Machine-Level Programming IV: Data

COMP400727: Introduction to Computer Systems

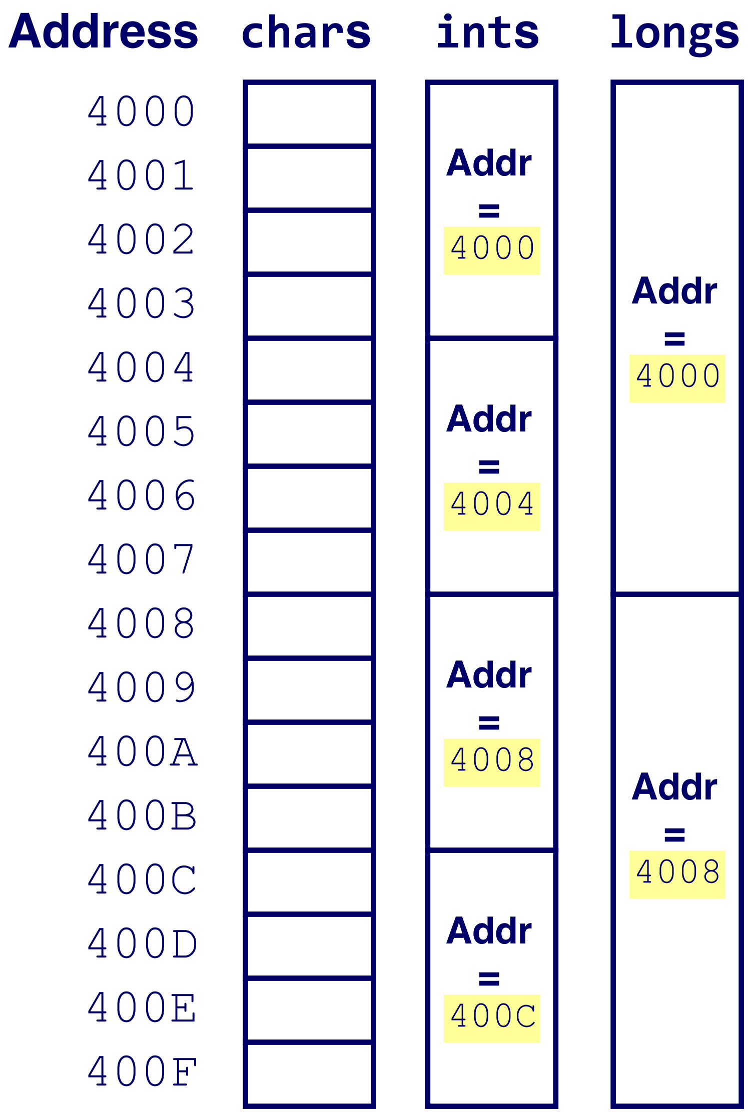
Danfeng Shan
Xi’an Jiaotong University

Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**
  - Allocation
  - Access
  - Alignment
- **If we have time: Union**

Reminder: Memory Organization

- **Memory locations do not have data types**
  - Types are implicit in how machine instructions *use* memory
- **Addresses specify byte locations**
  - Address of a larger datum is the address of its first byte
  - Addresses of successive items differ by the item's size



Array Allocation

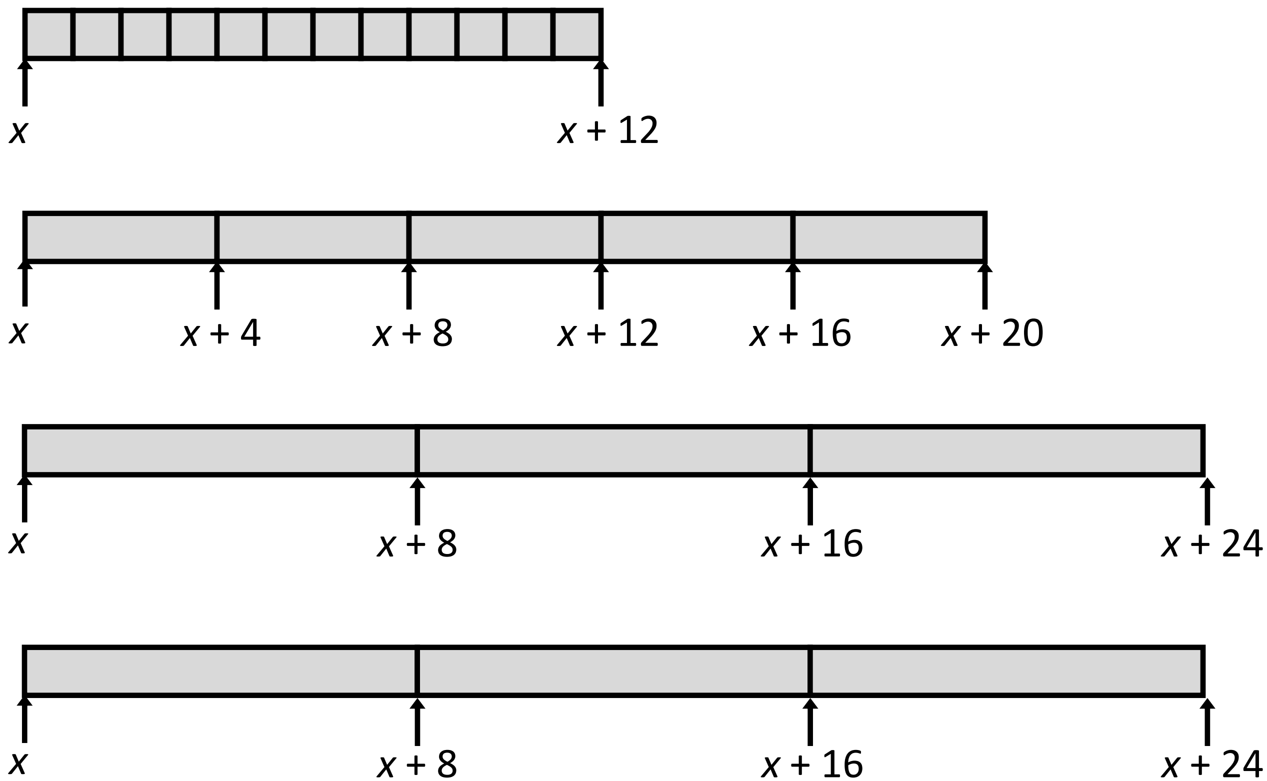
- C declaration ***Type*** **name[*Length*];**
  - Array of data type *Type* and length *Length*
  - Contiguously allocated region of *Length* * **sizeof**(*Type*) bytes in memory

`char string[12];`

`int val[5];`

`double a[3];`

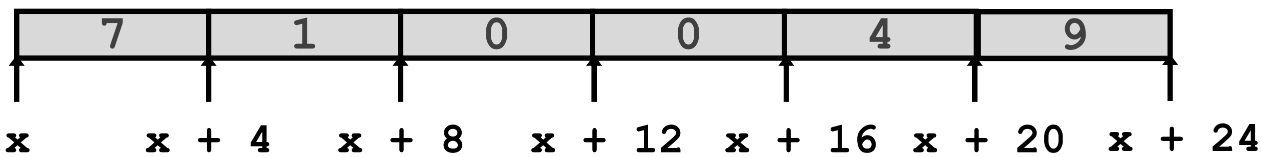
`char *p[3];`



Array Access

- **C declaration** ***`Type name[Length];`***
  - Array of data type *Type* and length *Length*
  - Identifier **`name`** acts like[1] a pointer to array element 0

`int val[6];`



- **Expression / Type / Value**

Expression	Type	Value	
`val[4]`	`int`	`4`	
`val[6]`	`int`	`??`	`// access past end`
`val`	`int *`	`x`	
`val+1`	`int *`	`x + 4`	
`&val[2]`	`int *`	`x + 8`	`// same as val+2`
`*(val+3)`	`int`	`0`	`// same as val[3]`
`val + i`	`int *`	`x + 4*i`	`// same as &val[i]`

[1] in most contexts (but not all)

Array Example

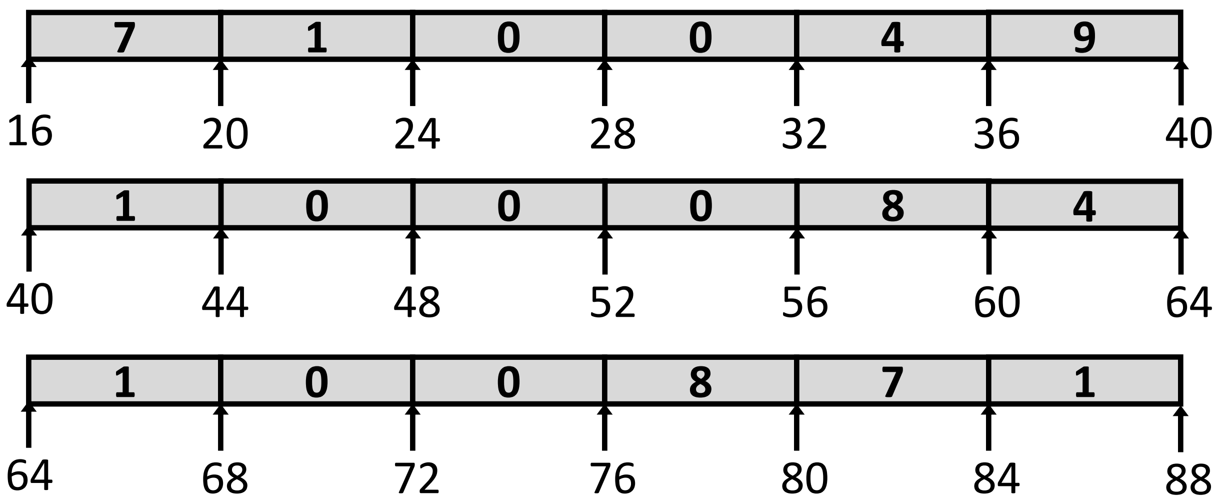
```
#define ZLEN 6
typedef int zip_dig[ZLEN];

zip_dig xjtu = { 7, 1, 0, 0, 4, 9 };
zip_dig thu  = { 1, 0, 0, 0, 8, 4 };
zip_dig pku  = { 1, 0, 0, 8, 7, 1 };
```

`zip_dig xjtu;`

`zip_dig  thu;`

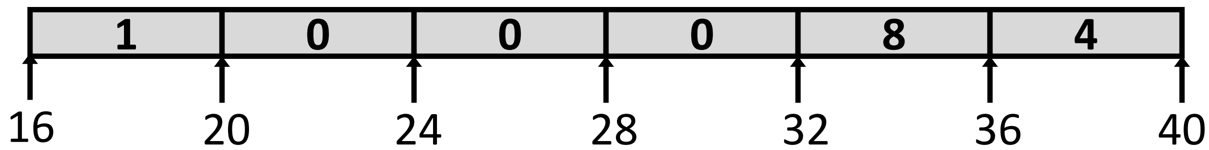
`zip_dig  pku;`



- **Declaration "`zip_dig xjtu`" equivalent to "`int xjtu[6]`"**
- **Example arrays were allocated in successive 24 byte blocks**
  - Not guaranteed to happen in general

