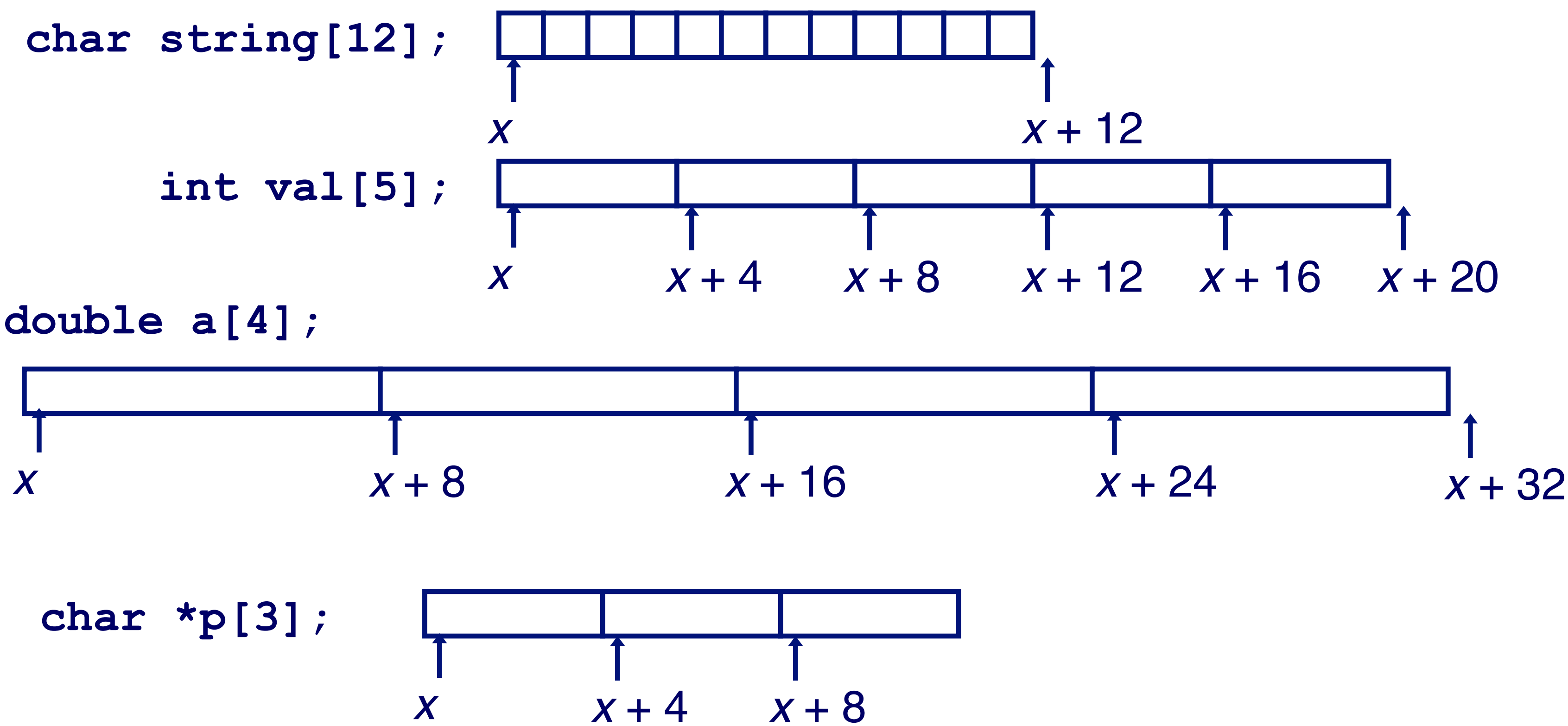
# Array Allocation

## Basic Principle

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

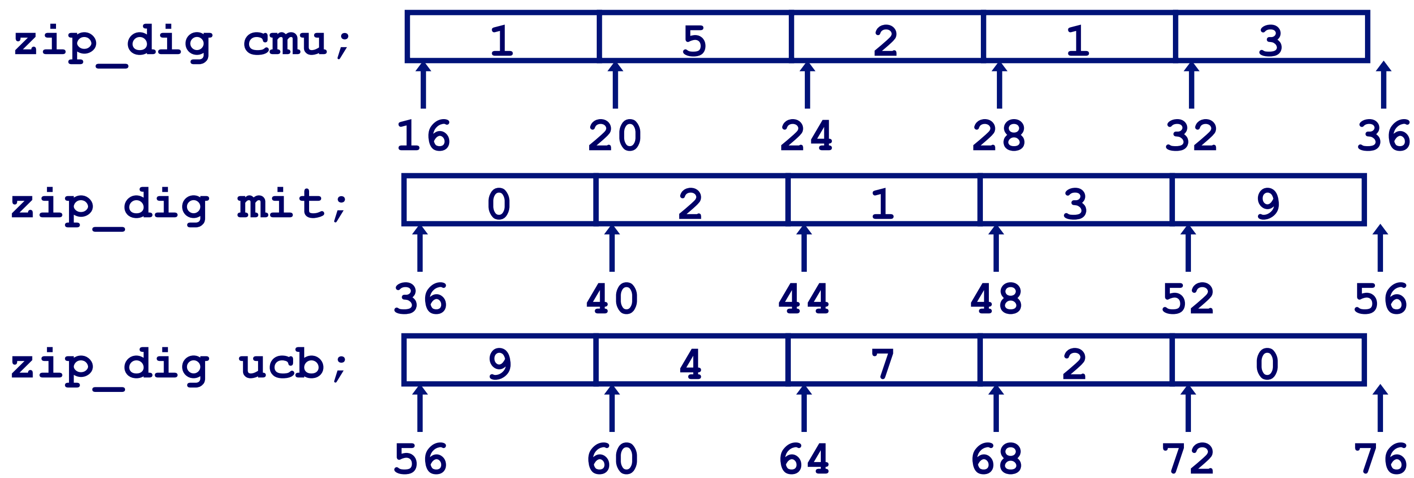
- Array of data type *T* and length *L*
- Contiguously allocated region of *L* * `sizeof(`*T*`)` bytes



