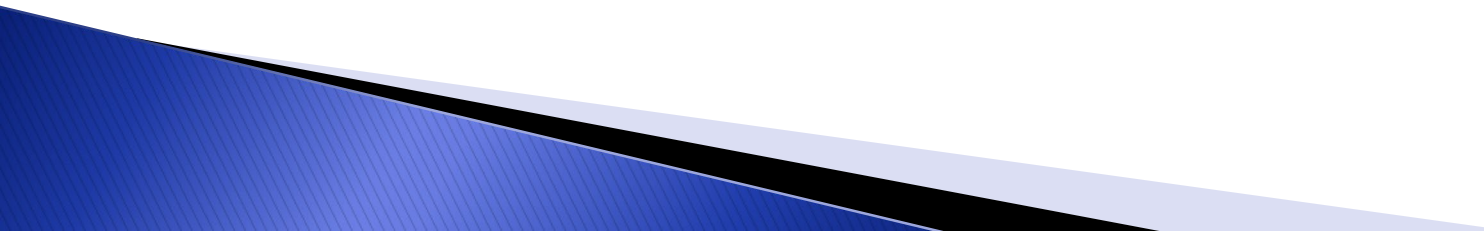


CSE 2421

Array and Structure Storage and Access

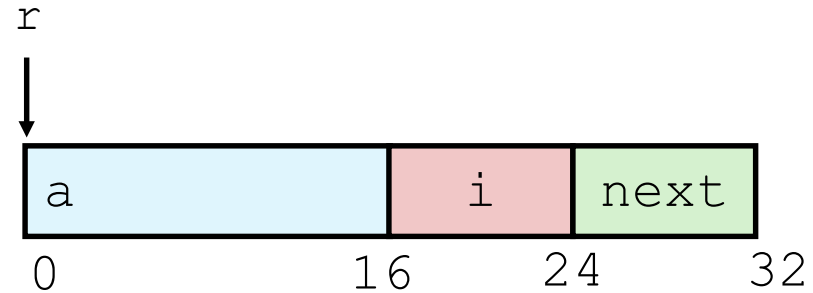


Today

- ▶ Arrays
 - One-dimensional
 - Multi-dimensional (nested)
 - Multi-level
 - ▶ Structures
 - Allocation
 - Access
 - Alignment
- 

Structure Representation

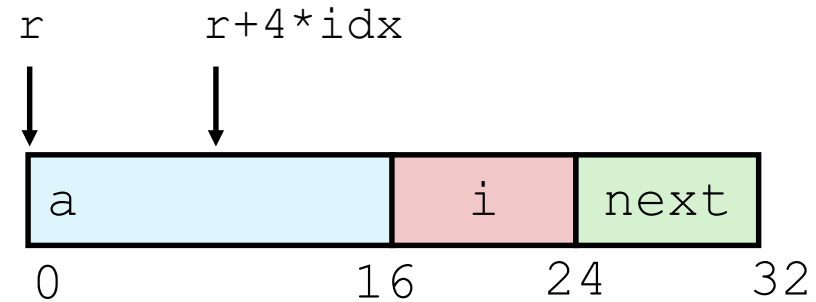
```
struct rec {  
    int a[4];  
    size_t i;  
    struct rec *next;  
};
```



- ▶ 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];  
    size_t i;  
    struct rec *next;  
};
```



► Generating Pointer to Array Element

- Offset of each structure member determined at compile time
- Compute as $r + 4 \cdot idx$

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

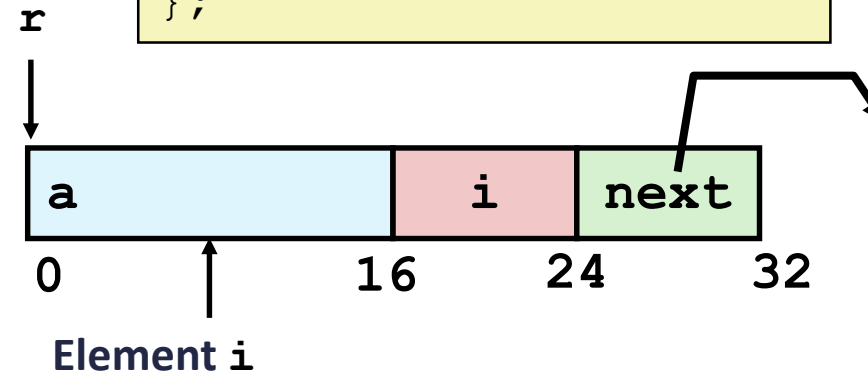
```
# r in %rdi, idx in %rsi  
leaq (%rdi,%rsi,4), %rax  
ret
```

Following Linked List

► C Code