Array Accessing Example

```
zip_dig xjtu;
```




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

x86-64

```
# %rdi = z
# %rsi = digit
movl (%rdi,%rsi,4), %eax  # 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(zip_dig z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax
  jmp     .L3
.L4:
  addl    $1, (%rdi,%rax,4)
  addq    $1, %rax
.L3:
  cmpq    $4, %rax
  jbe     .L4
  rep; ret
```

Array Loop Example

```
void zincr(zip_dig z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

```
  # %rdi = z
  movl    $0, %eax            #   i = 0
  jmp     .L3                 #   goto middle
.L4:                          # loop:
  addl    $1, (%rdi,%rax,4)   #   z[i]++
  addq    $1, %rax            #   i++
.L3:                          # middle
  cmpq    $5, %rax            #   i:4
  jbe     .L4                 #   if <=, goto loop
  rep; ret
```

Understanding Pointers & Arrays #1

Decl	A1 , A2			*A1 , *A2		
	Comp	Bad	Size	Comp	Bad	Size
`int A1[3]`						
`int *A2`						

- **Comp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

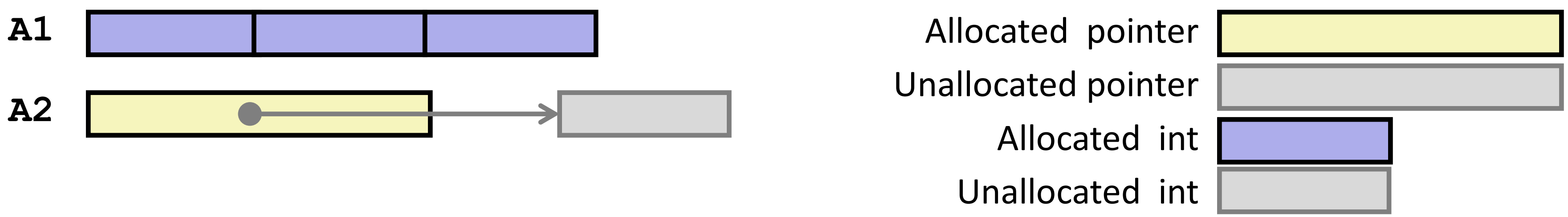
Understanding Pointers & Arrays #1

Decl	`A1 , A2`			`*A1 , *A2`		
	Comp	Bad	Size	Comp	Bad	Size
`int A1[3]`						
`int *A2`						

- **Comp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

Understanding Pointers & Arrays #1

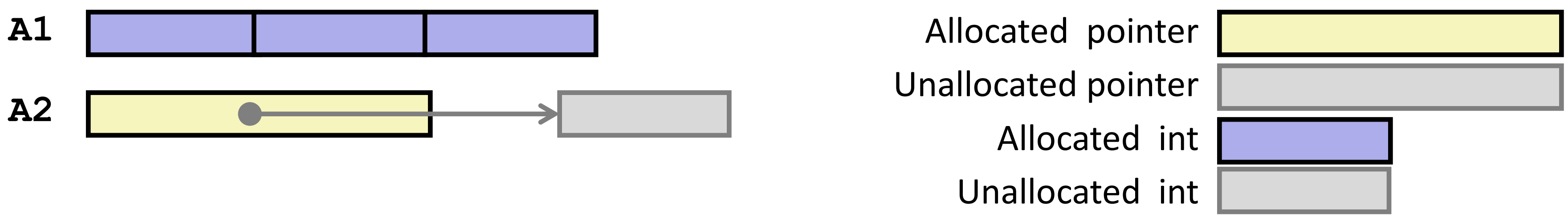
Decl	A1 , A2			*A1 , *A2		
	Comp	Bad	Size	Comp	Bad	Size
`int A1[3]`	Y	N	12			
`int *A2`						



- **Comp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

