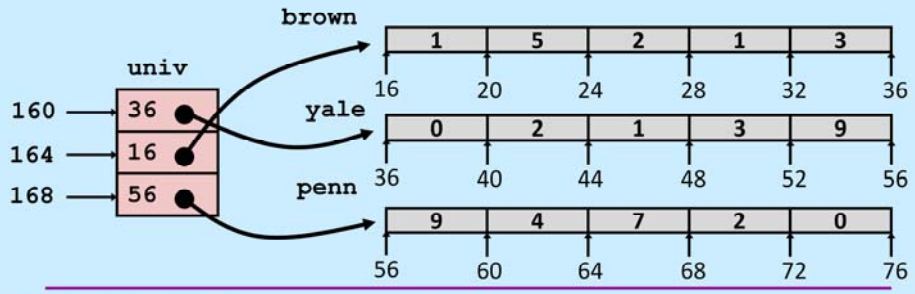
Supplied by CMU.

Multi-Level Array Example

```
zip_dig brown = { 1, 5, 2, 1, 3 };
zip_dig yale = { 0, 2, 1, 3, 9 };
zip_dig penn = { 9, 4, 7, 2, 0 };
```

```
#define UCOUNT 3
int *univ[UCOUNT] = {brown, yale, penn};
```

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



Supplied by CMU.

Element Access in Multi-Level Array

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

```
  movl   8(%ebp), %eax            # index
  movl   univ(,%eax,4), %edx      # p = univ[index]
  movl   12(%ebp), %eax           # dig
  movl   (%edx,%eax,4), %eax      # p[dig]
```

- Computation (IA32)
  - element access Mem[Mem[univ+4*index]+4*dig]
  - must do two memory reads
    - » first get pointer to row array
    - » then access element within array

Supplied by CMU.

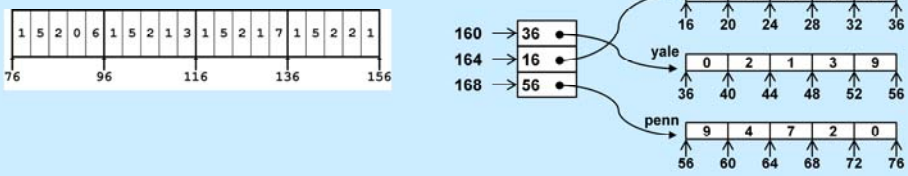
Array-Element Accesses

Nested array

```
int get_pvd_digit
   (int index, int dig)
{
  return pvd[index][dig];
}
```

Multi-level array

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



Accesses looks similar in C, but addresses very different:

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

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

Supplied by CMU.

N X N Matrix Code

- Fixed dimensions
  - know value of N at compile time

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element a[i][j] */
int fix_ele
  (fix_matrix a, int i, int j)
{
  return a[i][j];
}
```

- Variable dimensions
  - just recently supported by gcc

```
/* Get element a[i][j] */
int var_ele
 (int n, int a[n][n], int i, int j) {
  return a[i][j];
}
```


Supplied by CMU.

16 X 16 Matrix Access

- **Array Elements**
  - address **A** + $i * (C * K) + j * K$
  - C = 16, K = 4

```
/* Get element a[i][j] */
int fix_ele(fix_matrix a, int i, int j) {
  return a[i][j];
}
```

```
  movl   12(%ebp), %edx     # i
  sall   $6, %edx           # i*64
  movl   16(%ebp), %eax     # j
  sall   $2, %eax           # j*4
  addl   8(%ebp), %eax      # a + j*4
  movl   (%eax,%edx), %eax  # *(a + j*4 + i*64)
```


Supplied by CMU.

n X n Matrix Access

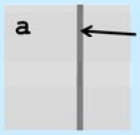
- **Array Elements**
  - address **A** + $i * (C * K) + j * K$
  - C = n, K = 4

```
/* Get element a[i][j] */
int var_ele(int n, int a[n][n], int i, int j) {
  return a[i][j];
}
```

