Arrays, Structs, and Memory

CMPU 224 – Computer Organization

Jason Waterman

Shift Operations in C

- Left Shift: `x << y`
  - Shift bit-vector **x** left **y** positions
    - Throw away extra bits on left
  - Fill with 0's on right
  - X * 2^y
- Right Shift: `x >> y`
  - Shift bit-vector **x** right **y** positions
    - Throw away extra bits on right
  - Logical shift
    - Fill with 0's on left
  - Arithmetic shift
    - Replicate most significant bit on left
- Undefined Behavior
  - Shift amount < 0 or ≥ data size

Argument **x**	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument **x**	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Multiplying by Constants

- Historically, the integer multiply instruction was very slow
  - Even on modern hardware it takes 3 cycles
- The compiler tries to replace multiplication by a constant with shifts and adds
- Multiplication by a power of two
  - A single left shift doubles a number

  (5) 00101 << 1 ⟹ 01010 (10)
- In general
  - $X << k = X \cdot 2^k$
  - Multiplication by a power of two can be performed by left shift
  - For both signed and unsigned numbers

Multiplying by a Constant Examples

- Examples
  - `u * 8 == u << 3`
  - `u * 5 == (u << 2) + u`
  - `u * 24 == (u << 5) - (u << 3)`

Unsigned Power-of-2 Divide with Shift

- Quotient of Unsigned by Power of 2
  - `u >> k` gives $\lfloor$ `u / 2^k` $\rfloor$ (greatest integer less than)
  - Uses logical right shift
  - Rounds down (towards zero) by truncating the decimal part

	Division	Computed	Hex	Binary
`x`	15213	15213	3B 6D	00111011 01101101
`x >> 1`	7606.5	7606	1D B6	00011101 10110110
`x >> 4`	950.8125	950	03 B6	00000011 10110110
`x >> 8`	59.4257813	59	00 3B	00000000 00111011

Signed Power-of-2 Divide with Shift

- Quotient of signed by Power of 2
  - `u >> k` gives $\lfloor$ `u / 2`^{k} $\rfloor$
  - Uses arithmetic right shift
  - Rounds down

	Division	Computed	Binary
x	-12,340.0	-12,340	11001111 11001100
x >> 1	-6,170.0	-6,170	*1*1100111 11100110
x >> 4	-771.25	-772	*1111*1100 11111100
x >> 8	-48.203125	-49	*11111111* 11001111
X >> 14	-0.75317382	-1	*11111111 11111111*

Masking Operations

- Goal: modify some bits of a bit vector while leaving the other bits unchanged
- Apply a mask to a value to change only certain bits
  - Example: value 0xAA (10101010), mask 0x0F (00001111)
- Setting bits to 1 (OR operator)
  - 0xAA | 0x0F // evaluates to 0xAF
  - Wherever the mask is 1, that bit will set be 1
  - Wherever the mask is 0, the bit will be unchanged
- Clearing bits to 0 (AND operator)
  - 0xAA & 0x0F // evaluates to 0x0A
  - Wherever the mask is 0, that bit will be cleared to 0
  - Wherever the mask is 1, that bit will remain the same.
- Flipping bits (XOR operator)
  - 0xAA ^ 0x0F // evaluates to 0x5A
  - Wherever the mask is 1, that bit will be flipped
  - Wherever the mask is 0, that bit will remain the same

```
  1010 1010  (value)
OR 0000 1111  (MASK)
-------------
  1010 1111
```

```
  1010 1010
AND 0000 1111
-------------
  0000 1010
```

```
  1010 1010
XOR 0000 1111
-------------
  1010 0101
```

Extracting Bits

- Goal: get the value for some subset of bits from a bit vector
- Can do this with a combination of shifting and masking
- Example: get the middle four bits out of a byte
  - `char a = 0x63; // 01**1000**11`
  - `a = a >> 2;  // 0001**1000**`
  - `a = a & 0xF;  // 00001000`
- Can extract any contiguous group of bits by varying the size of the shift and the masking bits

Logic Operations in C

- No Boolean type in C
  - View 0 as “False”
  - Anything nonzero as “True”
  - Logical operators always return 0 (False) or 1 (True)
  - Early termination
- Logical operators
  - &&, ||, ! (logical AND, OR, NOT)
- Examples (char data type)
  - !0x41 -> 0x00
  - !0x00 -> 0x01
  - !!0x41 -> 0x01
  - 0x69 && 0x55 -> 0x01
  - 0x69 || 0x55 -> 0x01

Data Sizes in C (Typical)