Understanding Pointers & Arrays #1

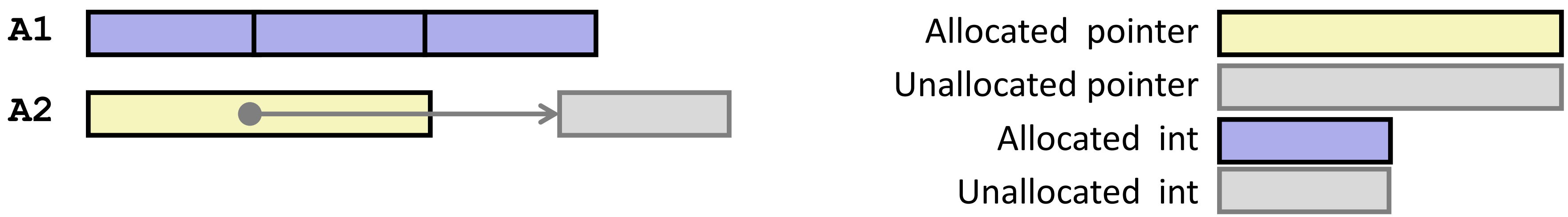
Decl	A1 , A2			*A1 , *A2		
	Comp	Bad	Size	Comp	Bad	Size
`int A1[3]`	Y	N	12			
`int *A2`	Y	N	8			



- Comp: Compiles (Y/N)
- Bad: Possible bad pointer reference (Y/N)
- Size: Value returned by `sizeof`

Understanding Pointers & Arrays #1

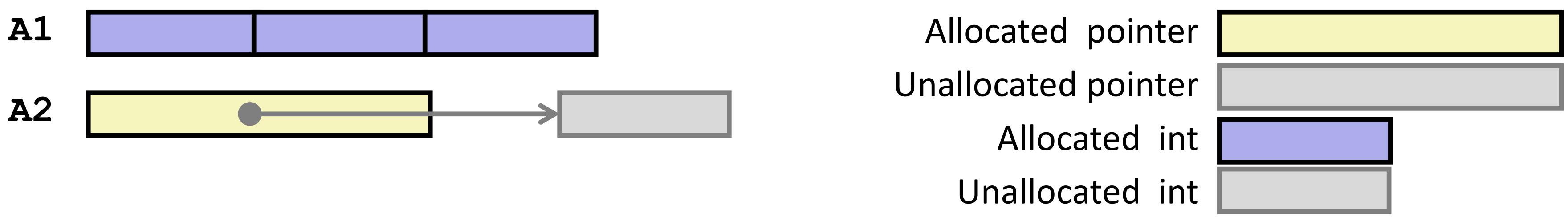
Decl	A1 , A2			*A1 , *A2		
	Comp	Bad	Size	Comp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4
`int *A2`	Y	N	8			



- **Comp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

Understanding Pointers & Arrays #1

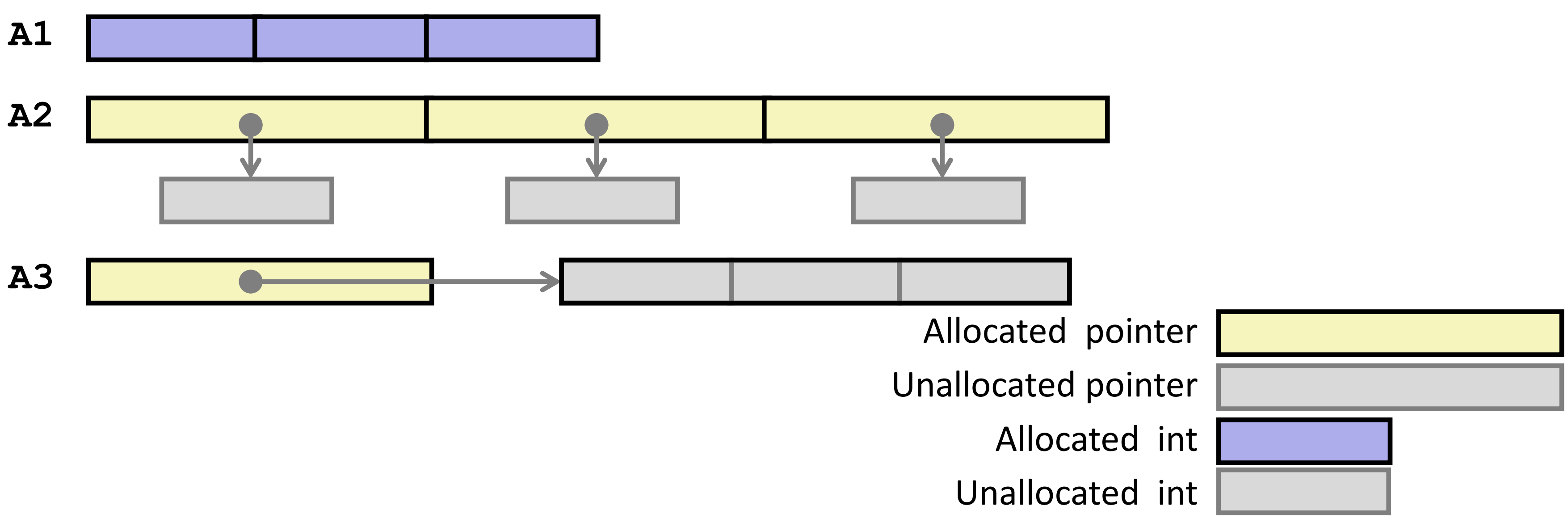
Decl	A1 , A2			*A1 , *A2		
	Comp	Bad	Size	Comp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4
`int *A2`	Y	N	8	Y	Y	4



- **Comp: Compiles (Y/N)**
- **Bad: Possible bad pointer reference (Y/N)**
- **Size: Value returned by `sizeof`**

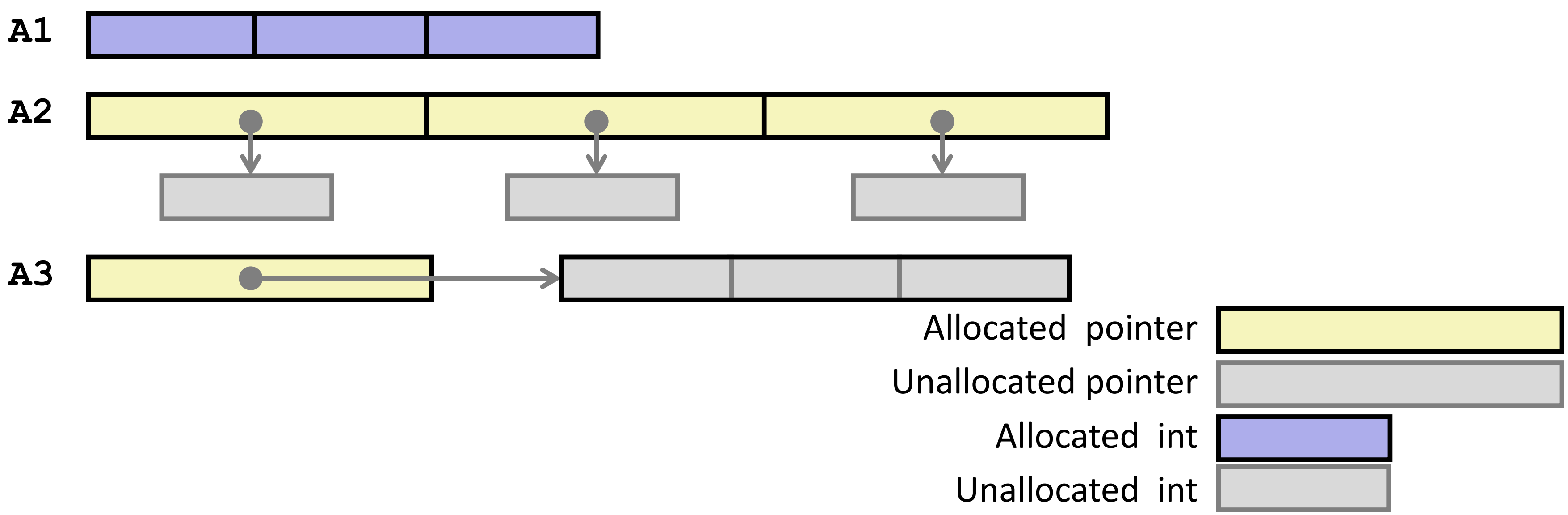
Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`									
`int *A2[3]`									
`int (*A3)[3]`									



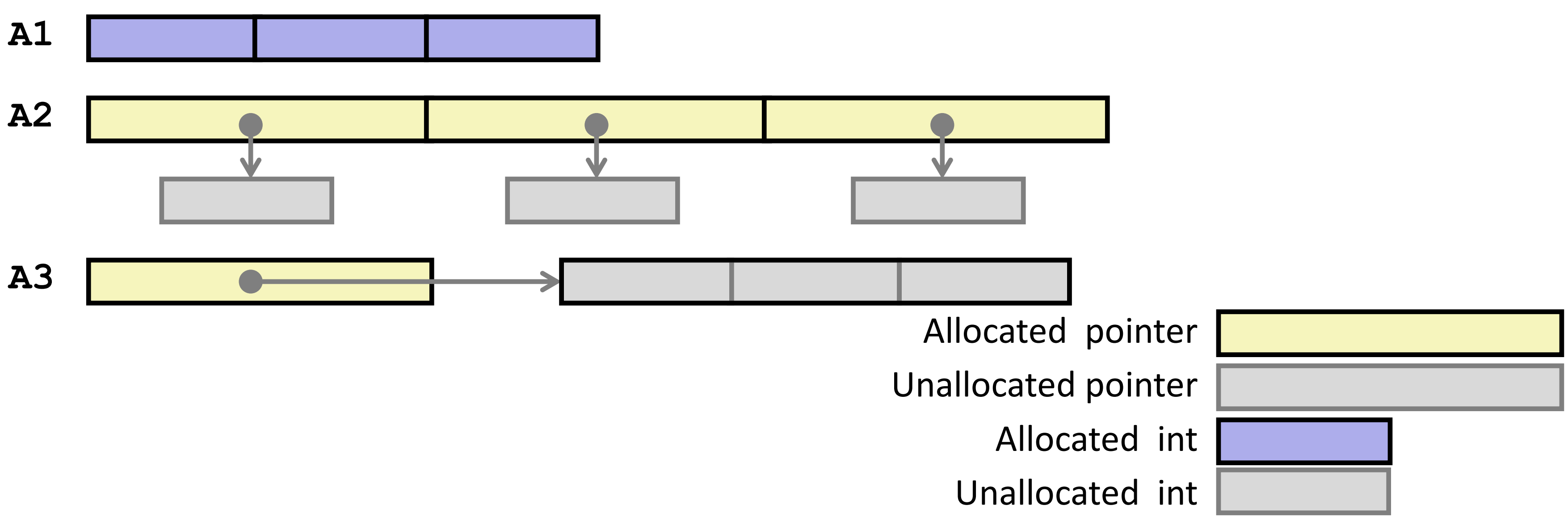
Understanding Pointers & Arrays #2

Decl	*An*			**An*			***An*		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12						
`int *A2[3]`									
`int (*A3)[3]`									



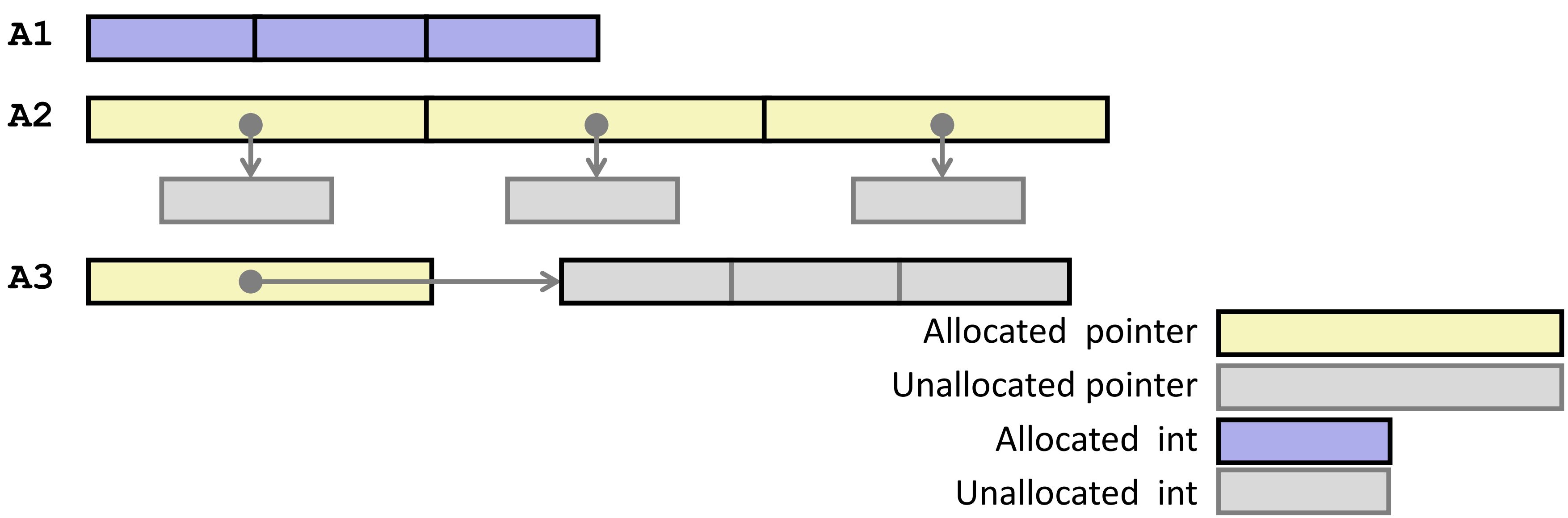
Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12						
`int *A2[3]`	Y	N	24						
`int (*A3)[3]`									



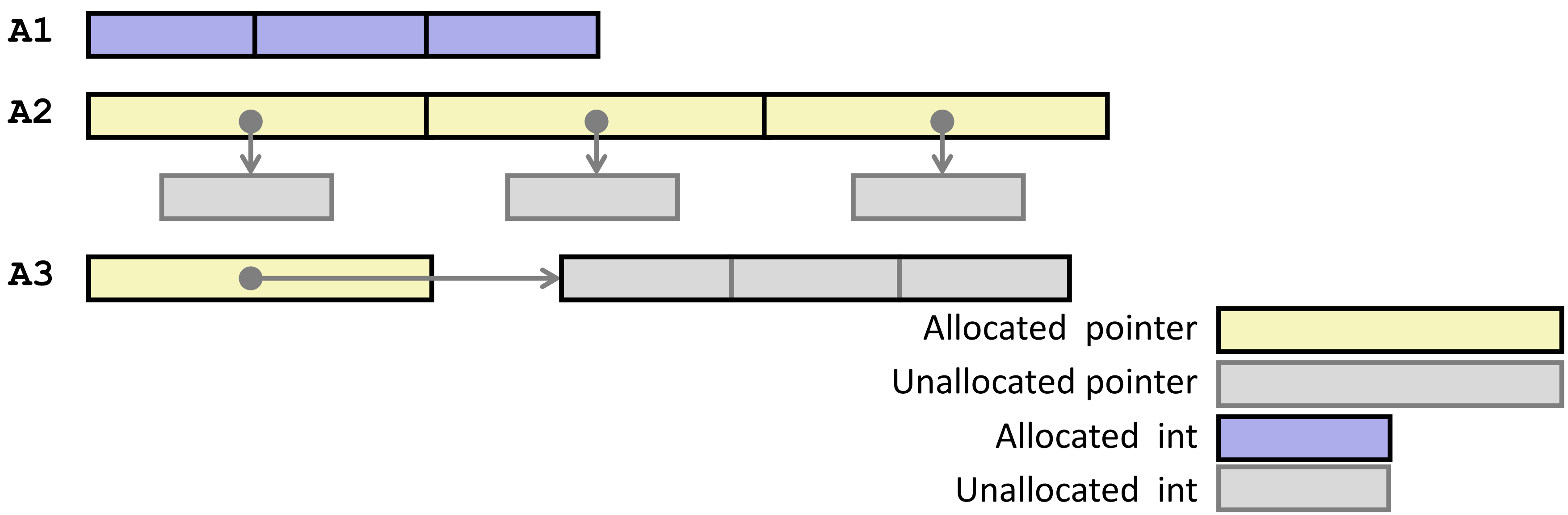
Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12						
`int *A2[3]`	Y	N	24						
`int (*A3)[3]`	Y	N	8						



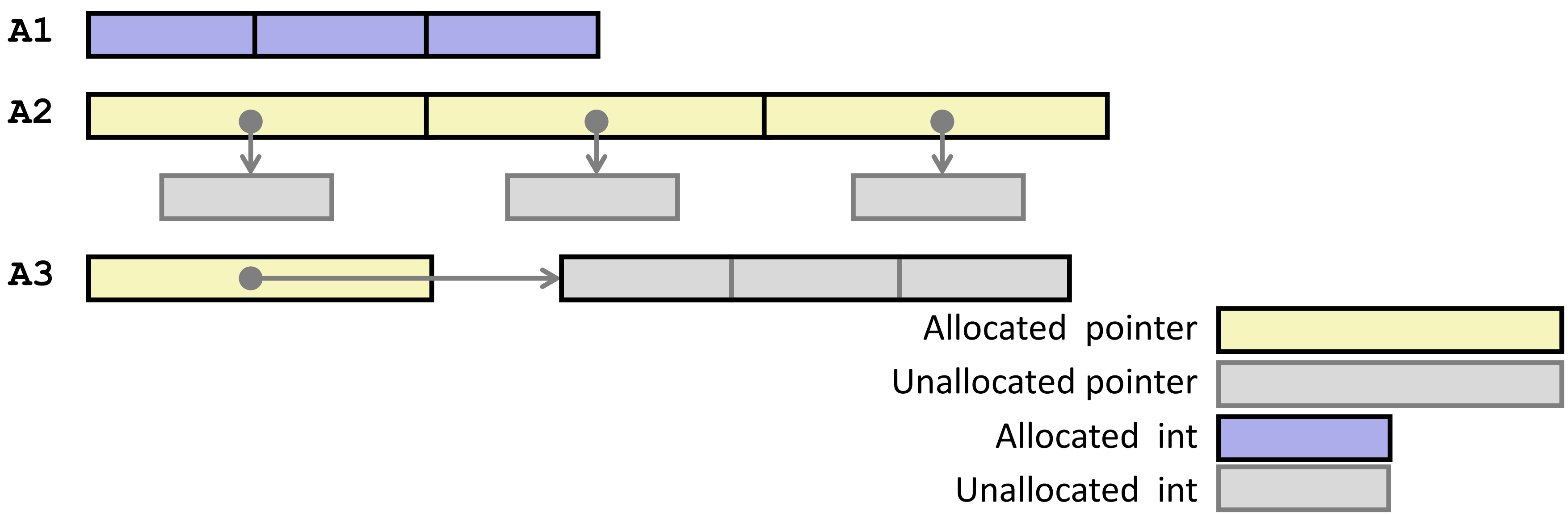
