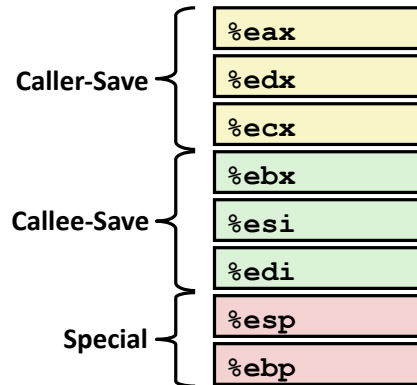


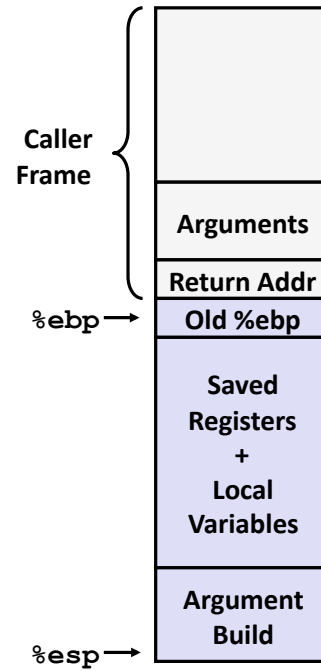
Last Time

■ Procedures (IA32)

- call / return
- %esp, %ebp
- local variables
- recursive functions



CSE351 - Autumn 2010



1

Today

■ Arrays

- One-dimensional
- Multi-dimensional (nested)
- Multi-level

■ Structs

- Alignment

■ Unions

CSE351 - Autumn 2010

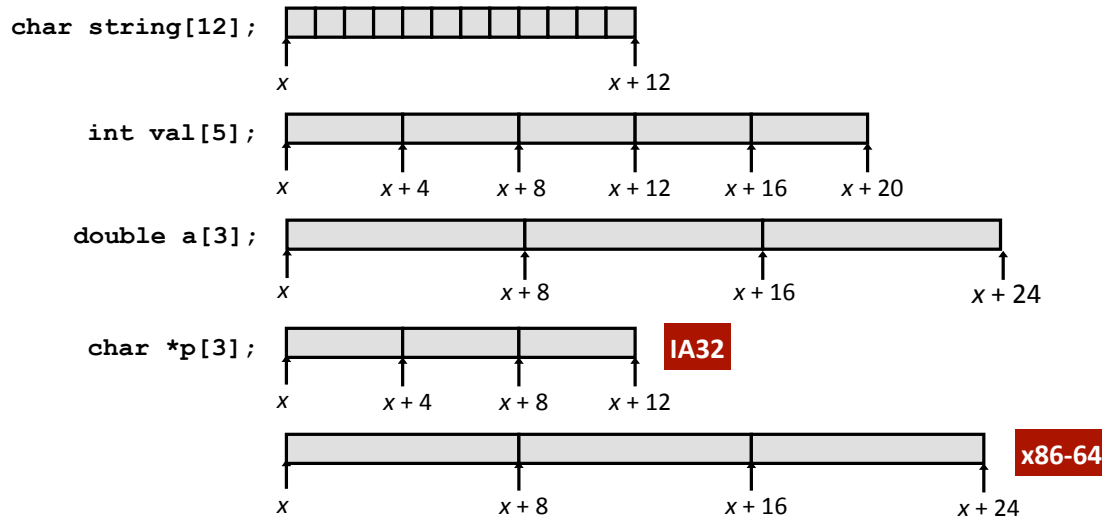
2

Array Allocation

■ Basic Principle

$T\ A[N];$

- Array of data type T and length N
- Contiguously allocated region of $N * \text{sizeof}(T)$ bytes



CSE351 - Autumn 2010

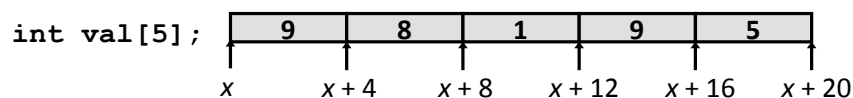
3

Array Access

■ Basic Principle

$T\ A[N];$

- Array of data type T and length N
- Identifier A can be used as a pointer to array element 0: Type T^*



