Last year, for the first time, spending by Apple and Google on patent lawsuits and unusually big-dollar patent purchases exceeded spending on research and development of new products

But in 2008, Mr. Phillips's company, Vlingo, had been contacted by a much larger voice recognition firm called Nuance. "I have patents that can prevent you from practicing in this market," Nuance's chief executive, Paul Ricci, told Mr. Phillips, according to executives involved in that conversation.

Mr. Ricci issued an ultimatum: Mr. Phillips could sell his firm to Mr. Ricci or be sued for patent infringements. When Mr. Phillips refused to sell, Mr. Ricci's company filed the first of six lawsuits.

Soon after, Apple and Google stopped returning phone calls. The company behind Siri switched its partnership from Mr. Phillips to Mr. Ricci's firm. And the millions of dollars Mr. Phillips had set aside for research and development were redirected to lawyers and court fees.

When the first lawsuit went to trial last year, Mr. Phillips won. In the companies' only courtroom face-off, a jury ruled that Mr. Phillips had not infringed on a broad voice recognition patent owned by Mr. Ricci's company.

But it was too late. The suit had cost $3 million, and the financial damage was done. In December, Mr. Phillips agreed to sell his company to Mr. Ricci. "We were on the brink of changing the world before we got stuck in this legal muck," Mr. Phillips said.

http://www.nytimes.com/2012/10/08/technology/patent-wars-among-tech-giants-can-stifle-competition.html

- Lab 5 extension: 11:55pm on Friday
- Bomb project: coming shortly (due after midterm)
- Practice questions and last year's midterm on webpage

- For Tuesday
  - Create a question (with answer) for the midterm. It shouldn't be from the textbook.
  - Read scanned chapter from A History of Modern Computing & be ready to discuss.

- Practice problem on webpage

```
#include <stdio.h>

int globalvar;

int  func1(int *x);
int  func2(int  x, int *y);
void func3_hint_seepage215(int x);

main() {
  int mainvar = 0;
  int ret =0;

  globalvar = 364;

  printf("%d\t%d\t%d\n", mainvar, ret, globalvar);

  ret = func1(&mainvar);
  printf("%d\t%d\t%d\n", mainvar, ret, globalvar);

  ret = func2(mainvar, &globalvar);
  printf("%d\t%d\t%d\n", mainvar, ret, globalvar);

  func3_hint_seepage215(ret);
}
```

```
int func1(int *x) {

  *x = 1 + 364;

  return 0;
}

int func2(int x, int *y) {

  return x * *y;
}
```

```
void func3_hint_seepage215(int x) {
  int val = 2;

  switch(x) {
    case 1: {
      val = x + 1;
      break;
    }
    case 2: {
      val = x / 10;
      break;
    }
    case 3: {
      val = x * 10;
      break;
    }
    case 4: {
      val = x % 10;
      break;
    }
    case 128: {
      val = 3;
      break;
    }
    default: {val = 0; }
  }
  printf("Something %d\n", val);
}
```