Understanding Pointers & Arrays #2

Decl	*An*			**An*			***An*		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4			
`int *A2[3]`	Y	N	24						
`int (*A3)[3]`	Y	N	8						



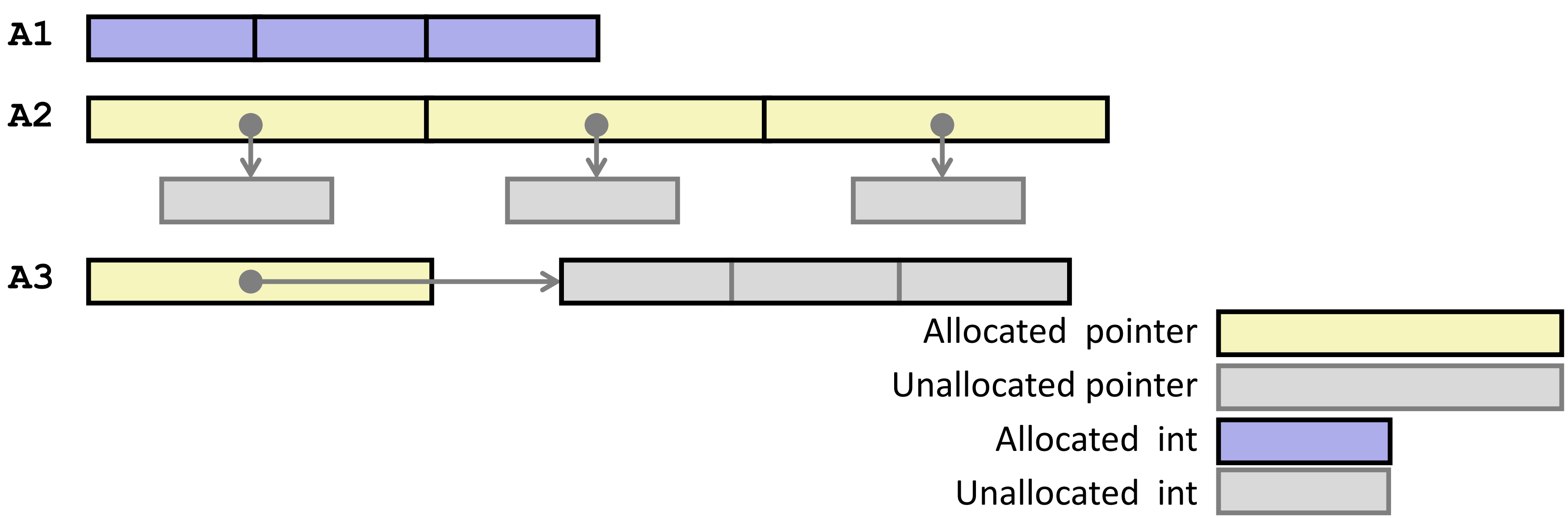
Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4			
`int *A2[3]`	Y	N	24	Y	N	8			
`int (*A3)[3]`	Y	N	8						



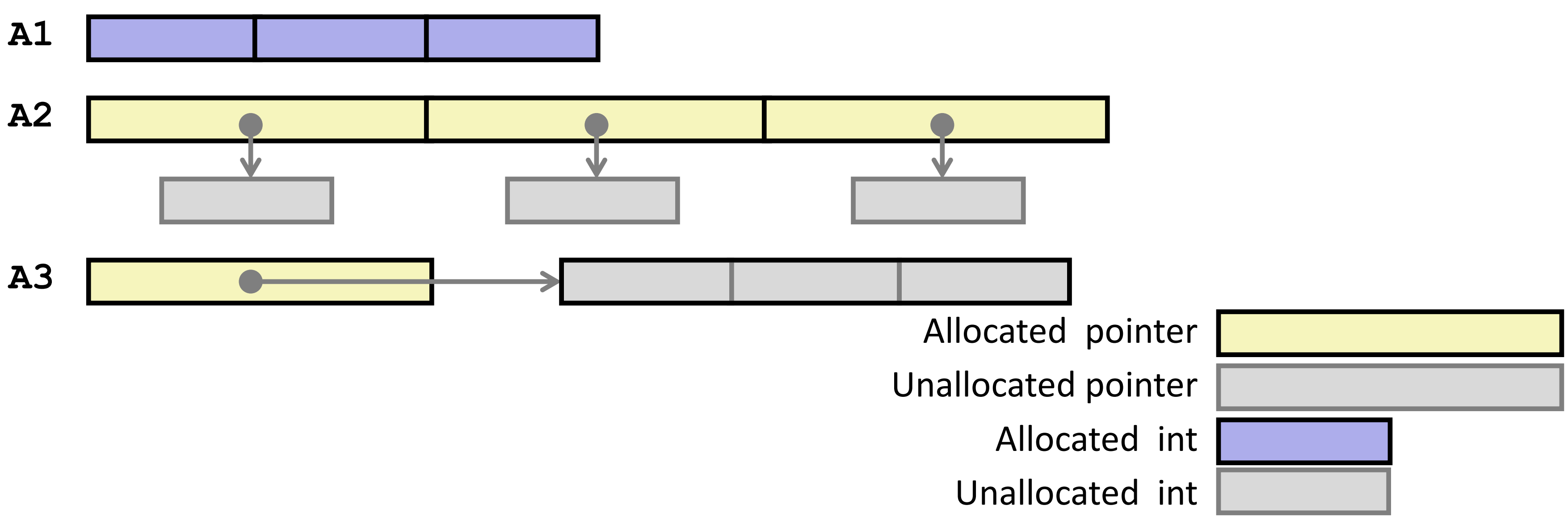
Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4			
`int *A2[3]`	Y	N	24	Y	N	8			
`int (*A3)[3]`	Y	N	8	Y	Y	12			



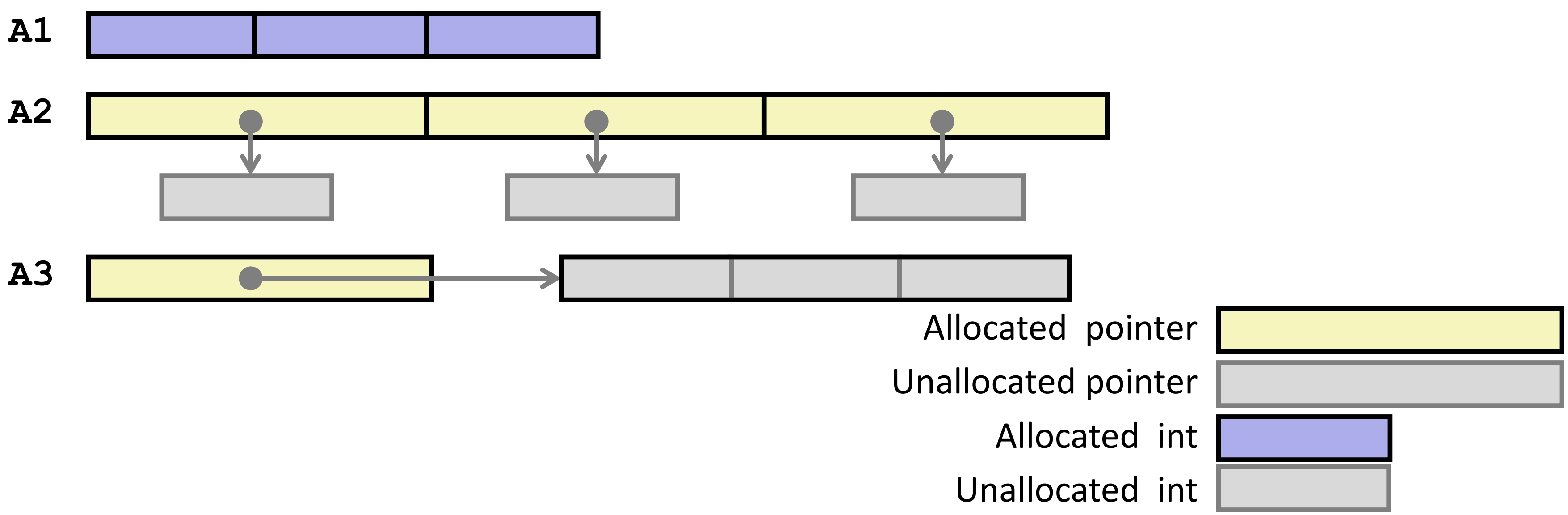
Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4	N	-	-
`int *A2[3]`	Y	N	24	Y	N	8			
`int (*A3)[3]`	Y	N	8	Y	Y	12			



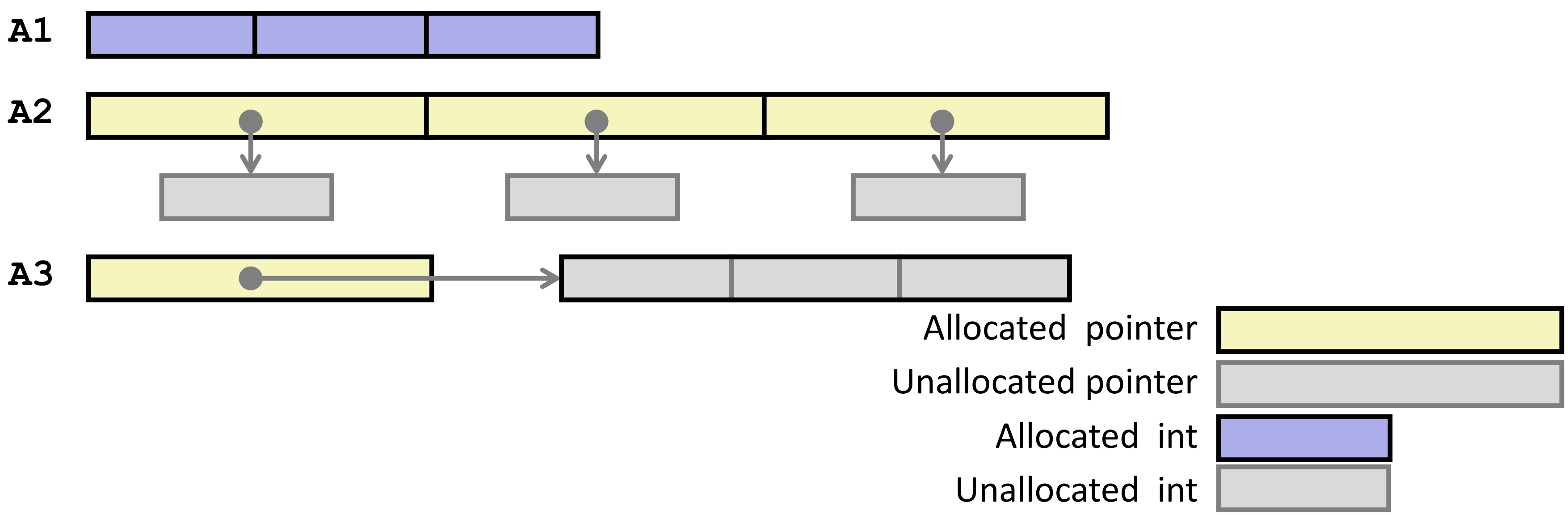
Understanding Pointers & Arrays #2

Decl	*An*			****An*			***An*		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4	N	-	-
`int *A2[3]`	Y	N	24	Y	N	8	Y	Y	4
`int (*A3)[3]`	Y	N	8	Y	Y	12			



Understanding Pointers & Arrays #2

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
`int A1[3]`	Y	N	12	Y	N	4	N	-	-
`int *A2[3]`	Y	N	24	Y	N	8	Y	Y	4
`int (*A3)[3]`	Y	N	8	Y	Y	12	Y	Y	4



Multidimensional (Nested) Arrays

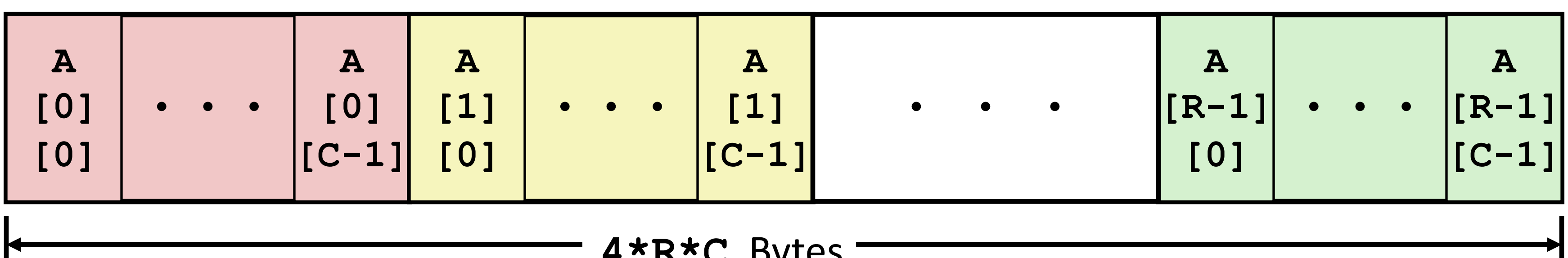
- **Declaration**

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

  - 2D array of data type *T*
  - *R* rows, *C* columns
- **Array Size**
  - *R* * *C* * **`sizeof`**(*T*) bytes
- **Arrangement**
  - Row-Major Ordering

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

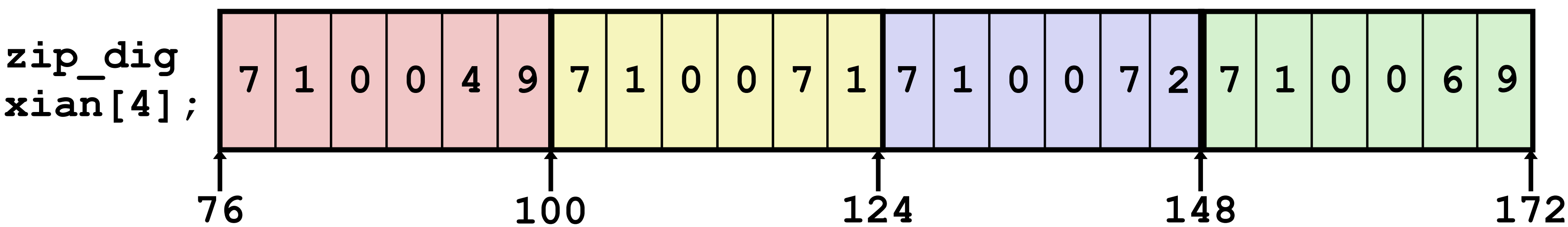
```
int A[R][C];
```



Nested Array Example

