# Procedures *con't*

# Register Saving Conventions

- When procedure yoo calls who:
  - `yoo()` is the *caller*
  - `who()` is the *callee*
- Can register be used for temporary storage?

```
yoo:
    • • •
    movq $15213, %rdx
    call who
    addq %rdx, %rax
    • • •
    ret
```

```
who:
    • • •
    subq $18213, %rdx
    • • •
    ret
```

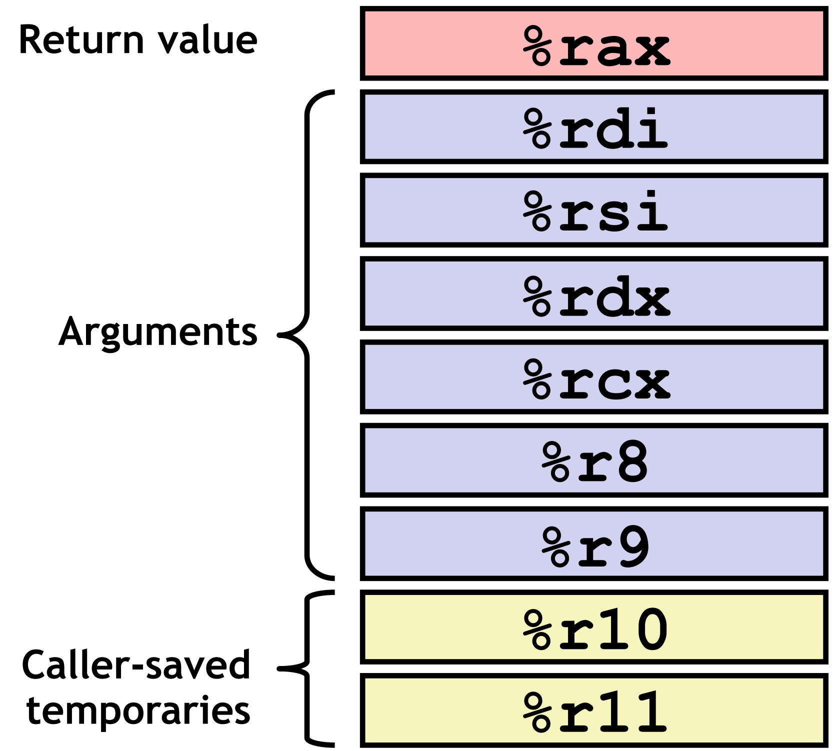
  - Contents of register **%rdx** overwritten by **who**
- Machine-Level programmer needs to solve for this.

# + Register Saving Conventions

- When procedure yoo calls who:
  - **`yoo()`** is the ***caller***
  - **`who()`** is the ***callee***
- Can register be used for temporary storage?
- **Conventions**
  - *"Caller Saved"*
    - Caller saves temporary values in its frame before the call
  - *"Callee Saved"*
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

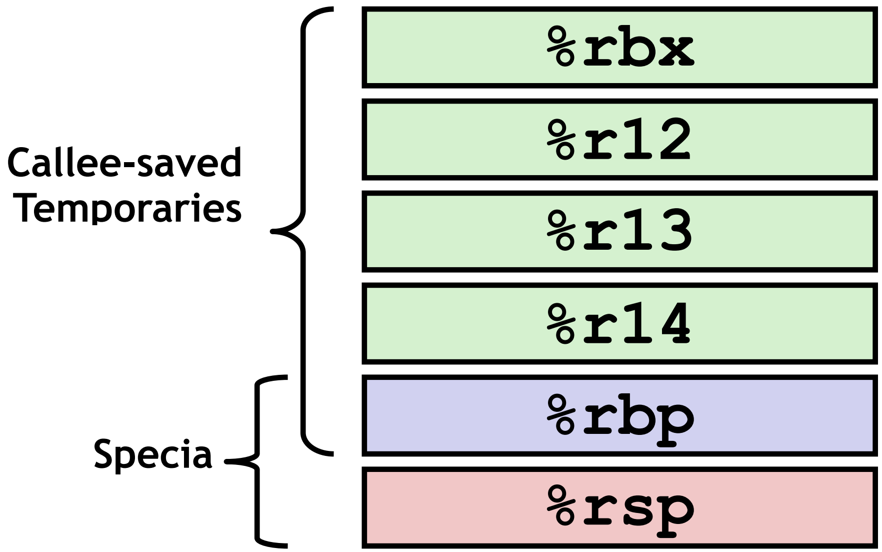
# x86-64 Linux Caller-saved Registers

- `%rax`
  - Return value
  - Also caller-saved
  - Can be modified by procedure
- `%rdi, ..., %r9`
  - Arguments
  - Also caller-saved
  - Can be modified by procedure
- `%r10, %r11`
  - Caller-saved
  - Can be modified by procedure



# + x86-64 Linux Callee-saved Registers

- **`%rbx, %r12, %r13, %r14`**
  - Callee-saved
  - Callee must save & restore
- **`%rbp`**
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
- **`%rsp`**
  - Special form of callee save
  - Restored to original value upon exit from procedure