```
void set_val
(struct rec *r, int val)
{
    while (r) {
        int i = r->i;
        r->a[i] = val;
        r = r->next;
    }
}
```

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

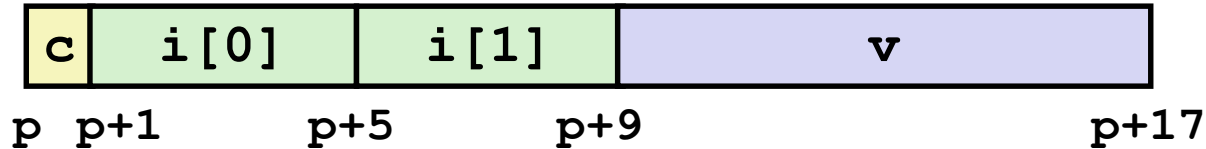


Register	Value
%rdi	r
%rsi	val

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

Structures & Alignment

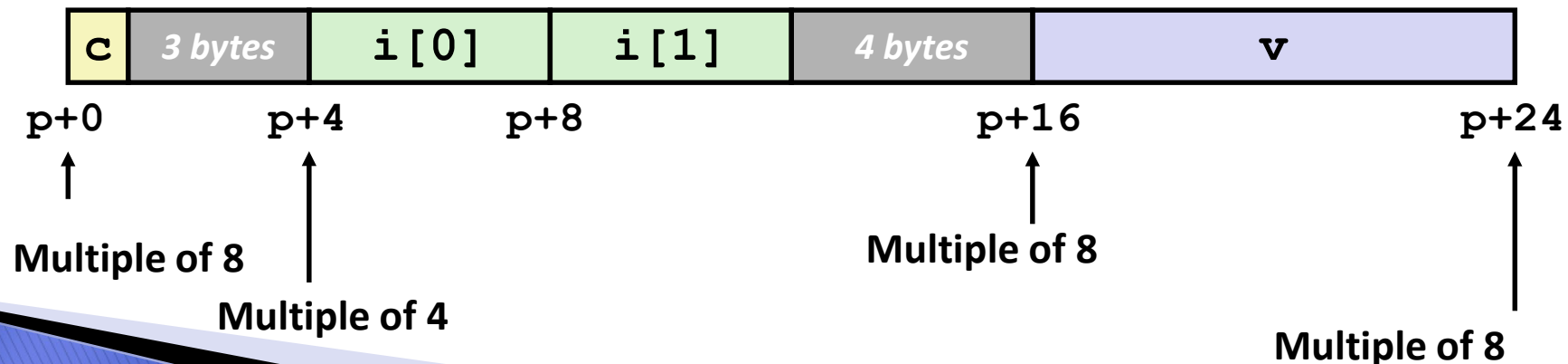
▶ Unaligned Data



```
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 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 quad word boundaries
 - Virtual memory trickier when datum spans 2 pages

▶ Compiler

- Inserts gaps in structure to ensure correct alignment of fields

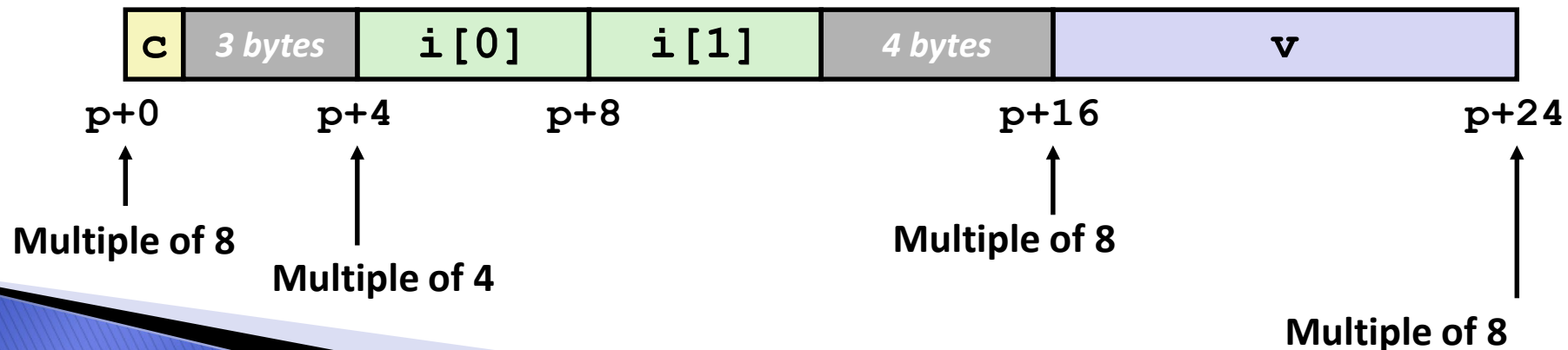
Specific Cases of Alignment (x86-64)