- The sizes of the basic data types can vary based on compiler and machine settings
- These are typical values on 32-bit and 64-bit systems

declaring a variable

```
int x;
```

Type — Variable name

Tells the compiler

The variable exists

A variable can be declared many TIMES

C Declaration	Size in Bytes (32-bit)	Size in Bytes (64-bit)
char	1	1
short	2	2
int	4	4
long	4	8
void*	4	8
float	4	4
double	8	8

defining a variable

```
int x = 42;
```

Tells the compiler the variable's initial value

should only be defined once

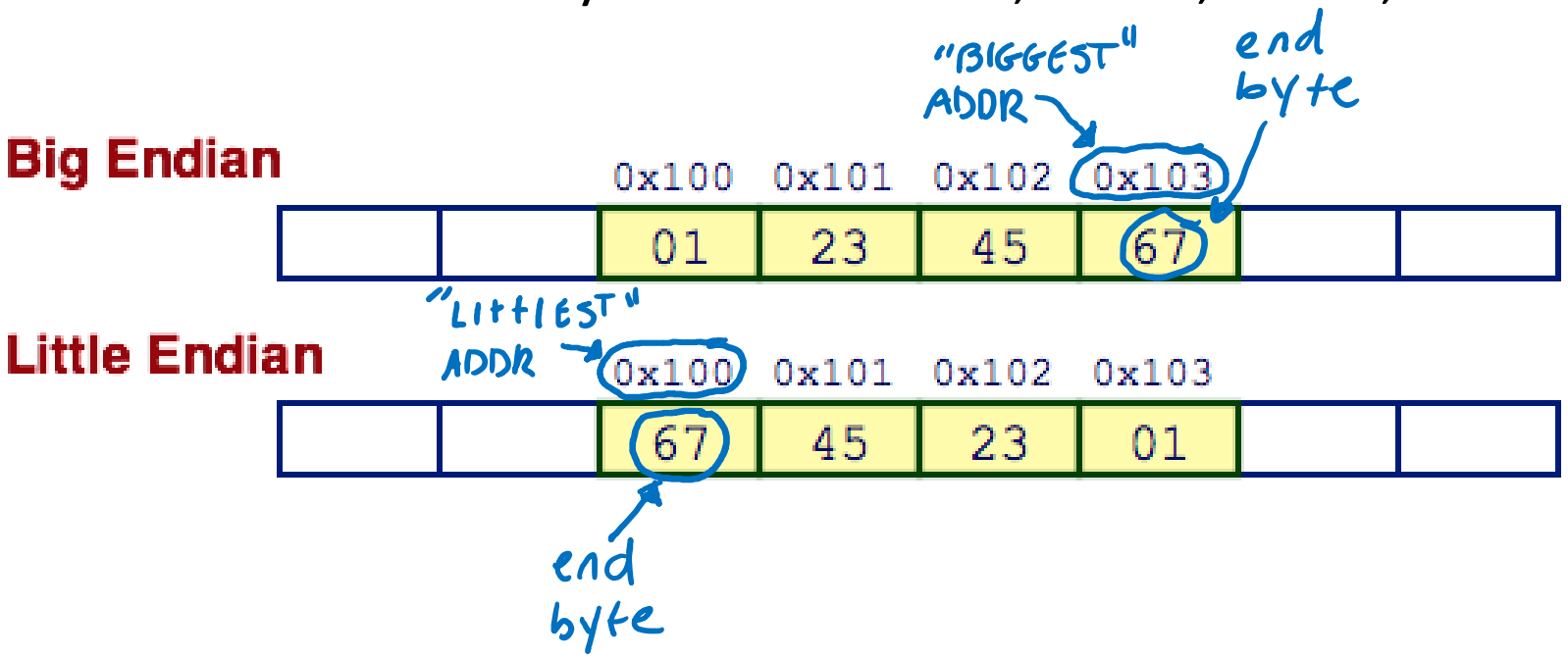
Storing Multi-Byte Data

- Memory is byte addressable
  - Every byte of memory has an address
  - Think of memory as a large array with the address as the index in the array
- For multi-byte data
  - The address specifies starting byte location of the data in memory
  - The rest of the data is in the increasing memory addresses that follow
- Example
  - The `int` at address 4 contains the four bytes: `31 76 D9 5C`

int	Address	Bytes
ADDR 0	0	94
	1	53
	2	7F
	3	EA
ADDR 4	4	**31**
	5	**76**
	6	**D9**
	7	**5C**
ADDR 8	8	AB
	9	83
	10	75
	11	BB
ADDR 12	12	28
	13	39
	14	4E
	15	05

Byte Ordering

- There are two conventions for the layout of multi-byte objects
  - Big Endian and Little Endian
- Example
  - 4-byte integer of `0x01234567` is located at memory address 0x100
  - This value exists in memory locations 0x100, 0x101, 0x102, 0x103