# 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

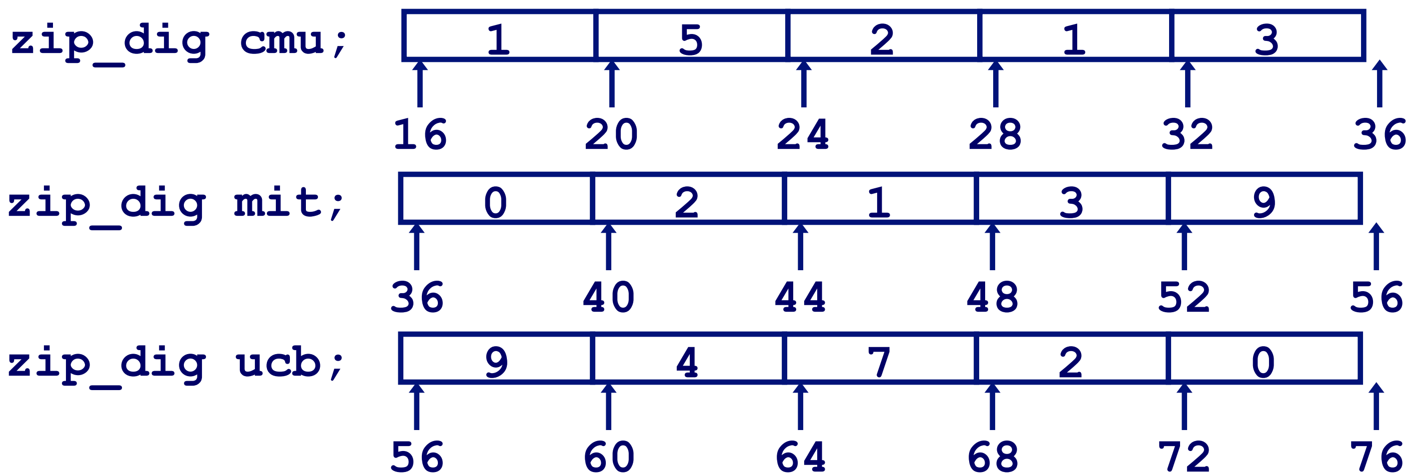
- Register `%edx` contains starting address of array
- Register `%eax` contains array index
- Desired digit at `4*%eax + %edx`
- Use memory reference `(%edx,%eax,4)`

```
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;
}
```

