Machine-Level Prog. IV - Structured Data

Today

- Arrays
- Structures
- Unions

Next time

- Buffers

Chris Riesbeck, Fall 2011

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`

- 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	`long double`

Array allocation

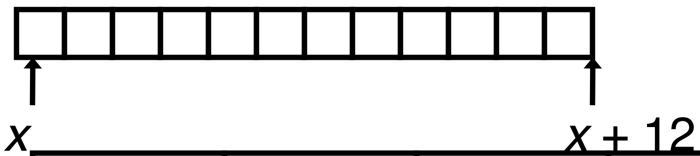
- Basic principle

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

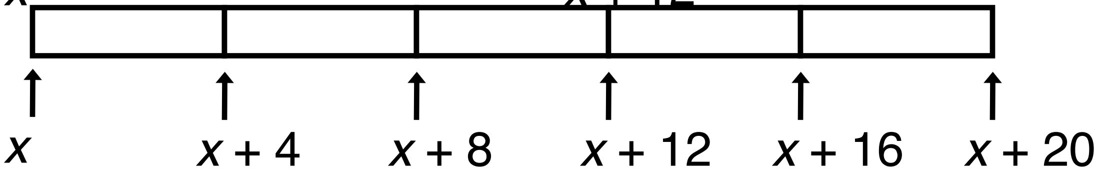
  – Array of data type ***T*** and length ***L***

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

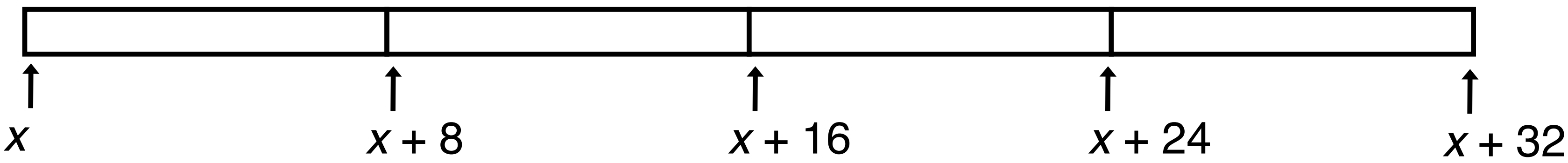
```
char string[12];
```



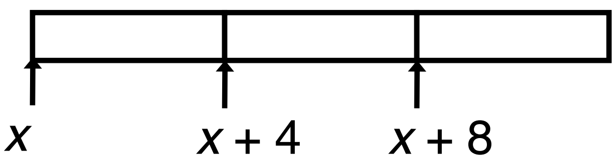

```
int val[5];
```




```
double a[4];
```




```
char *p[3];
```



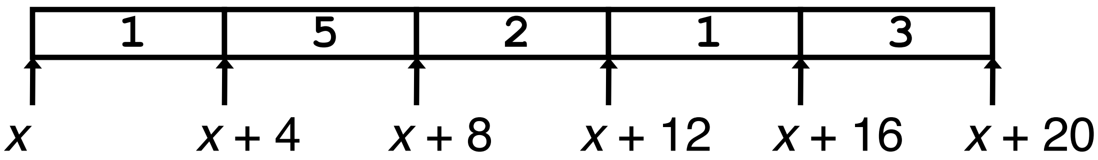
Array access

- Basic principle

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

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

`int val[5];`

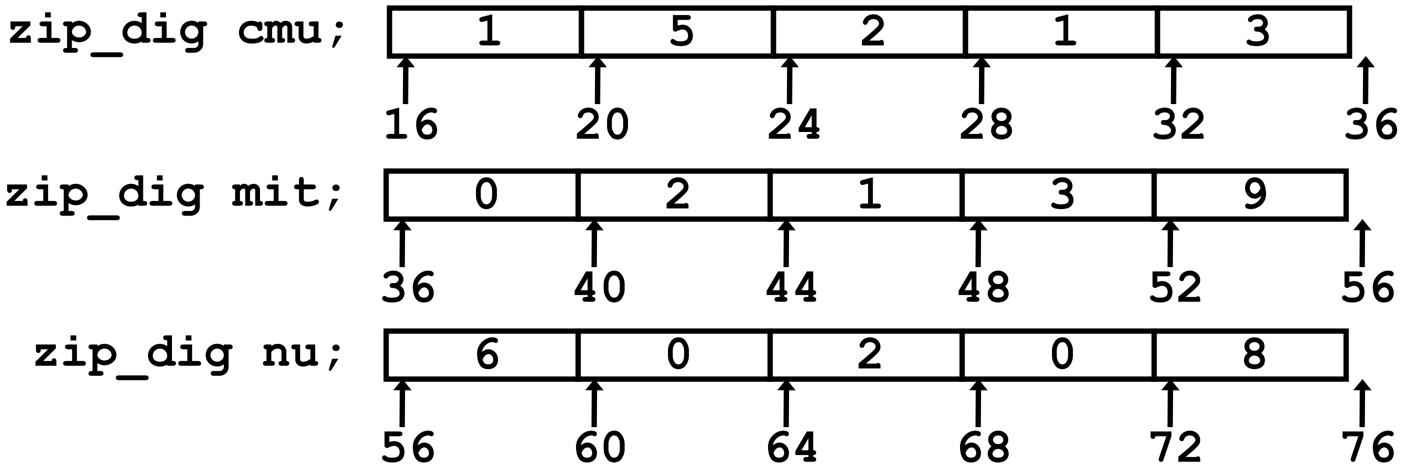


- Reference Type Value

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 nu = { 6, 0, 2, 0, 8 };
```



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

Array accessing example

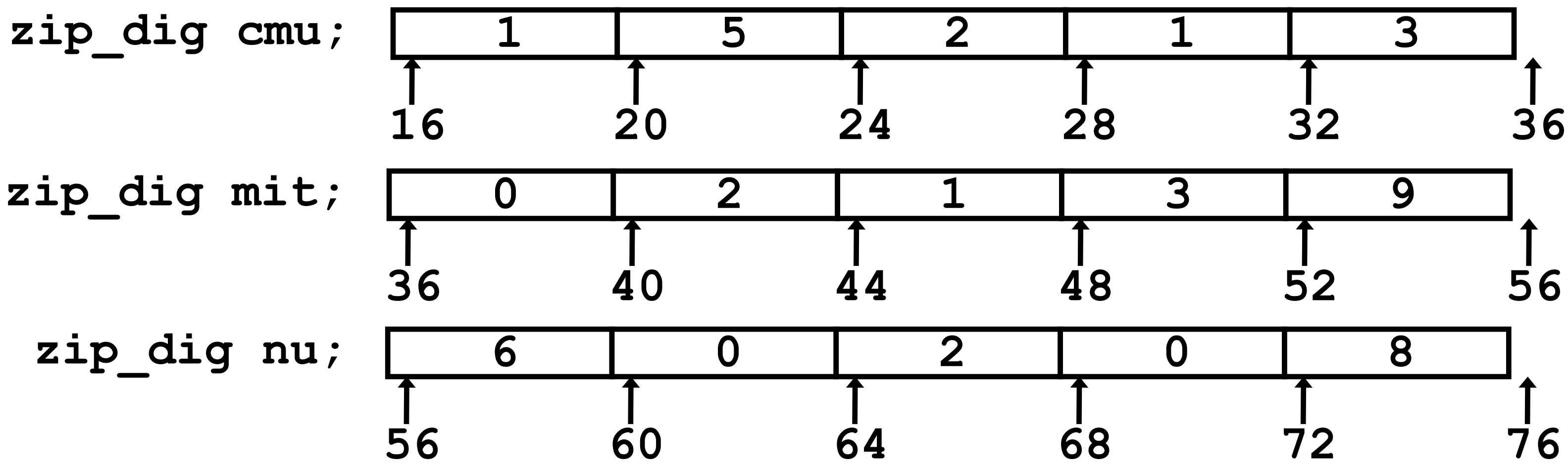
- Computation
  - Register `%edx` contains starting address of array
  - Register `%eax` contains array index
  - Desired digit at `4*%eax + %edx`
  - Use memory reference `(%edx,%eax,4)`

```
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



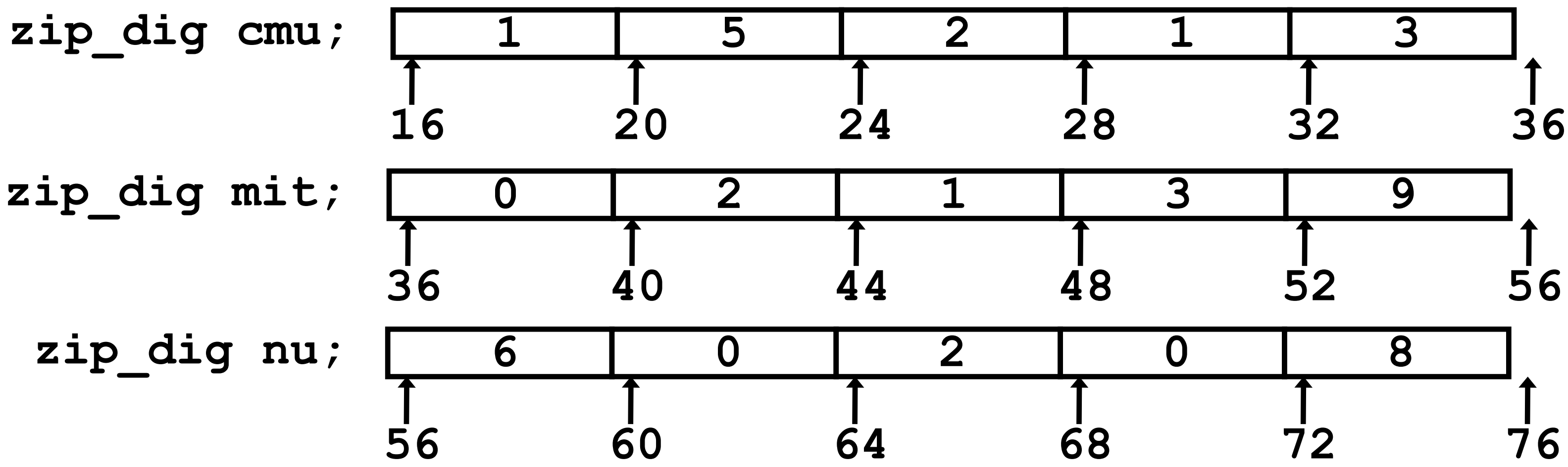
- Code does not do any bounds checking!