Byte Ordering Takeaways

- RISC-V (and most all other systems) uses **little endian**
  - When reading data from left to right in increasing memory order:
    - The bytes will be in reverse order from how the number is written
- **Bytes** are always written from msb (most significant bit) to lsb (least significant bit) in both endian conventions
- Little endian systems will always have the end byte at the smallest (littlest) address
- Big endian systems will always have the end byte at the largest (biggest) address

Pointers in C

- Pointers in C are all about memory addresses
  - Pointers are variables that store the location of other variables in memory
  - Think of them as shortcuts to data hidden in different corners of your program's storage
- Declaration
  - You introduce pointers with an asterisk before the variable type
  - **`int *ptr;`**
  - Creates a pointer named `ptr` that can hold the address of an integer variable.
- Getting an address of a variable
  - Use the ampersand **`&`** (address of operator) before a variable to get its address
  - **`int num = 10;`**
  - **`int *ptr = &num;`**
  - `ptr` now has the value of the memory location of where the data of `num` is stored (where the value 10 resides in memory)
- Accessing pointer data
  - Use the asterisk before the pointer to peek inside the memory location the pointer points to
  - **`int value = *ptr;`**
  - This assigns the value stored at the address held by `ptr` (which is 10) to the variable value.

Pointer Example in C

```
#include <stdio.h>

int main() {

  int num;  // integer declaration of variable x
  int *ptr; // int pointer declaration of variable ptr

  num = 10;

  printf("value of num: %d\n", num);
  printf("address of num in memory: %p\n", &num);

  ptr = &num; // Address of the variable num

  printf("value of ptr: %p\n", ptr);
  printf("value at the memory address pointed to by ptr: %d\n", *ptr);
}
```

Pointer Example in C

```
#include <stdio.h>

int main() {

  int num;  // integer declaration of variable x
  int *ptr; // int pointer declaration of variable ptr

  num = 10;

  printf("value of num: %d\n", num);
  printf("address of num in memory: %p\n", &num);

  ptr = &num; // Address of the variable num

  printf("value of ptr: %p\n", ptr);
  printf("value at the memory address pointed to by ptr: %d\n", *ptr);
}
```

```
prompt> gcc -o pointer_slide pointer_slide.c
prompt> ./pointer_slide
value of num: 10
address of num in memory: 0x40800188
value of ptr: 0x40800188
value at the memory address pointed to by ptr: 10
prompt>
```

Arrays in C

- Declaring arrays
  - `type array_name [ array_size ];`
    - Example: `int count [5];`
  - All elements of the array have the same type
- Declaring and initializing
  - `int count[] = {4, 2, 9, 3, 5};`

First Element

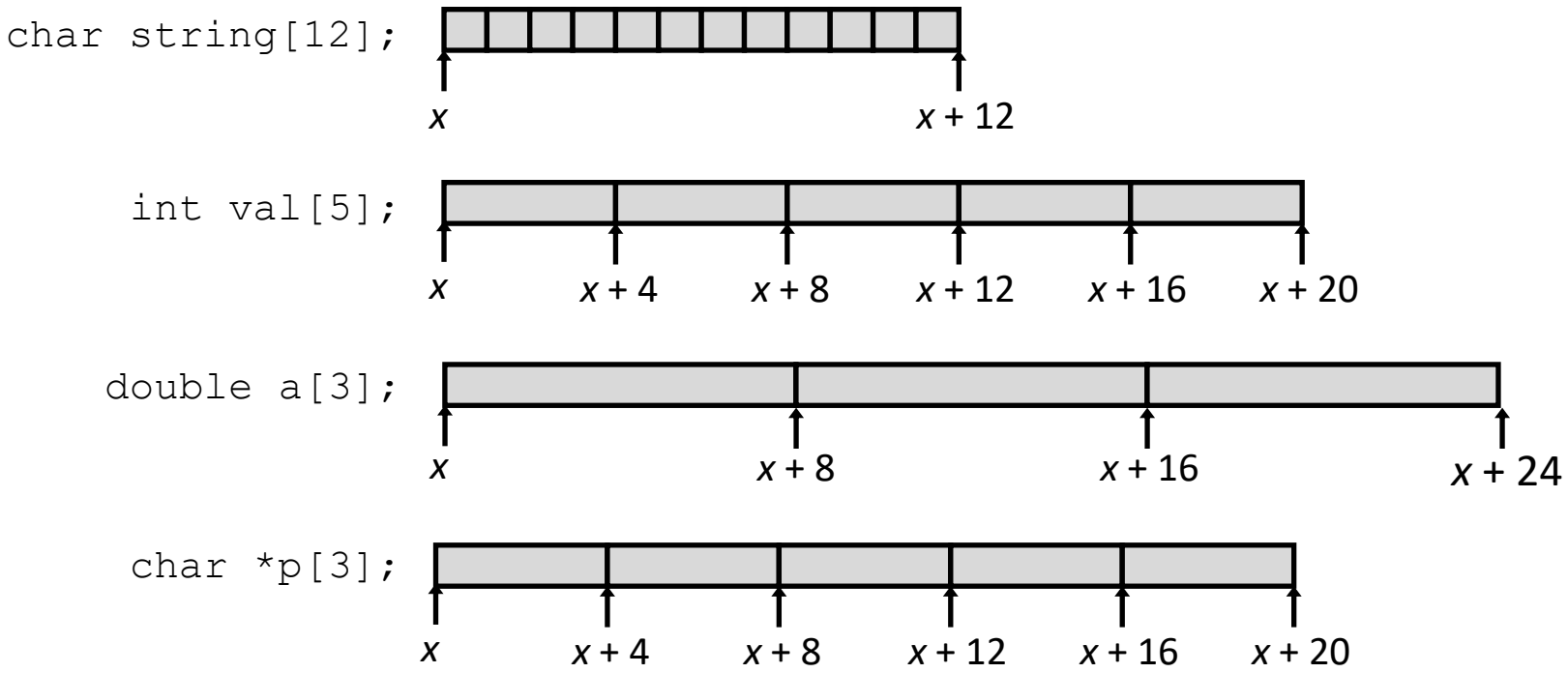
count[0]	count[1]	count[2]	count[3]	count[4]
4	2	9	3	5