```
  movl   8(%ebp), %eax      # n
  sall   $2, %eax           # n*4
  movl   %eax, %edx         # n*4
  imull  16(%ebp), %edx     # i*n*4
  movl   20(%ebp), %eax     # j
  sall   $2, %eax           # j*4
  addl   12(%ebp), %eax     # a + j*4
  movl   (%eax,%edx), %eax  # *(a + j*4 + i*n*4)
```


Supplied by CMU.

Optimizing Fixed-Array Access

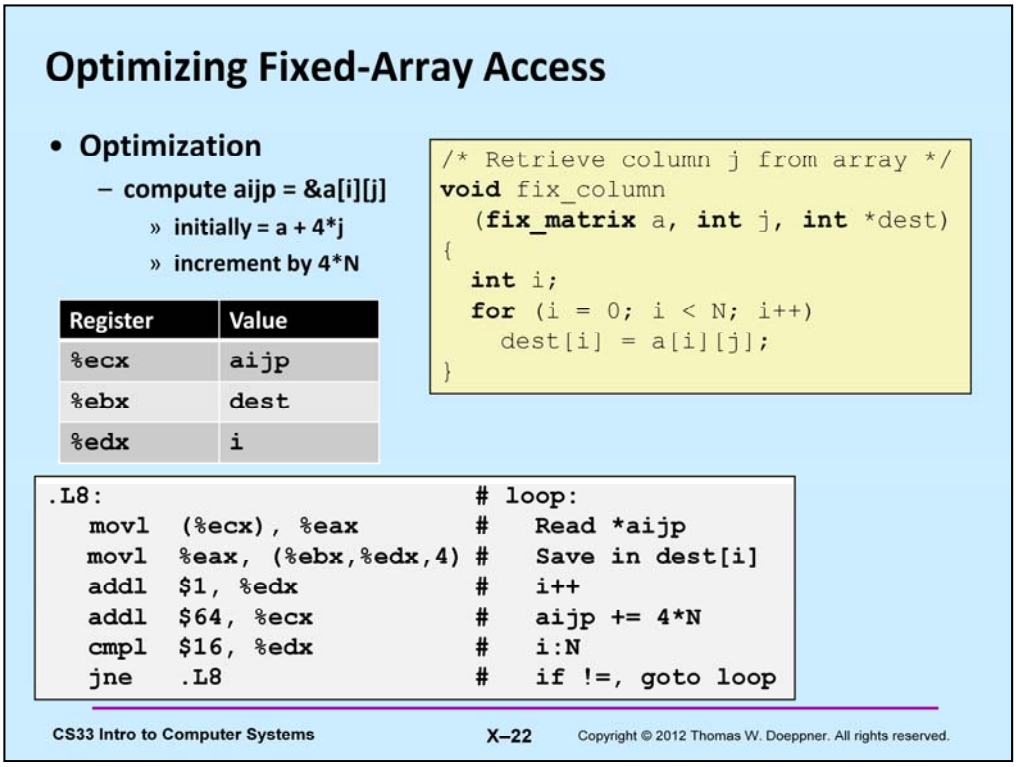


- **Computation**
  - step through all elements in column j
- **Optimization**
  - retrieving successive elements from single column

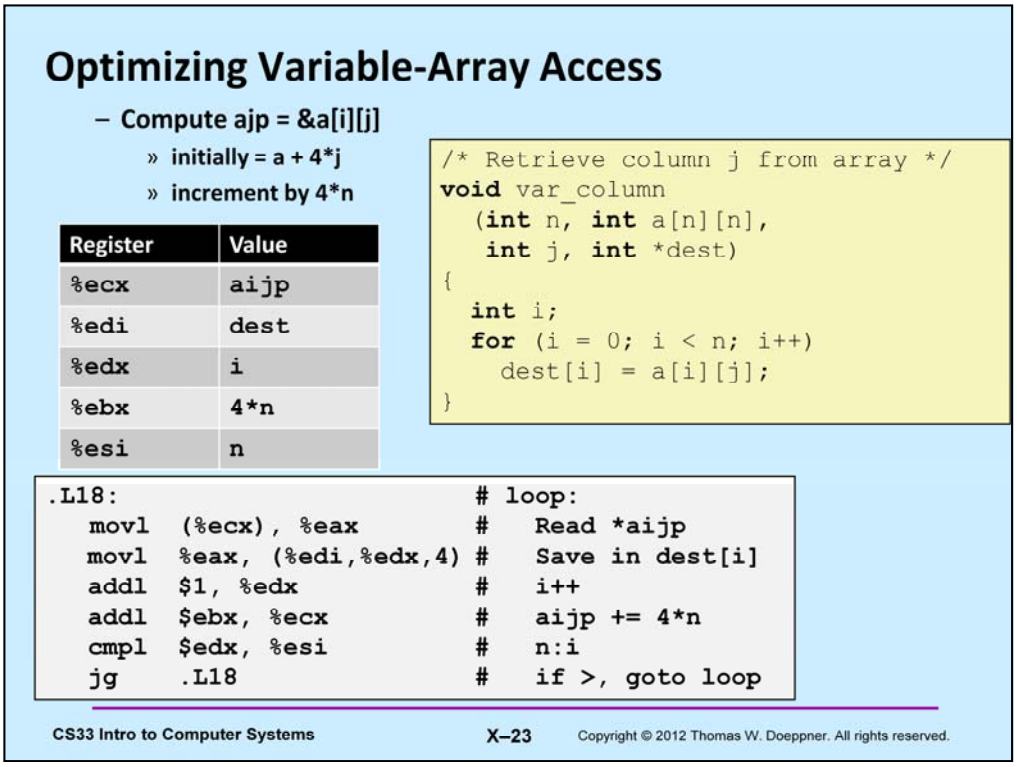
```
#define N 16
typedef int fix_matrix[N][N];
```

