

Lecture 5: Outline

- I. Multi-dimensional arrays
- II. Multi-level arrays
- III. Structures
- IV. Data alignment
- V. Linked Lists

Multidimensional arrays: 2D

- Declaration

`int a[3][4];` */*Conceptually 2D matrix with 3 rows 4 columns */*

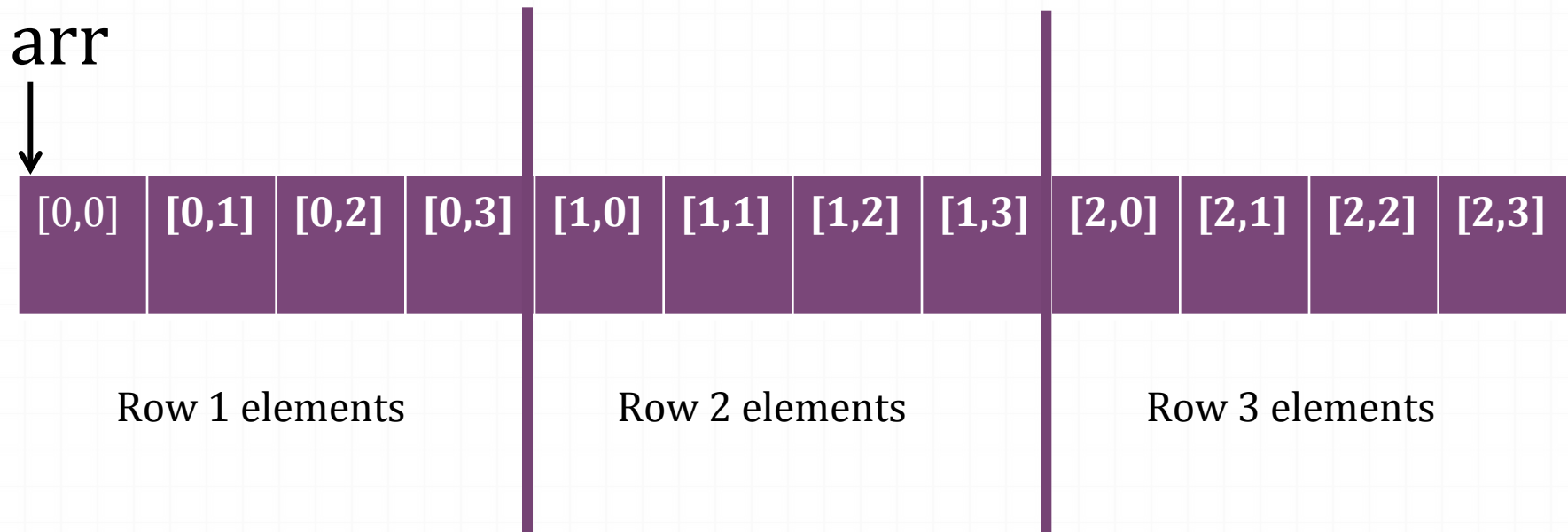
	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Element in row i, column j is retrieved as `a[i][j]`
- 'a' is a pointer to an integer array of size 4

Representation in memory

```
int arr[3][4];    /*Conceptually 2D matrix with 3 rows 4 columns */
```

In memory the elements of 'arr' are stored in a contiguous memory block



(arr+i) increments the address of 'arr' by how many bytes?



- A. $i * \text{sizeof}(\text{int})$
- B. $i * \text{sizeof}(\text{int}^*)$
- C. $i * \text{sizeof}(\text{int}) * \text{number of columns}$

(arr+i) increments the address of 'arr' by how many bytes?