Array Allocation

- Basic Principle

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

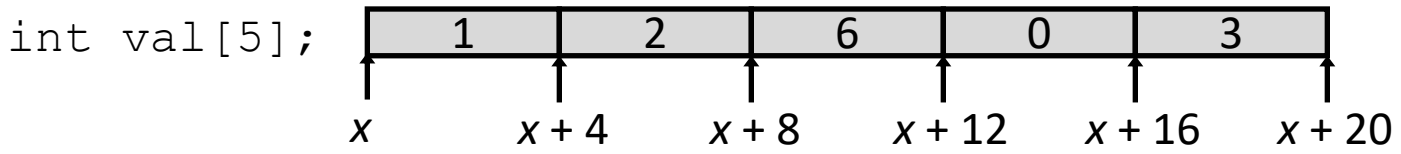
  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **`sizeof`**`(`*T*`)` bytes in memory



Array Access

- Basic Principle

  T **`A[`***L***`];`**
  - Array of data type *T* and length *L*
  - Identifier **`A`** can be used as a pointer to array element 0: Type *T**



Reference	Type	Value
`val`	`int *`	*x*
`val[4]`	`int`	3
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	2
`val + i`	`int *`	*x* + 4 *i*

Multidimensional (Nested) Arrays

- Declaration

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

  - 2D array of data type *T*
  - *R* rows, *C* columns
  - Type *T* element requires *K* bytes

- Arrangement
  - **Row-Major Ordering**

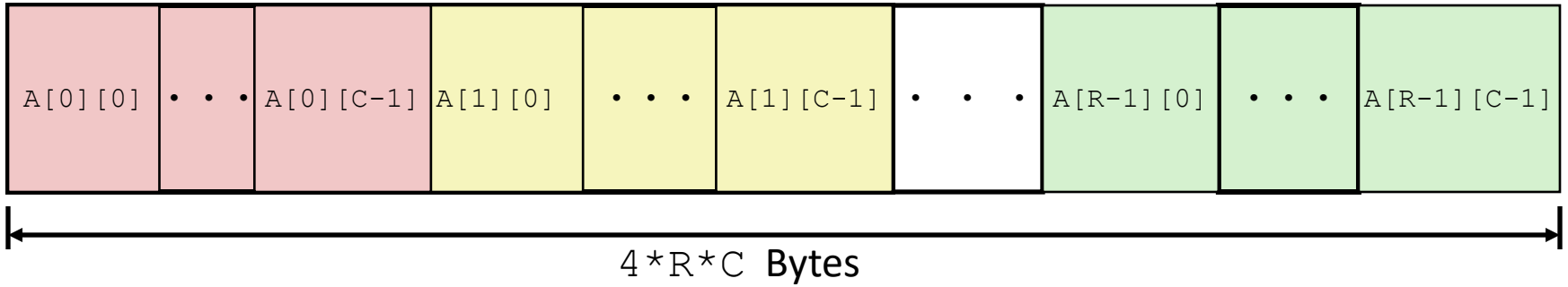
- Array Size
  - *R* * *C* * *K* bytes

  # OF ELEMENTS IN ARRAY

  SIZE OF AN ELEMENT

```
[ A[0][0]    . . .    A[0][C-1]
     .                    .
     .                    .
     .                    .
  A[R-1][0]  . . .  A[R-1][C-1] ]
```