```
#define XCOUNT 4
typedef int zip_dig[6];

zip_dig xian[XCOUNT] =
  {{7, 1, 0, 0, 4, 9 },
   {7, 1, 0, 0, 7, 1 },
   {7, 1, 0, 0, 7, 2 },
   {7, 1, 0, 0, 6, 9 }};
```

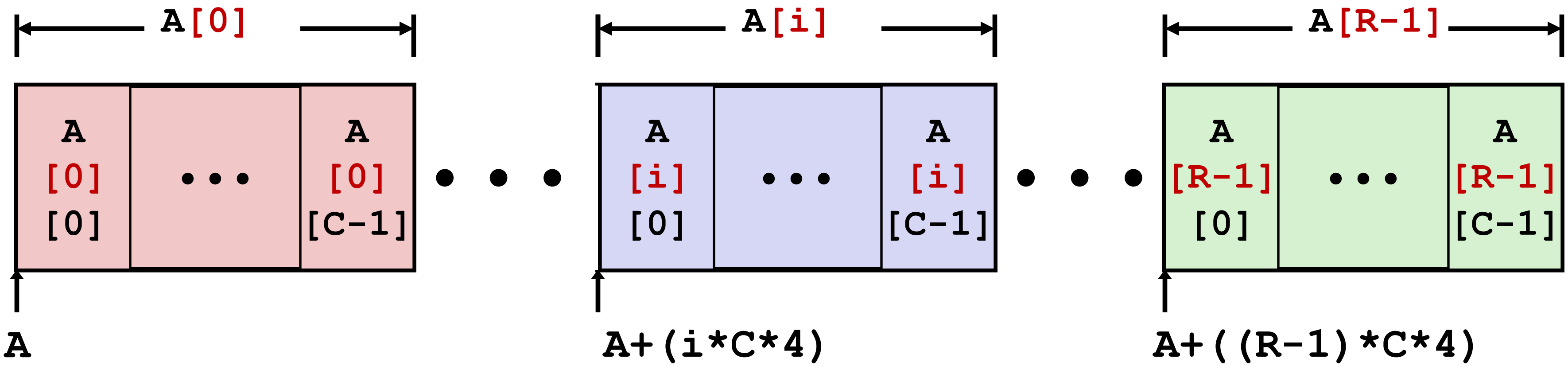


- "`zip_dig xian[4]`" equivalent to "`int xian[4][6]`"
  - Variable **`xian`**: array of 4 elements, allocated contiguously
  - Each element is an array of 6 **`int`**'s, allocated contiguously
- "Row-Major" ordering of all elements in memory

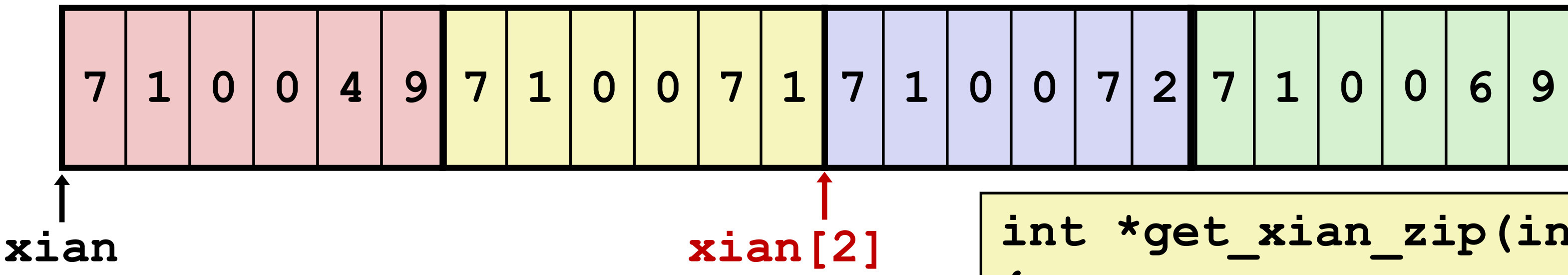
Nested Array Row Access

- **Row Vectors**
  - `A[i]` is array of *C* elements
  - Each element of type *T* requires *sizeof(T)* bytes
  - Starting address `A + i * (C * sizeof(T))`

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



Nested Array Row Access Code