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;
}
```

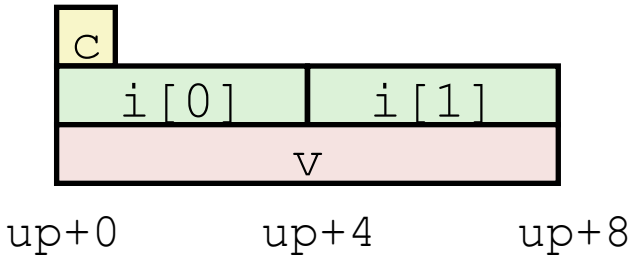
IA32 Assembly

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

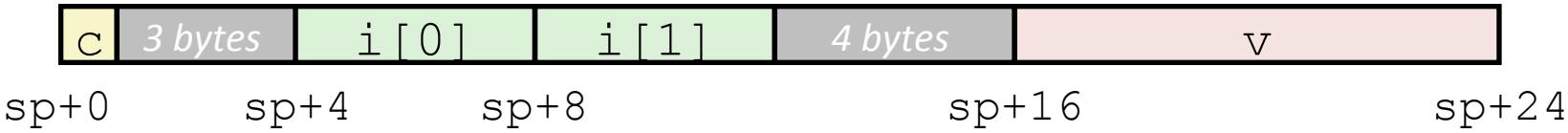
Union Allocation

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

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

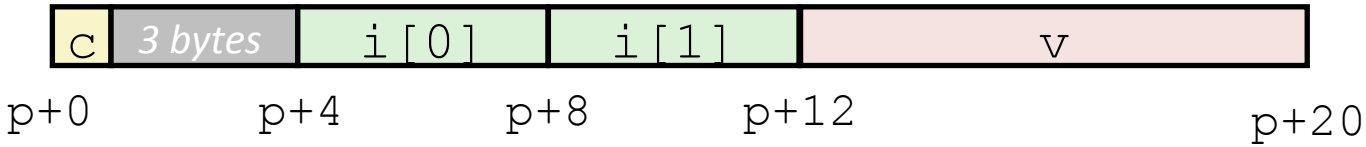



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Different Alignment Conventions

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



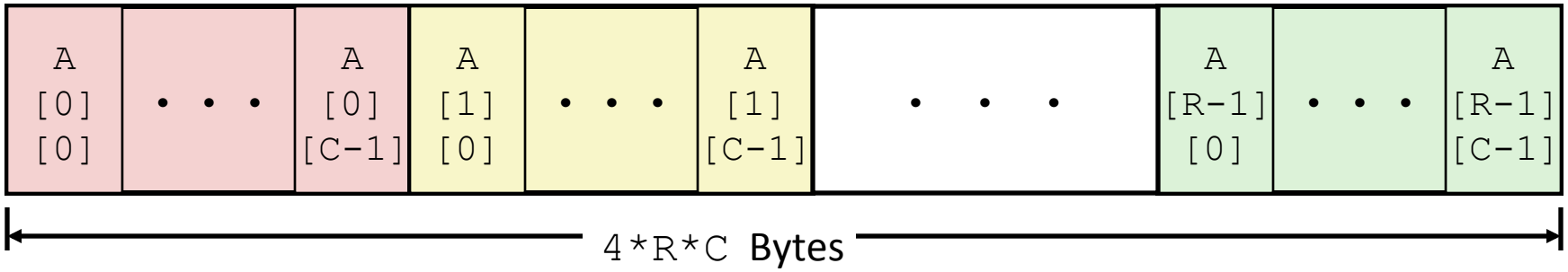
- IA32 Linux
  - K = 4; **`double`** treated like a 4-byte data type

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

$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A[R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

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

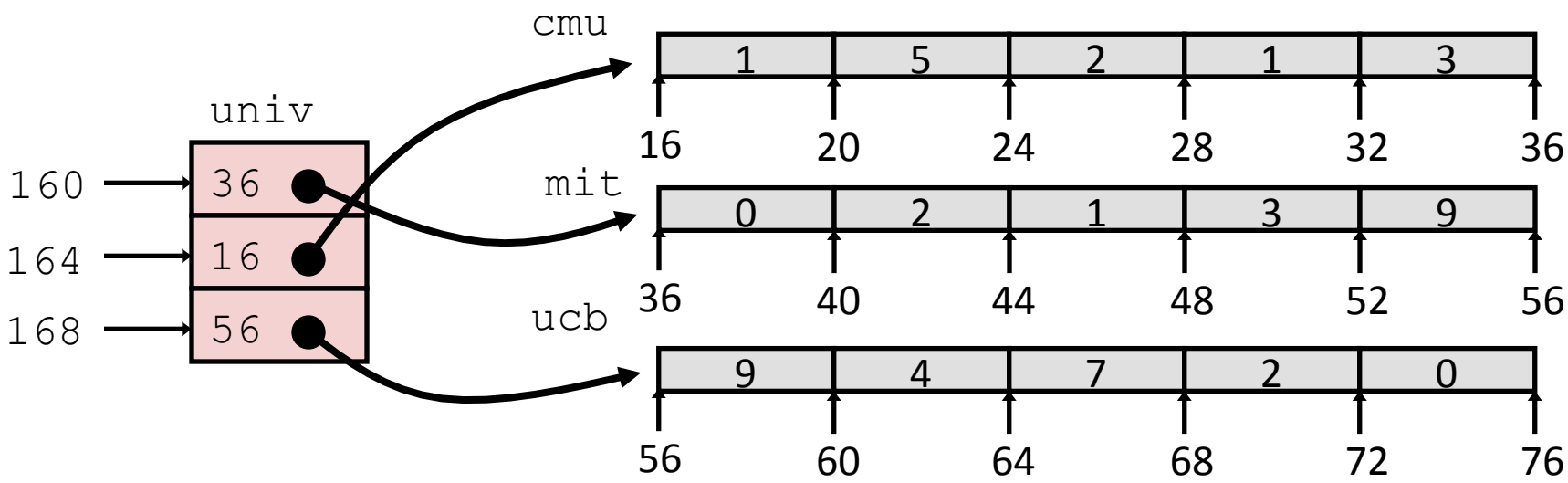


Multi-Level Array Example

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

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

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



Element Access in Multi-Level Array

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