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

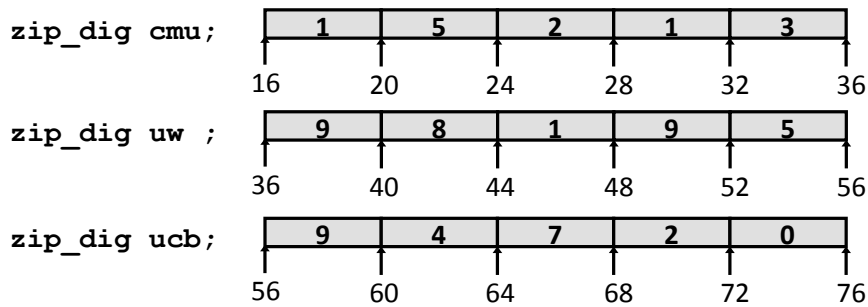
CSE351 - Autumn 2010

4

Array Example

```
typedef int zip_dig[5];

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



- Declaration “`zip_dig uw`” equivalent to “`int uw[5]`”
- Example arrays were allocated in successive 20 byte blocks
 - Not guaranteed to happen in general

Array Accessing Example



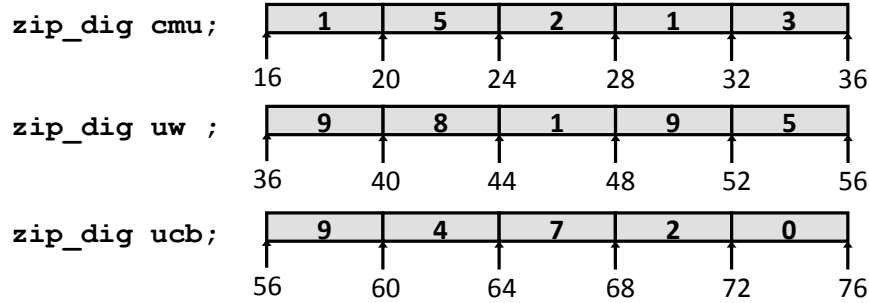
```
int get_digit
(zip_dig z, int dig)
{
    return z[dig];
}
```

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 $4 * \text{eax} + \text{edx}$
- Use memory reference `(%edx,%eax,4)`

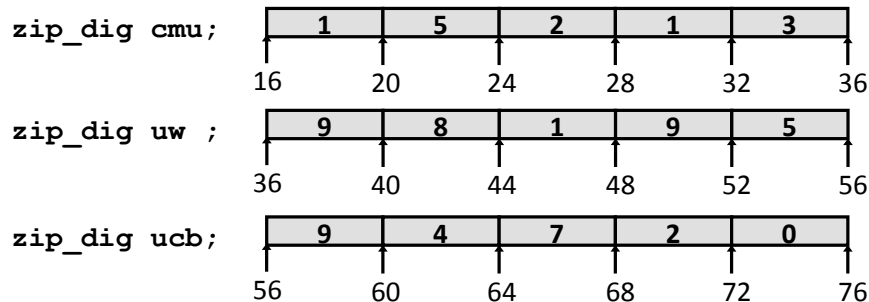
Referencing Examples



Reference	Address	Value	Guaranteed?
<code>uw[3]</code>			
<code>uw[6]</code>			
<code>uw[-1]</code>			
<code>cmu[15]</code>			

What are these values?

Referencing Examples



Reference	Address	Value	Guaranteed?
<code>uw[3]</code>	$36 + 4 * 3 = 48$	9	Yes
<code>uw[6]</code>	$36 + 4 * 6 = 60$	4	No
<code>uw[-1]</code>	$36 + 4 * -1 = 32$	3	No
<code>cmu[15]</code>	$16 + 4 * 15 = 76$??	No

- No bound checking
- Out-of-range behavior implementation-dependent
- No guaranteed relative allocation of different arrays

Array Loop Example

■ Original