```
/* Retrieve column j from array */
void fix_column
  (fix_matrix a, int j, int *dest)
{
  int i;
  for (i = 0; i < N; i++)
    dest[i] = a[i][j];
}
```

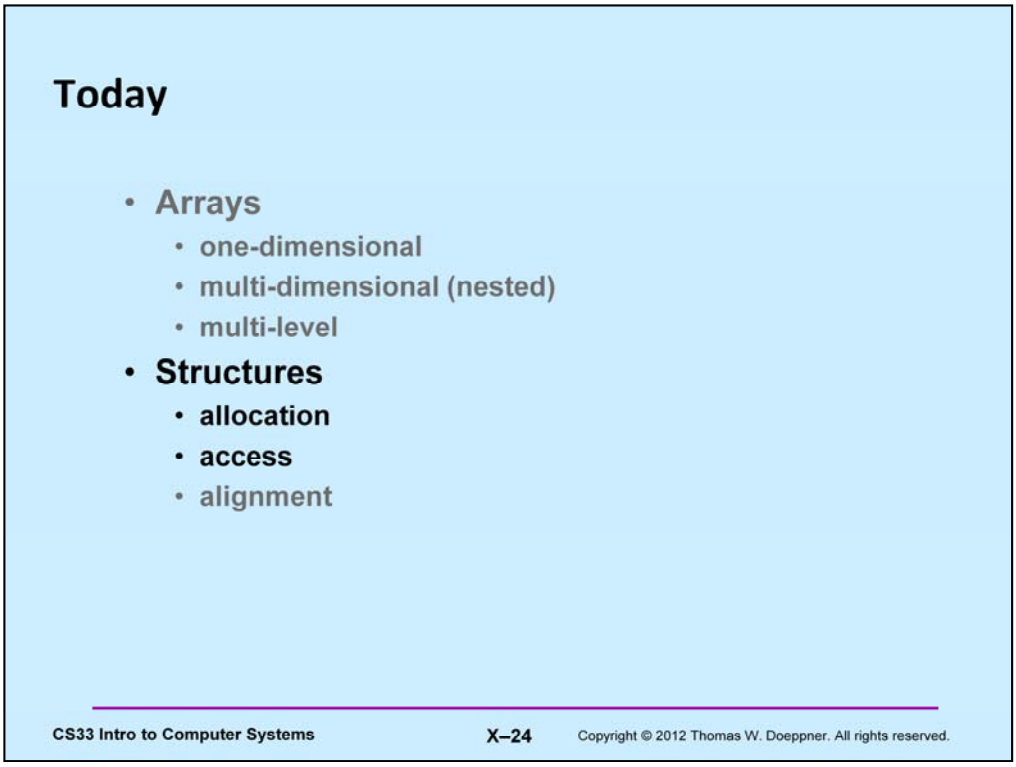

Supplied by CMU.



Supplied by CMU.



Supplied by CMU.

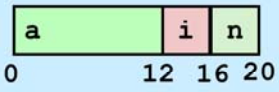


Supplied by CMU.

Structure Allocation

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

Memory layout

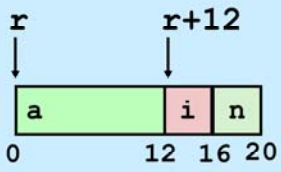


- Concept
  - contiguously allocated region of memory
  - refer to members within structure by names
  - members may be of different types

Supplied by CMU.

Structure Access

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



- Accessing structure member
  - pointer indicates first byte of structure
  - access elements with offsets

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

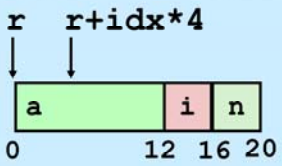
IA32 Assembly