# + Callee-Saved Example

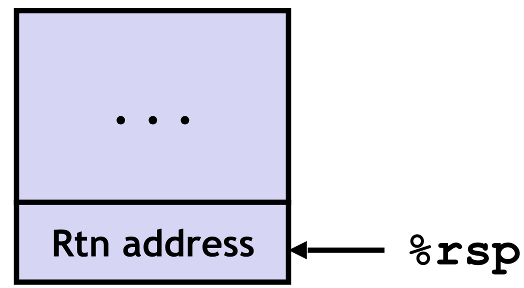
```
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```
long call_incr(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

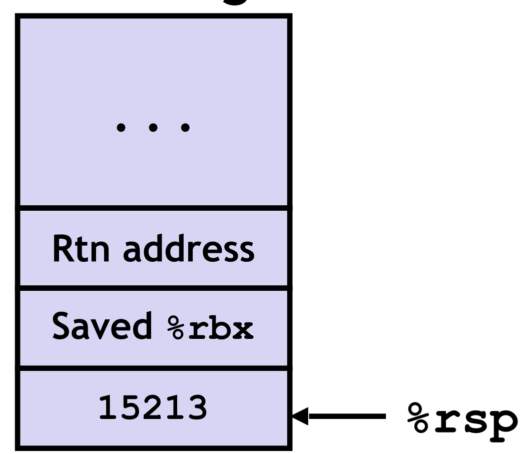
```
call_incr:
  pushq    %rbx             # save %rbx
  subq     $8, %rsp         # allocate
  movq     $15213, (%rsp)   # “push”
  movq     %rdi, %rbx
  movq     $3000, %rsi
  leaq     (%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $8, %rsp
  popq     %rbx
  ret
```

## call_incr’s stack

### Initial Stack




### Resulting Stack



# Callee-Saved Example *con't*

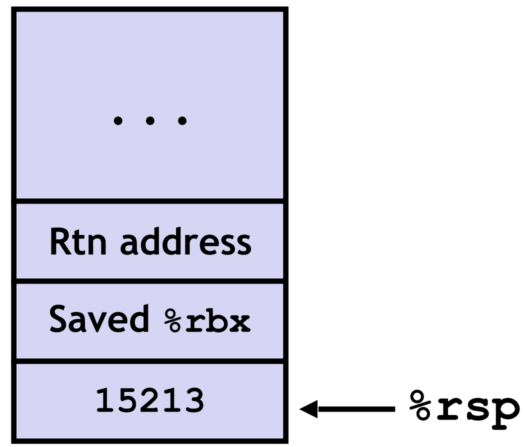
```c
long incr(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

```c
long call_incr(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

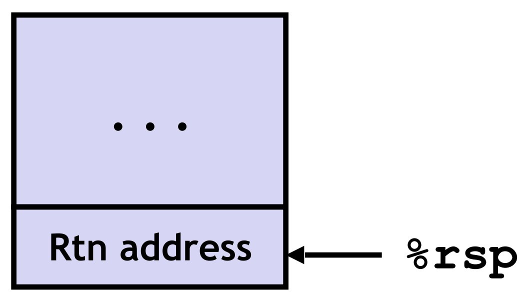
```
call_incr:
  pushq    %rbx
  subq     $8, %rsp
  movq     %rdi, %rbx
  movq     $15213, (%rsp)
  movq     $3000, %rsi
  leaq     (%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $8, %rsp         # deallocate
  popq     %rbx             # restore %rbx
  ret
```

## call_incr's stack

### Resulting Stack



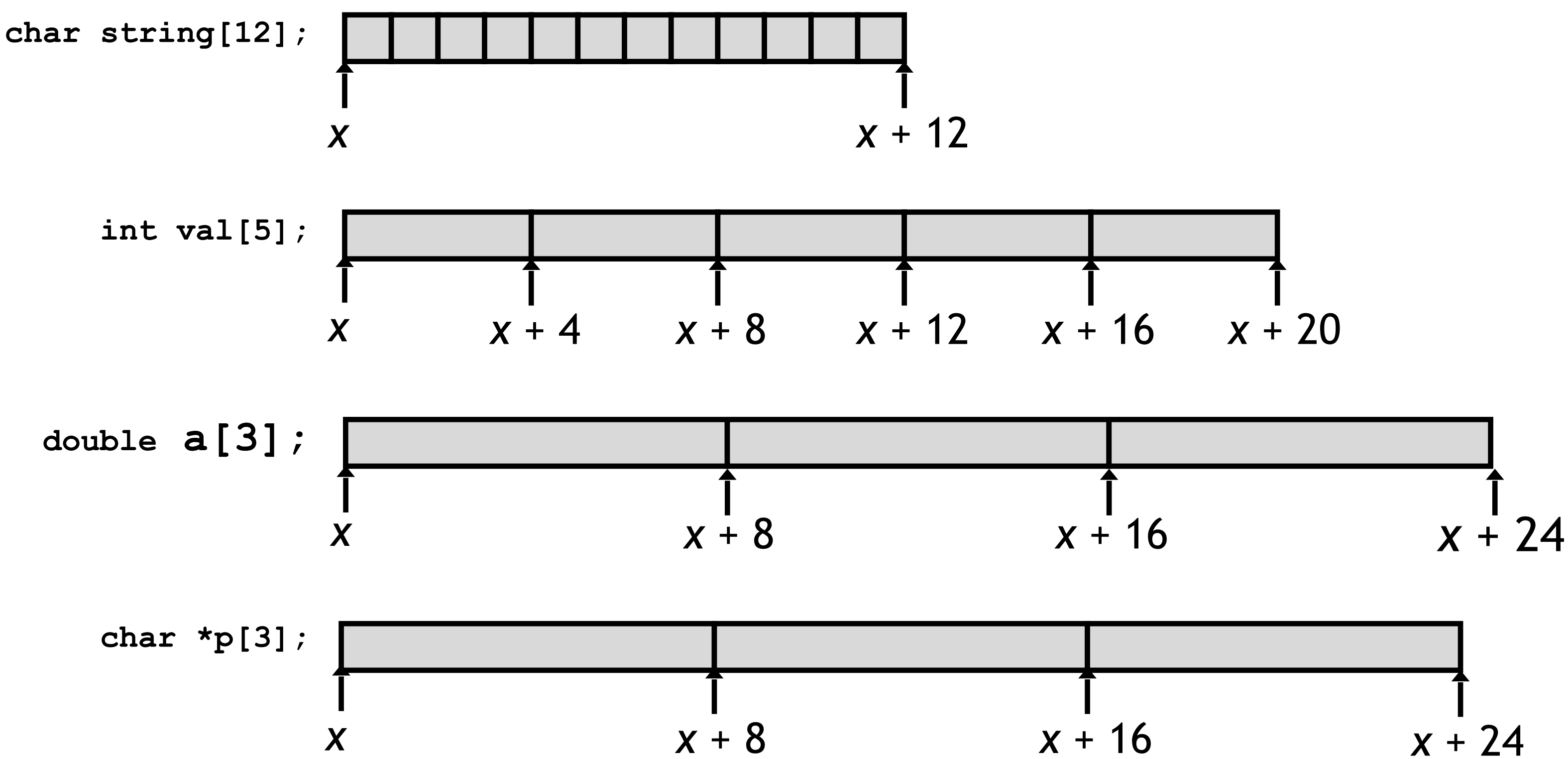

### Pre-return Stack



# Data: Arrays

# + Array Allocation

- **Basic Principle**
  *T* **A**[*N*];
  - Array of data type ***T*** and length ***N***
  - Contiguously allocated region of ***N*** * **sizeof**(*T*) bytes in memory



# + 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****

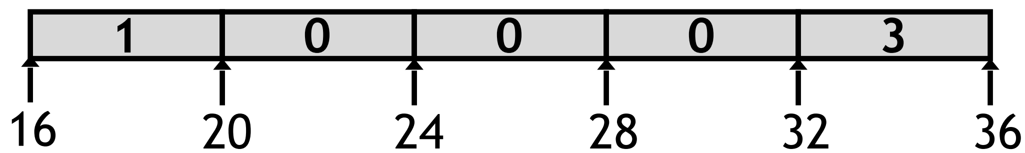
`int val[5];`

| 1 | 0 | 0 | 0 | 3 | |
|---|---|---|---|---|---|
| *x* | *x* + 4 | *x* + 8 | *x* + 12 | *x* + 16 | *x* + 20 |

| 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` | 0 |
| `val + `*i* | `int*` | *x* + 4 *i* |

# + Array Accessing Example

`int[] nyu;`




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

