Machine-Level Programming: Structured Data

Chapter 3 of B&O

Some notes adopted from Bryant and O'Hallaron

Basic Data Types

- Integral
  - Stored & operated on in general registers
  - Signed vs. unsigned depends on instructions used

Intel	GAS	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	GAS	Bytes	C
Single	`s`	4	`float`
Double	`l`	8	`double`
Extended	`t`	10/12/16	`long double`

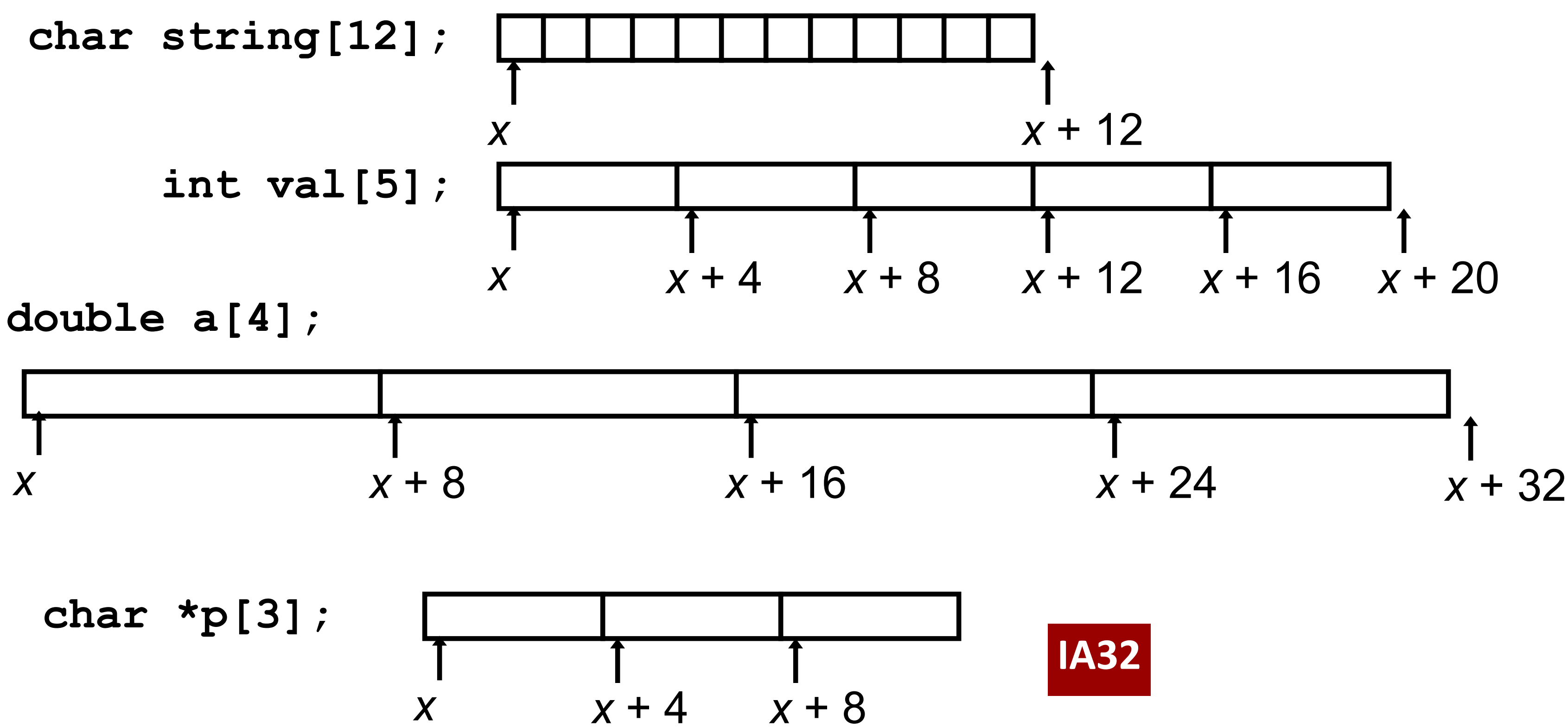
Array Allocation

- Basic Principle

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

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

  – Contiguously allocated region of ***L*** * `sizeof(`***T***`)` bytes



Array Access

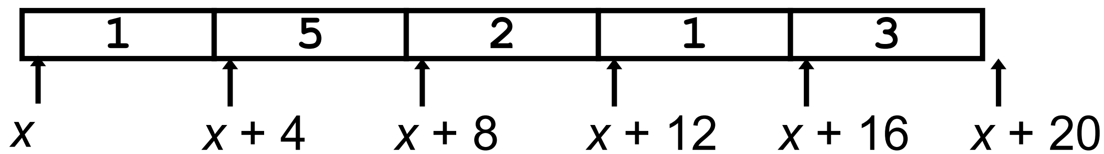
- Basic Principle

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

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

  – Identifier `A` can be used as a pointer to array element 0

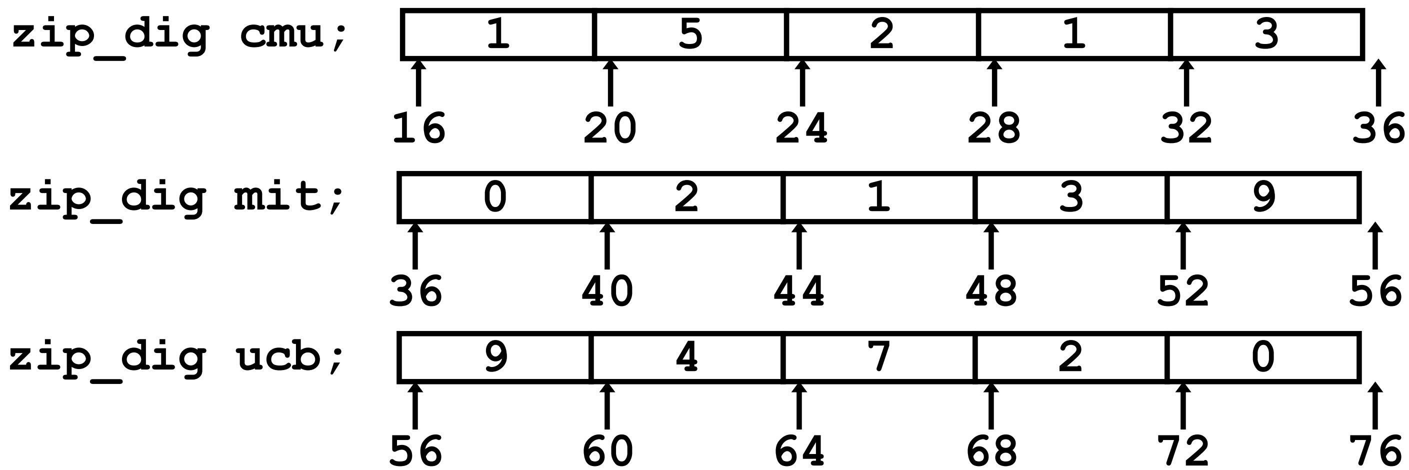
`int val[5];`



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***

Array Example

```
typedef int zip_dig[5];

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



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

Array Accessing Example

Computation

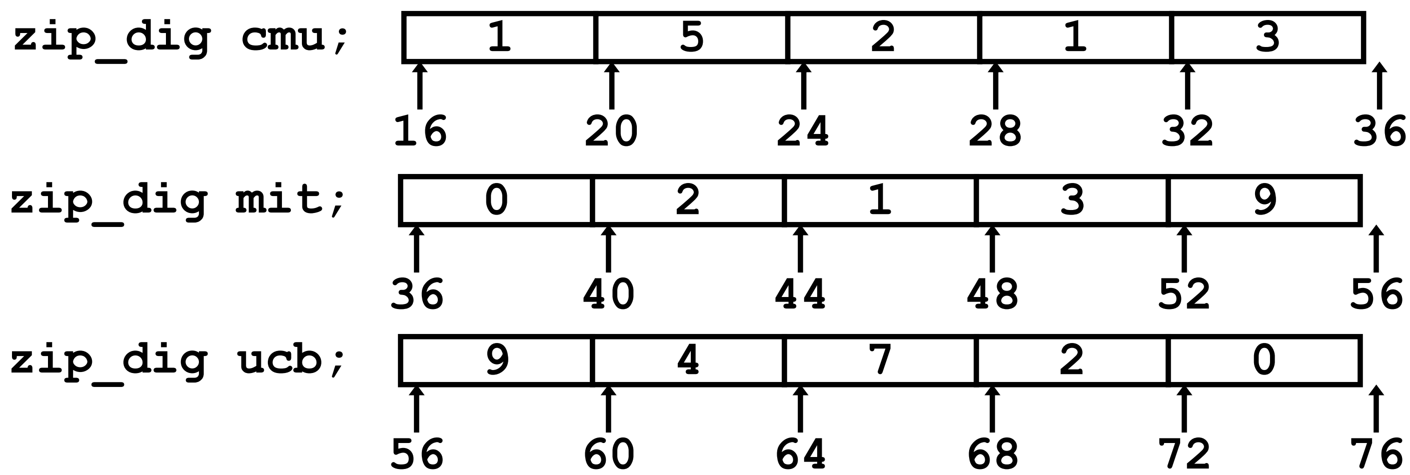
- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at `4*%eax + %edx`
- Use memory reference `(%edx,%eax,4)`

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

Memory Reference Code

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

Referencing Examples



Code Does Not Do Any Bounds Checking!

Reference	Address	Value	Guaranteed?
`mit[3]`	`36 + 4* 3 = 48`	`3`	**Yes**
`mit[5]`	`36 + 4* 5 = 56`	`9`	**No**
`mit[-1]`	`36 + 4*-1 = 32`	`3`	**No**
`cmu[15]`	`16 + 4*15 = 76`	`??`	**No**

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

Array Loop Example

Original Source

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

Transformed Version

- As generated by GCC
- Eliminate loop variable `i`
- Convert array code to pointer code
- Express in do-while form
  - No need to 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

- 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
```

Nested Array Example

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

zip_dig pgh[4];

1	5	2	0	6	1	5	2	1	3	1	5	2	1	7	1	5	2	2	1

76 96 116 136 156

- Declaration "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  - Variable `pgh` denotes array of 4 elements
    - Allocated contiguously
  - Each element is an array of 5 `int`'s
    - Allocated contiguously
- "Row-Major" ordering of all elements guaranteed

Nested Array Allocation

- Declaration

  T A[*R*][*C*];