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

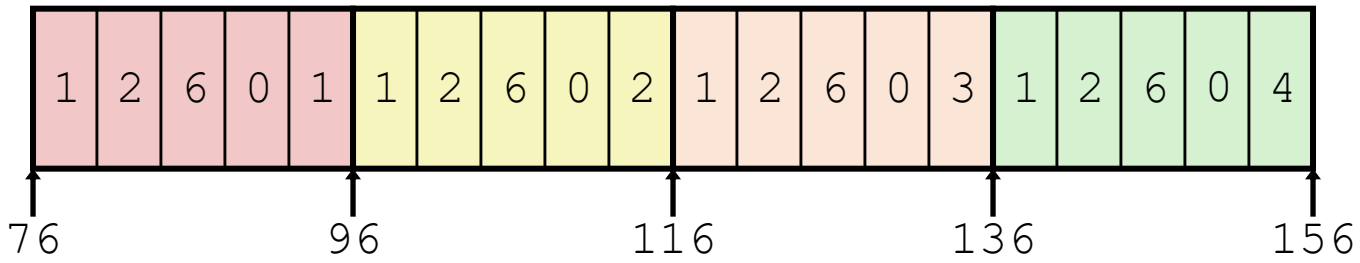


Nested Array Example

- `int pok[4][5];`
  - Variable **`pok`**: 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