```
int zd2int(zip_dig z)
{
    int i;
    int zi = 0;
    for (i = 0; i < 5; i++) {
        zi = 10 * zi + z[i];
    }
    return zi;
}
```

■ Transformed

- As generated by GCC
- Eliminate loop variable *i*
- Convert array code to pointer code
- Express in do-while form (no test at entrance)

```
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while (z <= zend);
    return zi;
}
```

Array Loop Implementation (IA32)

```
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```

```
# %ecx = z
xorl %eax,%eax
leal 16(%ecx),%ebx
.L59:
leal (%eax,%eax,4),%edx
movl (%ecx),%eax
addl $4,%ecx
leal (%eax,%edx,2),%eax
cmpl %ebx,%ecx
jle .L59
```

Translation?

Array Loop Implementation (IA32)

Registers

```
%ecx z
%eax zi
%ebx zend
```

Computations

- 10*zi + *z implemented as *z + 2*(zi+4*zi)
- z++ increments by 4

```
int zd2int(zip_dig z)
{
    int zi = 0;
    int *zend = z + 4;
    do {
        zi = 10 * zi + *z;
        z++;
    } while(z <= zend);
    return zi;
}
```

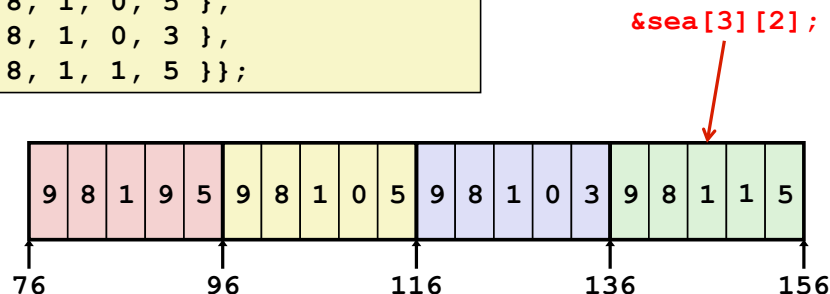
```
# %ecx = z
xorl %eax,%eax          # zi = 0
leal 16(%ecx),%ebx      # zend = z+4
.L59:
leal (%eax,%eax,4),%edx # 5*zi
movl (%ecx),%eax        # *z
addl $4,%ecx            # z++
leal (%eax,%edx,2),%eax # zi = *z + 2*(5*zi)
cmpl %ebx,%ecx          # z : zend
jle .L59                # if <= goto loop
```