```
# %edx = val
# %eax = r
movl %edx, 12(%eax)  # Mem[r+12] = val
```


Supplied by CMU.

Generating Pointer to Structure Member

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

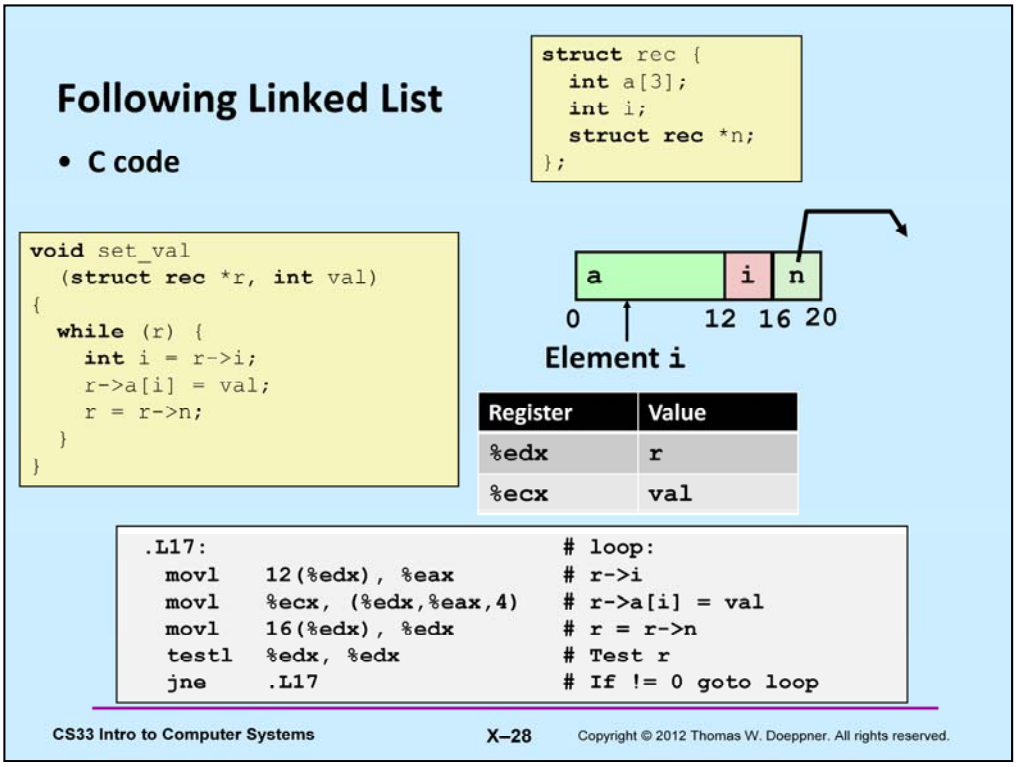


- **Generating pointer to array element**
  - **offset of each structure member determined at compile time**
  - **arguments**
    - » **Mem[%ebp+8]: r**
    - » **Mem[%ebp+12]: idx**

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