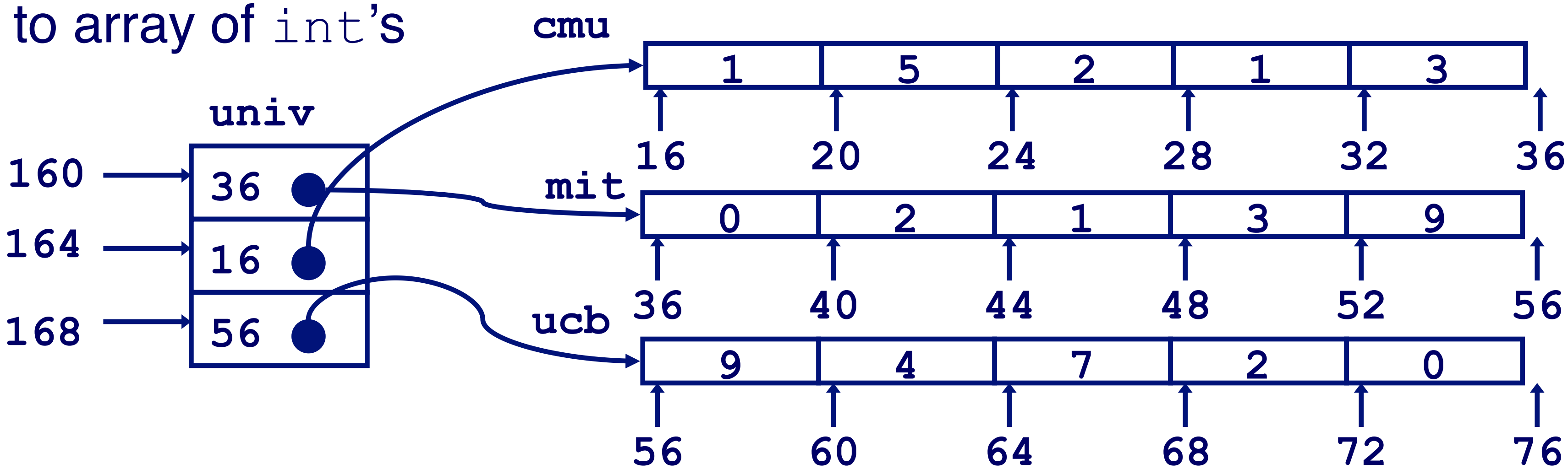
```
  # %ecx = z
  xorl %eax,%eax             # zi = 0
  leal 16(%ecx),%ebx         # zend  = z+4
.L59:
  leal (%eax,%eax,4),%edx    # 5*zi
  movl (%ecx),%eax           # *z
  addl $4,%ecx               # z++
  leal (%eax,%edx,2),%eax    # zi = *z + 2*(5*zi)
  cmpl %ebx,%ecx             # z : zend
 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

# 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