Reference	Address	Value	Guaranteed?
`mit[3]`	`36 + 4* 3 = 48`	`3`	
`mit[5]`	`36 + 4* 5 = 56`	`6`	
`mit[-1]`	`36 + 4*-1 = 32`	`3`	
`cmu[15]`	`16 + 4*15 = 76`	`??`	

– Out of range behavior implementation-dependent

- No guaranteed relative allocation of different arrays

Referencing examples



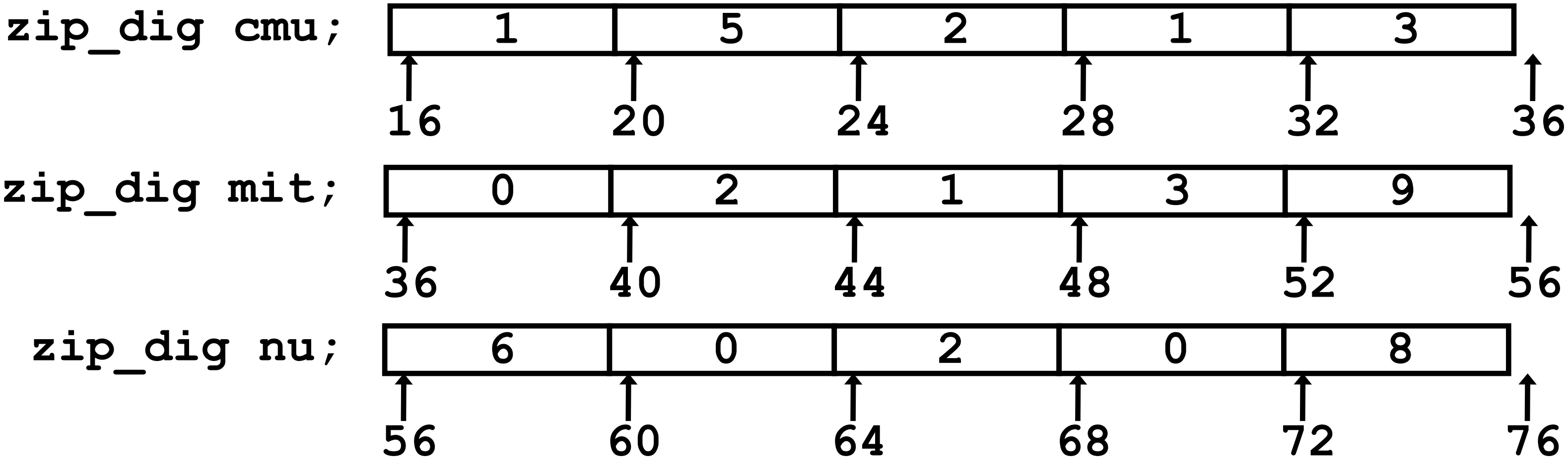
- Code does not do any bounds checking!

Reference	Address	Value	Guaranteed?
`mit[3]`	`36 + 4* 3 = 48`	`3`	**Yes**
`mit[5]`	`36 + 4* 5 = 56`	`6`	
`mit[-1]`	`36 + 4*-1 = 32`	`3`	
`cmu[15]`	`16 + 4*15 = 76`	`??`	

– Out of range behavior implementation-dependent

- No guaranteed relative allocation of different arrays

Referencing examples

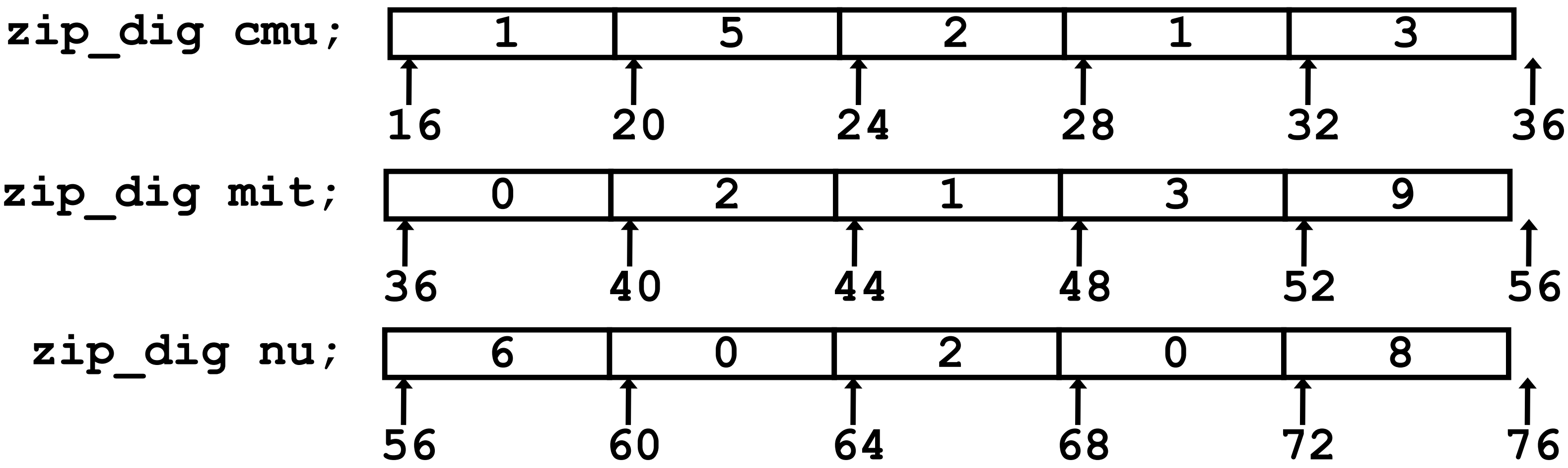


- Code does not do any bounds checking!

Reference	Address	Value	Guaranteed?
`mit[3]`	`36 + 4* 3 = 48`	`3`	**Yes**
`mit[5]`	`36 + 4* 5 = 56`	`6`	**No**
`mit[-1]`	`36 + 4*-1 = 32`	`3`	
`cmu[15]`	`16 + 4*15 = 76`	`??`	

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

Referencing examples



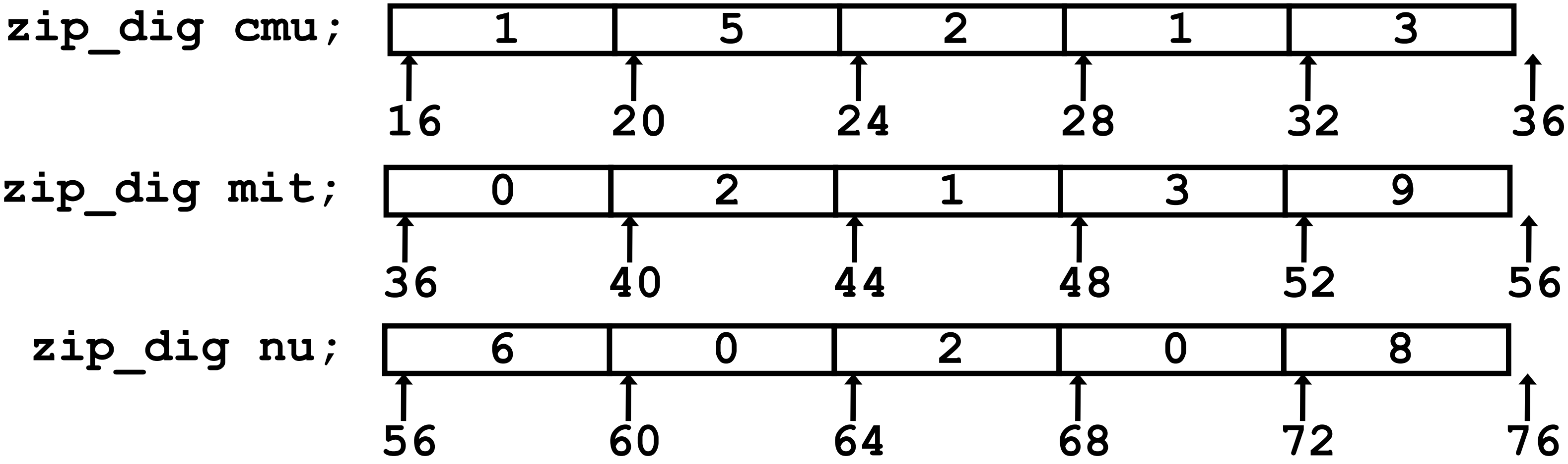
- Code does not do any bounds checking!