A. $i * \text{sizeof}(\text{int})$

B. $i * \text{sizeof}(\text{int}^*)$

C. $i * \text{sizeof}(\text{int}) * \text{number of columns}$

Express `arr[i][j]` using 'arr' as a pointer



- A. `*((arr+i) +j)`
- B. `*(* (arr+i)+j)`
- C. `*((int*)(arr+i)+j)`
- D. `*(arr+4*i*sizeof(int)+j*sizeof(int))`

Express `arr[i][j]` using 'arr' as a pointer



- A. $*((arr+i) +j)$
- B. $*(* (arr+i)+j)$: This is what is generally used
- C. $*((int*)(arr+i)+j)$
- D. $*(arr+4*i*sizeof(int)+j*sizeof(int))$

Multi-level arrays

- Declaration

```
char name_1[]="John";
```

```
char name_2[]="Paul";
```

```
char name_3[]="Rose";
```

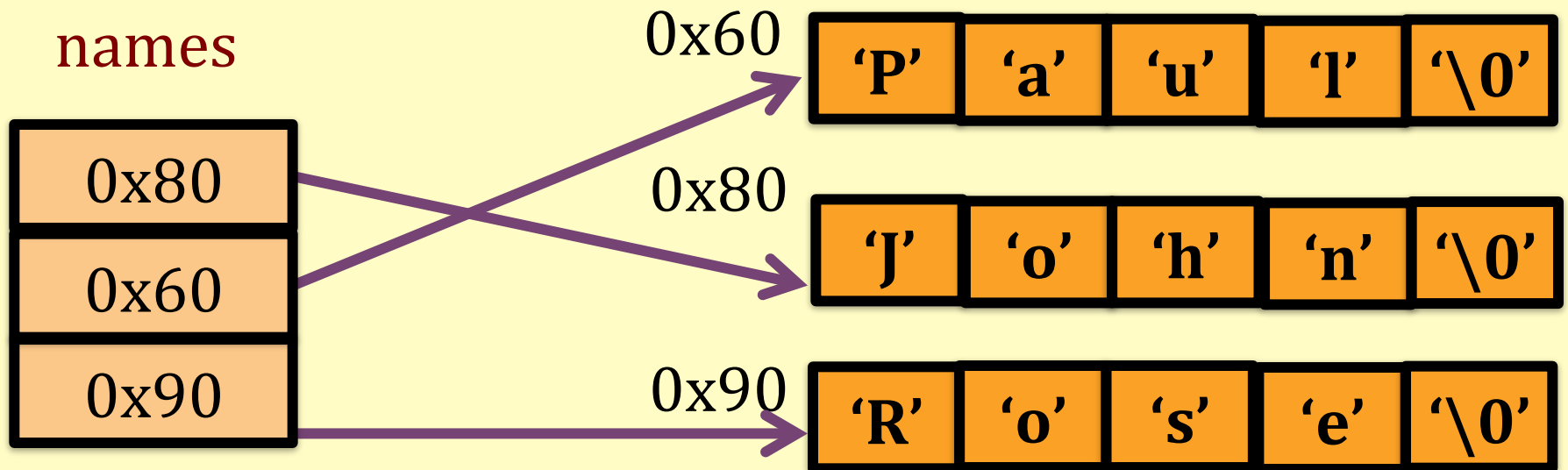
```
char * names[]={name_1, name_2, name_3};
```

What does the following statement print?

```
printf("\n Lettter: %c\n", names[1][2]);
```