```
# %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]
```

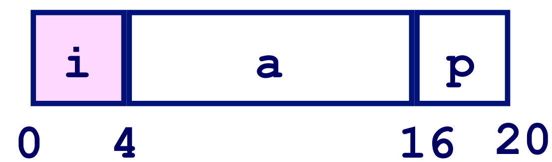
# Structures

## 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

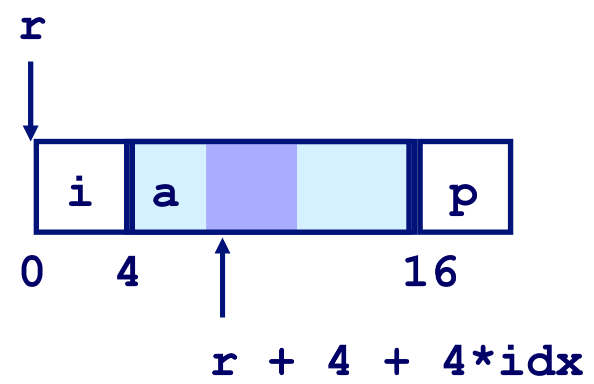
```
void
set_i(struct rec *r,
      int val)
{
  r->i = val;
}
```

### 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];
}
```

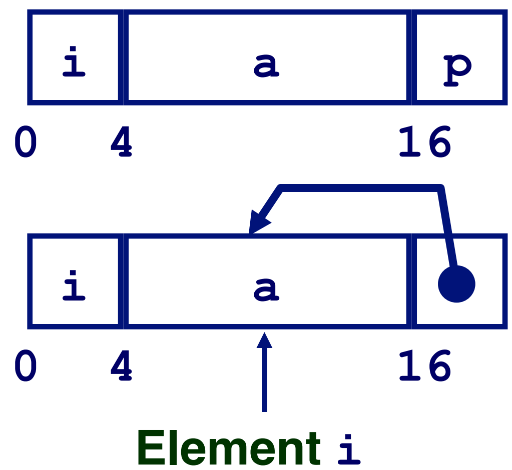
```
# %ecx = idx
# %edx = r
leal 0(,%ecx,4),%eax    # 4*idx
leal 4(%eax,%edx),%eax # r+4*idx+4
```

# 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 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

## 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`)
  - Linux:
    - » 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

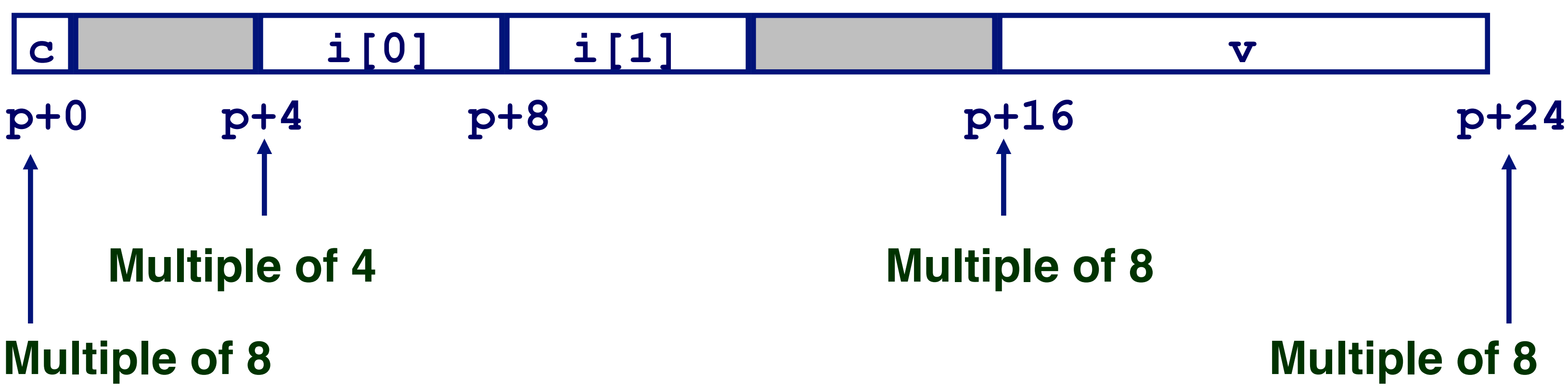
## Overall Structure Placement

- Each structure has alignment requirement K
  - Largest alignment of any element
- Initial address & structure length must be multiples of K