Reference	Address	Value	Guaranteed?
`mit[3]`	`36 + 4* 3 = 48`	`3`	**Yes**
`mit[5]`	`36 + 4* 5 = 56`	`6`	**No**
`mit[-1]`	`36 + 4*-1 = 32`	`3`	**No**
`cmu[15]`	`16 + 4*15 = 76`	`??`	

– Out of range behavior implementation-dependent

- No guaranteed relative allocation of different arrays

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`	`6`	**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

  Computes the integer represented by an array of 5 decimal digits.

```
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` and uses pointer arithmetic
  - Computes address of final element and uses that for test
  - 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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

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
```

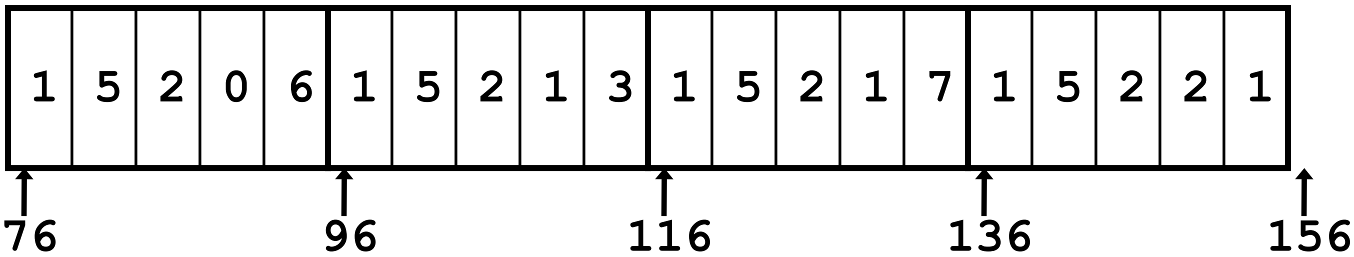
Checkpoint

Checkpoint

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 }};
```

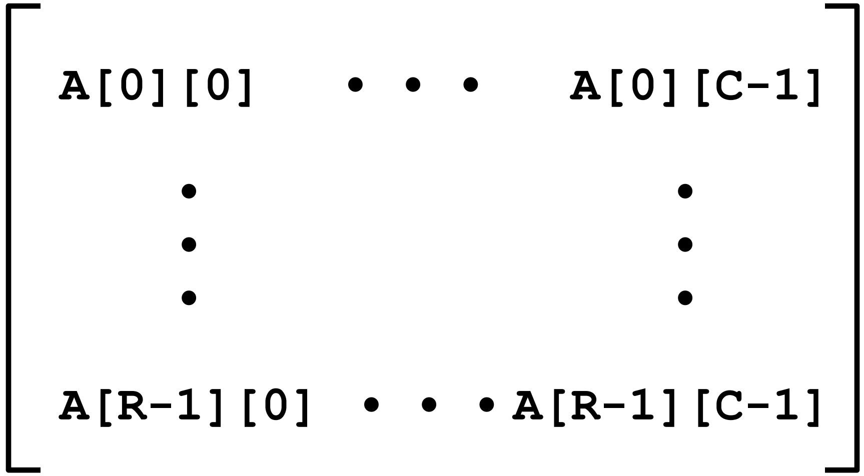
```
zip_dig
pgh[4];
```



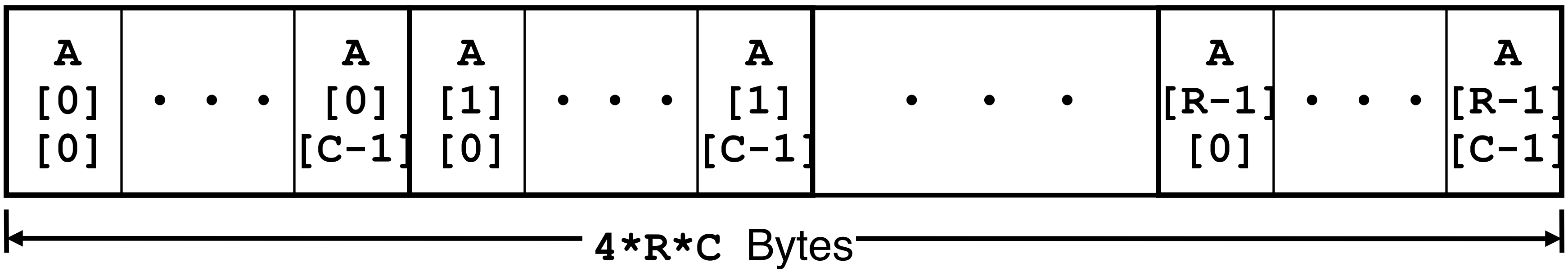
- Declaration "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  - Variable `pgh` denotes array of 4 elements
    - Allocated contiguously
  - Each element is an array of 5 `int`'s

Nested array allocation

- Declaration
  T `A[`*R*`][`*C*`];`
  – 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



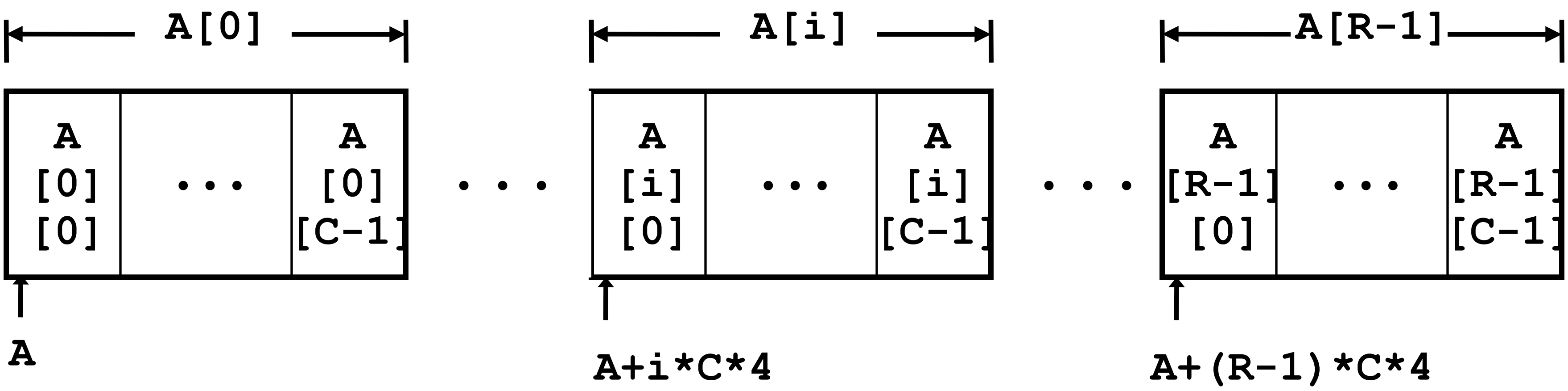
`int A[R][C];`



Nested array row access

- Row vectors
  - `A[i]` is array of *C* elements
  - Each element of type *T*
  - Starting address `A +` *i * C * K* *(sizeof(T) = K)*

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



Nested array row access code

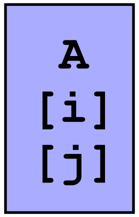
```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

- 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)`

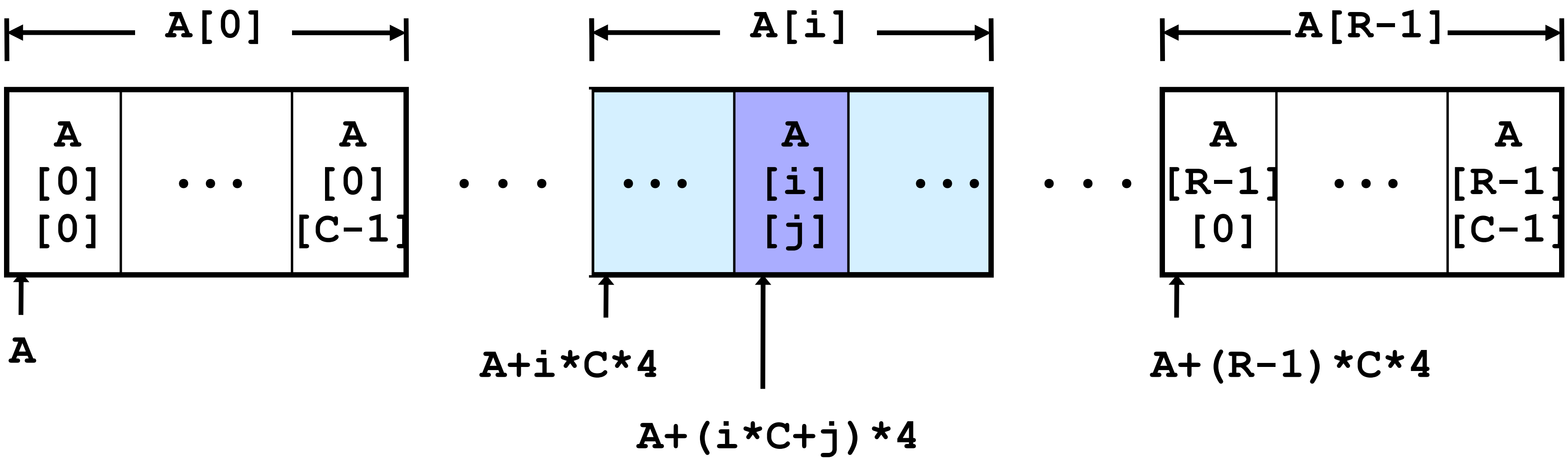
```
# %eax = index
 leal (%eax,%eax,4),%eax # 5 * index
 leal pgh(,%eax,4),%eax  # pgh + (20 * index)
```