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

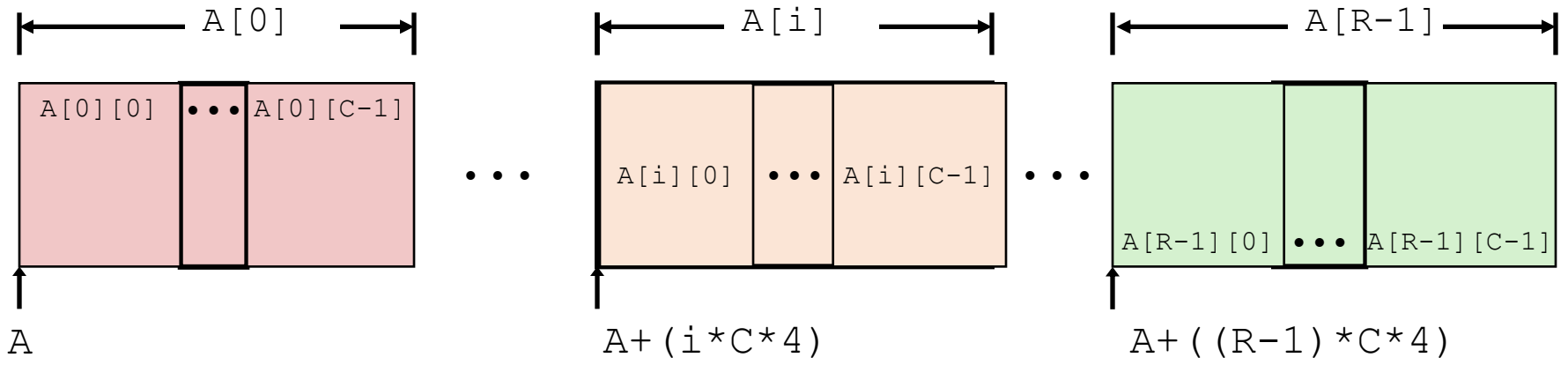
`int pok[4][5];`



Nested Array Row Access

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

SIZE OF ONE ROW

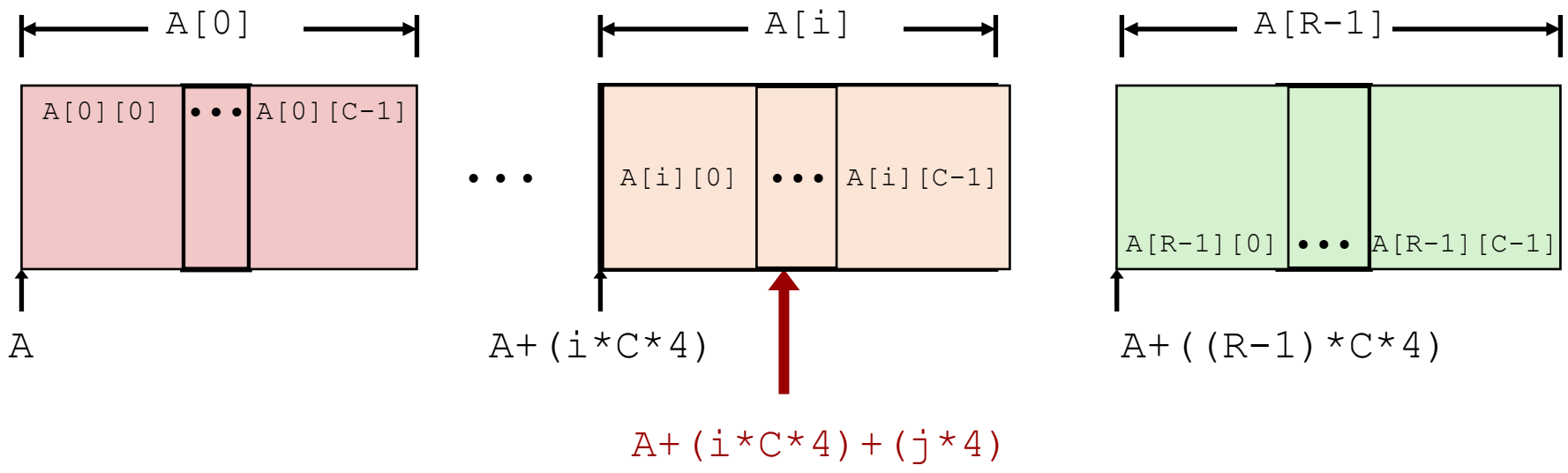


Nested Array Element Access

- Array Elements
  - **`A[i][j]`** is element of type *T,* which requires *K* bytes and has *R* rows and *C* cols
  - Address **`A`** **+** *i* * (*C* * *K*) + *j* * *K* = ***A* + (*i* * *C* + *j*)* *K***

SIZE OF ROW

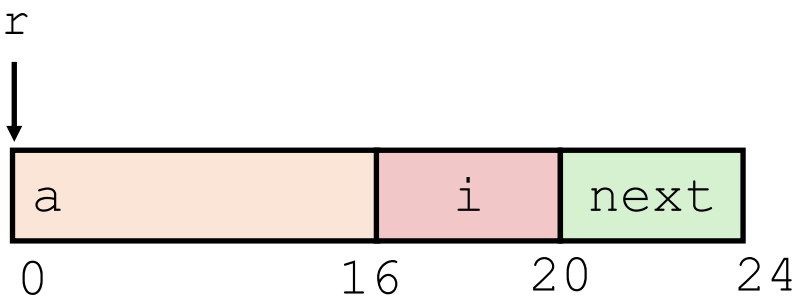
```
int A[R][C];
```



Structure Representation

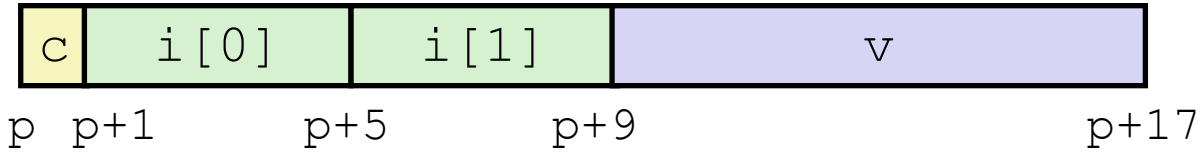
- Structure represented as block of memory
  - **Big enough to hold all of the fields**
- Fields ordered according to declaration
  - **Even if another ordering could yield a more compact representation**
- Compiler determines overall size + positions of fields

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



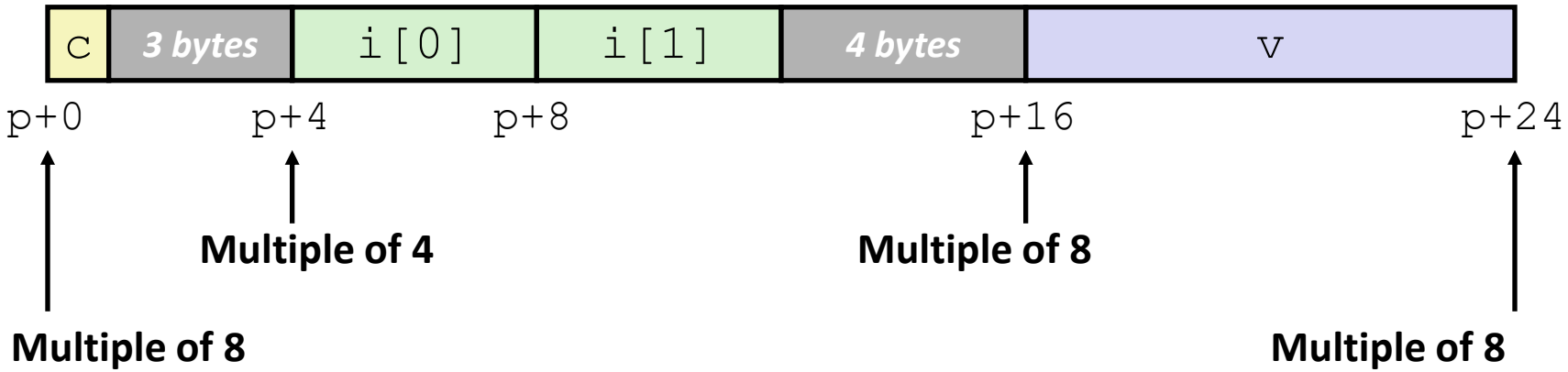
Structures & Alignment

- Unaligned Data




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

- Aligned Data
  - A primitive data type of K bytes must have an address that is 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 RISC-V