  – Array of data type *T*

  – *R* rows, *C* columns

  – Type *T* element requires *K* bytes

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

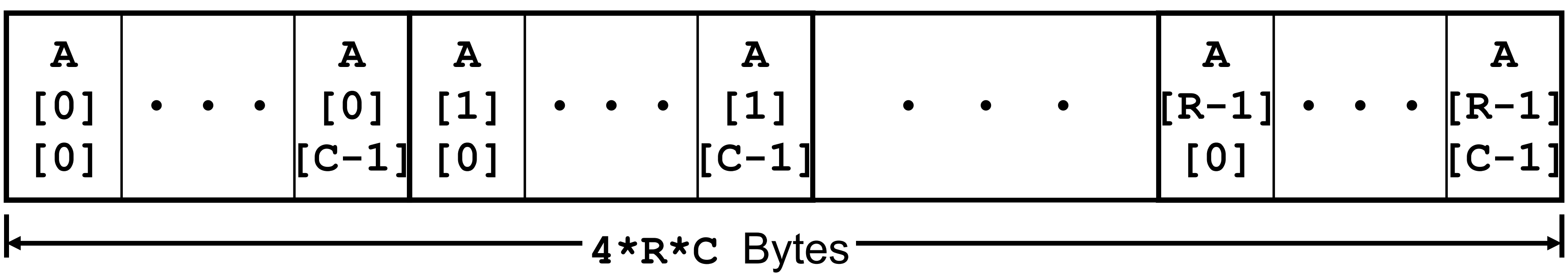
- Array Size

  – *R* * *C* * *K* bytes

- Arrangement

  – Row-Major Ordering

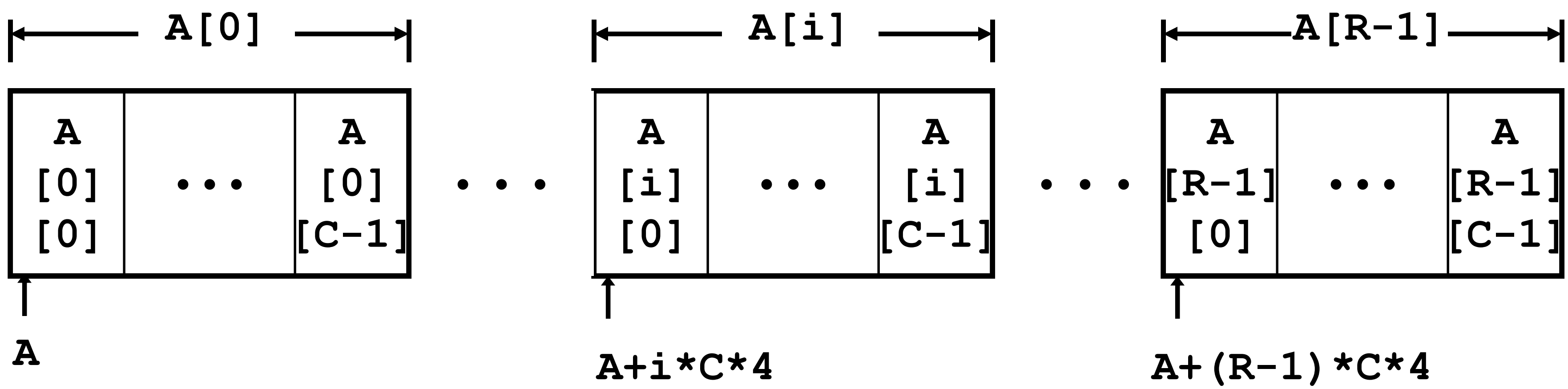
`int A[R][C];`



Nested Array Row Access

- Row Vectors
  - `A[i]` is array of ***C*** elements
  - Each element of type ***T***
  - Starting address `A + ` ***i * C * K***

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



Nested Array Row Access Code

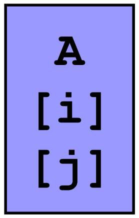
```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

- Row Vector
  - `pgh[index]` is array of 5 `int`'s
  - Starting address `pgh+20*index`
- Code
  - Computes and returns address
  - Compute as `pgh + 4*(index+4*index)`

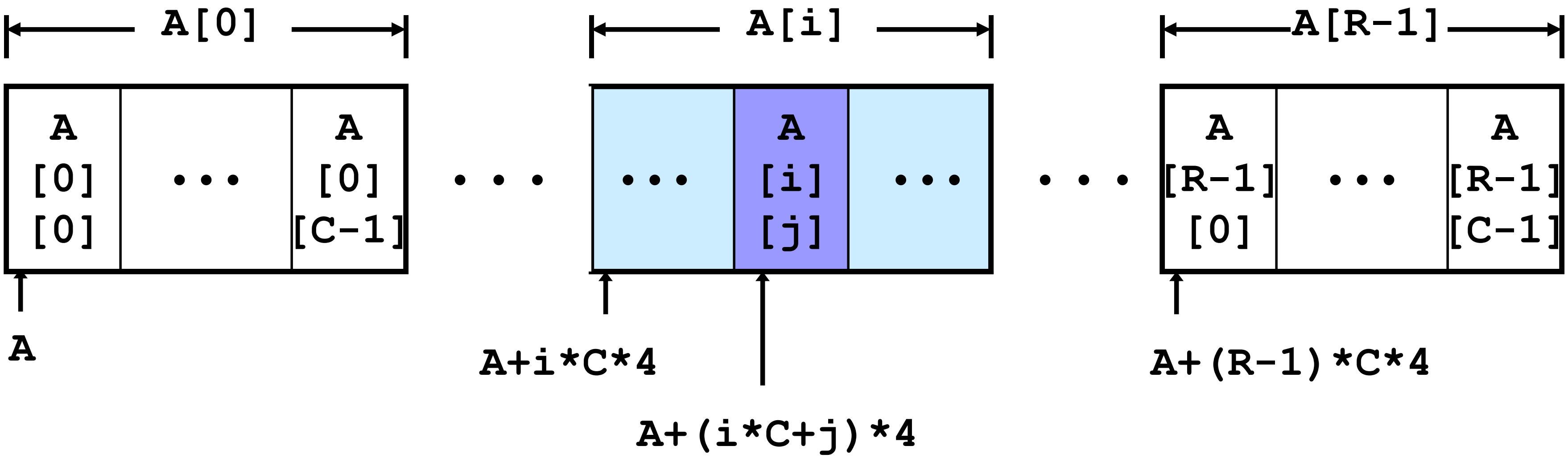
```
  # %eax = index
  leal  (%eax,%eax,4),%eax  # 5 * index