```
# %rdi = z
# %rsi = digit
movq (%rdi,%rsi,4), %rax  # z[digit]
```

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

# Array Loop Example

```
void zincr(int[] z) {
  int i;
  for (i = 0; i < 5; i++)
    z[i]++;
}
```

```
# %rdi = z, %rax = i
  movl    $0, %rax              #   i = 0
  jmp     .L3                   #   goto middle
.L4:                            # loop:
  addl    $1, (%rdi,%rax,4)     #   z[i]++
  addq    $1, %rax              #   i++
.L3:                            # middle
  cmpq    $4, %rax              #   i:4
  jbe     .L4                   #   if <=, goto loop
  rep; ret
```

# Multidimensional Arrays

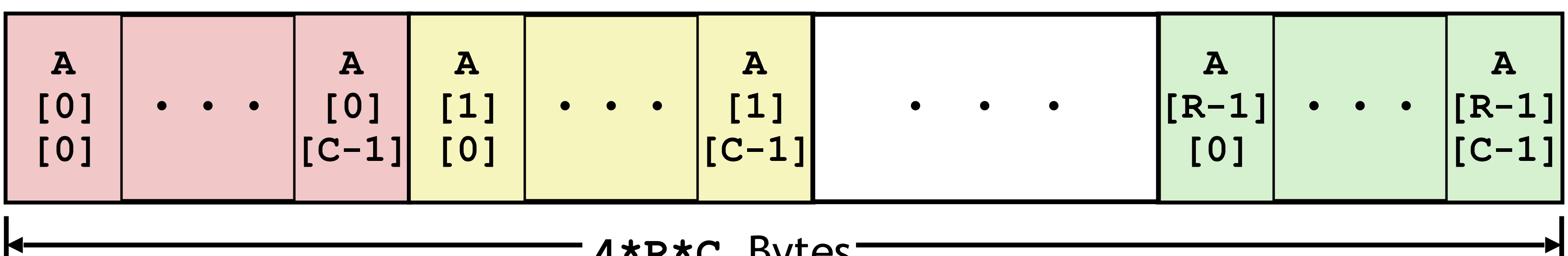
- **Declaration**

  `T  A[R][C];`

  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes
- **Array Size**
  - *R* * *C* * *K* bytes
- **Arrangement**
  - Row-Major Ordering

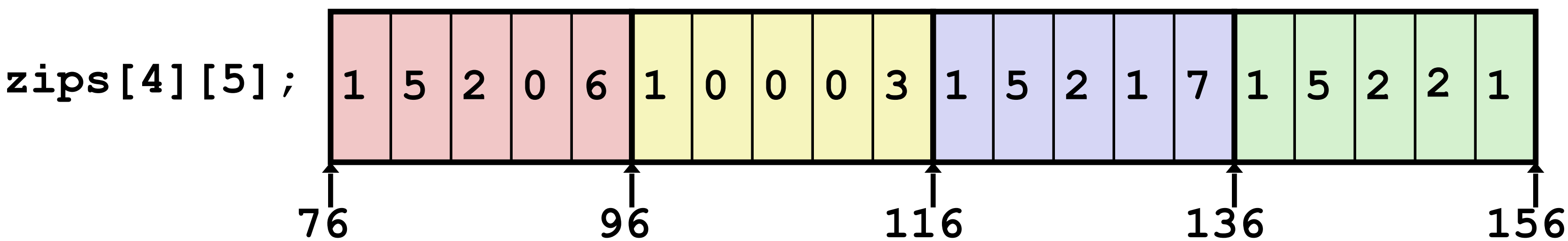
$$\begin{bmatrix} \texttt{A[0][0]} & \cdots & \texttt{A[0][C-1]} \\ \vdots & & \vdots \\ \texttt{A[R-1][0]} & \cdots & \texttt{A[R-1][C-1]} \end{bmatrix}$$

`int A[R][C];`



# + Nested Array Example

```
int[4][5] zips =
  {{1, 5, 2, 0, 6},
   {1, 0, 0, 0, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```




- Variable **zips**: array of 4 elements, allocated contiguously
- Each element is an array of 5 **int**’s, allocated contiguously
- “Row-Major” ordering of all elements in memory