- ▶ 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
- ▶ 16 bytes: **long double** (GCC on Linux)
 - lowest 4 bits of address must be 0000_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

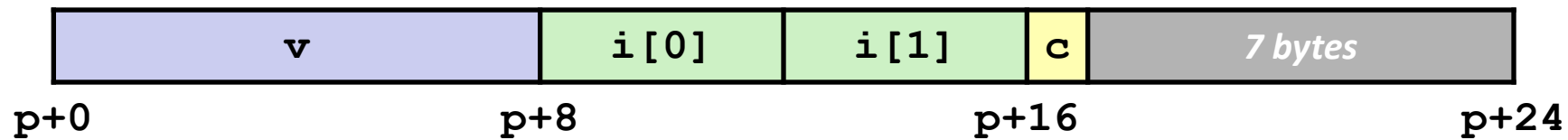
```
struct S1 {  
    char c;  
    int i[2];  
    double v;  
} *p;
```



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;
```

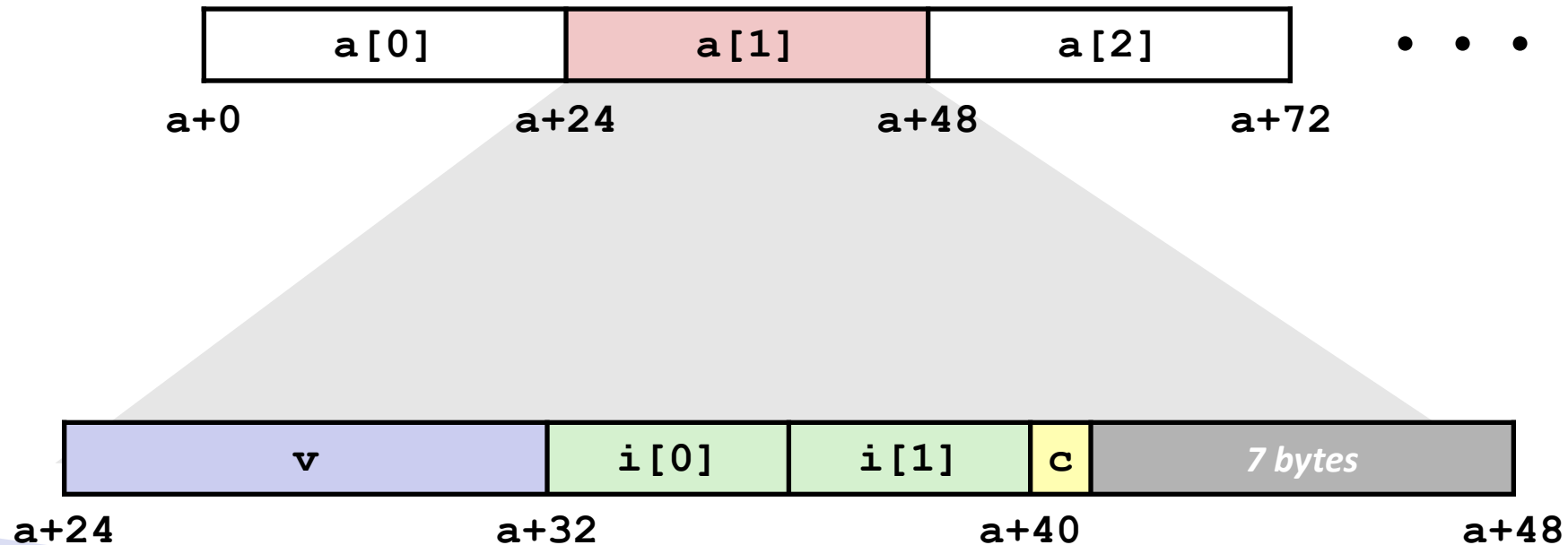


Multiple of $K=8$

Arrays of Structures

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

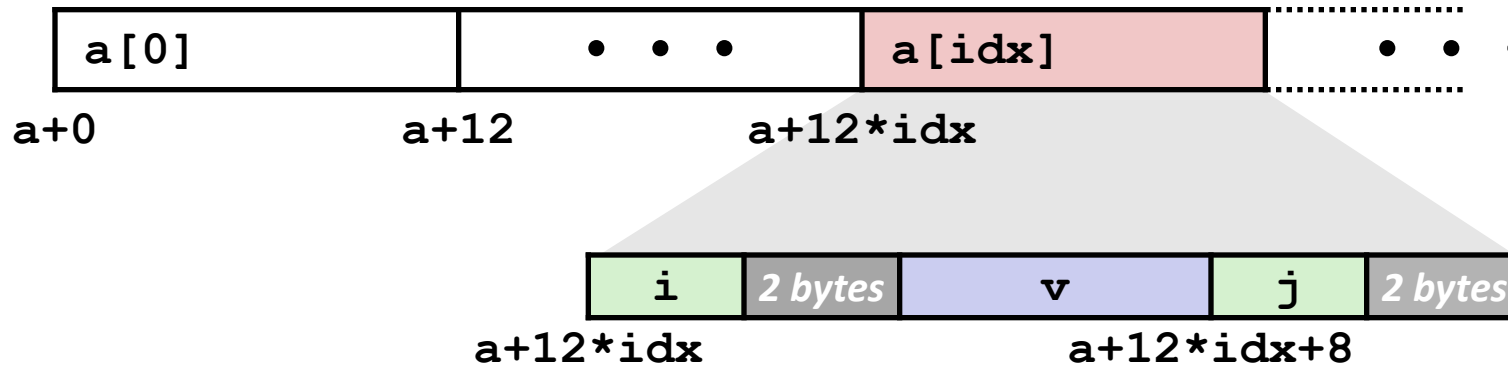
```
struct S2 {  
    double v;  
    int i[2];  
    char c;  
} a[10];
```



Accessing Array Elements

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

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



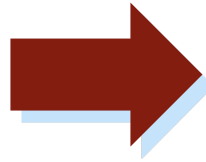
```
short get_j(int idx)  
{  
    return a[idx].j;  
}
```

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

Saving Space