```
movl   12(%ebp), %eax   # Get idx
sall   $2, %eax         # idx*4
addl   8(%ebp), %eax    # r+idx*4
```


Supplied by CMU.

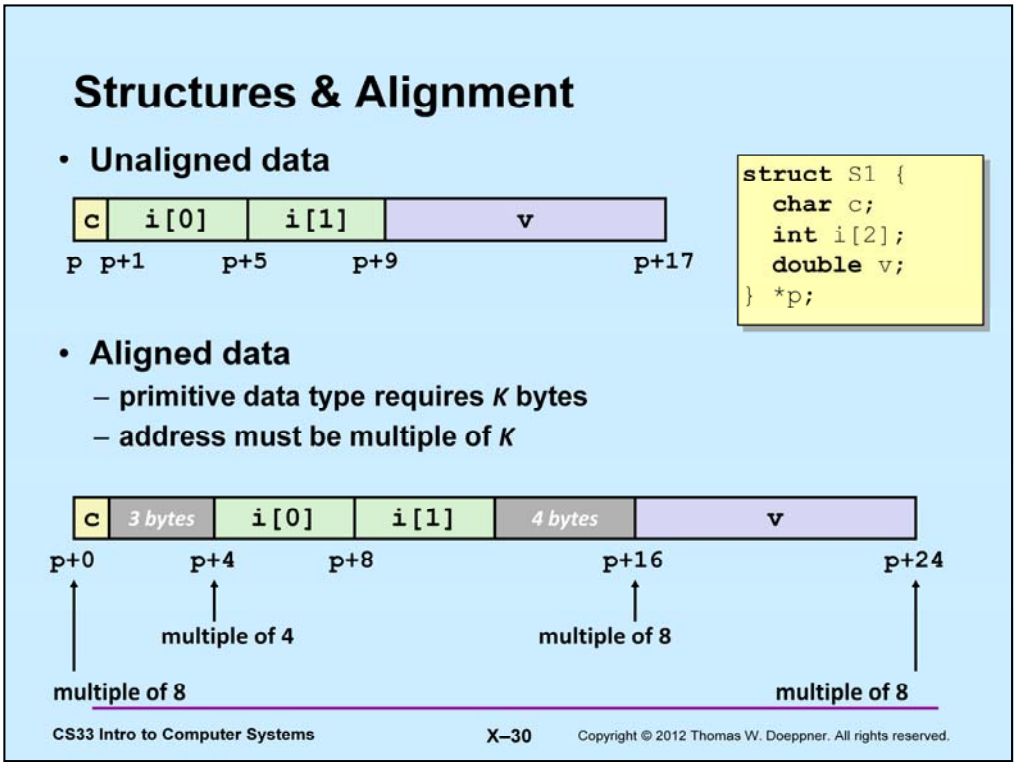


Supplied by CMU.

Today

- Arrays
  - one-dimensional
  - multi-dimensional (nested)
  - multi-level
- Structures
  - allocation
  - access
  - **alignment**

Supplied by CMU.



Supplied by CMU.

Note that in this and the following slides, p+i should be interpreted as meaning not p[i], as it would be in C, but the address p plus the integer value i. Thus if p is, for example, 0x1004, p+1 is 0x1005.

Alignment Principles

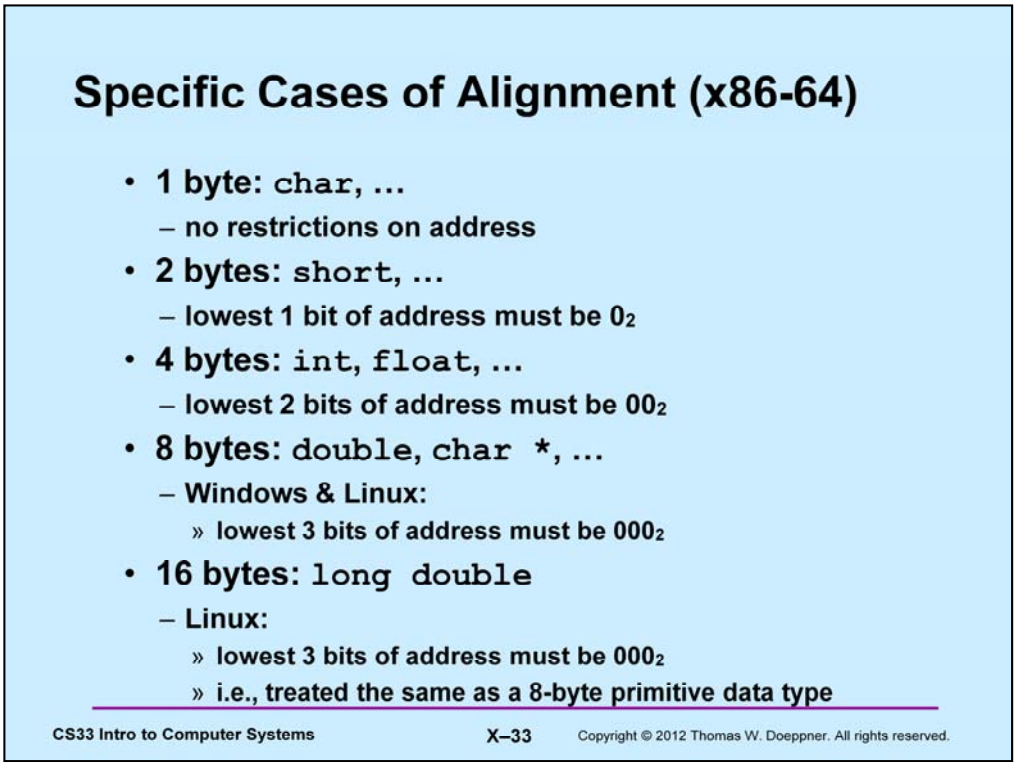
- Aligned data
  - primitive data type requires *K* bytes
  - address must be multiple of *K*
  - required on some machines; advised on IA32
    - » treated differently by IA32 Linux, x86-64 Linux, and Windows!
- Motivation for aligning data
  - memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - » inefficient to load or store datum that spans quad word boundaries
    - » virtual memory very tricky when datum spans 2 pages
- Compiler
  - inserts gaps in structure to ensure correct alignment of fields