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 + 4*(5*index + dig)=`

    `pgh + 20*index + 4*dig`

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

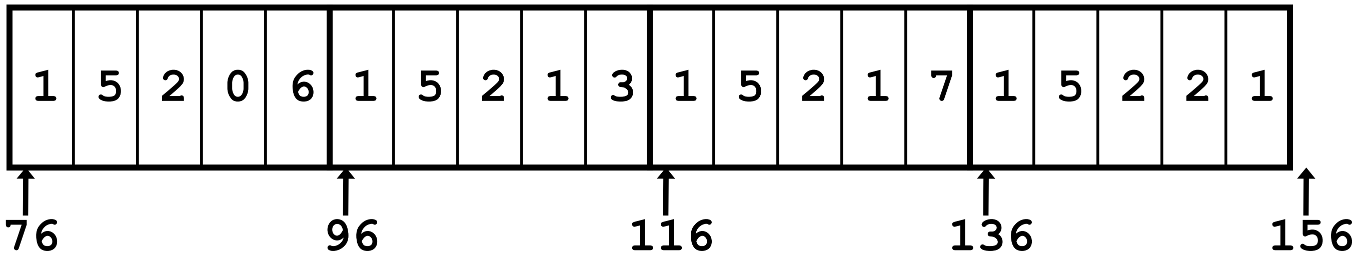
- Code
  - Computes address

    `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];
```



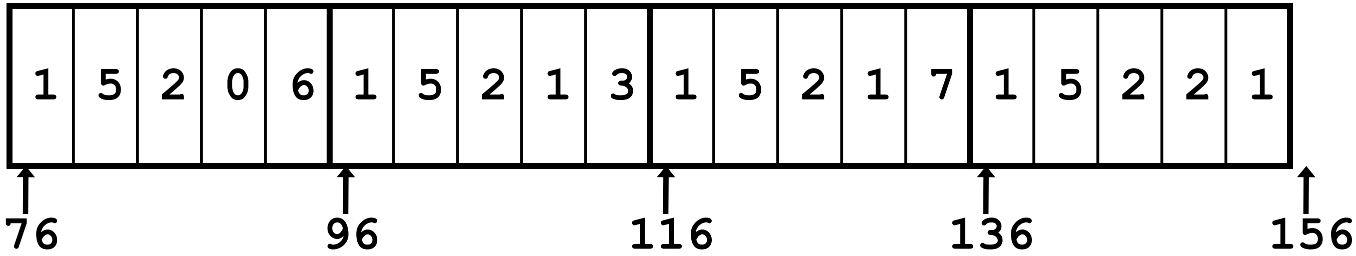
Reference	Address	Value	Guaranteed?
`pgh[3][3]`	`76+20*3+4*3 = 148`	`2`	
`pgh[2][5]`	`76+20*2+4*5 = 136`	`1`	
`pgh[2][-1]`	`76+20*2+4*-1 = 112`	`3`	
`pgh[4][-1]`	`76+20*4+4*-1 = 152`	`1`	
`pgh[0][19]`	`76+20*0+4*19 = 152`	`1`	
`pgh[0][-1]`	`76+20*0+4*-1 = 72`	`??`	

– Code does not do any bounds checking

– Ordering of elements within array guaranteed

Strange referencing examples

```
zip_dig
pgh[4];
```



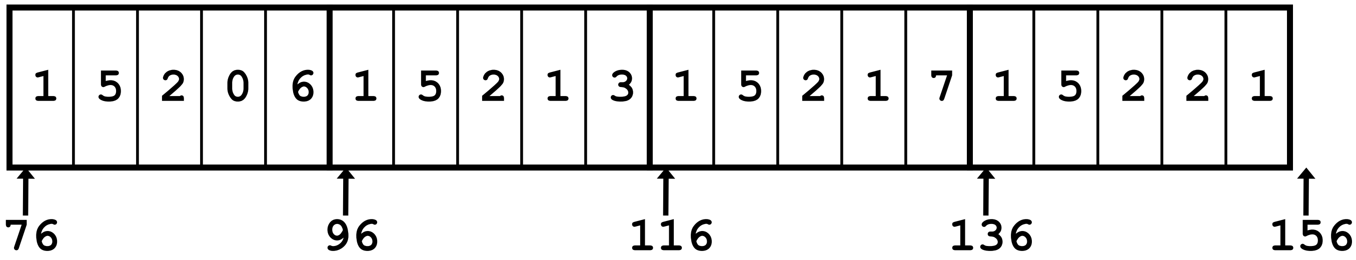
Reference	Address	Value Guaranteed?	
`pgh[3][3]`	`76+20*3+4*3 = 148`	`2`	**Yes**
`pgh[2][5]`	`76+20*2+4*5 = 136`	`1`	
`pgh[2][-1]`	`76+20*2+4*-1 = 112`	`3`	
`pgh[4][-1]`	`76+20*4+4*-1 = 152`	`1`	
`pgh[0][19]`	`76+20*0+4*19 = 152`	`1`	
`pgh[0][-1]`	`76+20*0+4*-1 = 72`	`??`	

– Code does not do any bounds checking

– Ordering of elements within array guaranteed

Strange referencing examples

```
zip_dig
pgh[4];
```



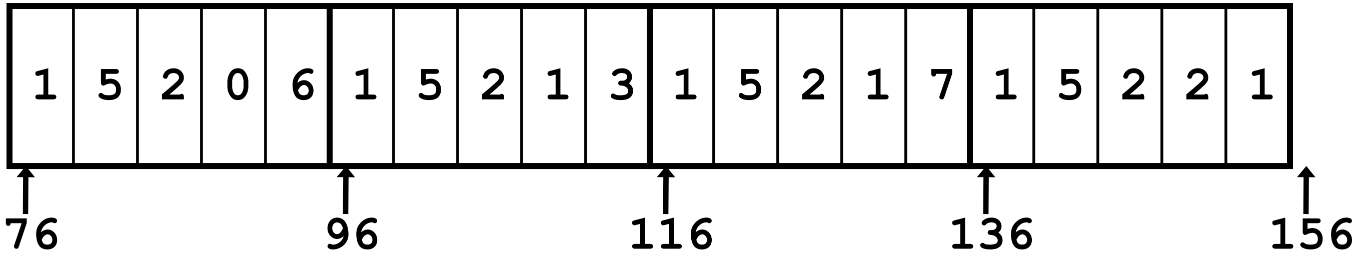
Reference	Address	Value	Guaranteed?
pgh[3][3]	76+20*3+4*3 = 148	2	Yes
pgh[2][5]	76+20*2+4*5 = 136	1	Yes
pgh[2][-1]	76+20*2+4*-1 = 112	3	
pgh[4][-1]	76+20*4+4*-1 = 152	1	
pgh[0][19]	76+20*0+4*19 = 152	1	
pgh[0][-1]	76+20*0+4*-1 = 72	??	

– Code does not do any bounds checking

– Ordering of elements within array guaranteed

Strange referencing examples

```
zip_dig
pgh[4];
```



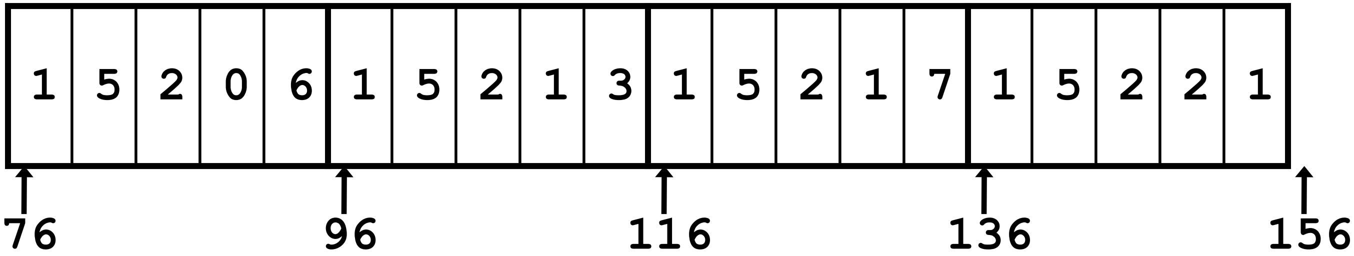
Reference	Address		Value	Guaranteed?
`pgh[3][3]`	`76+20*3+4*3`	`= 148`	`2`	**Yes**
`pgh[2][5]`	`76+20*2+4*5`	`= 136`	`1`	**Yes**
`pgh[2][-1]`	`76+20*2+4*-1`	`= 112`	`3`	**Yes**
`pgh[4][-1]`	`76+20*4+4*-1`	`= 152`	`1`	
`pgh[0][19]`	`76+20*0+4*19`	`= 152`	`1`	
`pgh[0][-1]`	`76+20*0+4*-1`	`= 72`	`??`	

– Code does not do any bounds checking

– Ordering of elements within array guaranteed

Strange referencing examples