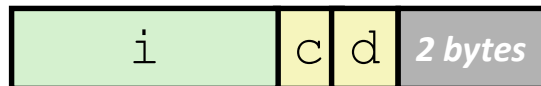
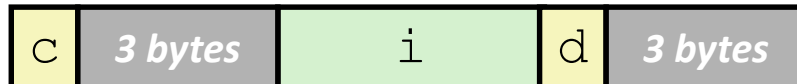
- ▶ Put large data types first

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



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

- ▶ Effect (K=4)



Summary

▶ Arrays

- Elements packed into contiguous region of memory
- Use index arithmetic to locate individual elements

▶ Structures

- Elements packed into single region of memory
- Access using offsets determined by compiler
- Possible require internal and external padding to ensure alignment

▶ Combinations

- Can nest structure and array code arbitrarily

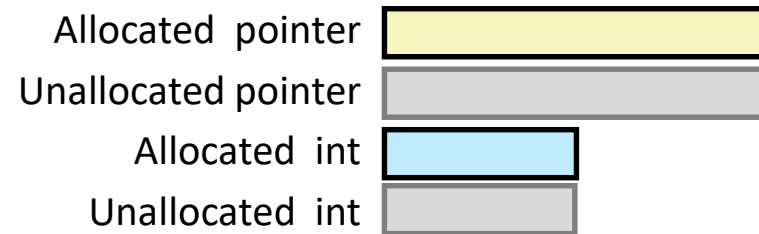
Understanding Pointers & Arrays #1

Decl	<i>An</i>			<i>*An</i>		
	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3]</code>						
<code>int *A2</code>						

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

Understanding Pointers & Arrays #1

Decl	An			*An		
	Cmp	Bad	Size	Cmp	Bad	Size