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

## Example (under Windows):

- K = 8, due to `double` element

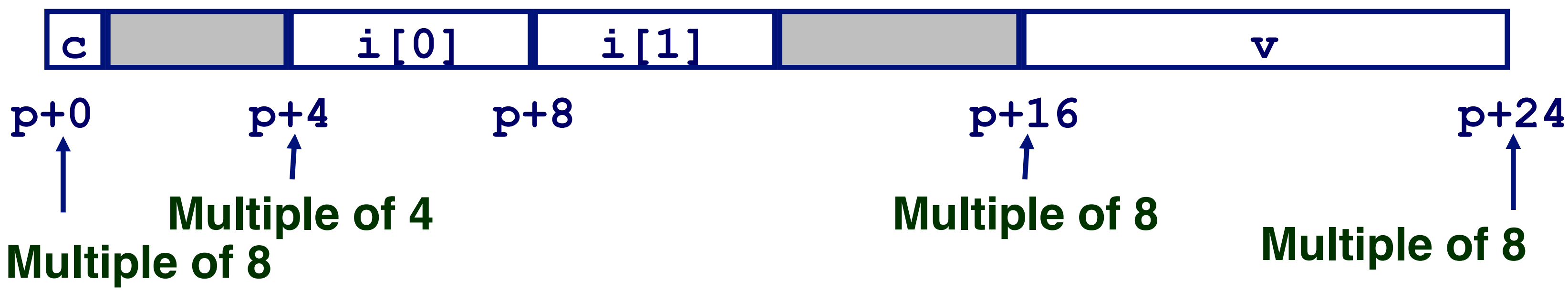


# Linux vs. Windows

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

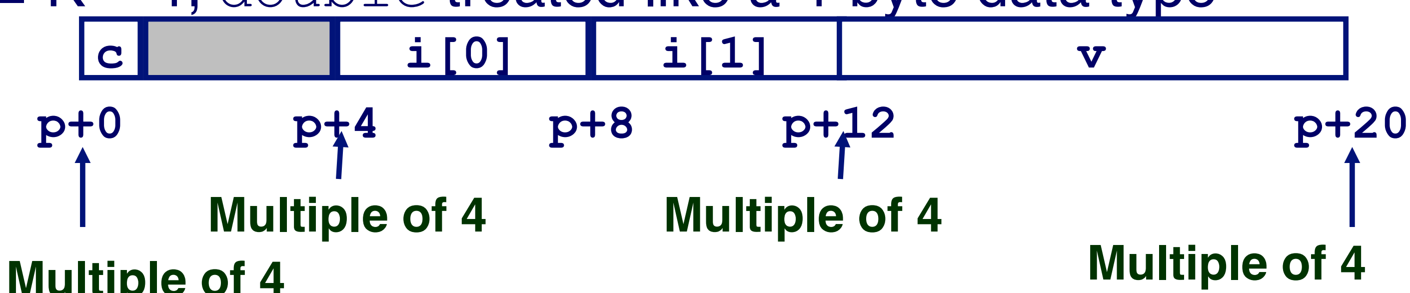
## Windows (including Cygwin):