# + Nested Array Row Access

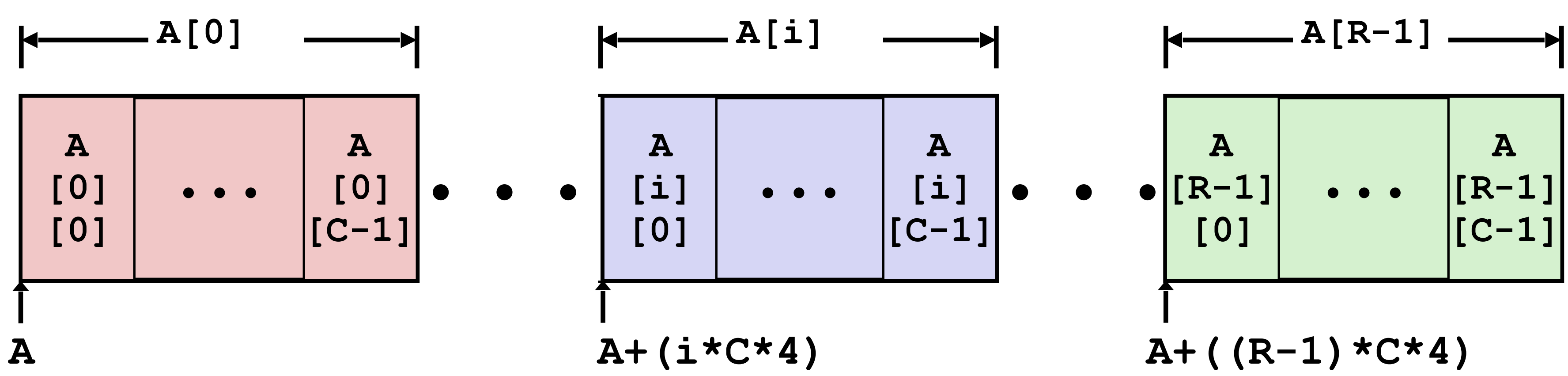
- **Declaration**

  `T A[R][C];`

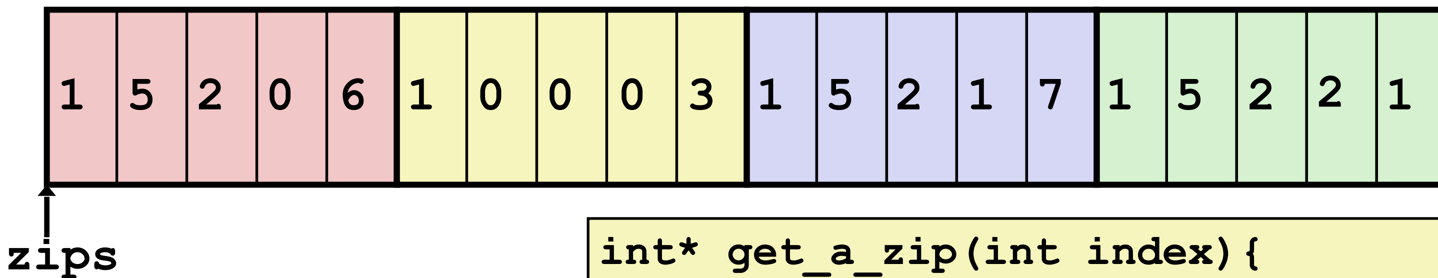
- **Row Vectors**
  - *A[i]* is an array of *C* elements, e.g. a "column"
  - Each element of type *T* requires *K* bytes
  - Therefore the starting address of column is `A + i *(C * K)`

- **Example**

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



# + Nested Array Row Access Code




```
int* get_a_zip(int index){
  return zips[index];
}
```

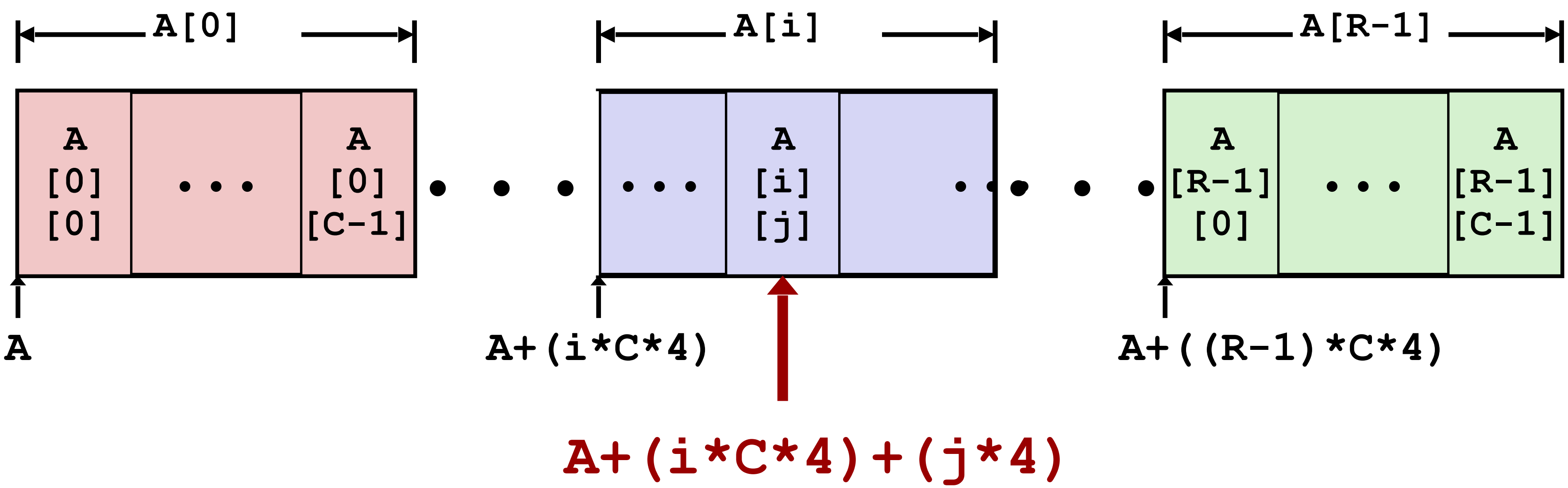
```
# %rdi = index
leaq (%rdi,%rdi,4), %rax        # 5 * index
leaq zips(,%rax,4), %rax        # zips + (20 * index)
```