int A1[3]	Y	N	12	Y	N	4
int *A2	Y	N	8	Y	Y	4



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

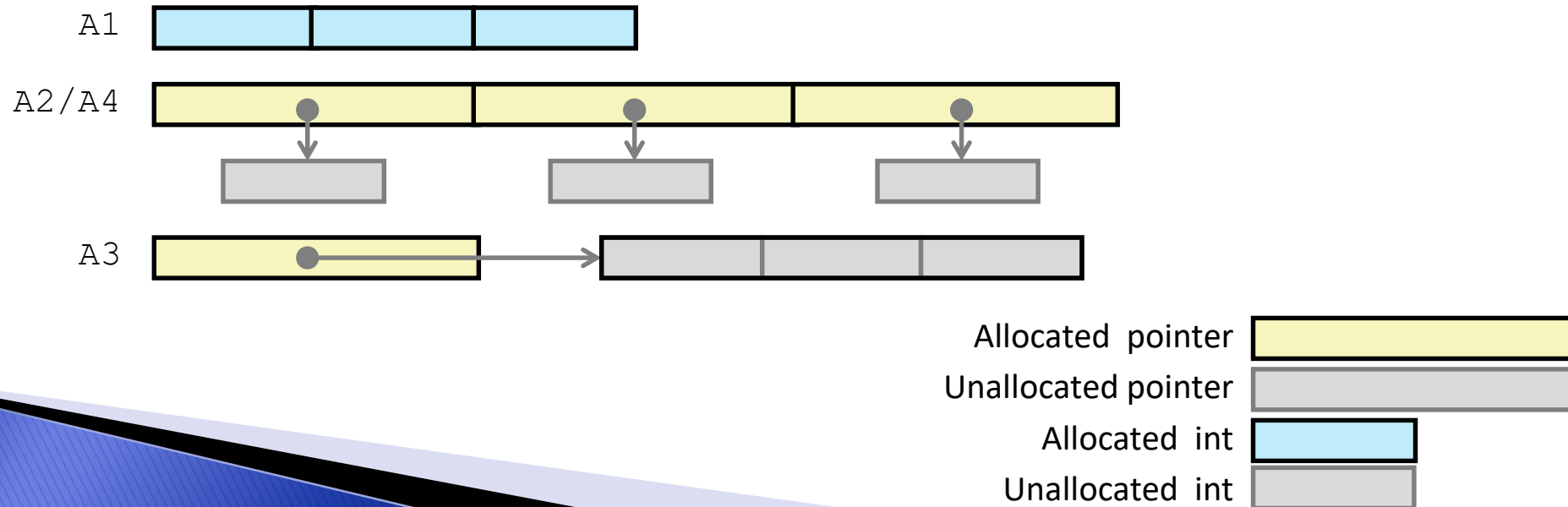
Understanding Pointers & Arrays #2

Decl	<i>A_n</i>			<i>*A_n</i>			<i>**A_n</i>		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3]</code>									
<code>int *A2[3]</code>									
<code>int (*A3)[3]</code>									
<code>int (*A4[3])</code>									

- ▶ Cmp: 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]	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
int (*A4[3])	Y	N	24	Y	N	8	Y	Y	4

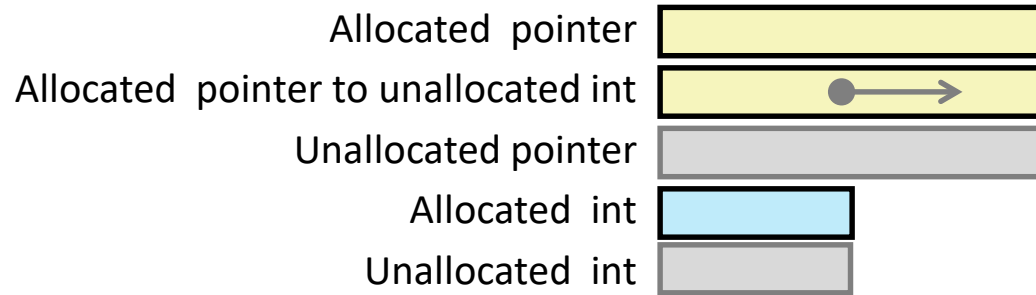


Understanding Pointers & Arrays #3

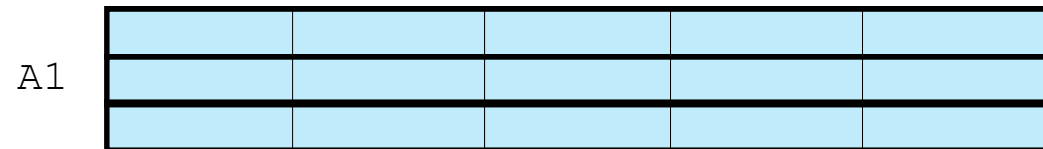
Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
<code>int A1[3][5]</code>									
<code>int *A2[3][5]</code>									
<code>int (*A3)[3][5]</code>									
<code>int *(A4[3][5])</code>									
<code>int (*A5[3])[5]</code>									

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