CSE351 - Autumn 2010

11

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 }};
```



- “zip_dig sea[4]” equivalent to “int sea[4][5]”
 - Variable `sea`: array of 4 elements, allocated contiguously
 - Each element is an array of 5 `ints`, allocated contiguously
- “row-major” ordering of all elements guaranteed

CSE351 - Autumn 2010

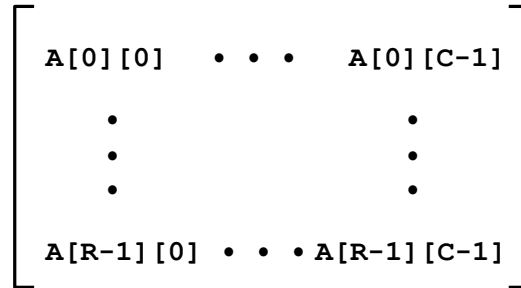
12

Multidimensional (Nested) Arrays

Declaration

T $A[R][C];$

- 2D array of data type T
- R rows, C columns
- Type T element requires K bytes



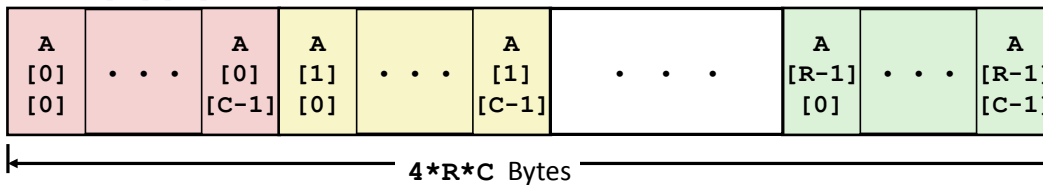
Array size

- $R * C * K$ bytes

Arrangement

- Row-major ordering

`int A[R][C];`

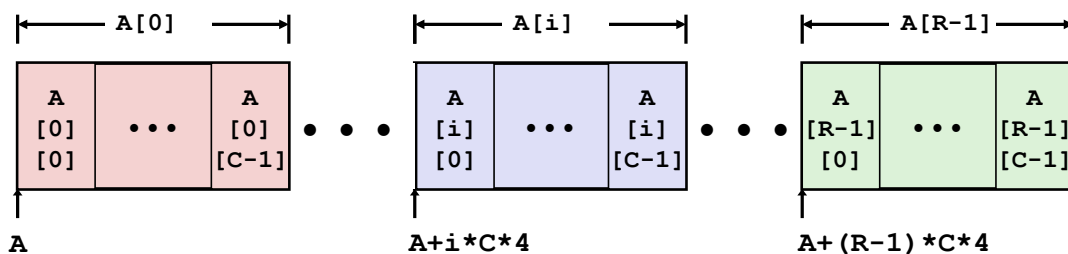


Nested Array Row Access

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];`



Nested Array Row Access Code

```
int *get_sea_zip(int index)
{
    return sea[index];
}
```

```
#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 }};
```

- What data type is `sea[index]`?
- What is its starting address?

```
# %eax = index
leal (%eax,%eax,4),%eax
leal sea(,%eax,4),%eax
```

Translation?

Nested Array Row Access Code

```
int *get_sea_zip(int index)
{
    return sea[index];
}
```

```
#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 }};
```

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

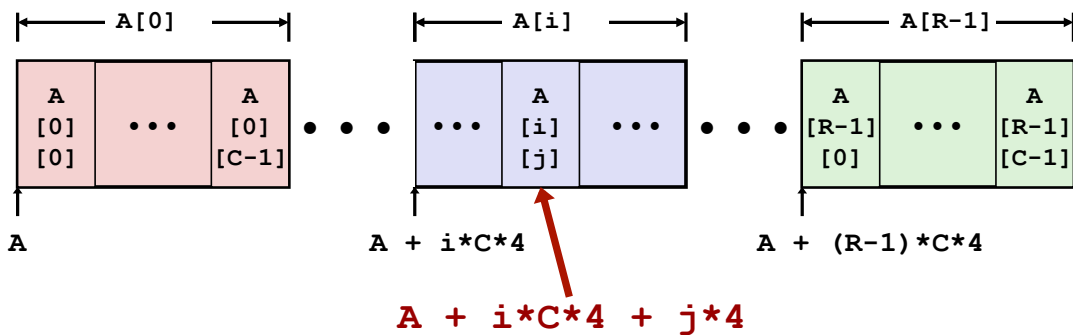
- **Row Vector**
 - `sea[index]` is array of 5 ints
 - Starting address `sea+20*index`
- **IA32 Code**
 - Computes and returns address
 - Compute as `sea+4*(index+4*index)=sea+20*index`

Nested Array Row 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];
```



CSE351 - Autumn 2010

17

Nested Array Element Access Code

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

```
# %ecx = dig
# %eax = index
leal 0(,%ecx,4),%edx      # 4*dig
leal (%eax,%eax,4),%eax   # 5*index
movl sea(%edx,%eax,4),%eax # *(sea + 4*dig + 20*index)
```