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

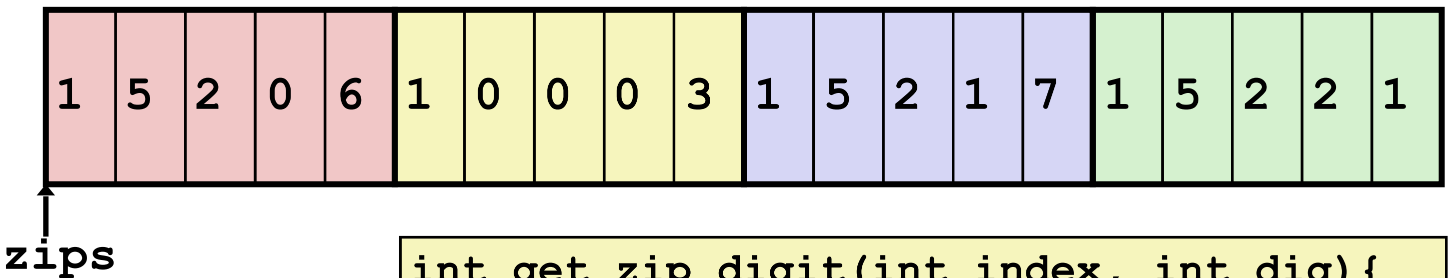
# + Nested Array Element Access

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

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



# Nested Array Element Access Code




```
int get_zip_digit(int index, int dig){
  return zips[index][dig];
}
```

```
leaq   (%rdi,%rdi,4), %rax   # 5 * index (%rdi is index)
addl   %rax, %rsi            # 5 * index + dig
movl   zips(,%rsi,4), %rax   # M[zips + 4*(5 * index + dig)]
```

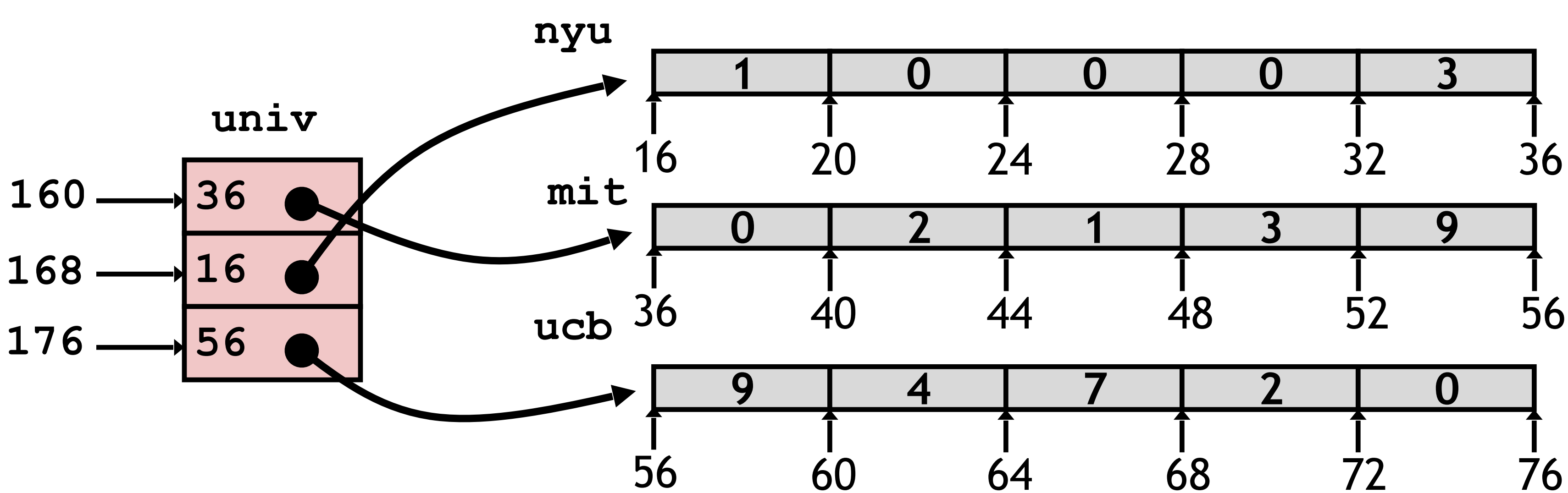
- Array Elements
  - `zips[index][dig]` is an int
  - Address: `zips + 20*index + 4*dig`
    - Expressed in assembly as `zips + 4*(5 * index + dig)`

# Multi-Level Array Example

```
int* nyu = { 1, 0, 0, 0, 3 };
int* mit = { 0, 2, 1, 3, 9 };
int* ucb = { 9, 4, 7, 2, 0 };
```

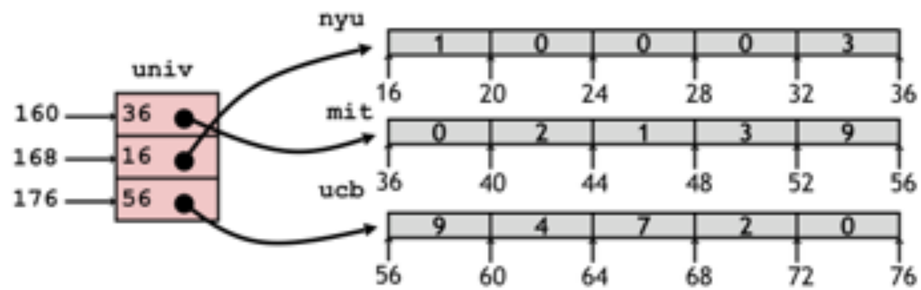
```
#define UCOUNT 3
int *univ[UCOUNT] = {mit, nyu, ucb};
```

- Variable **`univ`** denotes an array of 3 elements
- Each element is a pointer (8 bytes)
- Each pointer points to array of int's