- Motivation for Aligning Data
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - Inefficient to load or store data that spans word boundaries

- Compiler
  - Inserts gaps in structure to ensure correct alignment of fields

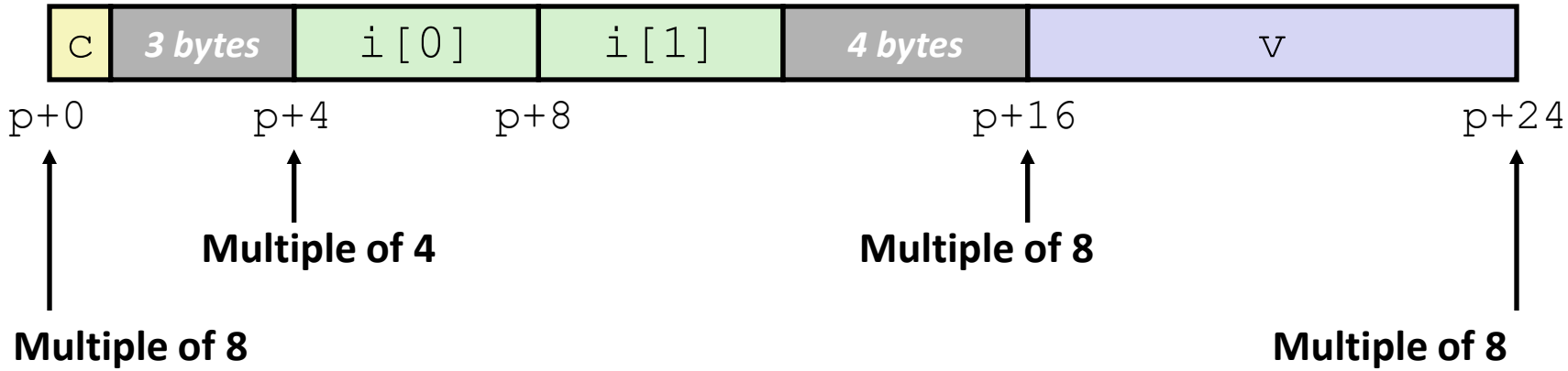
Specific Cases of Alignment

- 1 byte: **`char`**, …
  - no restrictions on address
- 2 bytes: **`short`**, …
  - lowest 1 bit of address must be 0_2
- 4 bytes: **`int`**, **`float`**, **`char *`**, …
  - lowest 2 bits of address must be 00_2
- 8 bytes: **`double`**, **`long long`**, …
  - 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**
    - $\mathbf{K_{struct}}$ = Largest alignment of any element in struct
  - Initial address & structure length must be multiples of $\mathbf{K_{struct}}$
- Example:
  - K_{struct} = 8, due to **`double`** element

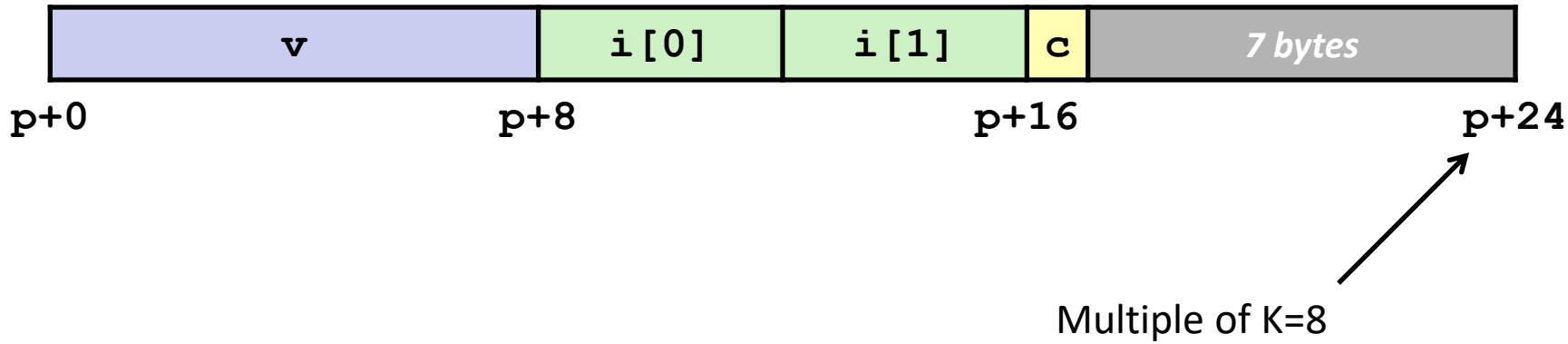
```
struct S1 {
  char c;
  int i[2];
  double v;
} *p;
```