Supplied by CMU.

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 of address must be 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:
    - » lowest 2 bits of address must be 00_2
    - » i.e., treated the same as a 4-byte primitive data type

Supplied by CMU.

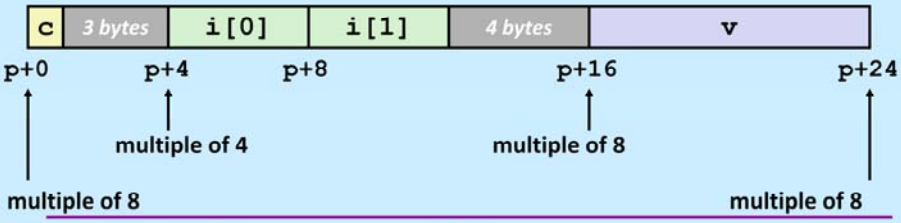


Supplied by CMU.

Alignment with Structures

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

- **Within structure:**
  - must satisfy each 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

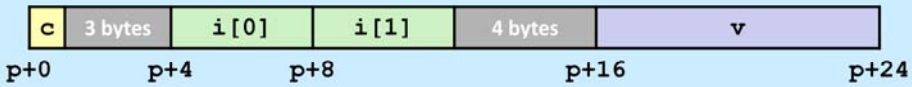


Supplied by CMU.

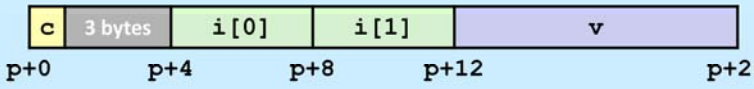
Different Alignment Conventions

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

- **x86-64 or IA32 Windows:**
  - **K = 8, due to `double` element**



- **IA32 Linux**
  - **K = 4; `double` treated like a 4-byte data type**

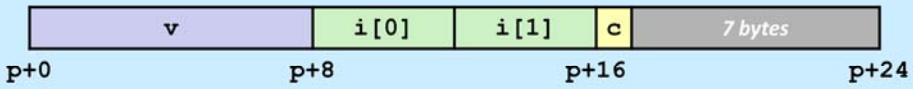


Supplied by CMU.

Meeting Overall Alignment Requirement

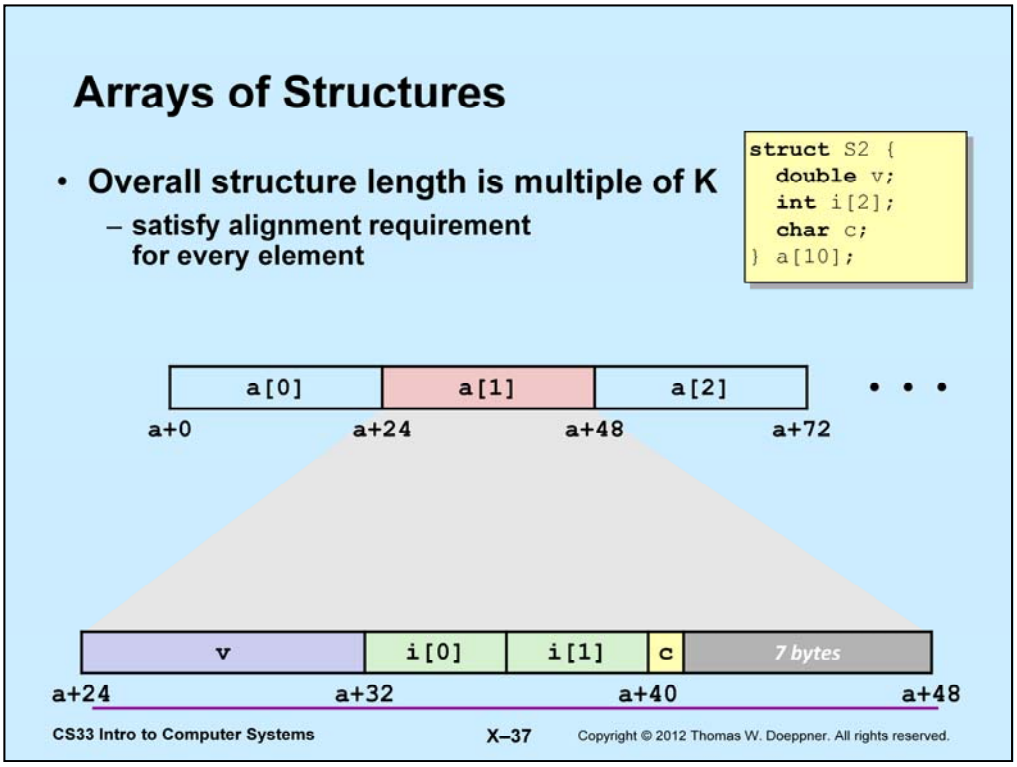
- For largest alignment requirement K
  - overall length of structure must be multiple of K