```
  # %ecx = index
  # %eax = dig
  leal 0(,%ecx,4),%edx     # 4*index
  movl univ(%edx),%edx     # Mem[univ+4*index]
  movl (%edx,%eax,4),%eax  # Mem[...+4*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

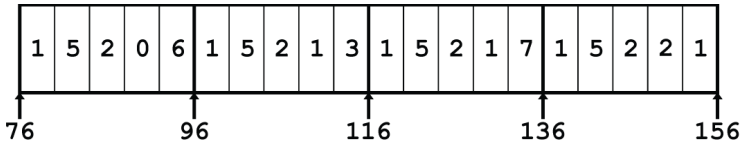
Array Element Accesses

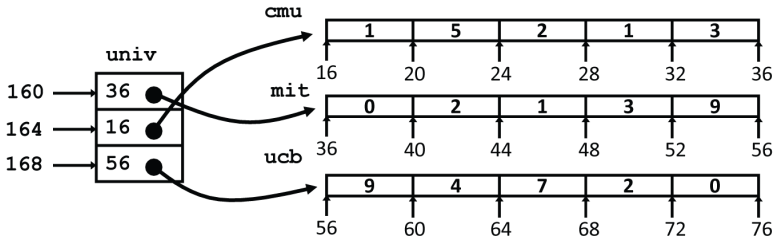
Nested array

```
int get_pgh_digit
  (int index, int dig)
{
  return pgh[index][dig];
}
```