  leal pgh(,%eax,4),%eax    # pgh + (20 * index)
```

Nested Array Element Access

- Array Elements
  - `A[i][j]` is element of type ***T***
  - Address `A + ` ***(i * C + j) * K***




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



Nested Array Element Access Code

- Array Elements
  - `pgh[index][dig]` is `int`
  - Address:

    `pgh + 20*index + 4*dig`

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

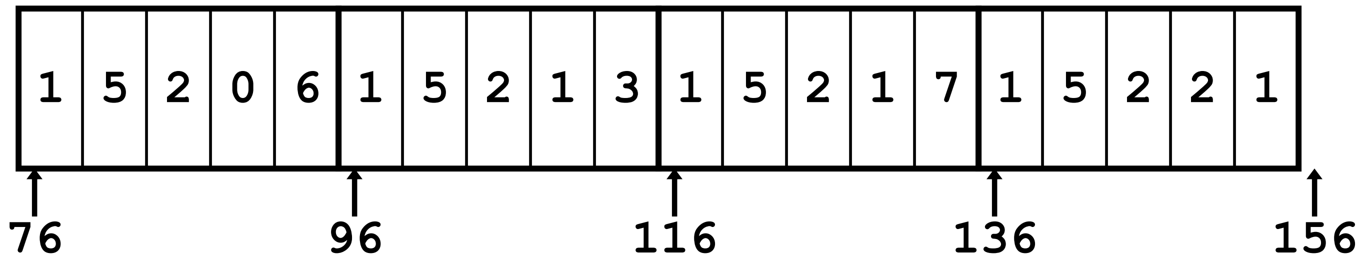
- Code
  - Computes address

    `pgh + 4*dig + 4*(index+4*index)`
  - `movl` performs memory reference

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

Strange Referencing Examples

`zip_dig pgh[4];`



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

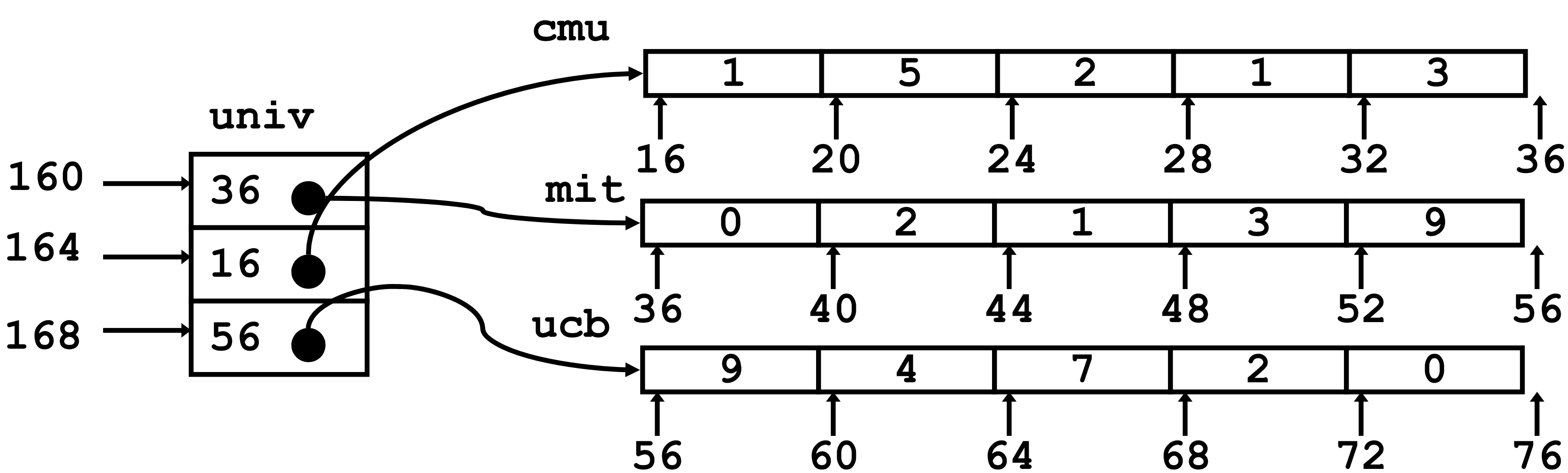
- Code does not do any bounds checking
- Ordering of elements within array guaranteed

Multi-Level Array Example

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