■ Array Elements

- $sea[index][dig]$ is `int`
- Address: $sea + 20 * index + 4 * dig$

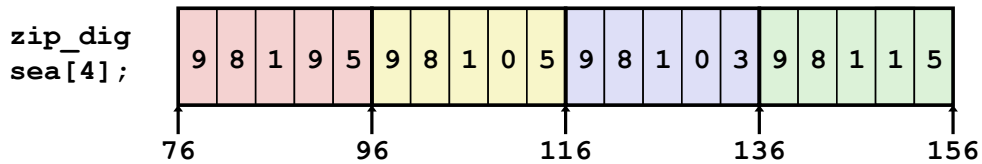
■ IA32 Code

- Computes address $sea + 4 * dig + 4 * (index + 4 * index)$
- `movl` performs memory reference

CSE351 - Autumn 2010

18

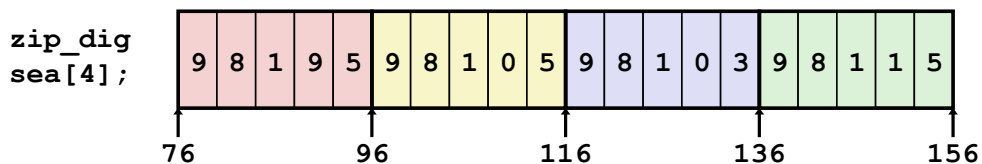
Strange Referencing Examples



Reference	Address	Value	Guaranteed?
sea[3][3]			
sea[2][5]			
sea[2][-1]			
sea[4][-1]			
sea[0][19]			
sea[0][-1]			

What are these values?

Strange Referencing Examples



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

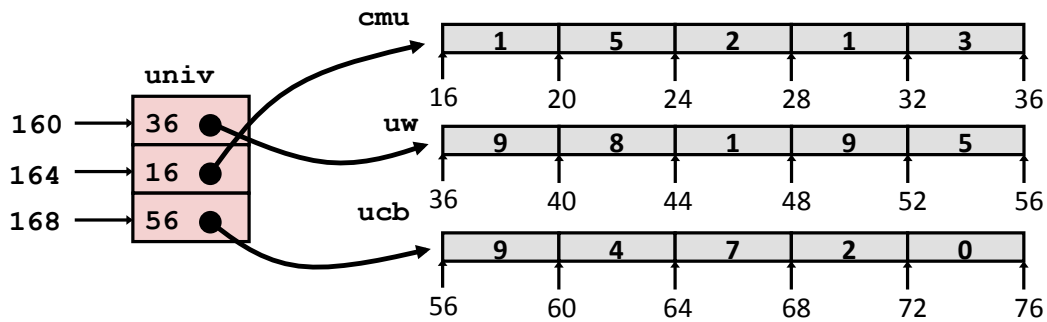
- Code does not do any bounds checking
- Ordering of elements within array guaranteed

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};
```

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



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
movl univ(%edx),%edx
movl (%edx,%eax,4),%eax
```

Translation?

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]
```

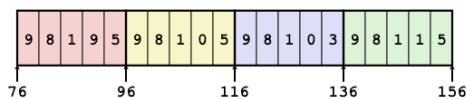
■ Computation (IA32)

- Element access $\text{Mem}[\text{Mem}[\text{univ}+4*\text{index}]+4*\text{dig}]$
- Must do two memory reads
 - First get pointer to row array
 - Then access element within array

Array Element Accesses

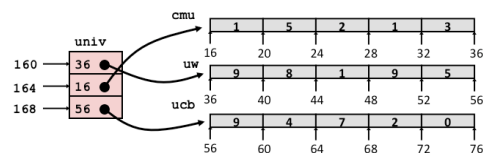
Nested array