Meeting Overall Alignment Requirement

- Largest alignment requirement K_{struct}
- Overall structure must be multiple of K_{stuct}

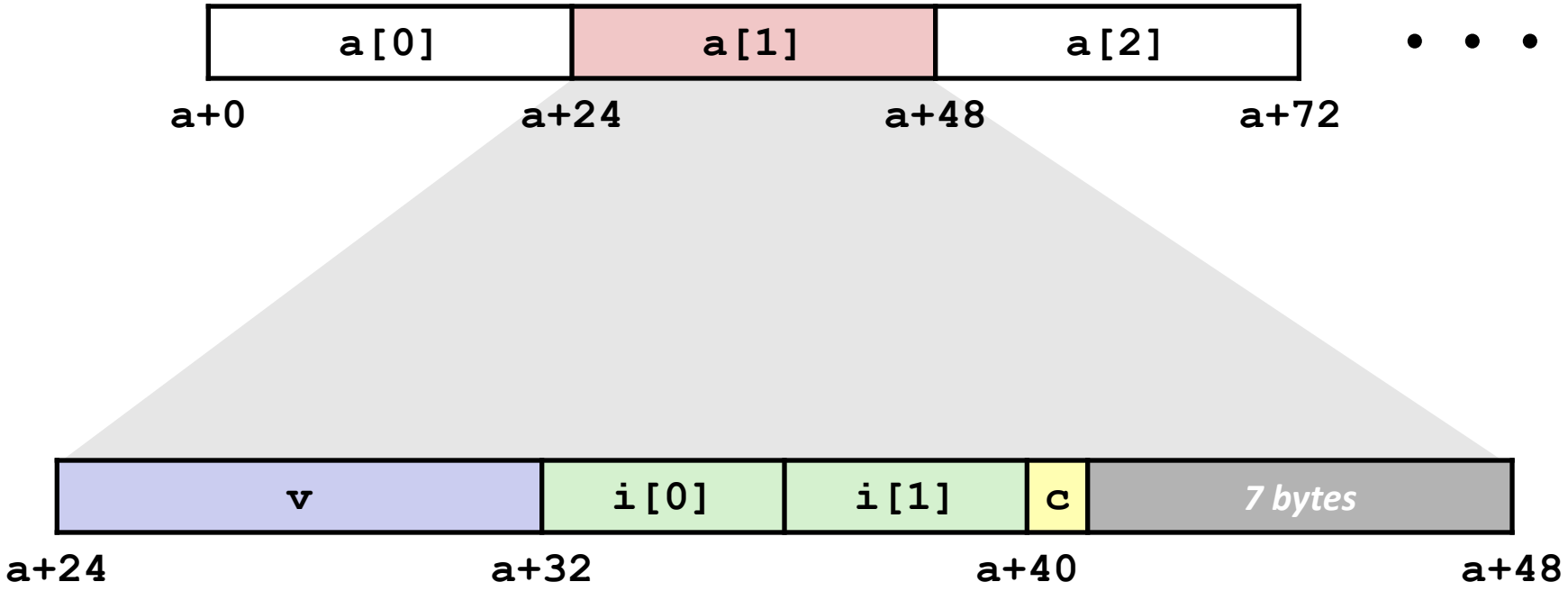
```
struct S2 {
  double v;
  int i[2];
  char c;
} *p;
```



Arrays of Structures

- Overall structure length multiple of K_{struct}
- Satisfy alignment requirement for every element

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

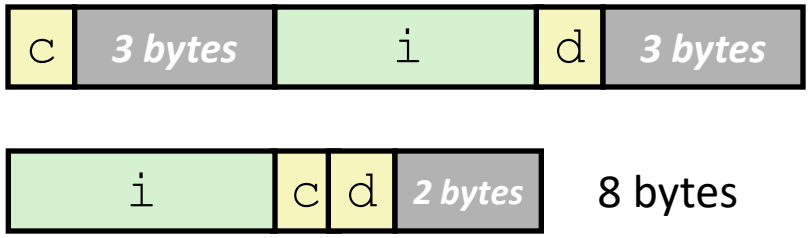


Saving Space

- Put large data types first

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

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



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