```
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};
```



Element Access in Multi-Level Array

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

- Computation
  - 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

```
# %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]
```

Array Element Accesses

Similar C references

Different address computation

- Nested Array

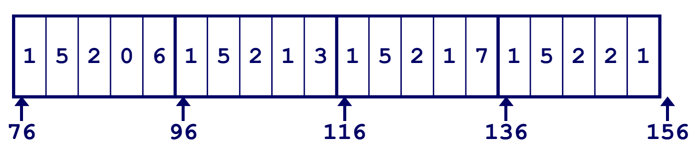
```
int get_pgh_digit
  (int index, int dig)
{
  return pgh[index][dig];
}
```

- Multi-Level Array

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

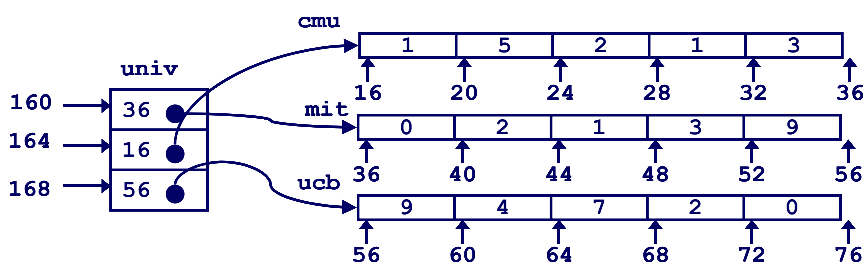
– Element at

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

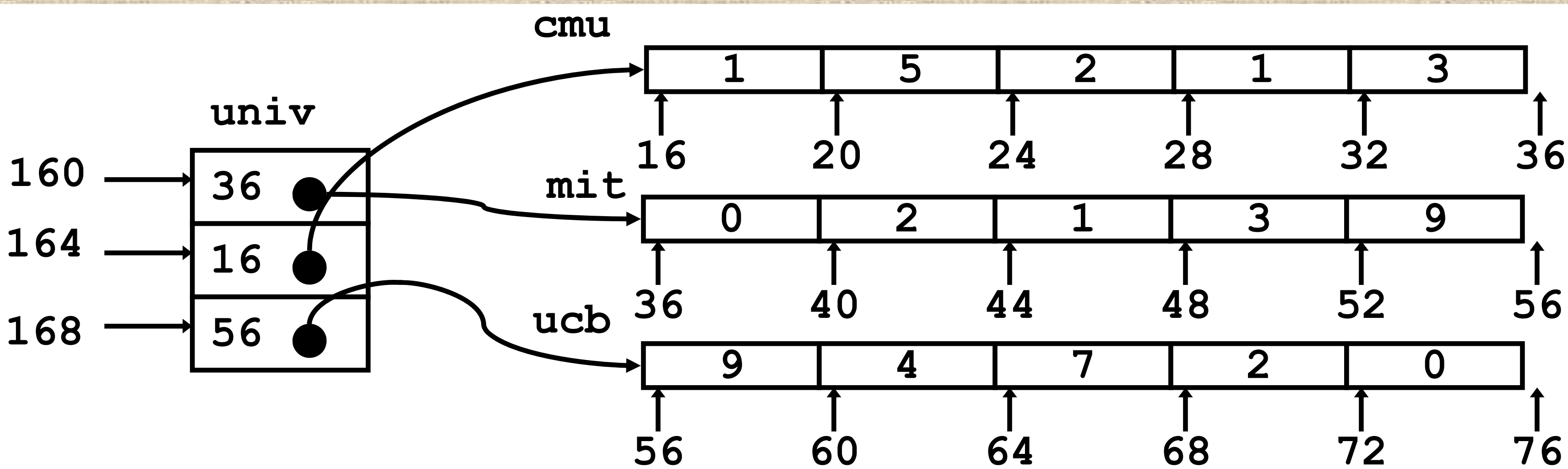


– Element at

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



Strange Referencing Examples

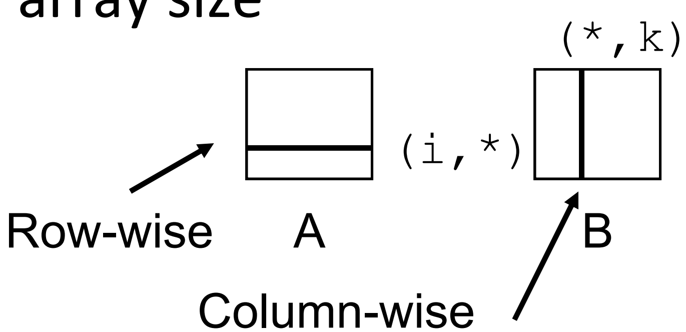


Reference	Address	Value	Guaranteed?
`univ[2][3]`	`56+4*3 = 68`	`2`	**Yes**
`univ[1][5]`	`16+4*5 = 36`	`0`	**No**
`univ[2][-1]`	`56+4*-1 = 52`	`9`	**No**
`univ[3][-1]`	`??`	`??`	**No**
`univ[1][12]`	`16+4*12 = 64`	`7`	**No**

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

Using Nested Arrays

- Strengths
  - C compiler handles doubly subscripted arrays
  - Generates very efficient code
    - Avoids multiply in index computation
- Limitation
  - Only works if have 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 arbitrary size
- Programming
  - Must do index computation explicitly
- Performance
  - Accessing single element costly
  - Must do multiplication

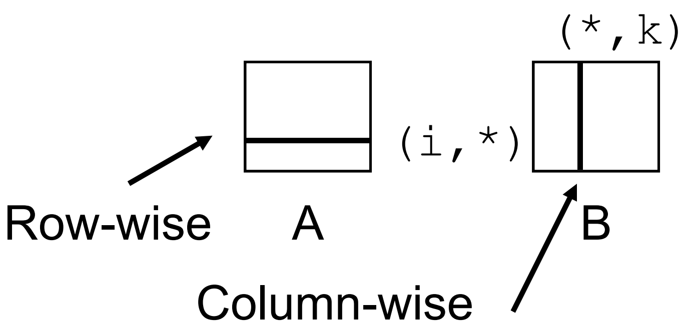
```
int * new_var_matrix(int n)
{
  return (int *)
    calloc(sizeof(int), n*n);
}
```