```
int *get_xian_zip(int index)
{
  return xian[index];
}
```

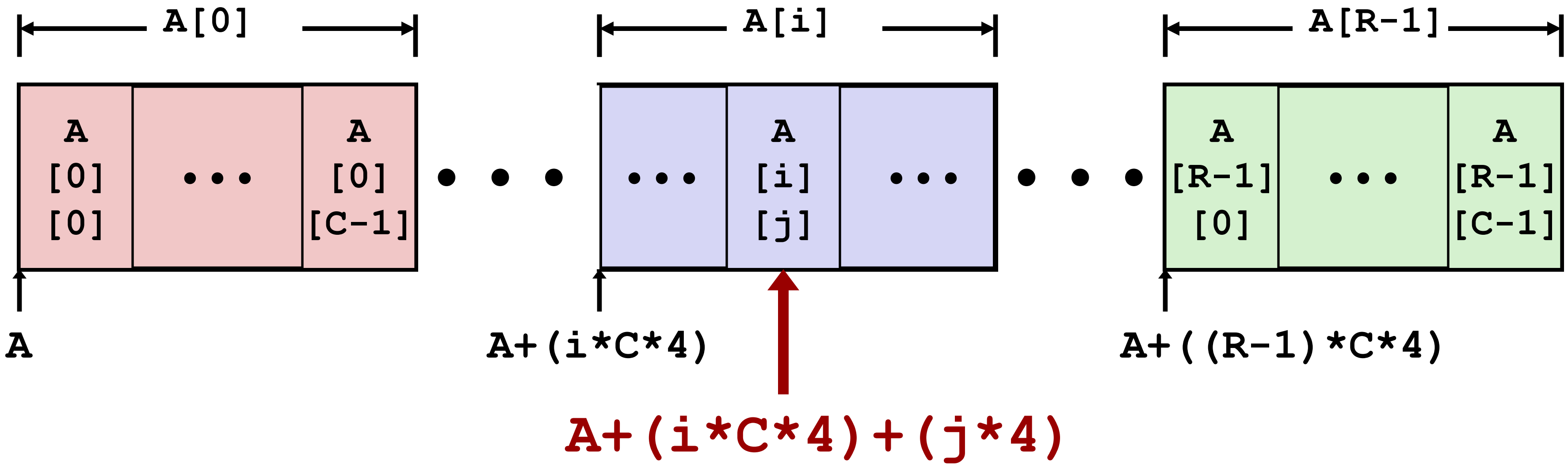
```
 # %rdi = index
  leaq (%rdi,%rdi,2),%rax # 3 * index
  leaq xian(,%rax,8),%rax # xian + (24 * index)
```

- **Row Vector**
  - `xian[index]` is array of 6 `int`'s
  - Starting address `xian+24*index`
- **Machine Code**
  - Computes and returns address
  - Compute as `xian + 8*(index+2*index)`

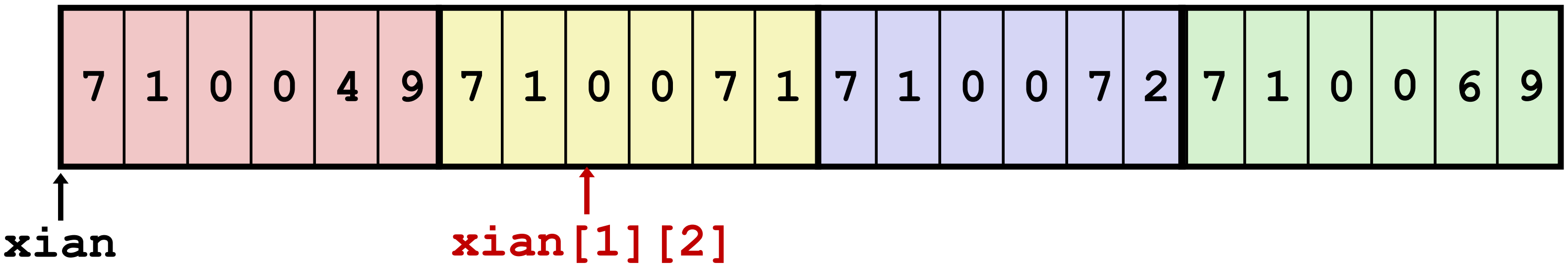
Nested Array Element Access

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

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



Nested Array Element Access Code




```
int get_xian_digit(int index, int dig)
{
  return xian[index][dig];
}
```

```
leaq   (%rdi,%rdi,2), %rax    # 3*index
leaq   (%rsi,%rax,2), %rsi    # 6*index+dig
movl   xian(,%rsi,4), %eax    # xian + 4*(6*index+dig)
```

- **Array Elements**
  - `xian[index][dig]` is `int`
  - Address: `xian + 24*index + 4*dig`

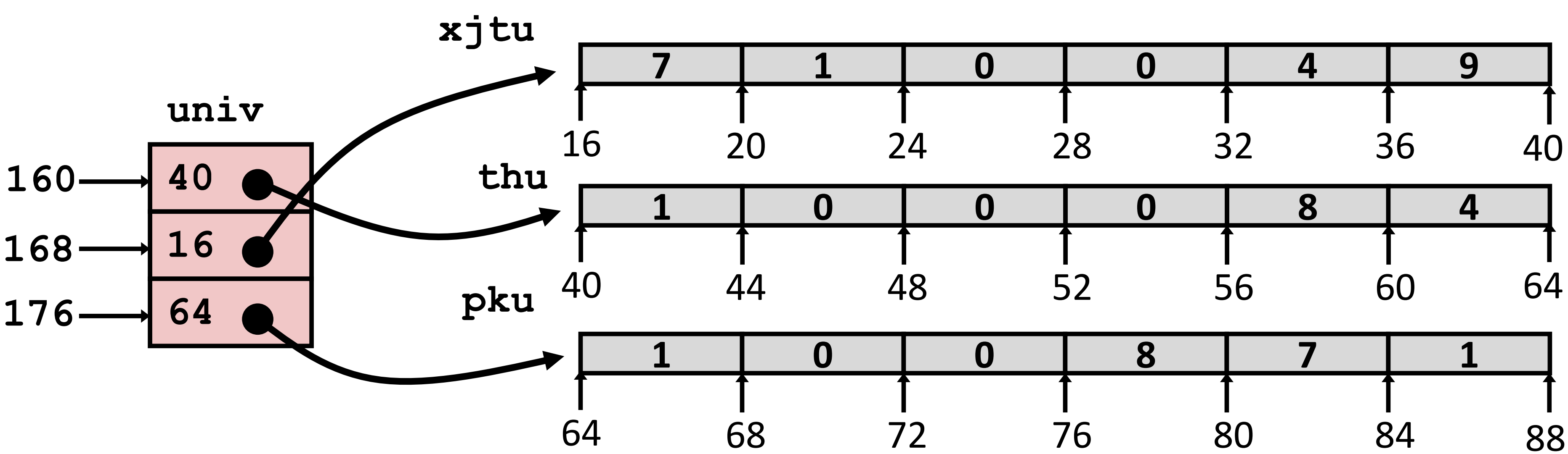
    `= xian + 4*(6*index + dig)`

Multi-Level Array Example

```
zip_dig xjtu = { 7, 1, 0, 0, 4, 9 };
zip_dig thu  = { 1, 0, 0, 0, 8, 4 };
zip_dig pku  = { 1, 0, 0, 8, 7, 1 };
```

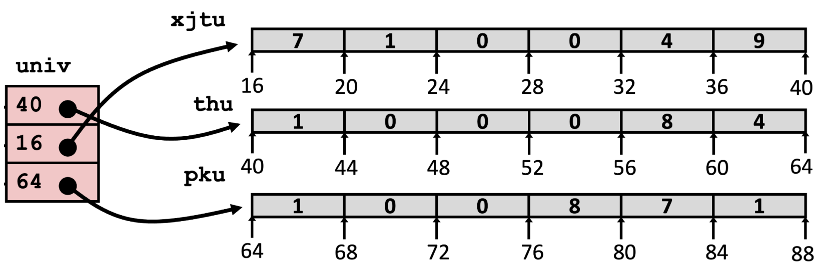
```
#define UCOUNT 3
int *univ[UCOUNT] = {thu, xjtu, pku};
```

- Variable `univ` denotes 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_univ_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

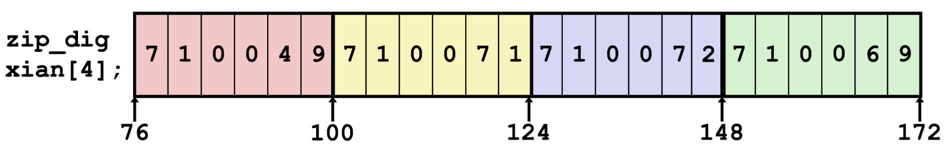
Array Element Accesses

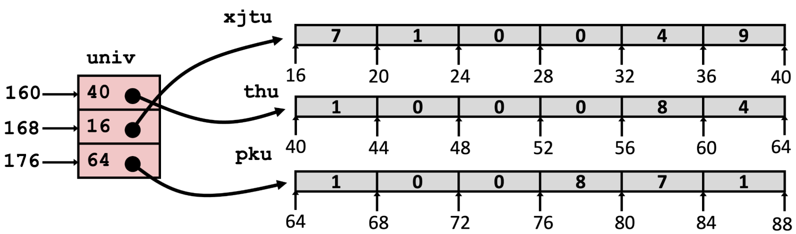
Nested array

```
int get_xian_digit
   (size_t index, size_t digit)
{
  return xian[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[xian+24*index+4*digit]` `Mem[Mem[univ+8*index]+4*digit]`

N X *N* Matrix Code

- **Fixed dimensions**
  - Know value of *N* at compile time

```
#define N 16
typedef int fix_matrix[N][N];
/* Get element A[i][j] */
int fix_ele(fix_matrix A,
            size_t i, size_t j)
{
  return A[i][j];
}
```

- **Variable dimensions, explicit indexing**
  - Traditional way to implement dynamic arrays

```
#define IDX(n, i, j) ((i)*(n)+(j))
/* Get element A[i][j] */
int vec_ele(size_t n, int *A,
            size_t i, size_t j)
{
  return A[IDX(n,i,j)];
}
```

- **Variable dimensions, implicit indexing**
  - Added to language in 1999

```
/* Get element A[i][j] */
int var_ele(size_t n, int A[n][n],
            size_t i, size_t j) {
  return A[i][j];
}
```

16 X 16 Matrix Access

- Array Elements
  - `int A[16][16];`
  - Address `A + i * (C * sizeof(int)) + j * sizeof(int)`
  - C = 16, sizeof(int) = 4

```
/* Get element A[i][j] */
int fix_ele(fix_matrix A, size_t i, size_t j) {
  return A[i][j];
}
```

```
  # A in %rdi, i in %rsi, j in %rdx
  salq    $6, %rsi               # 64*i
  addq    %rsi, %rdi             # A + 64*i
  movl    (%rdi,%rdx,4), %eax    # Mem[A + 64*i + 4*j]
  ret
```

n X *n* Matrix Access

- **Array Elements**
  - `size_t n;`
  - `int A[n][n];`
  - Address `A + i * (C * sizeof(int)) + j * sizeof(int)`
  - C = n, sizeof(int) = 4
  - Must perform integer multiplication

```
/* Get element A[i][j] */
int var_ele(size_t n, int A[n][n], size_t i, size_t j)
{
  return A[i][j];
}
```

```
  # n in %rdi, A in %rsi, i in %rdx, j in %rcx
  imulq   %rdx, %rdi               # n*i
  leaq    (%rsi,%rdi,4), %rax      # A + 4*n*i
  movl    (%rax,%rcx,4), %eax      # Mem[A + 4*n*i + 4*j]
  ret
```

Example: Array Access

```
#include <stdio.h>
#define ZLEN 6
#define XCOUNT 4
typedef int zip_dig[ZLEN];

int main(int argc, char** argv) {
zip_dig xian[XCOUNT] =
    {{7, 1, 0, 0, 4, 9},
     {7, 1, 0, 0, 7, 1 },
     {7, 1, 0, 0, 7, 2 },
     {7, 1, 0, 0, 6, 9 }};
    int *linear_zip = (int *) xian;
    int *zip2 = (int *) xian[2];
    int result =
        xian[0][0] +
        linear_zip[8] +
        *(linear_zip + 10) +
        zip2[1];
    printf("result: %d\n", result);
    return 0;
}
```

```
linux> ./array
```