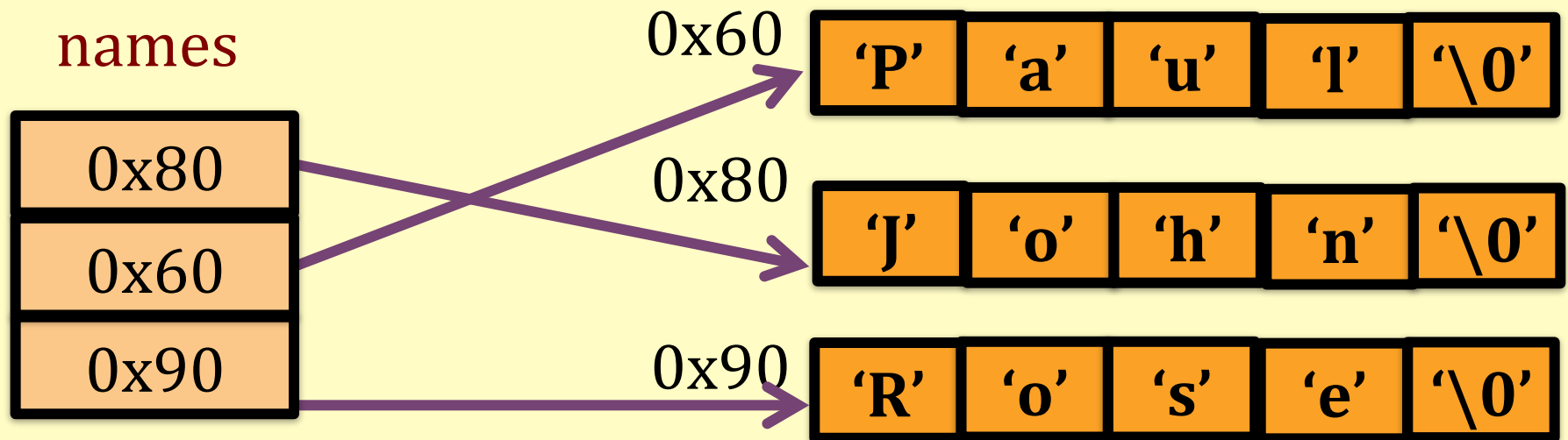
Multi-level arrays: Representation in memory

```
char name_1[]="John";  
char name_2[]="Paul";  
char name_3[]="Rose";  
char * names[]={name_1, name_2, name_3};
```



Multi-level vs multi-dimensional arrays

1. What does `names[1][2]` give in each case?
2. Which one needs more memory accesses?
3. When would we prefer multi-level arrays?



names

`'J'` `'o'` `'h'` `'n'` `'\0'` `'P'` `'a'` `'u'` `'l'` `'\0'` `'R'` `'o'` `'s'` `'e'` `'\0'`

Pointers in C

- Why use pointers?
 - If we want to pass a huge struct or array, it's easier to pass a pointer than the whole thing.
 - In general, pointers allow cleaner, more compact code.
- So what are the drawbacks?
 - Pointers are probably the single largest source of bugs in software, so be careful anytime you deal with them.



Derived data types: Structures

C structures : Overview

- A `struct` is a data structure composed of simpler data types.
 - Like a class in Java/C++ but without methods or inheritance.

```
struct point {  
    int x;  
    int y;  
}  
void PrintPoint(struct point p)  
{  
    printf( “ (%d,%d) ” , p.x, p.y) ;  
}
```

Pointers to structures

- The C arrow operator ($\rightarrow$) dereferences and extracts a structure field with a single operator.
- The following are equivalent:

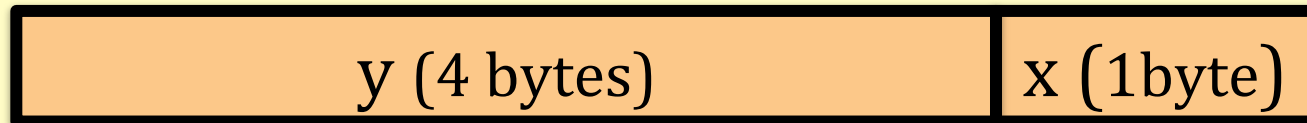
```
struct point *p;
```

```
printf( "x is %d\n" , (*p).x ) ;  
printf( "x is %d\n" , p->x ) ;
```