```
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
    - 2 for subscripts
    - 1 for data
  - Adds
    - 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 Mult.

- 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`
- Performance
  - 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;
}
```

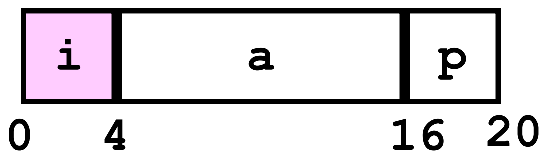
```
{
  int j;
  int result = 0;
  int iTn = i*n;
  int jTnPk = k;
  for (j = 0; j < n; j++) {
    result +=
      a[iTn+j] * b[jTnPk];
    jTnPk += n;
  }
  return result;
}
```

Structures

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

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

Memory Layout



- Accessing Structure Member

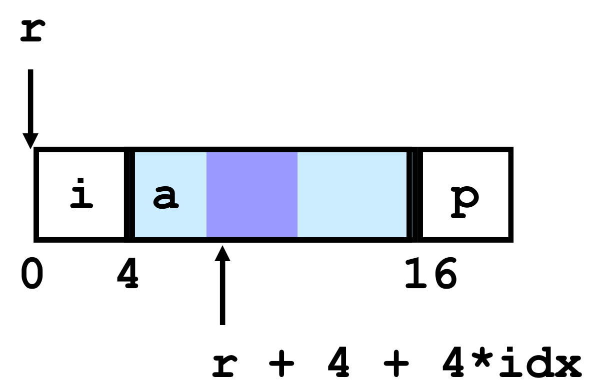
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

Assembly

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

Generating Pointer to Struct. Member

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



- Generating Pointer to Array Element
  - Offset of each structure member determined at compile time

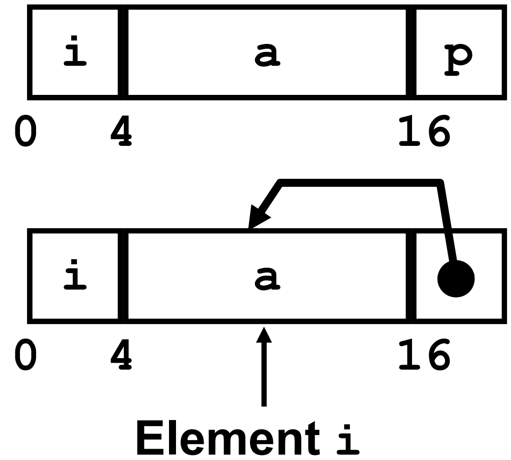
```
int *
find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

```
# %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];
}
```

```
 # %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!
- 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

Specific Cases of Alignment

- Size of Primitive Data Type:
  - 1 byte (e.g., `char`)
    - no restrictions on address
  - 2 bytes (e.g., `short`)
    - lowest 1 bit of address must be 0_2
  - 4 bytes (e.g., `int`, `float`, `char *`, etc.)
    - lowest 2 bits of address must be 00_2
  - 8 bytes (e.g., `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

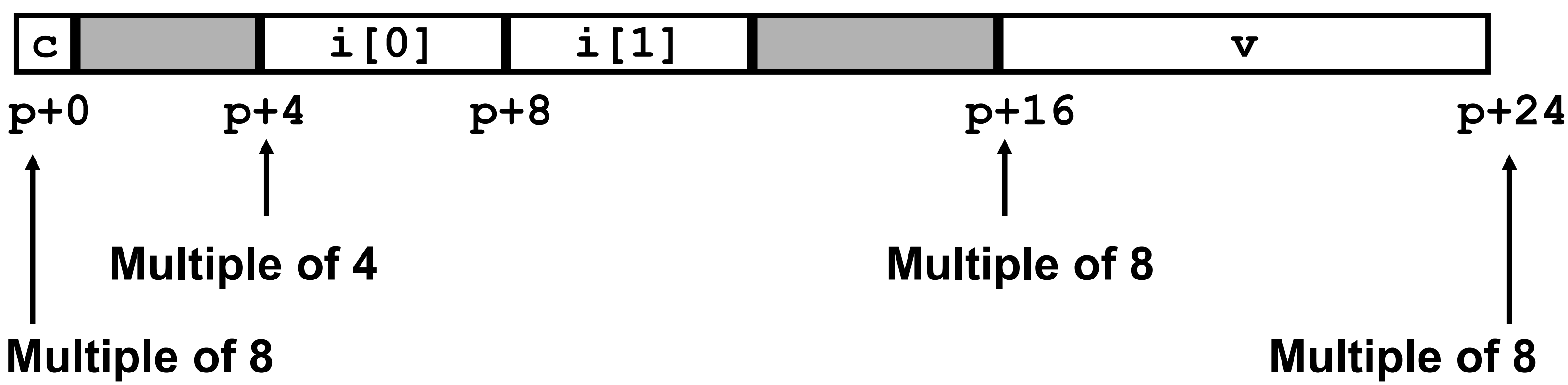
Specific Cases of Alignment (x86-64)

- 1 byte: `char, …`
  - no restrictions on address
- 2 bytes: `short`, …
  - lowest 1 bit of address must be 0_2
- 4 bytes: `int`, `float`, …
  - lowest 2 bits of address must be 00_2
- 8 bytes: `double`, `char *`, …
  - Windows & Linux:
    - lowest 3 bits of address must be 000_2
- 16 bytes: `long double`
  - Linux:
    - lowest 3 bits of address must be 000_2

Satisfying Alignment with Structures

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

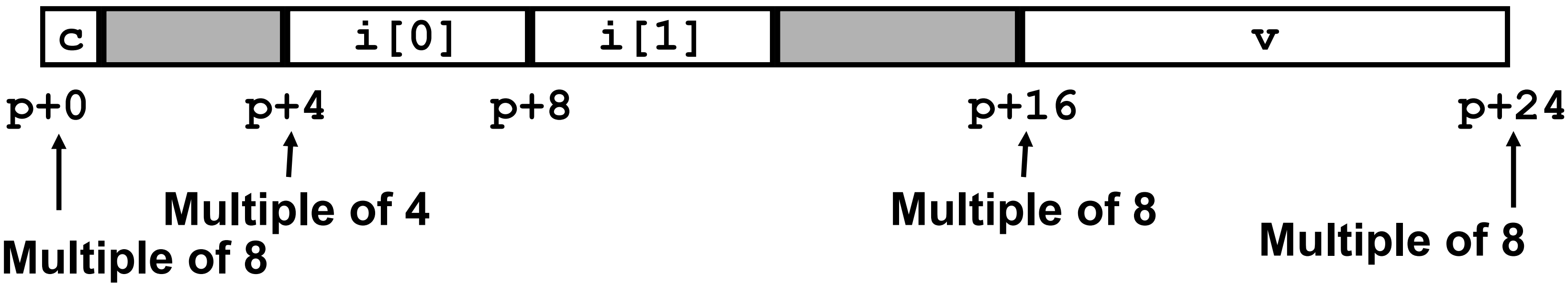
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



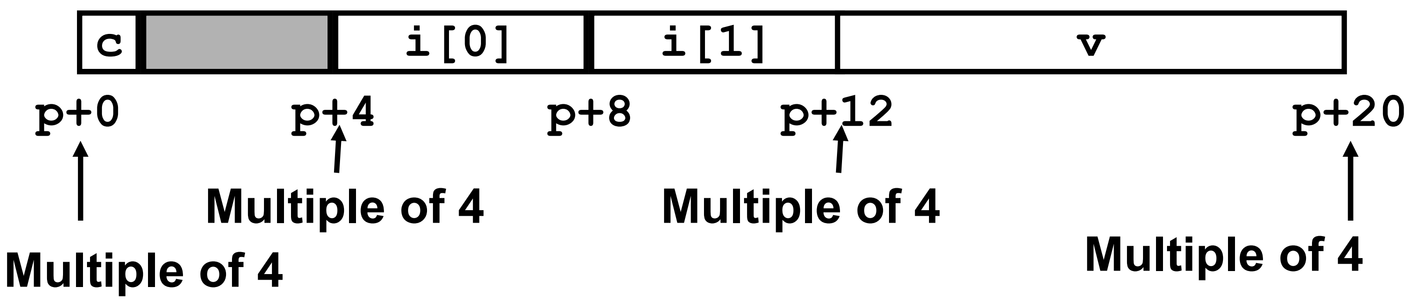
Different Alignment Conventions

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

- X86-64 Linux or IA32 Windows
  - K = 8, due to `double` element



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



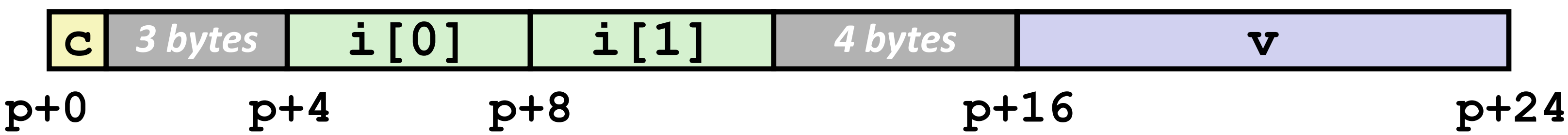
Saving Space

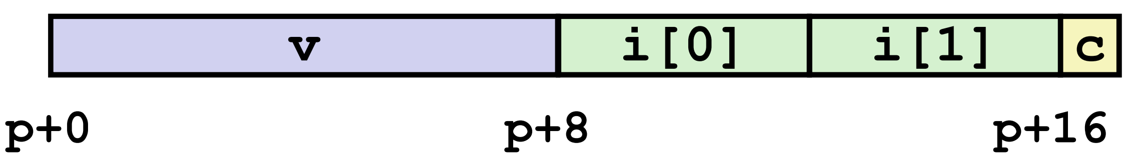
- Put large data types first