```
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];
}
```

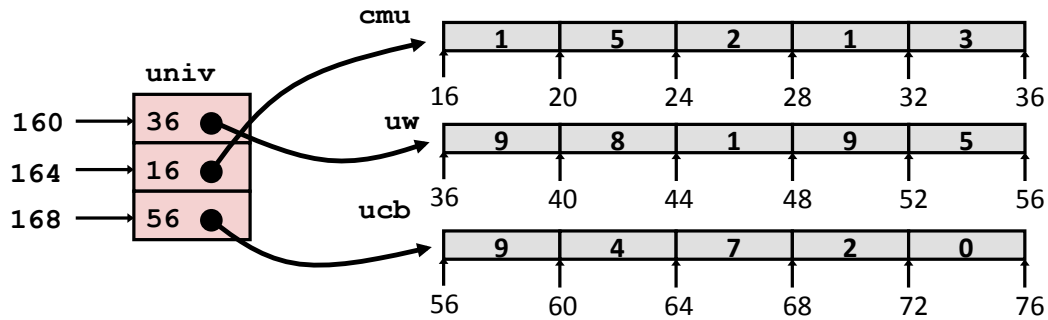


Access looks similar, but it isn't:

$\text{Mem}[\text{sea}+20*\text{index}+4*\text{dig}]$

$\text{Mem}[\text{Mem}[\text{univ}+4*\text{index}]+4*\text{dig}]$

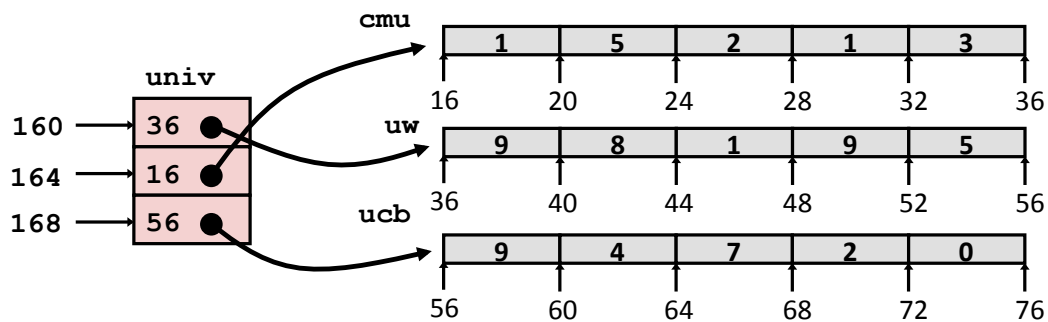
Strange Referencing Examples



Reference	Address	Value	Guaranteed?
<code>univ[2][3]</code>			
<code>univ[1][5]</code>			
<code>univ[2][-1]</code>			
<code>univ[3][-1]</code>			
<code>univ[1][12]</code>			

What values go here?

Strange Referencing Examples



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

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

Using Nested Arrays

Strengths

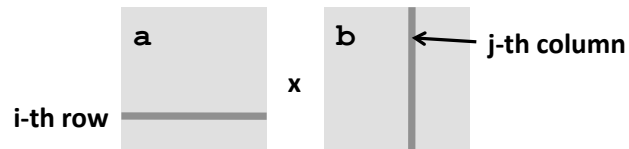
- C compiler handles doubly subscripted arrays
- Generates very efficient code
- Avoids multiply in index computation

Limitation

- Only works for fixed array size

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

```
/* Compute element i,k of
   fixed matrix product */
int fix_prod_ele
(fix_matrix a, fix_matrix b,
 int i, int k)
{
    int j;
    int result = 0;
    for (j = 0; j < N; j++)
        result += a[i][j]*b[j][k];
    return result;
}
```



Dynamic Nested Arrays

Strength

- Can create matrix of any size

Programming

- Must do index computation explicitly

Performance

- Accessing single element costly
- Must do multiplication

```
int * new_var_matrix(int n)
{
    return (int *)
        calloc(sizeof(int), n*n);
}
```

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

```
movl 12(%ebp),%eax    # i
movl 8(%ebp),%edx    # a
imull 20(%ebp),%eax  # n*i
addl 16(%ebp),%eax   # n*i+j
movl (%edx,%eax,4),%eax # Mem[a+4*(i*n+j)]
```

Dynamic Array Multiplication

■ Without Optimizations

- Multiplies: 3
 - 2 for subscripts
 - 1 for data
- Adds: 4
 - 2 for array indexing
 - 1 for loop index
 - 1 for data