```
zip_dig
pgh[4];
```



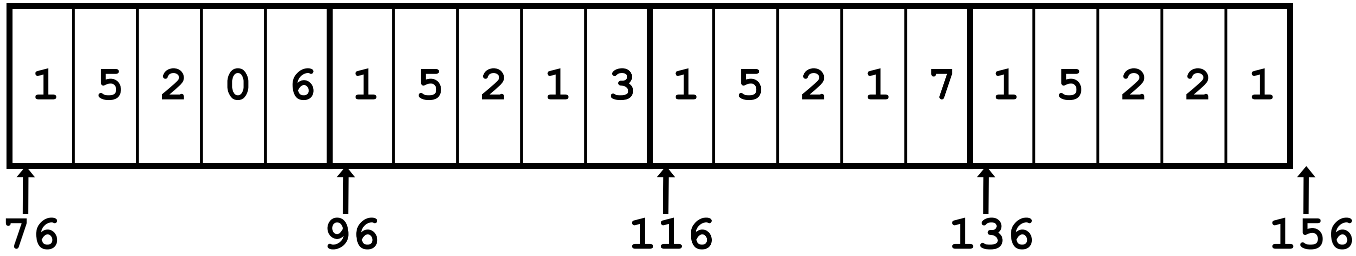
Reference	Address		Value	Guaranteed?
`pgh[3][3]`	`76+20*3+4*3`	`= 148`	`2`	**Yes**
`pgh[2][5]`	`76+20*2+4*5`	`= 136`	`1`	**Yes**
`pgh[2][-1]`	`76+20*2+4*-1`	`= 112`	`3`	**Yes**
`pgh[4][-1]`	`76+20*4+4*-1`	`= 152`	`1`	**Yes**
`pgh[0][19]`	`76+20*0+4*19`	`= 152`	`1`	
`pgh[0][-1]`	`76+20*0+4*-1`	`= 72`	`??`	

– Code does not do any bounds checking

– Ordering of elements within array guaranteed

Strange referencing examples

```
zip_dig
pgh[4];
```



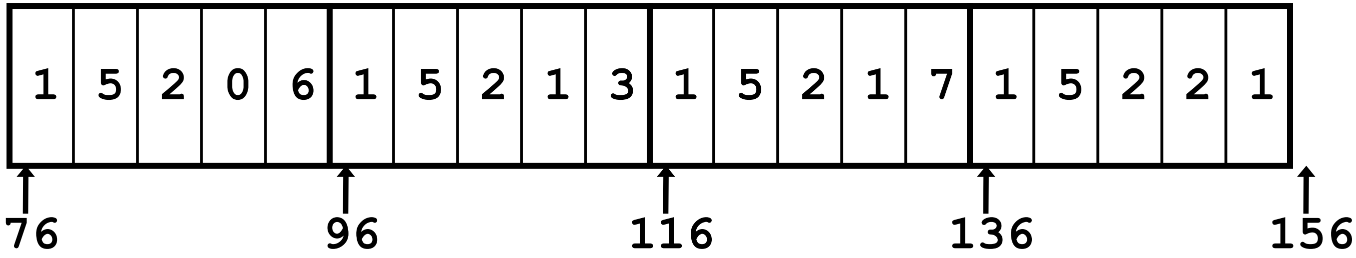
Reference	Address	Value	Guaranteed?
pgh[3][3]	76+20*3+4*3 = 148	2	Yes
pgh[2][5]	76+20*2+4*5 = 136	1	Yes
pgh[2][-1]	76+20*2+4*-1 = 112	3	Yes
pgh[4][-1]	76+20*4+4*-1 = 152	1	Yes
pgh[0][19]	76+20*0+4*19 = 152	1	Yes
pgh[0][-1]	76+20*0+4*-1 = 72	??	

– Code does not do any bounds checking

– Ordering of elements within array guaranteed

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

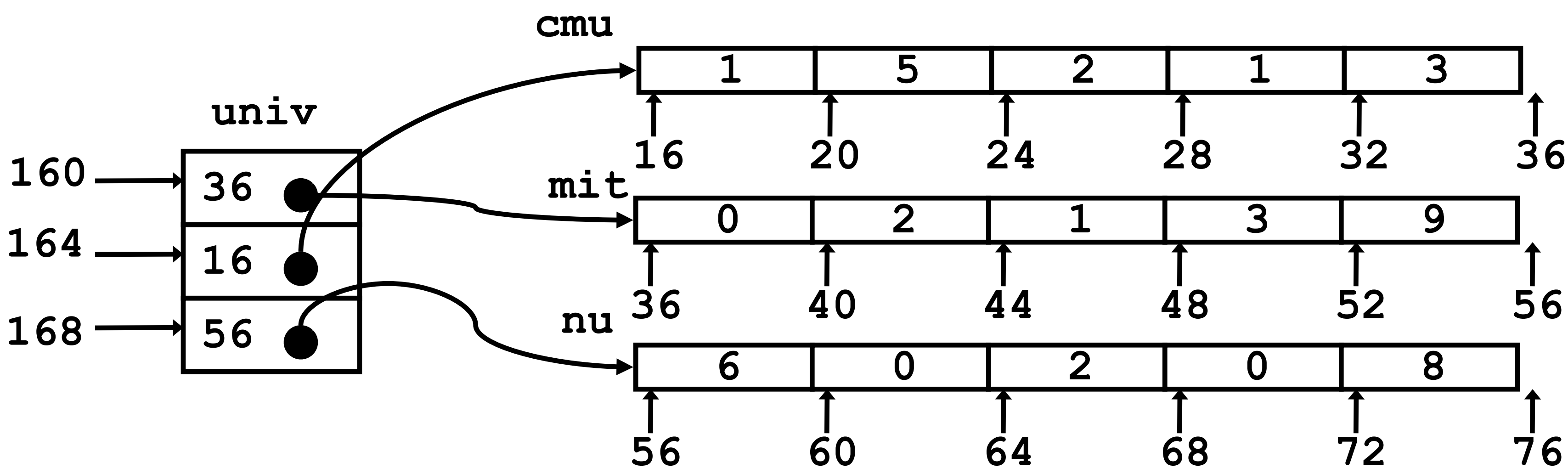
– Ordering of elements within array guaranteed

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 nu = { 6, 0, 2, 0, 8 };
```

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



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

- Nested Array

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

- Element at

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

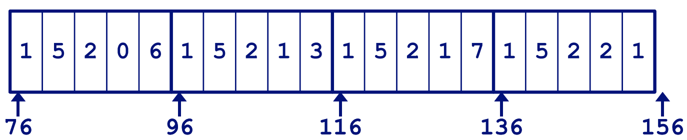
Different address computation

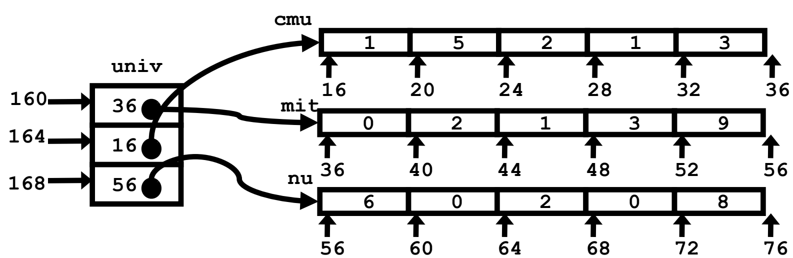
- Multi-Level Array

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

- Element at

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



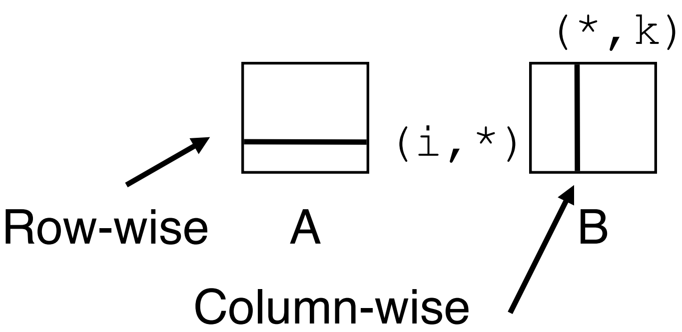


Using nested arrays

- 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);
}
```

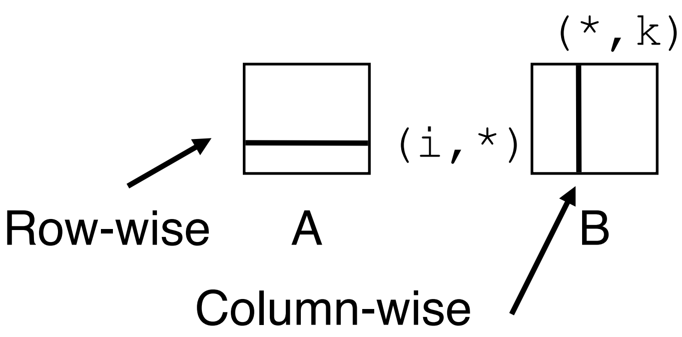
```
int var_ele (int *a, int i,
             int j, int n)
{
  return a[i*n+j];
}
```