```
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)*

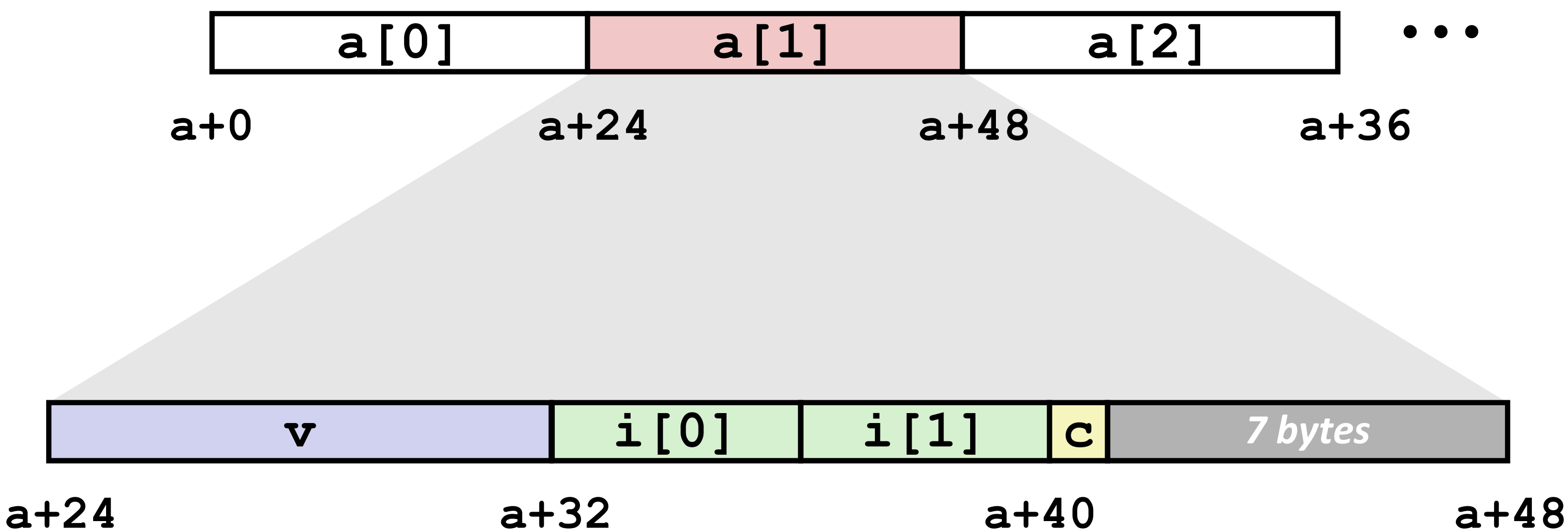




Arrays of Structures

- Satisfy alignment requirement for every element

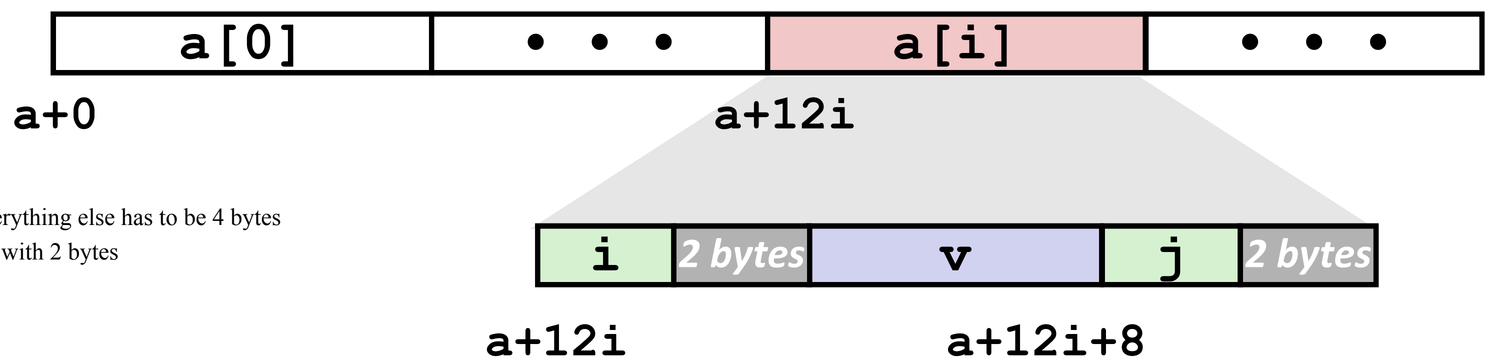
```
struct S2 {
    double v;
    int i[2];
    char c;
} a[10];
```



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];
```



If one thing is 4 bytes then everything else has to be 4 bytes as well, so that is why you fill with 2 bytes

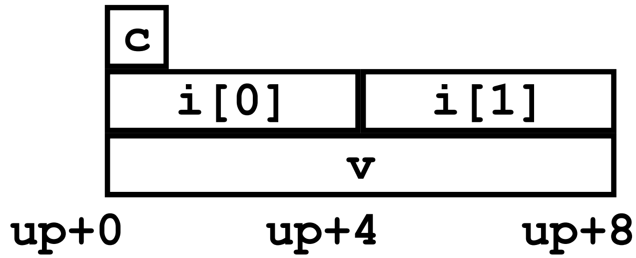
```
short get_j(int idx)
{
  return a[idx].j;
}
```

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

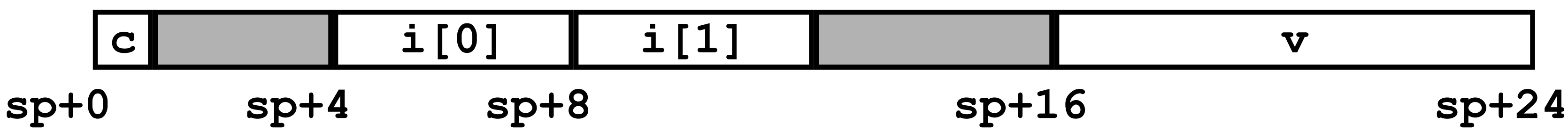
Union Allocation

- Principles
  - Overlay union elements
  - Allocate according to largest element
  - Can only use one field at a time

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

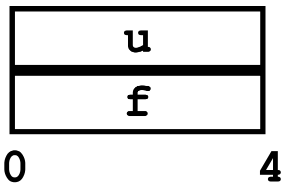



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



Using Union to Access Bit Patterns

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



- Get direct access to bit representation of float
- `bit2float` generates float with given bit pattern
  - NOT the same as `(float) u`
- `float2bit` generates bit pattern from float
  - NOT the same as `(unsigned) f`

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

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

Byte Ordering Revisited

- Idea
  - Short/long/quad words stored in memory as 2/4/8 consecutive bytes
  - Which is most (least) significant?
  - Can cause problems when exchanging binary data between machines
- Big Endian
  - Most significant byte has lowest address
  - PowerPC, Sparc
- Little Endian
  - Least significant byte has lowest address
  - Intel x86, Alpha

Byte Ordering Example

```
union {
  unsigned char c[8];
  unsigned short s[4];
  unsigned int i[2];
  unsigned long l[1];
} dw;
```

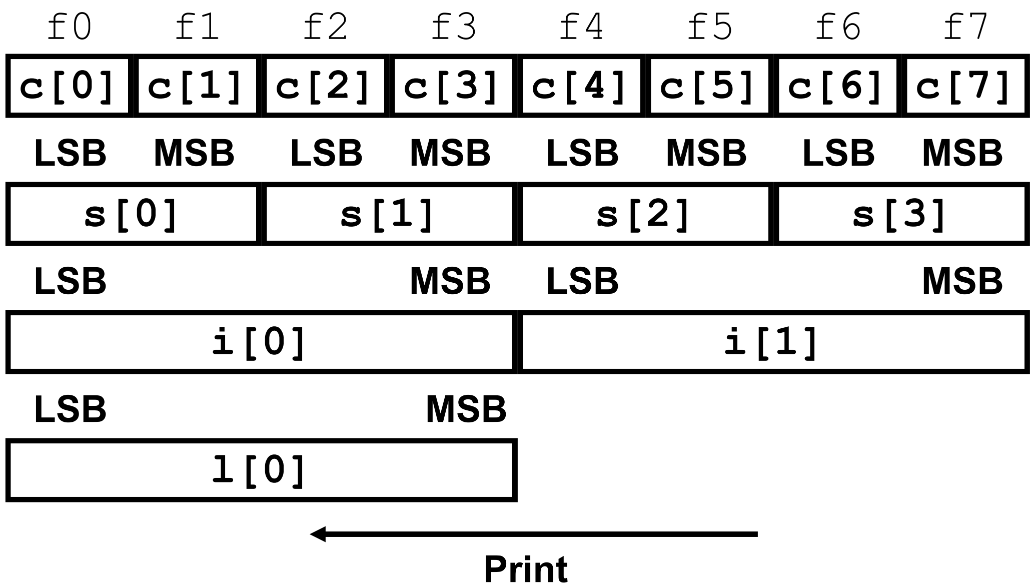
<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="4">l[0]</td><td colspan="4"></td></tr>
</table>

Byte Ordering Example (Cont).