```

/* Compute element i,k of
   variable matrix product */
int var_prod_ele
(int *a, int *b,
 int i, int k, int n)
{
  int j;
  int result = 0;
  for (j = 0; j < n; j++)
    result +=
      a[i*n + j] * b[j*n + k];
  return result;
}

```

Optimizing Dynamic Array Multiplication

■ Optimizations

- Performed when set optimization level to `-O2`

■ Code Motion

- Expression `i*n` can be computed outside loop

■ Strength Reduction

- Incrementing `j` has effect of incrementing `j*n+k` by `n`

■ Operations count

- 4 adds, 1 mult

■ Compiler can optimize regular access patterns

```

{
  int j;
  int result = 0;
  for (j = 0; j < n; j++)
    result +=
      a[i*n + j] * b[j*n + k];
  return result;
}

```

```

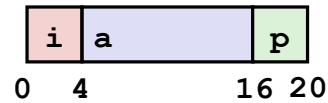
{
  int j;
  int result = 0;
  int iTn = i*n;
  int jTnPk = k;
  for (j = 0; j < n; j++) {
    result +=
      a[iTn + j] * b[jTnPk];
    jTnPk += n;
  }
  return result;
}

```

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;
}
```

CSE351 - Autumn 2010

In java: `r.i = val;`

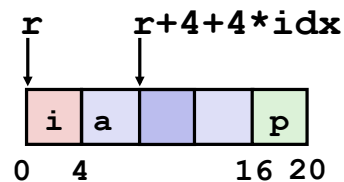
IA32 Assembly

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

31

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);
}
```

In java? ...

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

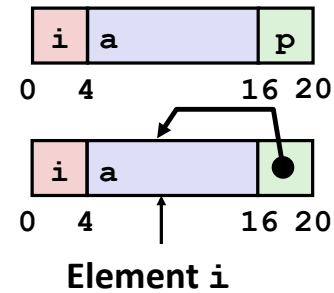

Structure Referencing (Cont.)

■ 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);
}
```

```
# %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
```



In java? ...

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

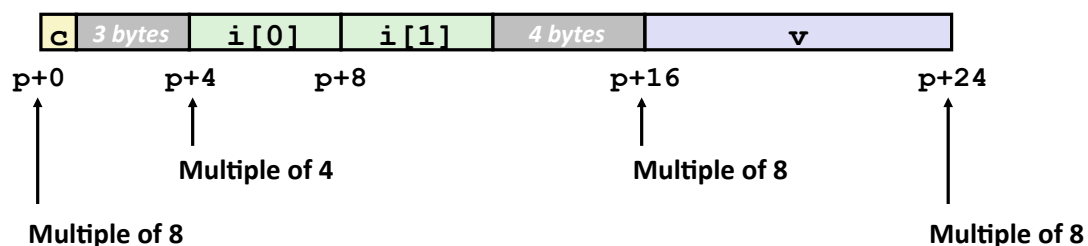
Specific Cases of Alignment (IA32)

- **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)