# + Element Access in Multi-Level Array

```
int get_uni_digit
  (size_t index, size_t digit){
   return univ[index][digit];
}
```




```
salq    $2, %rsi              # 4*digit
addq    univ(,%rdi,8), %rsi  # p = univ[index] + 4*digit
movl    (%rsi), %eax          # return *p
ret
```

- Computation
  - Element access **Mem[Mem[univ+8*index]+4*digit]**
  - Must do two memory reads
    - First get pointer to row array
    - Then access element within array

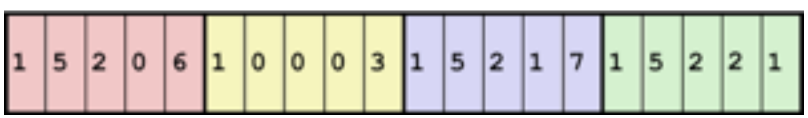
# + Array Element Accesses

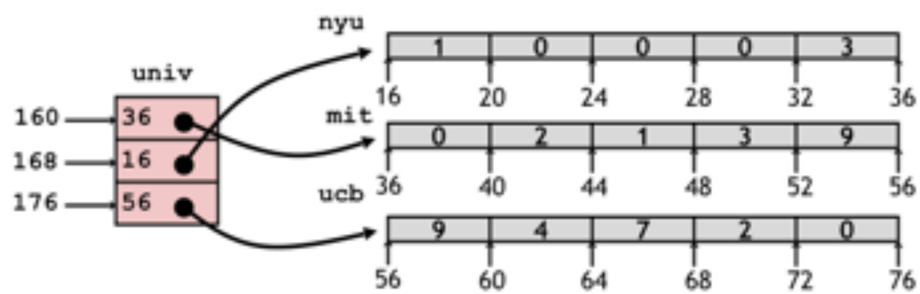
**Nested array**

```
int get_zip_digit
  (size_t index, size_t digit)
{
  return zips[index][digit];
}
```

**Multi-level array**

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




Accesses looks similar in C, but address computations very different:

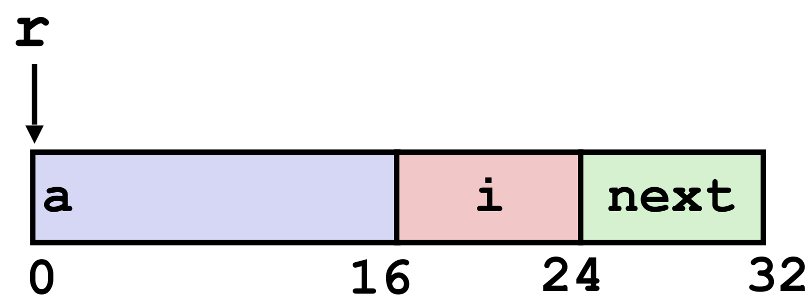
`Mem[zips+20*index+4*digit]`

`Mem[Mem[univ+8*index]+4*digit]`

# Data: Structs

# Structure Representation

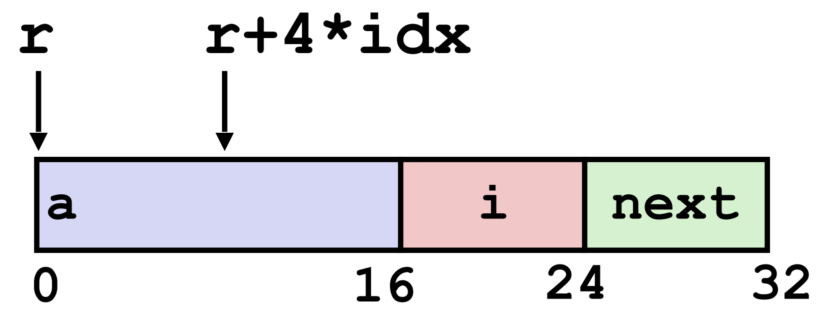
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```




- **Structure represented as block of memory**
  - Big enough to hold all of the fields
- **Fields ordered according to declaration**
  - Even if another ordering could yield a more compact representation
- **Compiler determines overall size + positions of fields**
  - Machine-level program has no understanding of the structures in the source code

# + Generating Pointer to Structure Member

```
struct rec {
    int a[4];
    int i;
    struct rec* next;
};
```



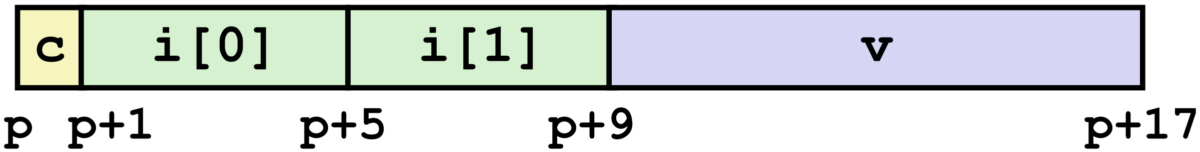

```
int* get_array_ptr(struct rec* r, int idx){
  return &r->a[idx];
}
```

```
leaq   (%rdi,%rsi,4), %rax    # r in %rdi, idx in %rsi
ret                           # move ptr into %rax, return
```

- **Generating pointer to array element**
  - Offset of each structure member determined at compile time
  - Compute as `r + 4 * idx`