Representation in memory

```
struct p {  
    int y;  
    char x;  
};  
struct p sp;
```

sp



0x100

0x104

0x105

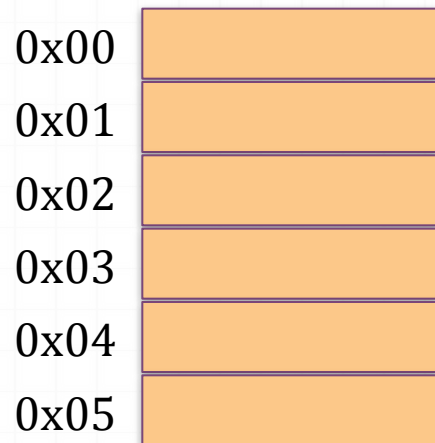
Data Alignment

```
struct p {  
    char x;  
    int y;  
};  
Struct p sp;
```

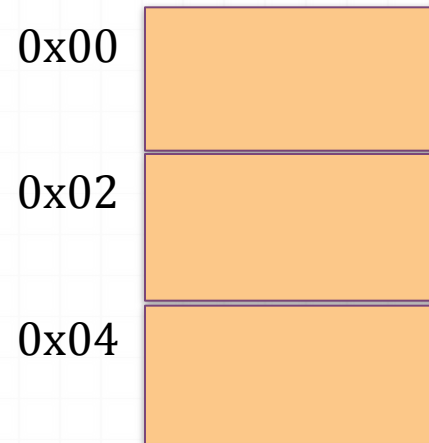
Q: How is the above struct laid out in memory?

Alignment Fundamentals

- Processors do not always access memory in byte sized chunks, instead in 2, 4, 8, even 16 or 32 byte chunks
- Boundaries at which data objects are stored affects the behavior of read/write operations into memory