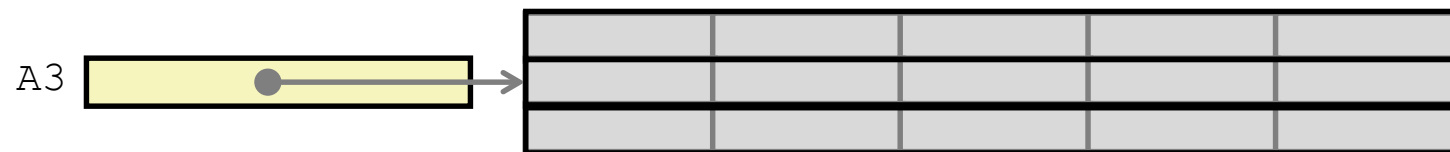
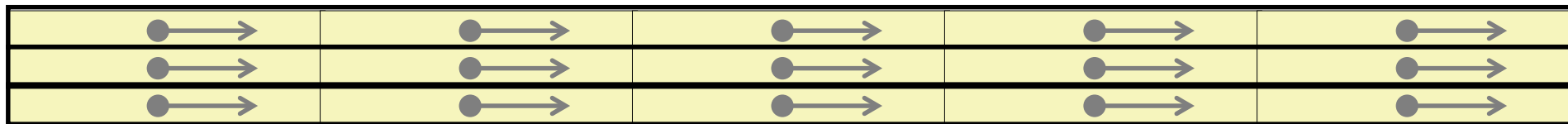
Decl	***An		
	Cmp	Bad	Size
<code>int A1[3][5]</code>			
<code>int *A2[3][5]</code>			
<code>int (*A3)[3][5]</code>			
<code>int *(A4[3][5])</code>			
<code>int (*A5[3])[5]</code>			



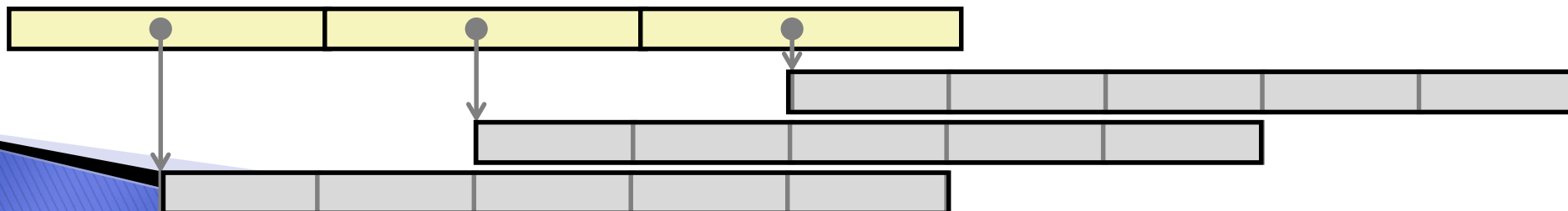
Declaration
<code>int A1[3][5]</code>
<code>int *A2[3][5]</code>
<code>int (*A3)[3][5]</code>
<code>int *(A4[3][5])</code>
<code>int (*A5[3])[5]</code>



A2/A4



A5



Understanding Pointers & Arrays #3

Decl	An			*An			**An		
	Cmp	Bad	Size	Cmp	Bad	Size	Cmp	Bad	Size
int A1[3][5]	Y	N	60	Y	N	20	Y	N	4
int *A2[3][5]	Y	N	120	Y	N	40	Y	N	8
int (*A3)[3][5]	Y	N	8	Y	Y	60	Y	Y	20
int *(A4[3][5])	Y	N	120	Y	N	40	Y	N	8
int (*A5[3])[5]	Y	N	24	Y	N	8	Y	Y	20

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

Decl	***An		
	Cmp	Bad	Size
int A1[3][5]	N	-	-
int *A2[3][5]	Y	Y	4
int (*A3)[3][5]	Y	Y	4
int *(A4[3][5])	Y	Y	4
int (*A5[3])[5]	Y	Y	4