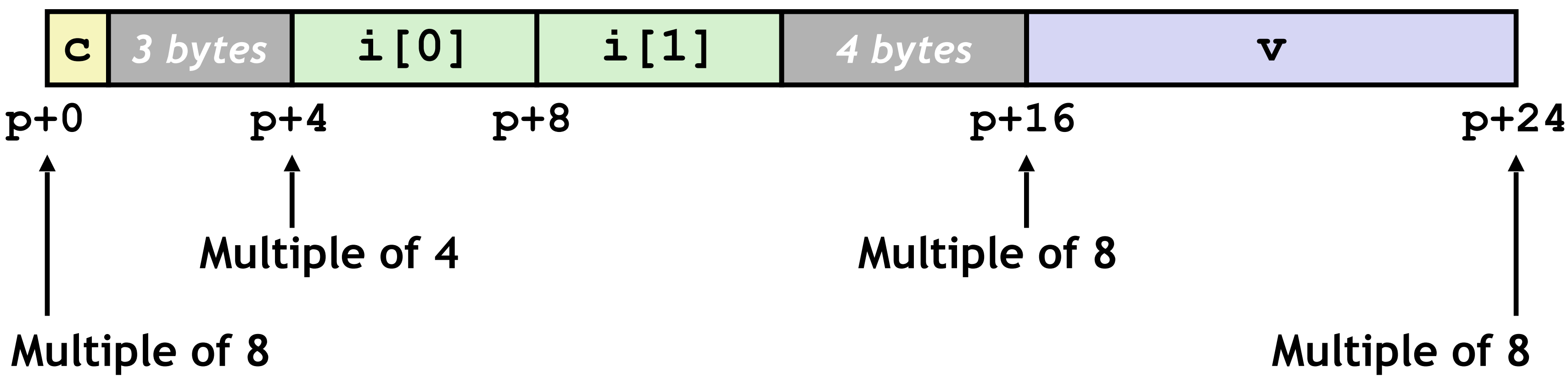
# + Structures & Alignment

- **Unaligned Data**




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

- **Aligned Data**
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***



# + Alignment Principles

- **Aligned Data**
  - Primitive data type requires ***K*** bytes
  - Address must be multiple of ***K***
  - Required on some machines; advised on x86-64
- **Motivation for Aligning Data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store datum that spans word boundaries
    - Virtual memory trickier when you allow unaligned data.
- **Compiler**
  - Inserts gaps in structure to ensure correct alignment of fields

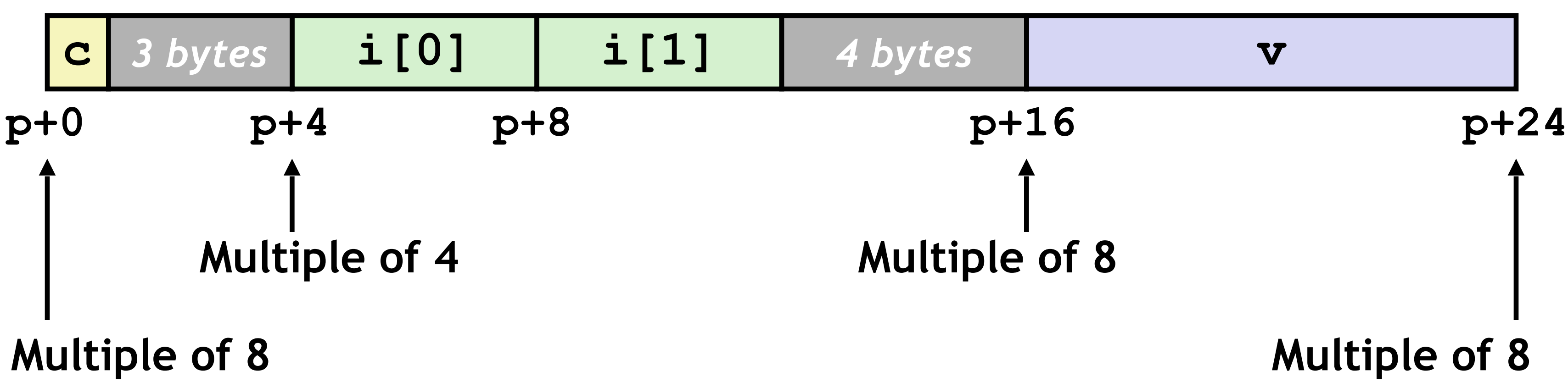
# + Specific Cases of Alignment (x86-64)

- **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`, `long`, `char *`, …**
  - lowest 3 bits of address must be $000_2$

# + Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each 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:**
  - K = 8, due to **double** element

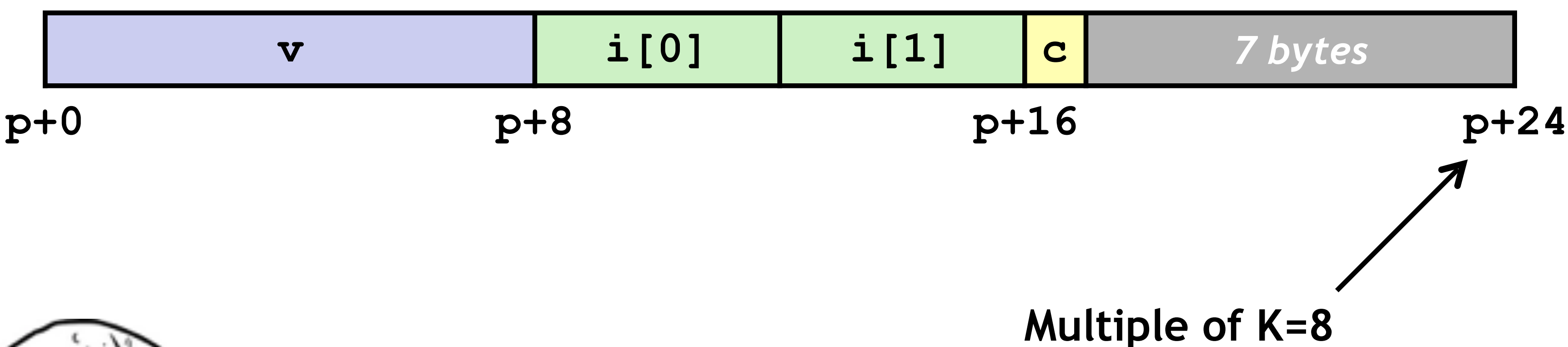
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