```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```

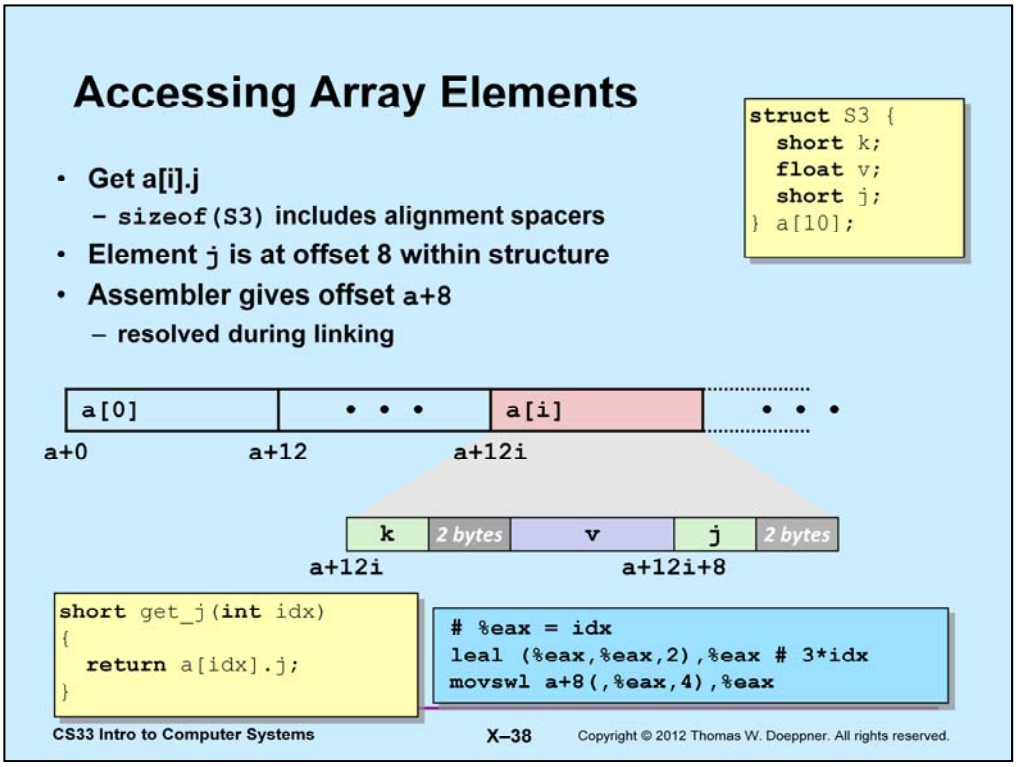


p+0 p+8 p+16 p+24

Supplied by CMU.



Supplied by CMU.



Supplied by CMU.

Note that even though the procedure *get_j* returns a short, its return value is converted to an int (a long word) — this is because of the convention that register eax (a 32-bit register) contains the return value.

Saving Space

- Put large data types first

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

- Effect (K=4)

Supplied by CMU.

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

Supplied by CMU.