Multi-level array

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



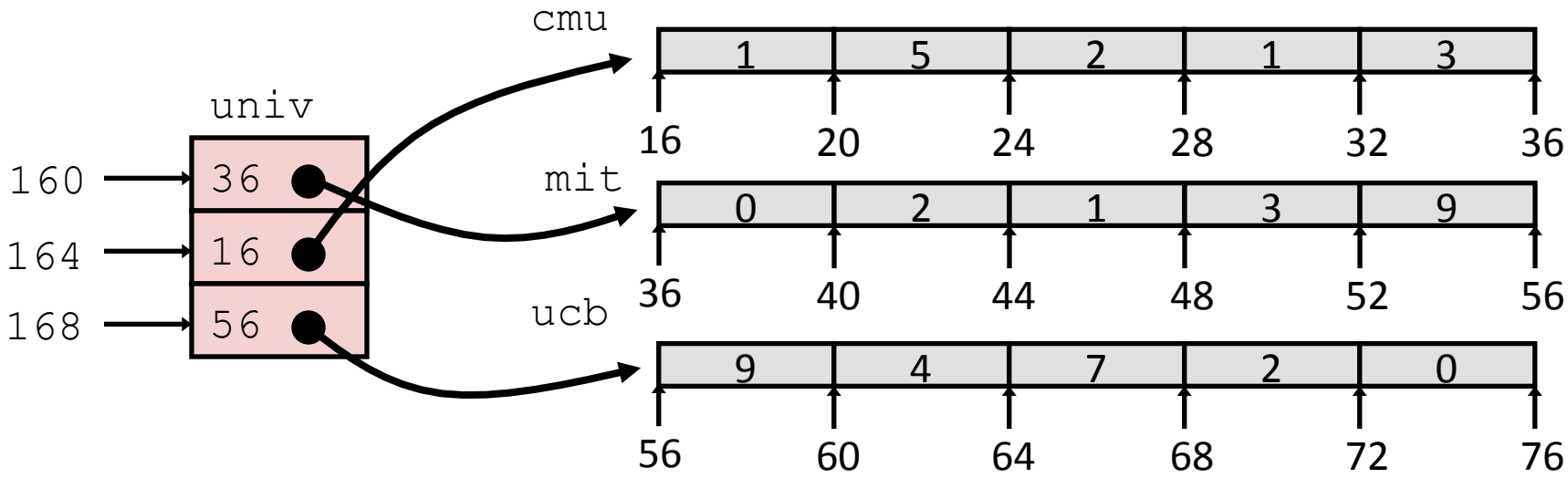


Access looks similar…

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

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

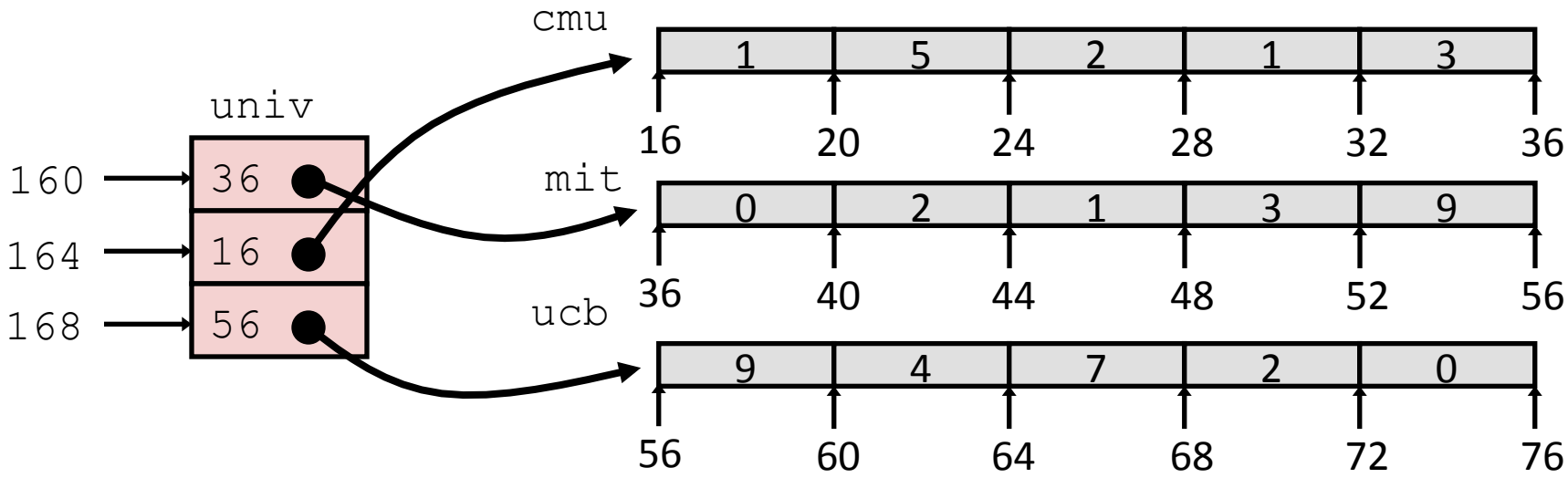
Strange Referencing Examples



- Reference Address Value Guaranteed?

univ[2][3]	**56+4*3 = 68**	**2**	
univ[1][5]			
univ[2][-1]			
univ[3][-1]			
univ[1][12]			

Strange Referencing Examples



- Reference Address Value Guaranteed?

univ[2][3]	**56+4*3 = 68**	**2**	Yes
univ[1][5]	**16+4*5 = 36**	**0**	No
univ[2][-1]	**56+4*-1 = 52**	**9**	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

Consider the source code below, used to keep track of the rooms currently reserved in a family-run hotel. Each entry in the *residents* array stores a name of the customer reserving the room. *FLOORS* represents the number of floors in the hotel. *ROOMS* represents the number of rooms per floor. Both are constants declared with #define. LEN, the maximum number of bytes allocated for a name, is defined to be 12.

```
char residents[FLOORS][ROOMS][LEN];

void reserve_room(int floor, int room, char *custname){
    strcpy(residents[floor][room], custname);
}
```

The assembly code for the function *reserve_room* looks like this:

```
reserve_room:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    movl 16(%ebp),%edx
    pushl %edx
    movl 8(%ebp),%edx
    sall $4,%edx
    subl 8(%ebp),%edx
    leal (%eax,%eax,2),%eax
    leal residents(,%eax,4),%eax
    leal (%eax,%edx,4),%edx
    pushl %edx
    call strcpy
    movl %ebp,%esp
    popl %ebp
    ret
```

Due to a strange bug, the program accesses *residents[0][1][-2]*. What value is actually being accessed? (Express your answer as an integer triplet (-,-,-). You may assume that FLOORS and ROOMS are both greater than 1)

What is the value of *ROOMS*?

```
char *gets(char *s)
{
   int c;
   char *dest = s;
   int gotchar = 0;
   while ((c = getchar()) != '\n' && c != EOF){
    *dest++ = c;
    gotchar = 1;
   }
   *dest++ = '\0';
   if (c == EOF && !gotchar)
    return NULL;
   return s;
}

void echo()
{
   char buf[8];
   gets(buf);
   puts(buf);
}
```