Example: Array Access

```
#include <stdio.h>
#define ZLEN 6
#define XCOUNT 4
typedef int zip_dig[ZLEN];

int main(int argc, char** argv) {
zip_dig xian[XCOUNT] =
    {{7, 1, 0, 0, 4, 9},
     {7, 1, 0, 0, 7, 1 },
     {7, 1, 0, 0, 7, 2 },
     {7, 1, 0, 0, 6, 9 }};
    int *linear_zip = (int *) xian;
    int *zip2 = (int *) xian[2];
    int result =
        xian[0][0] +
        linear_zip[8] +
        *(linear_zip + 10) +
        zip2[1];
    printf("result: %d\n", result);
    return 0;
}
```

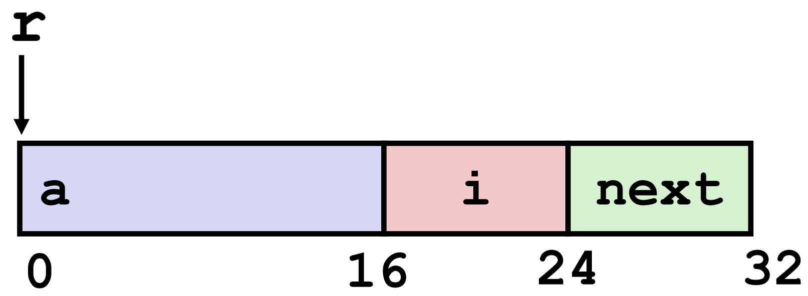
```
linux> ./array
result: 15
```

Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**
  - Allocation
  - Access
  - Alignment
- **If we have time: Union**

Structure Representation

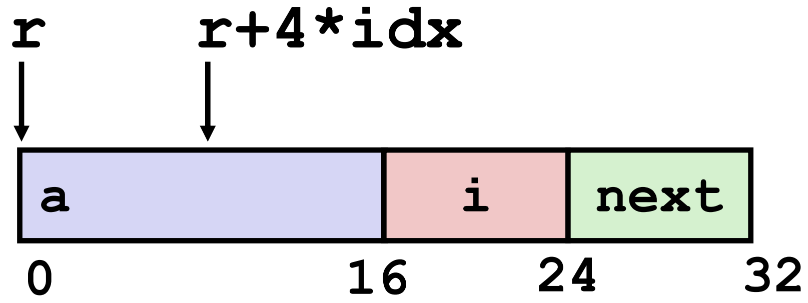
```
struct rec {
    int a[4];
    size_t i;
    struct rec *next;
};
```



- **Structure represented as block of memory**
  - Big enough to hold all the fields
- **Fields ordered according to declaration**
  - Even if another ordering could be more compact
- **Compiler determines overall size + positions of fields**
  - In assembly, we see only offsets, not field names

Generating Pointer to Structure Member

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



- **Generating Pointer to Array Element**
  - Offset of each structure member determined at compile time
  - Compute as `r + 4*idx`

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

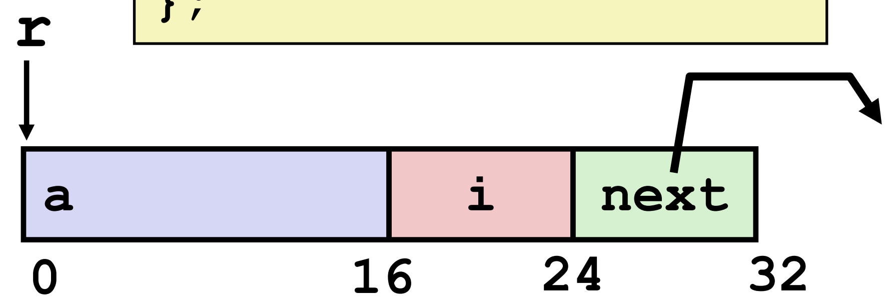
```
  # r in %rdi, idx in %rsi
  leaq  (%rdi,%rsi,4), %rax
  ret
```

Following Linked List #1

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

- C Code

```
long length(struct rec*r) {
    long len = 0L;
    while (r) {
        len ++;
        r = r->next;
    }
    return len;
}
```



Register	Value
%rdi	r
%rax	len

- Loop assembly code

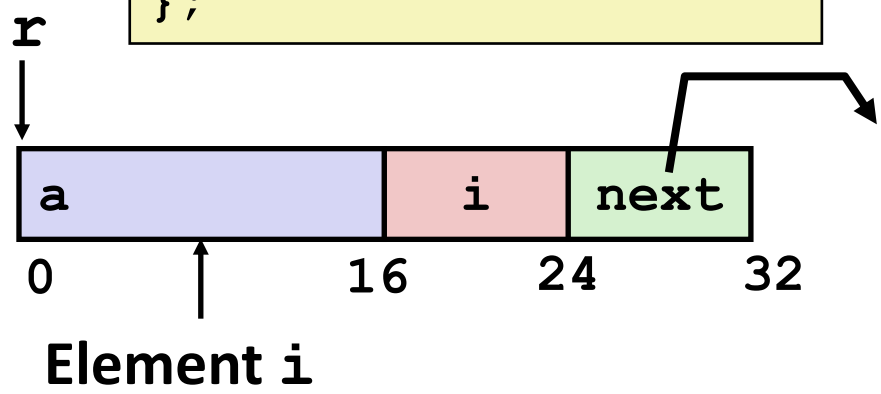
```
.L11:                         # loop:
  addq    $1, %rax            #   len ++
  movq    24(%rdi), %rdi      #   r = Mem[r+24]
  testq   %rdi, %rdi          #   Test r
  jne     .L11                #   If != 0, goto loop
```

Following Linked List #2

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

- C Code

```
void set_val
  (struct rec *r, int val)
{
  while (r) {
    size_t i = r->i;
    // No bounds check
    r->a[i] = val;
    r = r->next;
  }
}
```

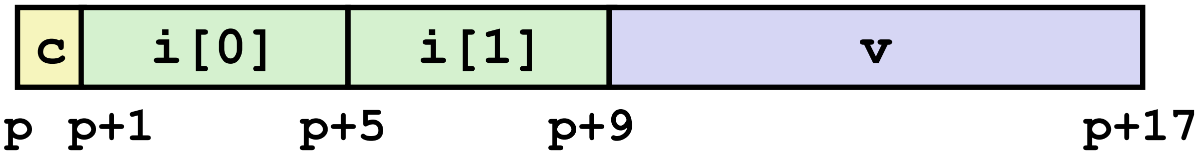


Register	Value
%rdi	r
%rsi	val

```
.L11:                           # loop:
  movq   16(%rdi), %rax         #   i = Mem[r+16]
  movl   %esi, (%rdi,%rax,4)    #   Mem[r+4*i] = val
  movq   24(%rdi), %rdi         #   r = Mem[r+24]
  testq  %rdi, %rdi             #   Test r
  jne    .L11                   #   if !=0 goto loop
```