```
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
    - 4 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;
}
```



Wednesday, October 19, 2011

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;
}
```

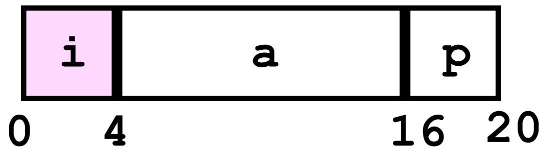
```
{
  int j;
  int result = 0;
  int iTn = i*n;
  int jTnPk = k;
  for (j = 0; j < n; j++) {
    result +=
      a[iTn+j] * b[jTnPk];
    jTnPk += n;
  }
  return result;
}
```

Structures

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

```
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;
}
```

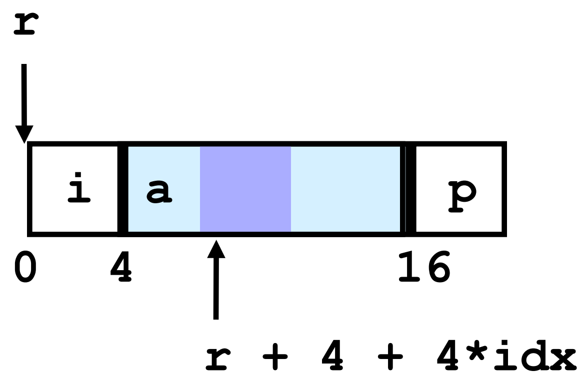
Assembly

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


Wednesday, October 19, 2011

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

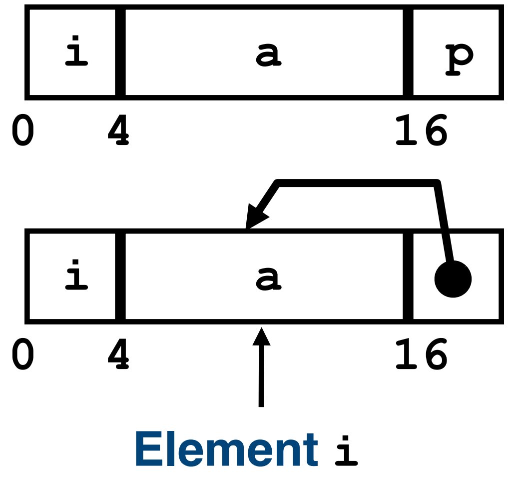
```
int *
find_a
 (struct rec *r, int idx)
{
  return &r->a[idx];
}
```

```
# %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
```

Checkpoint

Checkpoint

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 Linux and Windows!
- Motivation for aligning data
  - Memory accessed by (aligned) double or quad-words
    - 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

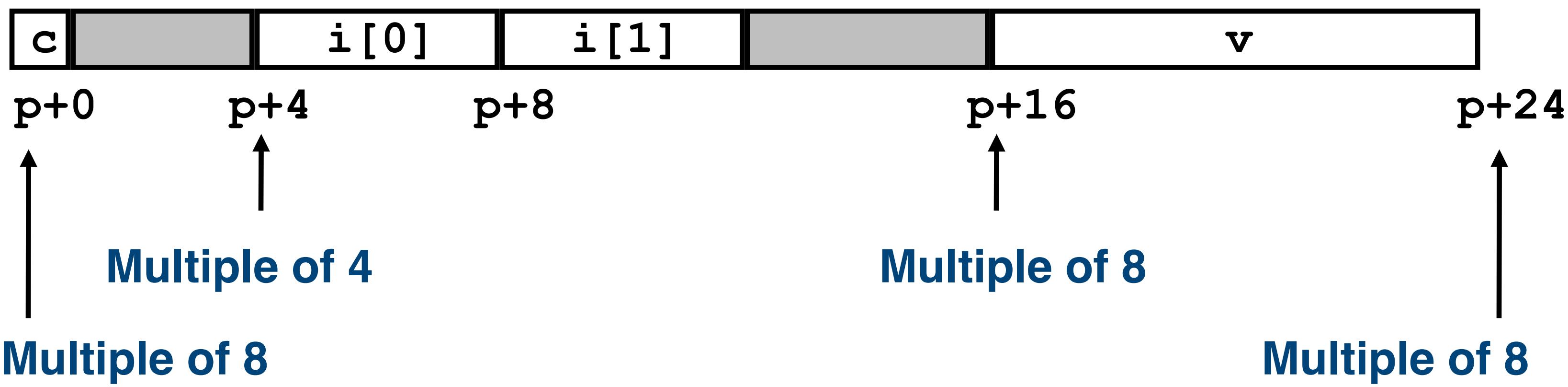
Specific cases of alignment

- 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`) [only 10 bytes needed]
    - Linux and Windows:
      - lowest 2 bits of address must be 00_2
      - i.e., treated the same as a 4-byte primitive data type

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):
  - K = 8, due to `double` element

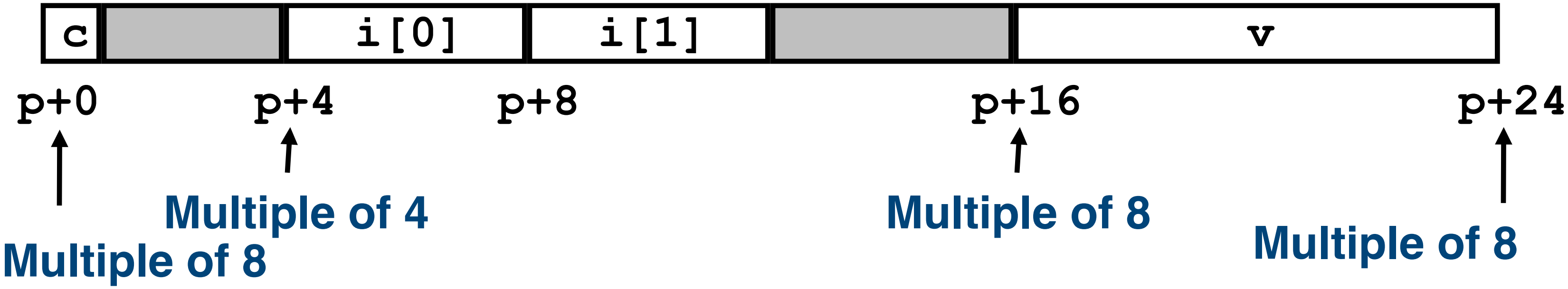
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