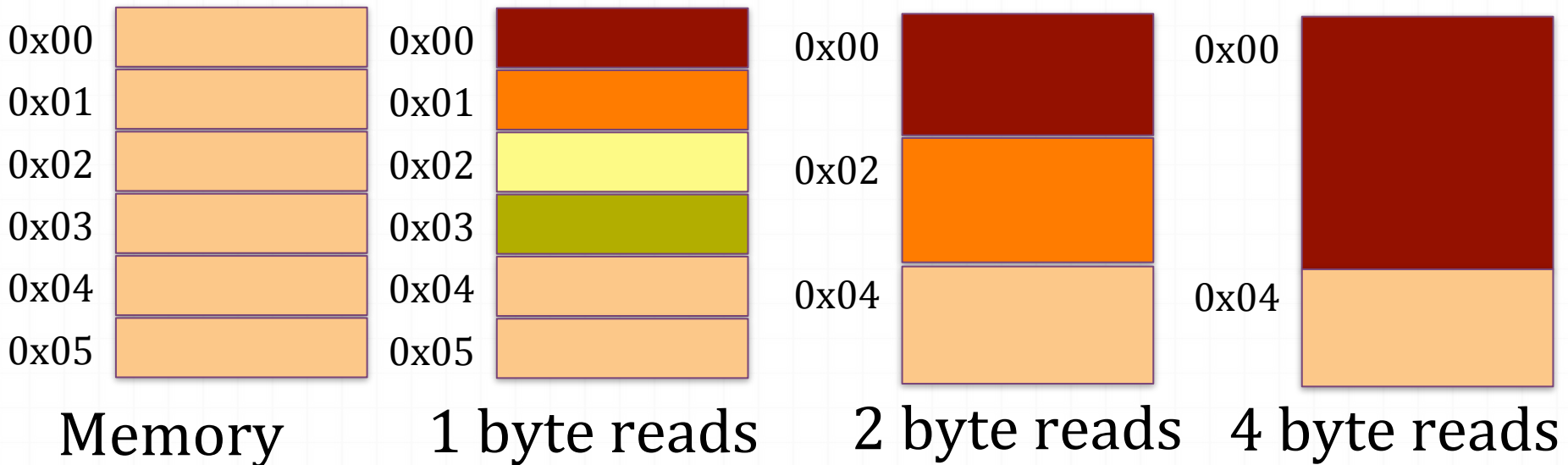
Programmer's view
of memory



Processor's view of
memory

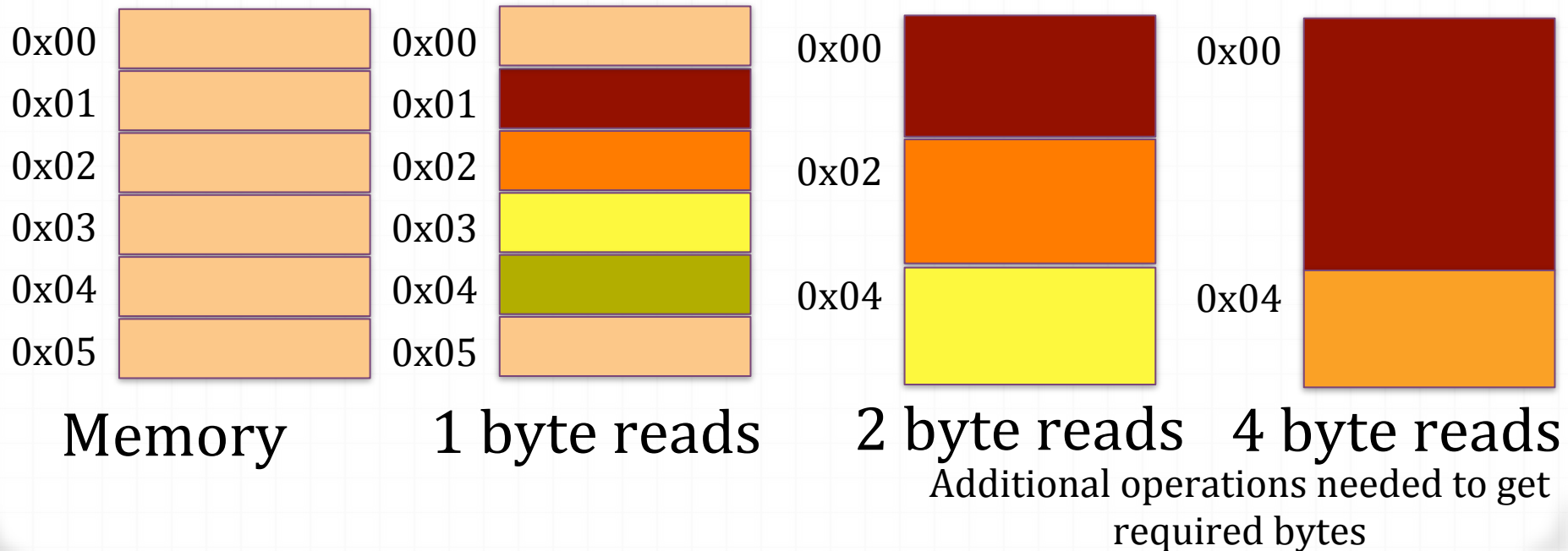
Alignment Fundamentals

- Consider the task of reading 4 bytes from memory starting at 0x00 for processors that read 1, 2, 4 bytes at a time.
- How many memory accesses are needed in each case?



Alignment Fundamentals

- Now consider the task of reading 4 bytes from memory starting at 0x01 for processors that read 1, 2, 4 bytes at a time.
- How many memory accesses are needed in each case?
 - Some processors just would not allow this scenario because it is extra work for the h/w and it affects the performance