- K = 8, due to `double` element




## Linux:

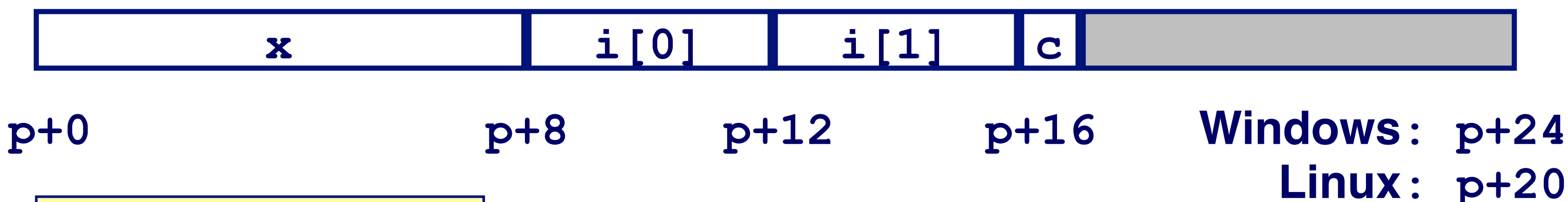
- K = 4; `double` treated like a 4-byte data type



# Overall Alignment Requirement

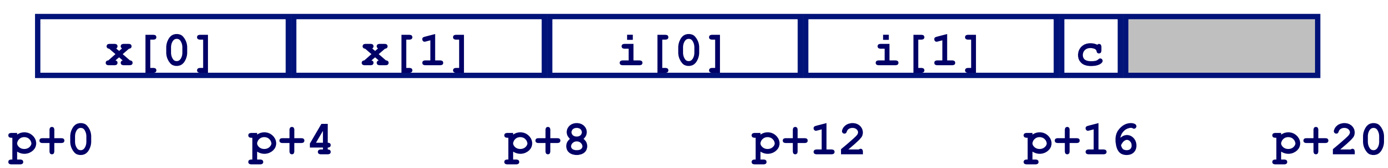
```
struct S2 {
  double x;
  int i[2];
  char c;
} *p;
```

`p` must be multiple of:
- 8 for Windows
- 4 for Linux




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

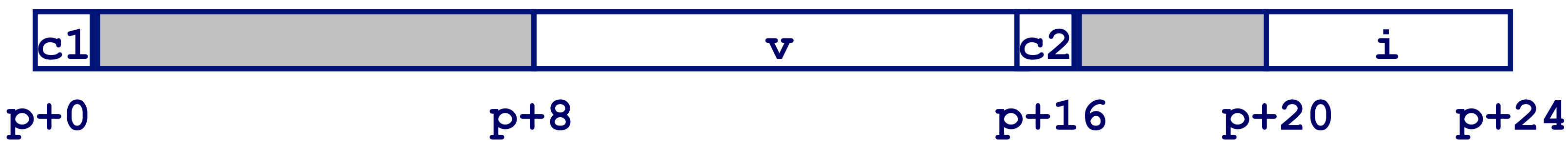
`p` must be multiple of 4 (in either OS)



# Ordering Elements Within Structure

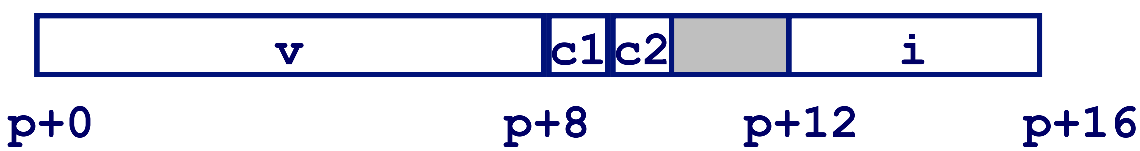
```
struct S4 {
  char c1;
  double v;
  char c2;
  int i;
} *p;
```