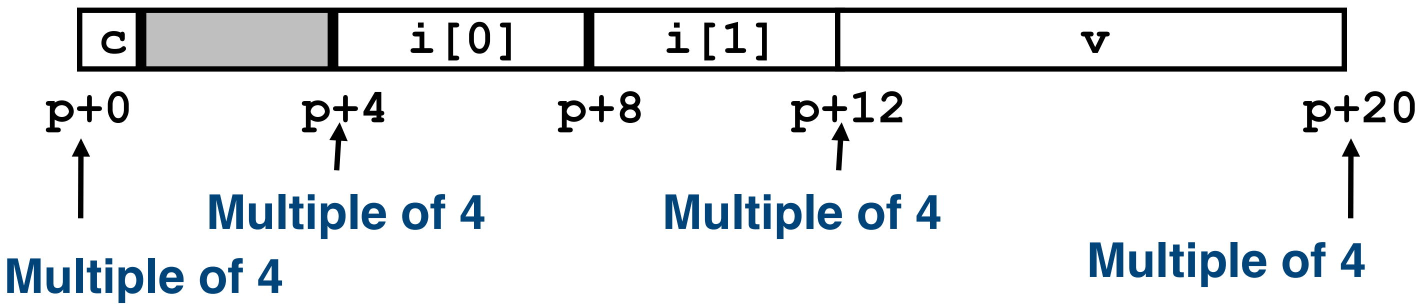
Linux vs. Windows

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

- Windows (including Cygwin):
  - K = 8, due to `double` element



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



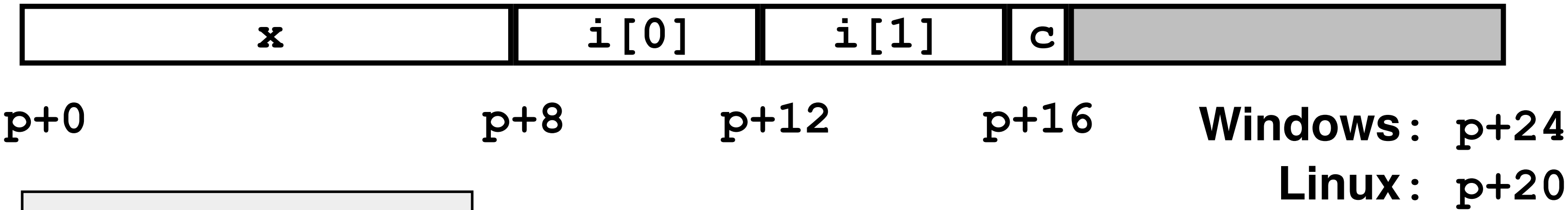
Overall alignment requirement

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

`p` **must be multiple of:**

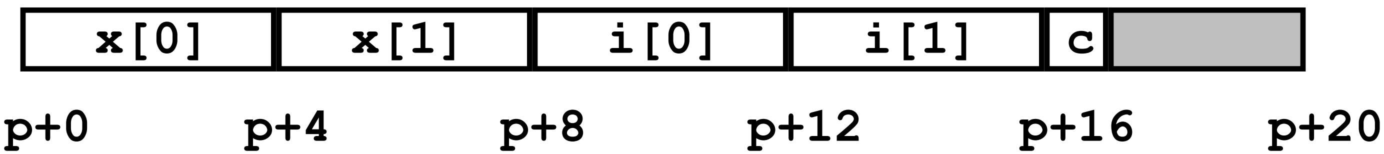
8 for Windows

4 for Linux




```
struct S3 {
  float x[2];
  int i[2];
  char c;
} *p;
```

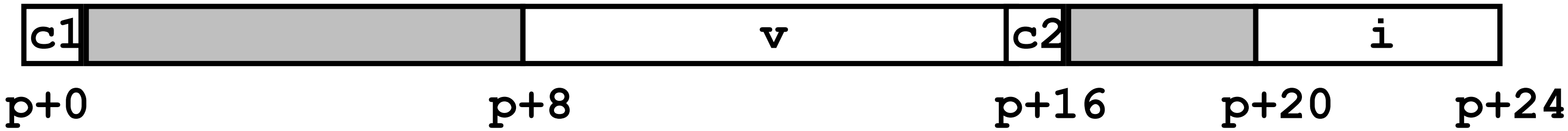
`p` **must be multiple of 4 (in either OS)**



Ordering elements within structure

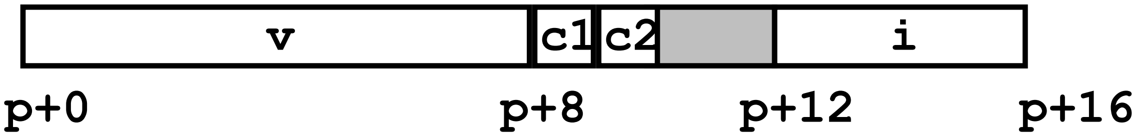
```
struct S4 {
  char c1;
  double v;
  char c2;
  int i;
} *p;
```

10 bytes wasted space in Windows




```
struct S5 {
  double v;
  char c1;
  char c2;
  int i;
} *p;
```

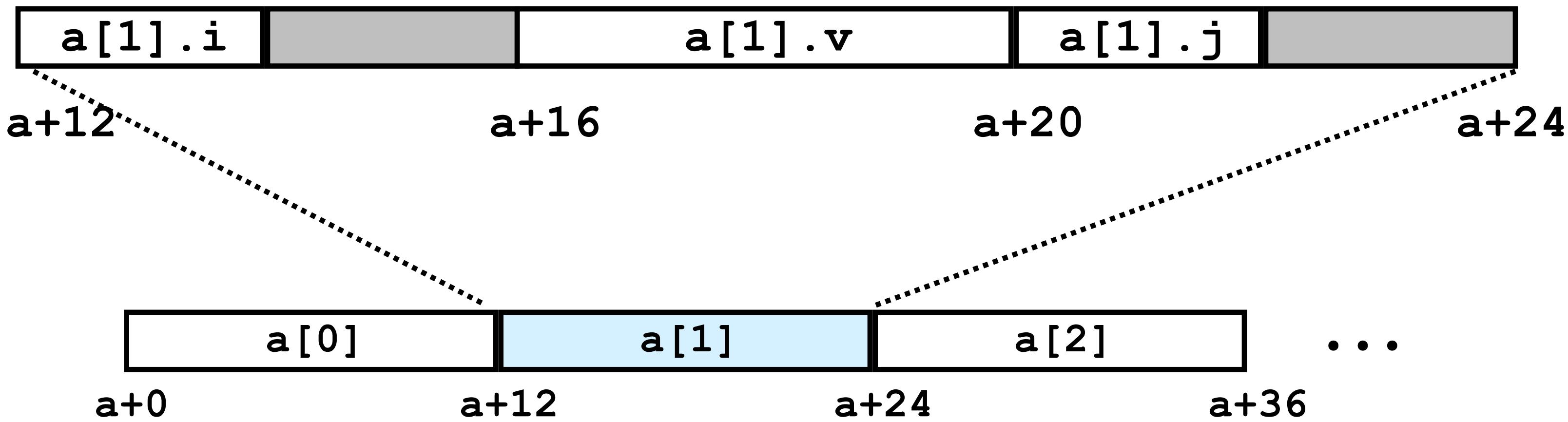
2 bytes wasted space



Arrays of structures

- Principle
  - Allocated by repeating allocation for array type
  - In general, may nest arrays & structures to arbitrary depth

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```



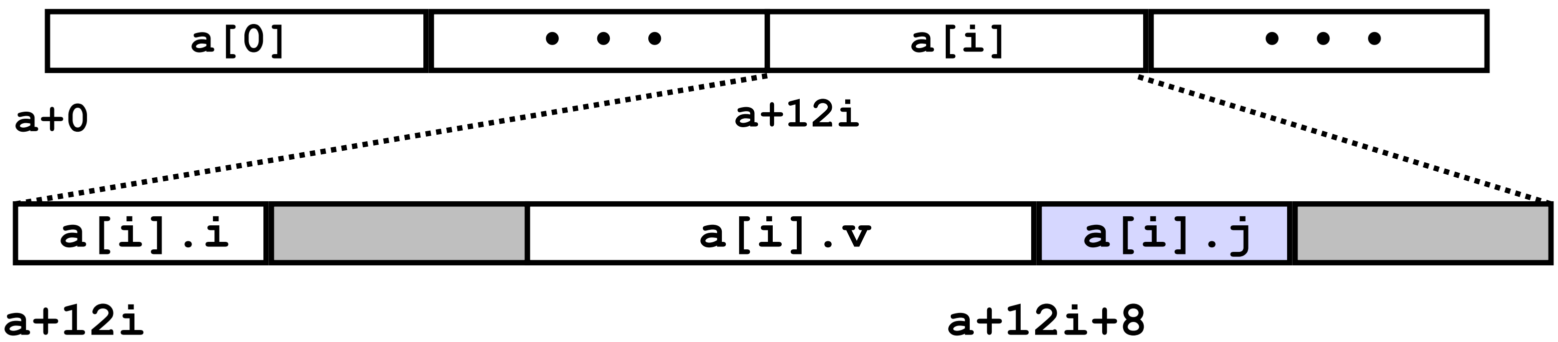
Accessing element within array

- Compute offset to start of structure
  - Compute 12**i* as 4*(*i*+2*i*)
- Access element according to its offset within structure
  - Offset by 8
  - Assembler gives displacement as a + 8
    - Linker must set actual value

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```

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

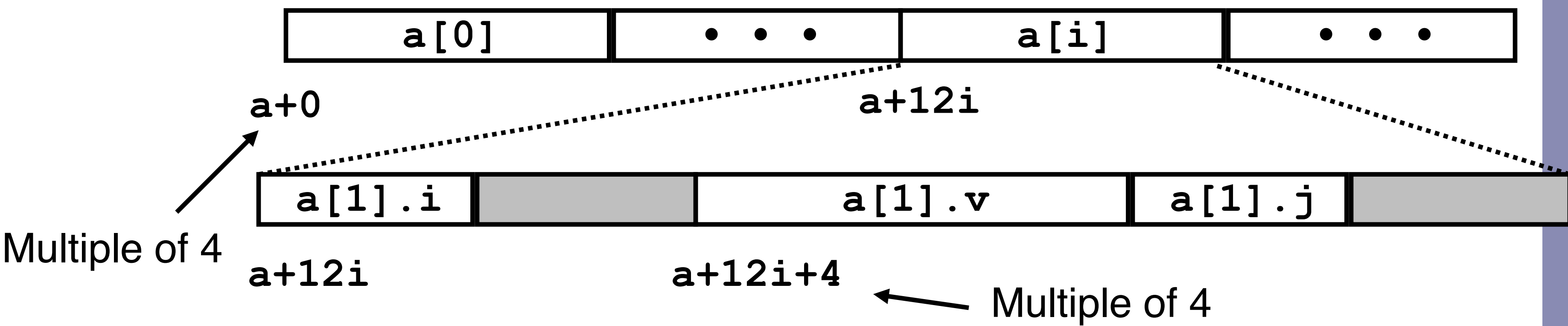
```
# %eax = idx
leal (%eax,%eax,2),%eax # 3*idx
movswl a+8(,%eax,4),%eax
```