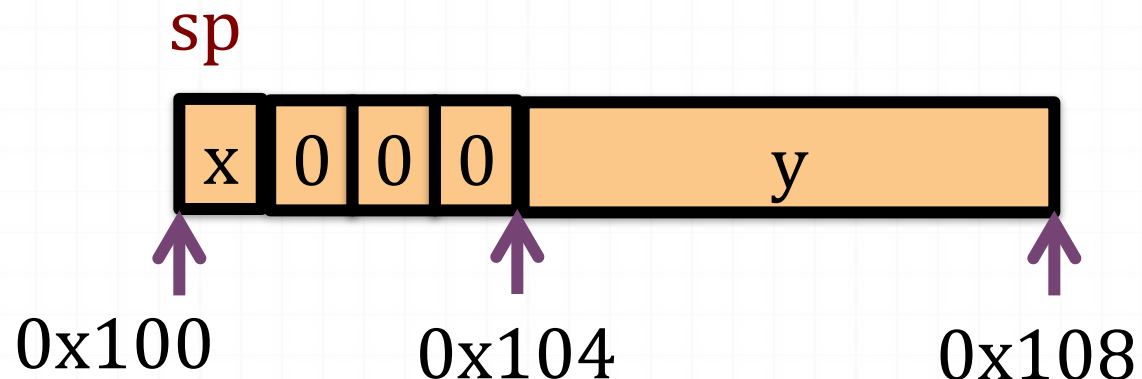


Alignment requirements

- Data objects should start at memory addresses that are divisible by the size of the data object.
 - short (2 byte) at address divisible by 2 0b_____0
 - int (4 byte) at address divisible by 4 0b_____00
 - double (8 byte) at address divisible by 8 0b_____000

```
struct p {  
    char x;  
    int y;  
};  
Struct p sp;
```

Q: How is 'sp' laid out in memory?
Compiler takes care of zero padding



How big are structs?

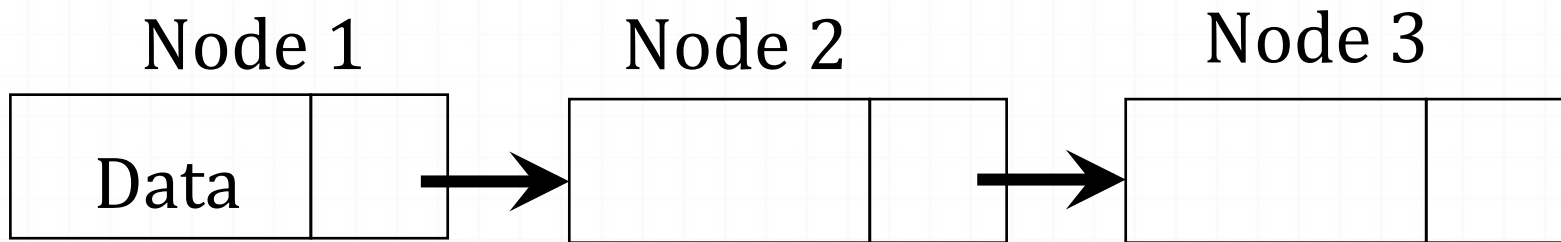
- Recall C operator `sizeof()` which gives size in bytes (of type or variable)

Q: How big is `sizeof(struct p)`?

```
struct p {  
    char x;  
    int y;  
};
```

- A. 5 bytes
- B. 8 bytes
- C. 12 bytes

Linked Lists



A generic linked list has:

- Multiple data objects (structs) also called nodes
- Each node linked to the next node in the list i.e. it knows the address of the next node
- Nodes located at different memory locations (unlike arrays where they are contiguous)

Advantages compared to arrays

- Multiple data members (of different data types) can be stored in each node
- Nodes can be easily inserted or removed from the list without modifying the whole list

Let's look at an example of using structures, pointers, `malloc()`, and `free()` to implement a **linked list of strings**.

```
typedef struct Node node;
```

```
struct Node {  
    char *value;  
    _____ next;  
};
```

node



Q:What is the data type of the variable '**next**'?

- A. struct Node
- B. Node
- C. node
- D. node *

Let's look at an example of using structures, pointers, `malloc()`, and `free()` to implement a **linked list of strings**.

```
typedef struct Node node;
```

```
struct Node {  
    char *value;  
    _____ next;  
};
```

node



Q:What is the data type of the variable '**next**'?