```
int j;
for (j = 0; j < 8; j++)
dw.c[j] = 0xf0 + j;

printf("Characters 0-7 ==
[0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x,0x%x]\n",
    dw.c[0], dw.c[1], dw.c[2], dw.c[3],
    dw.c[4], dw.c[5], dw.c[6], dw.c[7]);

printf("Shorts 0-3 ==
[0x%x,0x%x,0x%x,0x%x]\n",
    dw.s[0], dw.s[1], dw.s[2], dw.s[3]);

printf("Ints 0-1 == [0x%x,0x%x]\n",
    dw.i[0], dw.i[1]);

printf("Long 0 == [0x%lx]\n",
    dw.l[0]);
```

Byte Ordering on IA32

Little Endian

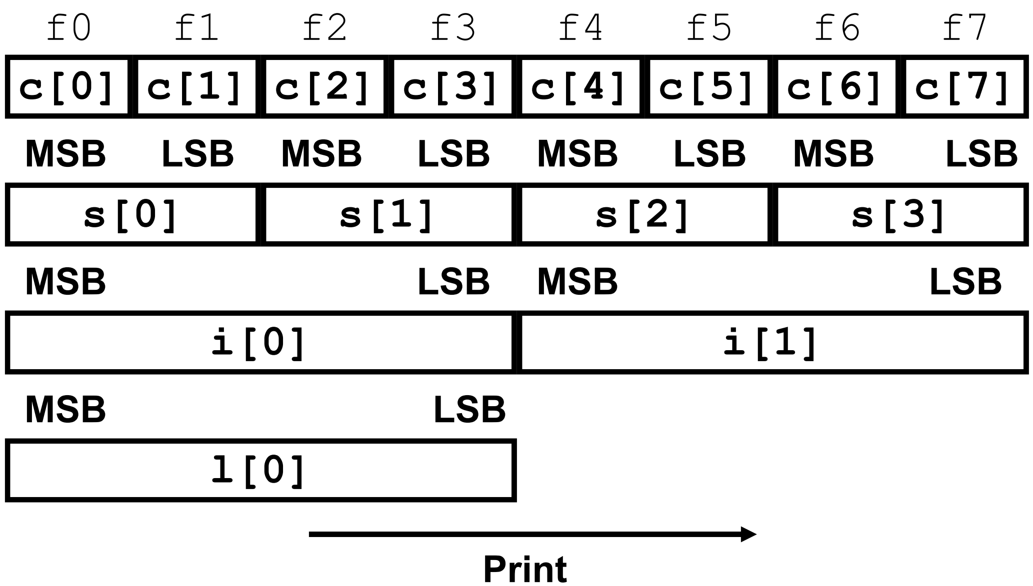


Output on IA32:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf3f2f1f0]
```

Byte Ordering on Sun

Big Endian

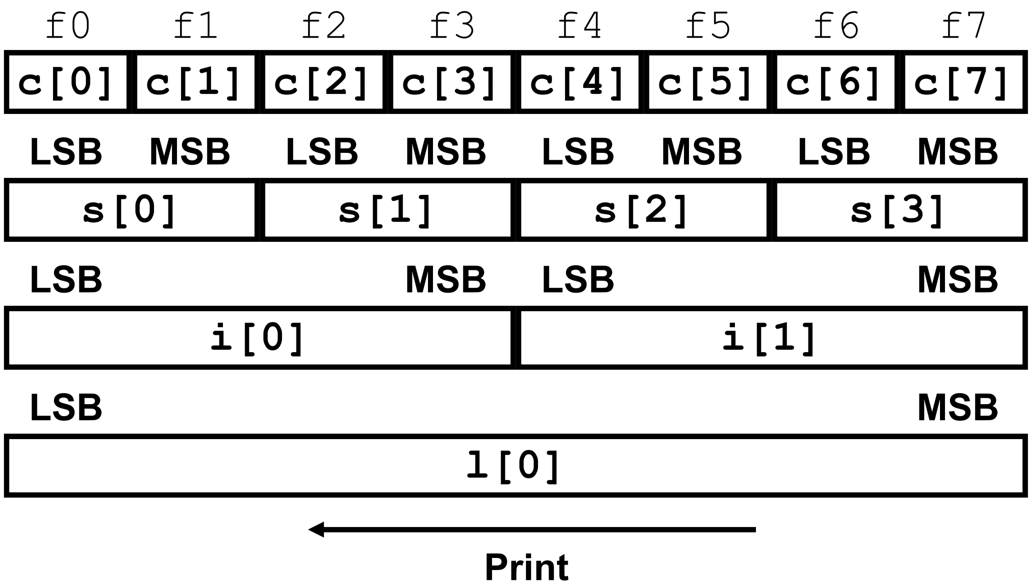


Output on Sun:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf0f1,0xf2f3,0xf4f5,0xf6f7]
Ints       0-1 == [0xf0f1f2f3,0xf4f5f6f7]
Long       0   == [0xf0f1f2f3]
```

Byte Ordering on x86-64

Little Endian



Output on x86-64:

```
Characters 0-7 == [0xf0,0xf1,0xf2,0xf3,0xf4,0xf5,0xf6,0xf7]
Shorts     0-3 == [0xf1f0,0xf3f2,0xf5f4,0xf7f6]
Ints       0-1 == [0xf3f2f1f0,0xf7f6f5f4]
Long       0   == [0xf7f6f5f4f3f2f1f0]
```

Summary

- Arrays in C
  - Contiguous allocation of memory
  - Pointer to first element
  - No bounds checking
- Compiler Optimizations
  - Compiler often turns array code into pointer code (zd2int)
  - Uses addressing modes to scale array indices
  - Lots of tricks to improve array indexing in loops
- Structures
  - Allocate bytes in order declared
  - Pad in middle and at end to satisfy alignment
- Unions
  - Overlay declarations
  - Way to circumvent type system