Satisfying alignment within structure

- Achieving Alignment
  - Starting address of structure array must be multiple of worst-case alignment for any element
    - `a` must be multiple of 4
  - Offset of element within structure must be multiple of element's alignment requirement
    - `v`'s offset of 4 is a multiple of 4
  - Overall size of structure must be multiple of worst-case alignment for any element
    - Structure padded with unused space to be 12 bytes

```
struct S6 {
  short i;
  float v;
  short j;
} a[10];
```



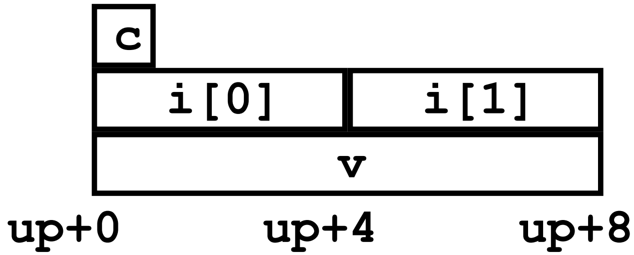
Checkpoint

Checkpoint

Union allocation

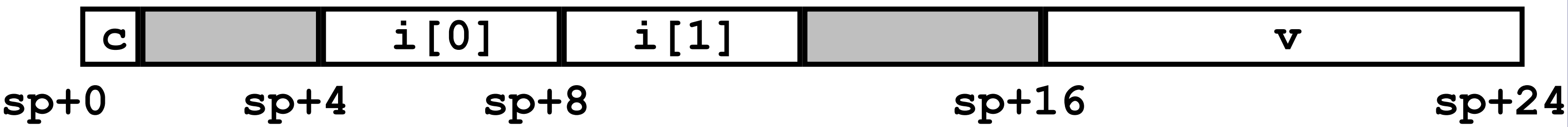
- Principles
  - Overlay union elements
  - Allocate according to largest element
  - Can only use one field at a time

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




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

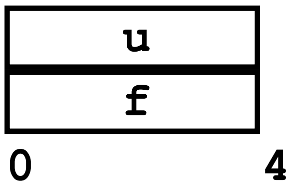
(Windows alignment)



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;
}
```




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

- 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`

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;
```

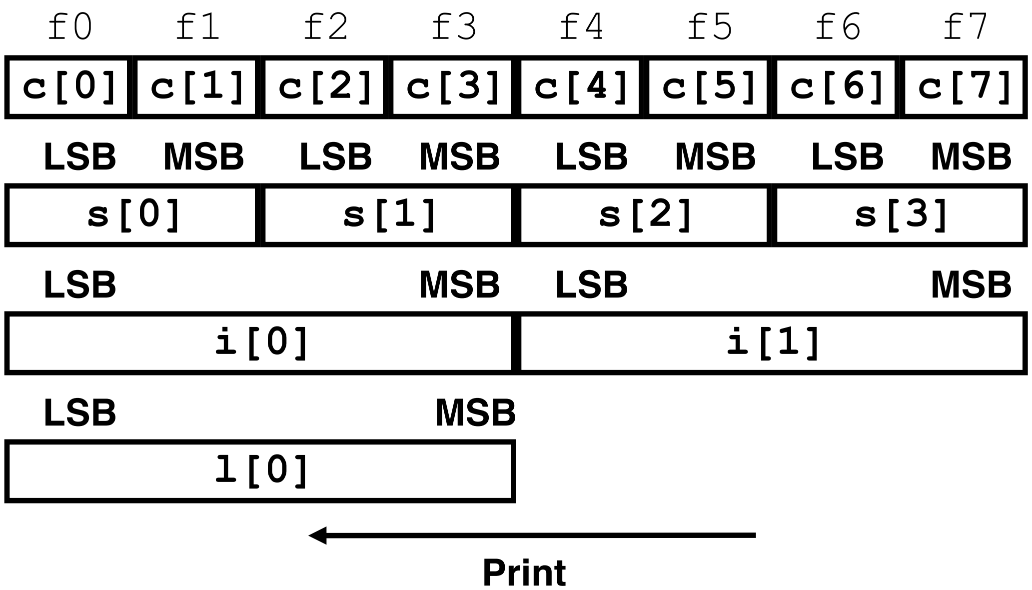
<table>
<tr><td>c[0]</td><td>c[1]</td><td>c[2]</td><td>c[3]</td><td>c[4]</td><td>c[5]</td><td>c[6]</td><td>c[7]</td></tr>
<tr><td colspan="2">s[0]</td><td colspan="2">s[1]</td><td colspan="2">s[2]</td><td colspan="2">s[3]</td></tr>
<tr><td colspan="4">i[0]</td><td colspan="4">i[1]</td></tr>
<tr><td colspan="4">l[0]</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 x86

Little Endian

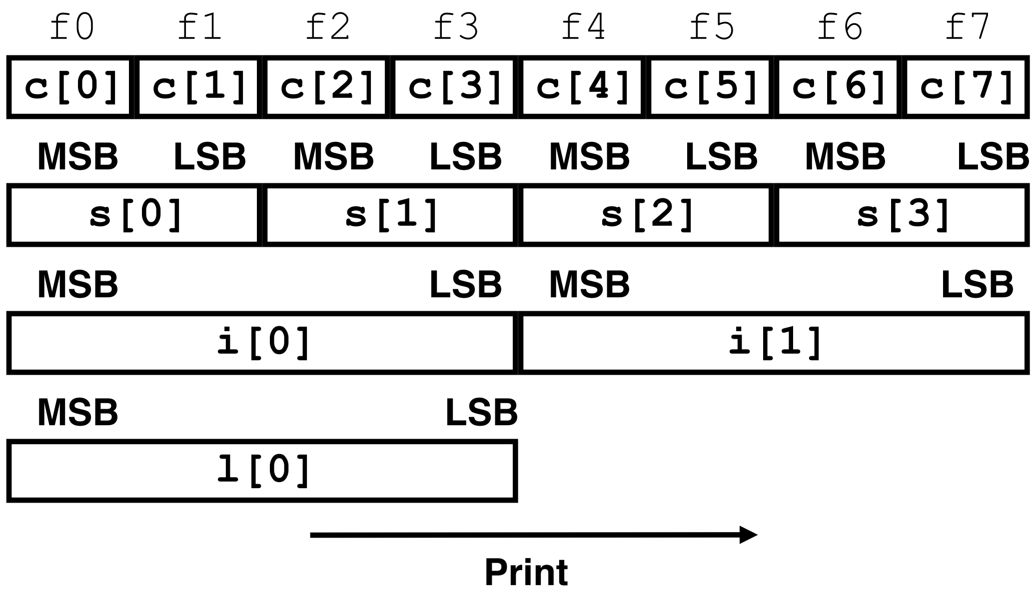


Output on Pentium:

```
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   == [f3f2f1f0]
```

Byte ordering on sun

Big Endian

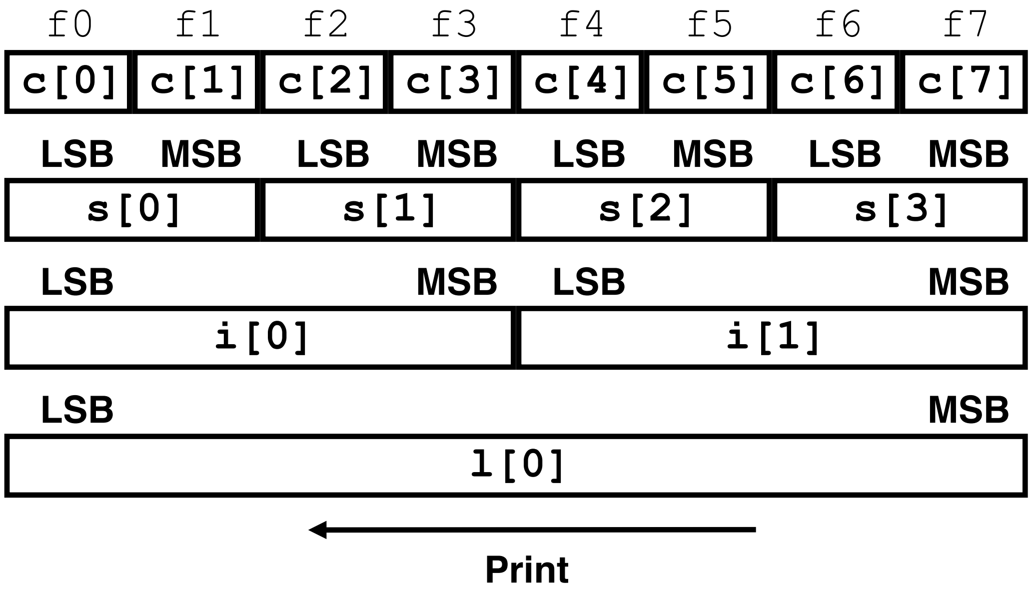


Output on Sun:

```
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 alpha

Little Endian



Output on Alpha:

```
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