10 bytes wasted space in Windows




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

2 bytes wasted space

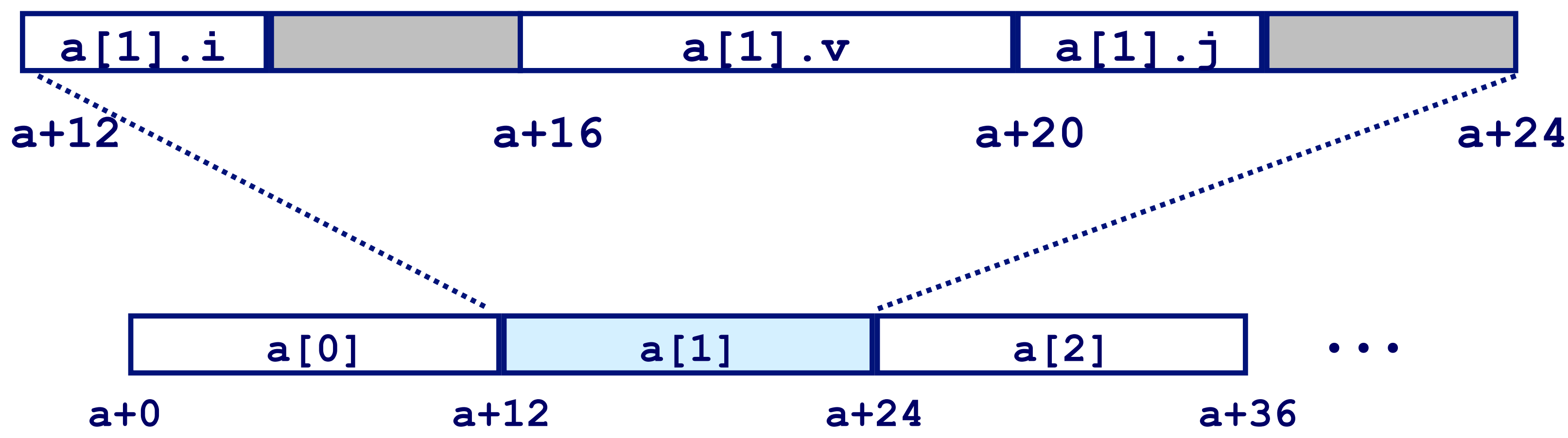


# Arrays of Structures

## 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];
```

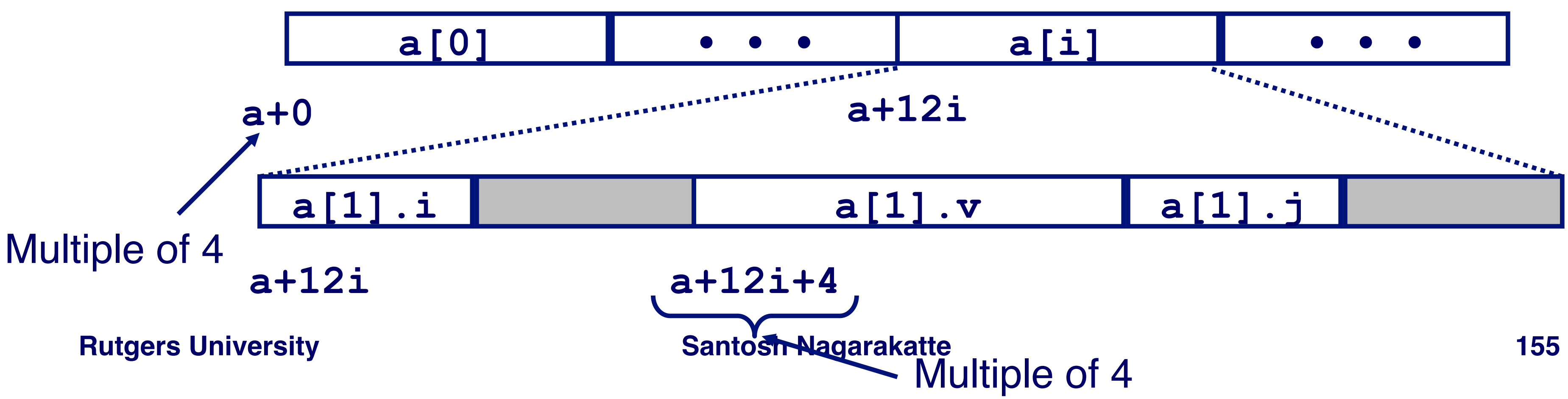


# Satisfying Alignment within Structure

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];
```



# 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