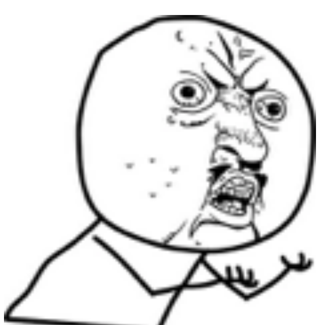


# Meeting Overall Alignment Requirement

- For largest alignment requirement K
- Overall structure must be multiple of K

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

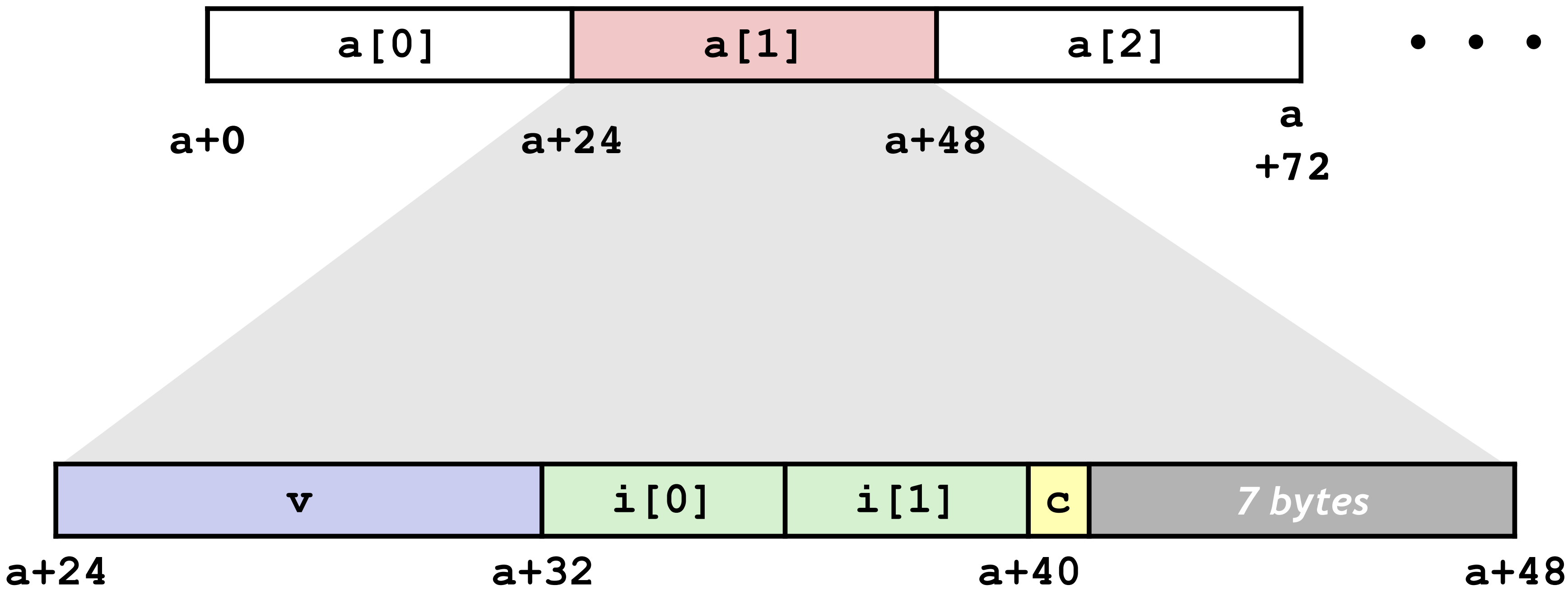


# Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

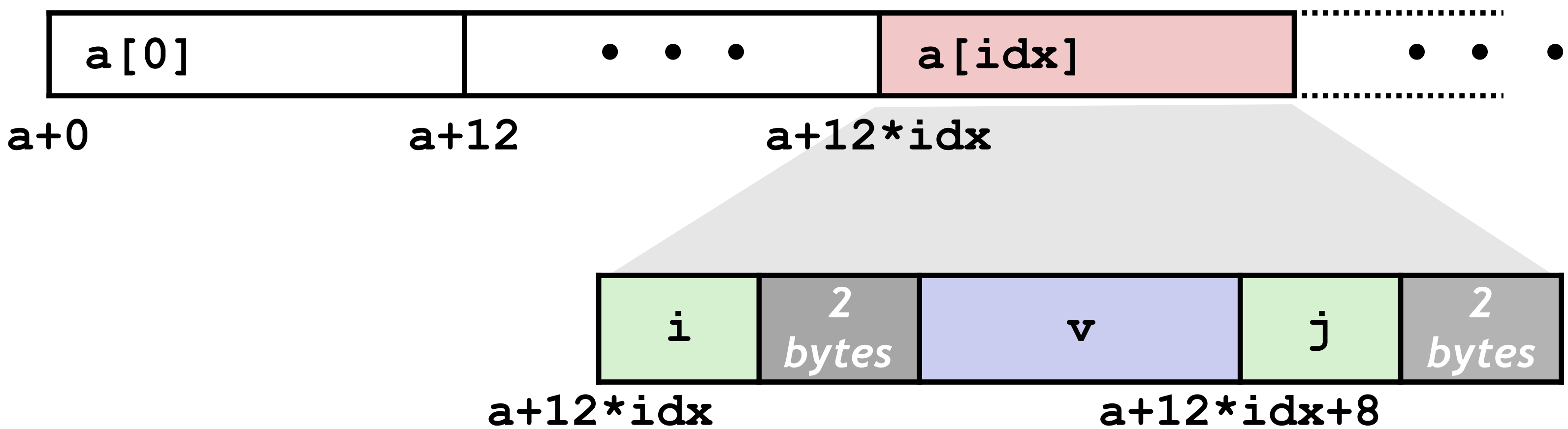
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



# + Accessing Members in Array of Struct

- **Compute array offset 12 `* idx`**
  - `sizeof(i) + sizeof(v) + sizeof(j) + 4 bytes == sizeof(S3) == 12 bytes` (4 bytes is the padding)
- **Element j is at an offset of 8 bytes within structure**
  - `sizeof(i) + 2 bytes + sizeof(float) == 8 bytes`

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



# + Saving Space (Struct packing)

- **Put large data types first**

```
struct S4 {
  char c;
  int i;
  char d;
} *p;
```

```
struct S5 {
  int i;
  char c;
  char d;
} *p;
```

- **Effect (K=12 -> K=4)**