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;
```

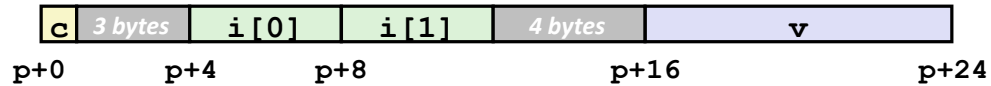


Different Alignment Conventions

- **IA32 Windows or x86-64:**

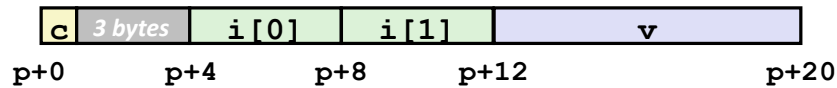
- $K = 8$, due to `double` element

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



- **IA32 Linux**

- $K = 4$; `double` treated like a 4-byte data type



Saving Space

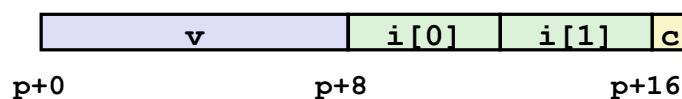
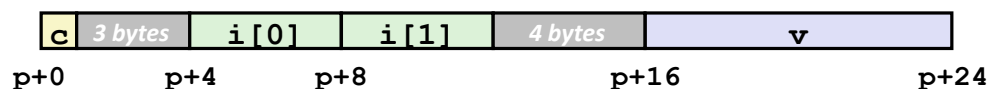
- **Put large data types first**

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



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

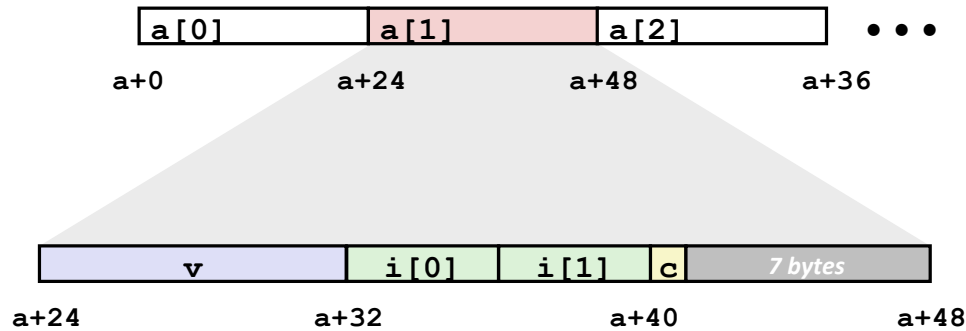
- **Effect (example x86-64, both have $K=8$)**



Arrays of Structures

- Satisfy alignment requirement for every element

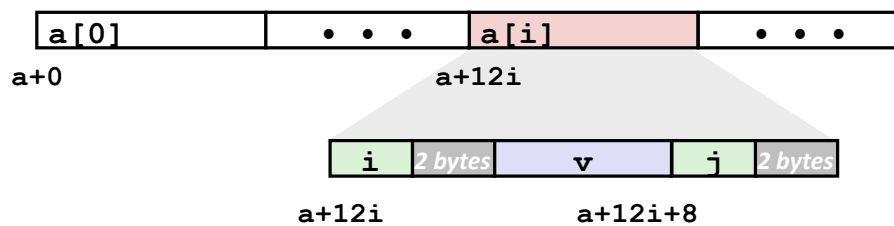
```
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```



Accessing Array Elements

- Compute array offset $12i$
- Compute offset 8 with structure
- Assembler gives offset $a+8$
 - Resolved during linking

```
struct S3 {
    short i;
    float v;
    short j;
} a[10];
```



```
short get_j(int idx)
{
    return a[idx].j;
// return (a + idx)->j;
}
```

```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```

Unions

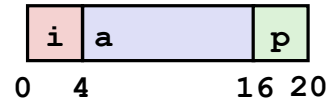
```
struct rec {
    int i;
    int a[3];
    int *p;
};
```

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

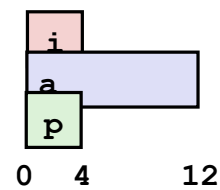
■ Concept

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

Structure Layout



Union Layout

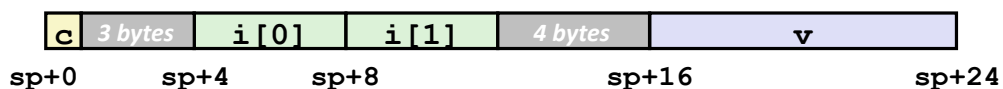
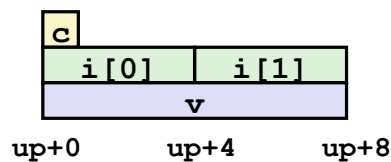


Union Allocation

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

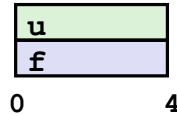
```
union U1 {
    char c;
    int i[2];
    double v;
} *up;
```

```
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