Structures & Alignment

Unaligned Data

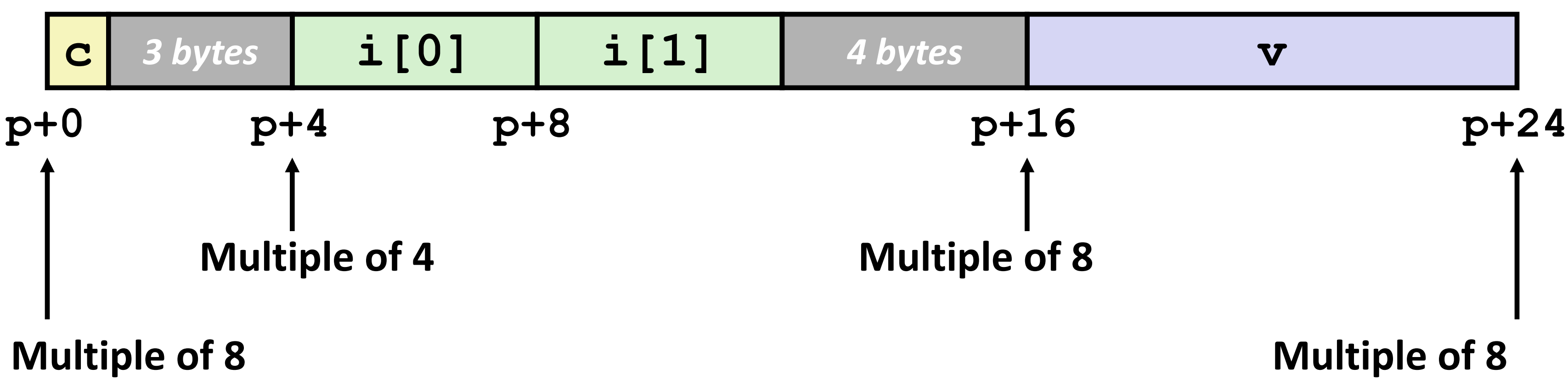



```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```

Aligned Data

Primitive data type requires ***K*** bytes

Address must be multiple of ***K***



Alignment Principles

- **Aligned Data**
  - Primitive data type requires B bytes
  - Address must be multiple of B
  - 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 quad word boundaries (8 bytes)
    - Inefficient to load or store datum that spans cache lines (64 bytes). Intel states should avoid crossing 16 byte boundaries.
    - Virtual memory trickier when datum spans 2 pages (4 KB pages)
- **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**

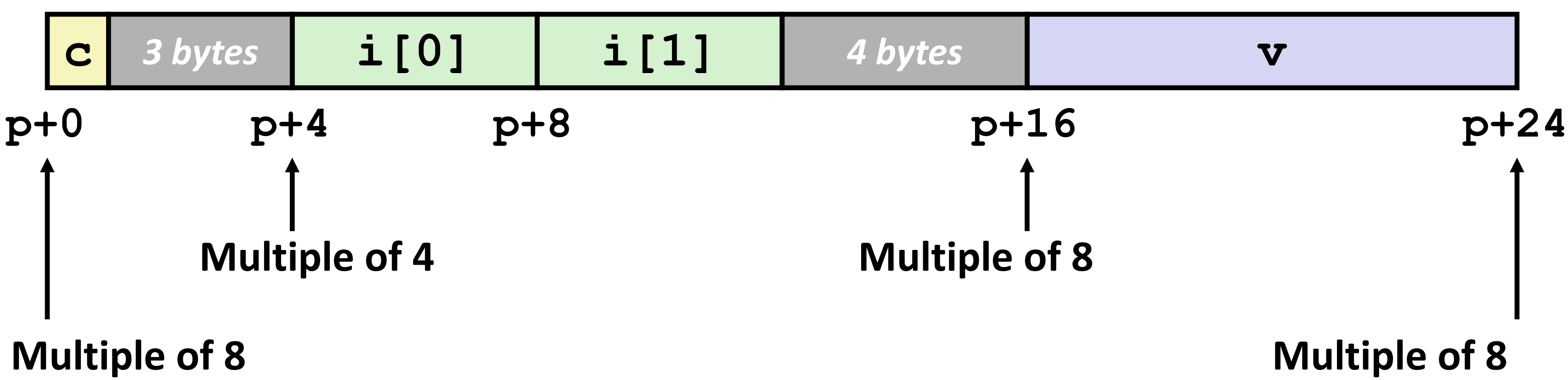
K = Largest alignment of any element

Initial address & structure length must be multiples of **K**

Example:

K = 8, due to **`double`** element

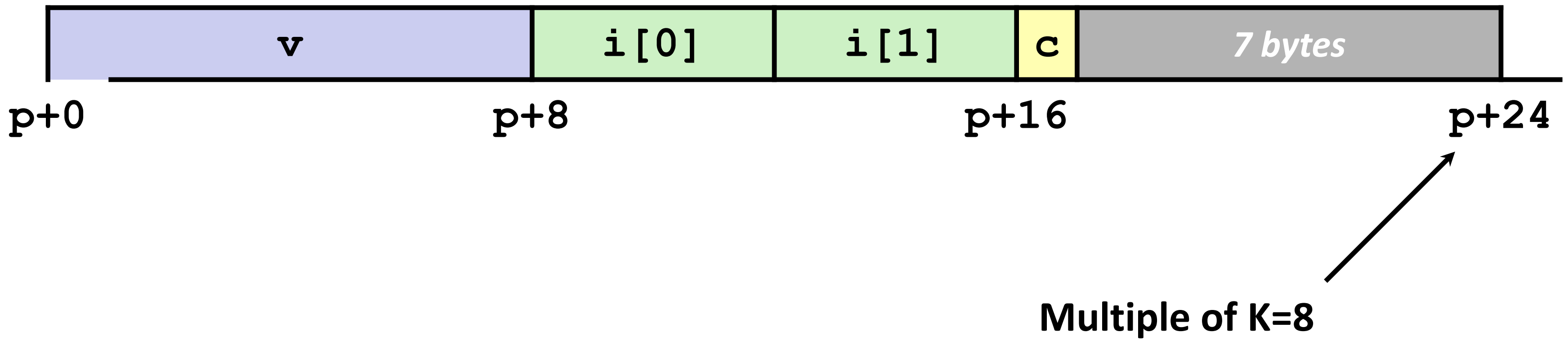
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Meeting Overall Alignment Requirement

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

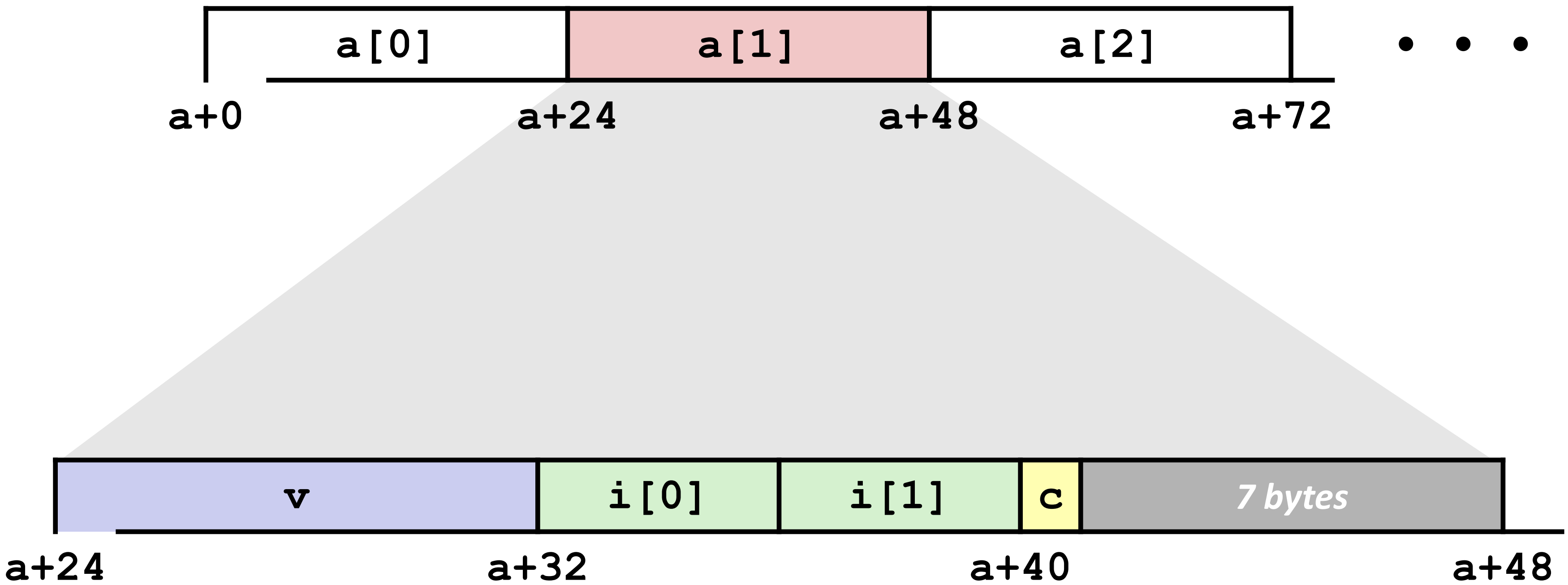
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures

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

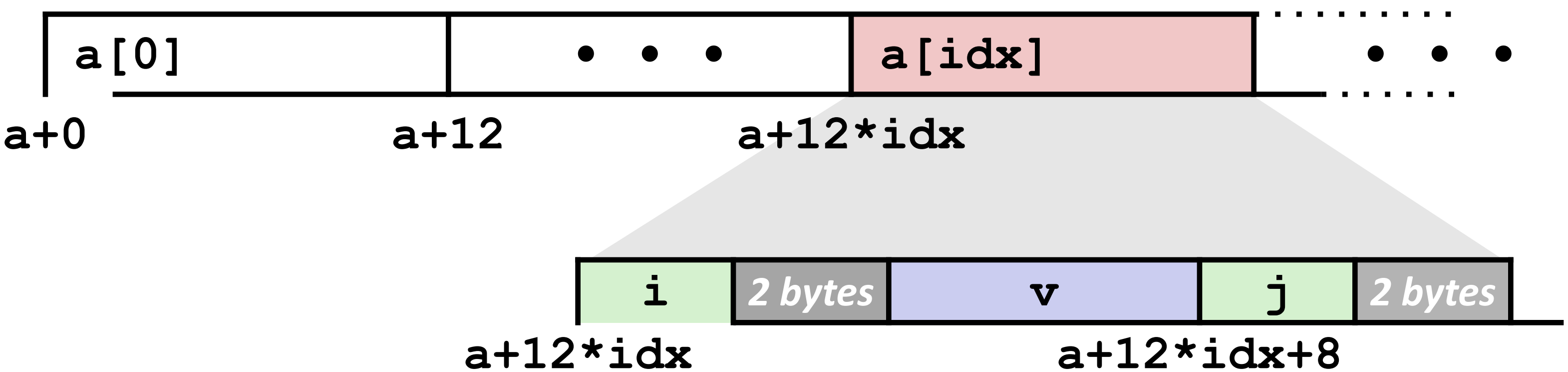
```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Accessing Array Elements

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

- Compute array offset 12*idx
  - `sizeof(S3)`, including alignment spacers
- Element `j` is at offset 8 within structure
- Assembler gives offset `a+8`
  - Resolved during linking




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

```
# %rdi = idx
leaq (%rdi,%rdi,2),%rax # 3*idx
movzwl a+8(,%rax,4),%eax
```

Saving Space

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=4)

c	*3 bytes*	i	d	*3 bytes*

i	c	d	*2 bytes*

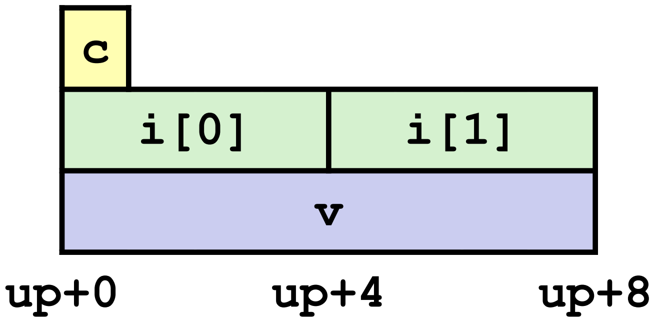
Today

- **Arrays**
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- **Structures**
  - Allocation
  - Access
  - Alignment
- **If we have time: Union**

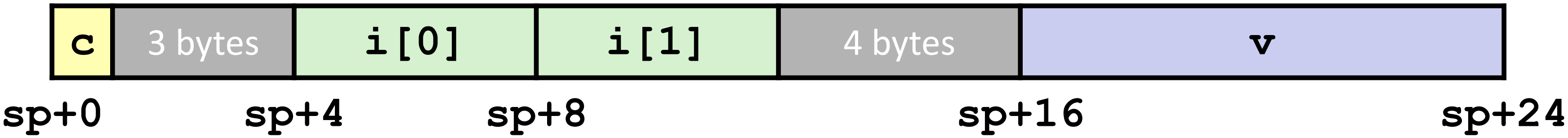
Union Allocation

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

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

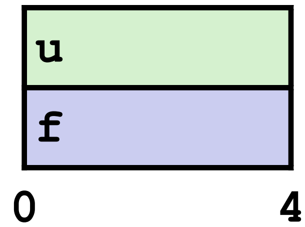



```
struct S1 {
  char c;
  int i[2];
  double v;
} *sp;
```



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;
}
```

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

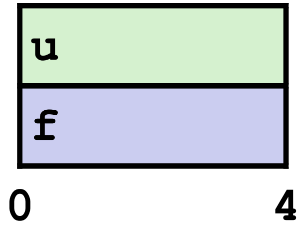
Same as `(float) u` ?

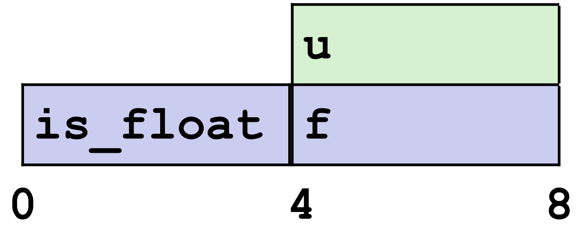
Same as `(unsigned) f` ?

Using Unions as Sum Types

```
typedef union {
  float f;
  unsigned u;
} num_t;

typedef struct {
  bool is_float;
  num_t val;
} value_t;
```





(technically is_float only takes 1 byte and then there's 3 bytes of padding)

Summary

- **Arrays**
  - Elements packed into contiguous region of memory
  - Aligned to satisfy every element's alignment requirement
  - Pointer to first element
  - Use index arithmetic to locate individual elements
  - No bounds checking
- **Structures**
  - Elements packed into single region of memory
  - Possible require internal and external padding to ensure alignment
  - Access using offsets determined by compiler
- **Unions**
  - Overlay declarations
  - Way to circumvent type system