- A. struct Node
- B. Node
- C. node
- D. node *

Adding a node to the list

```
node *list_add(node* head, char *string)
```

```
{  
    //Step 1:Create a new node
```

new_node

```
    _____;  
    return new_node;
```

```
}
```

value=?

next=?

Q: How should we declare and initialize **new_node**?

A. `node *new_node=(node*) malloc(sizeof(node)) ;`

B. `node new_node;`

C. `node *new_node=head;`

D. `node *new_node=(node *)malloc(sizeof(head)) ;`

Adding a node to the list

```
node *list_add(node* head, char *string)
```

```
{  
    //Step 1:Create a new node
```

new_node

```
    _____;  
    return new_node;  
}
```

value=?

next=?

Q: How should we declare and initialize **new_node**?

A. `node *new_node=(node*) malloc(sizeof(node)) ;`

B. `node new_node;`

C. `node *new_node=head;`

D. `node *new_node=(node *)malloc(sizeof(head)) ;`

Q: What is wrong with step 2?

```
node *list_add(node* head, char *string)
{
    //Step 1: Create a new node
    node *new_node=(node*) malloc(sizeof(node)) ;
    //Step 2: Fill in its value
    strcpy(new_node->value, string);
    return new_node;
}
```

- A. Step 2 is correct.
- B. We should use the operator '.' instead of '->'
- C. Memory is not allocated for 'value'

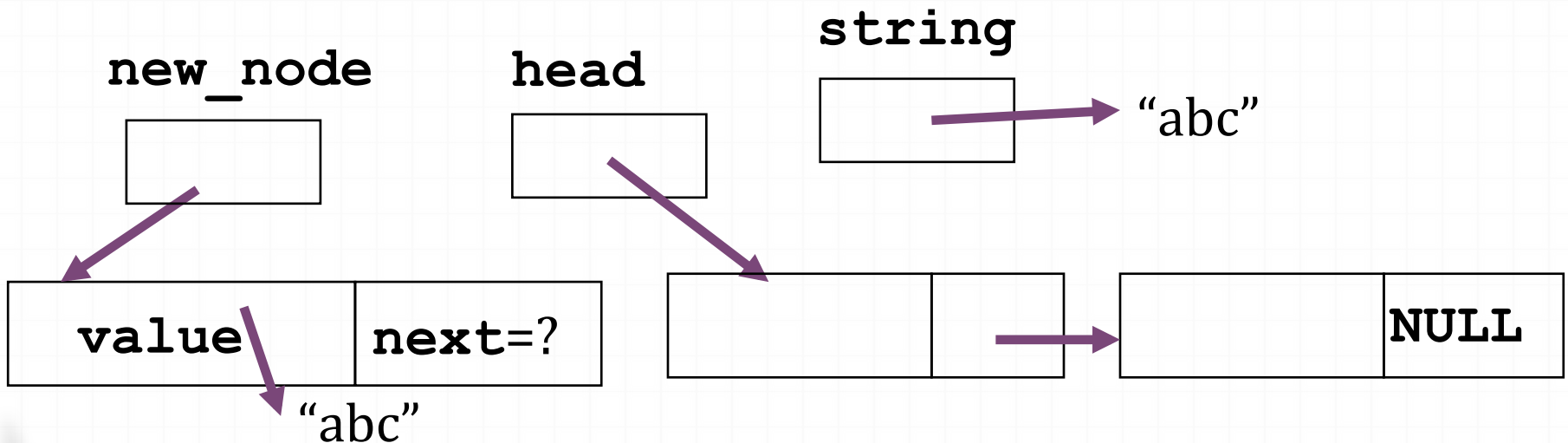
Q: What is wrong with step 2?

```
node *list_add(node* head, char *string)
{
    //Step 1: Create a new node
    node *new_node=(node*) malloc(sizeof(node)) ;
    //Step 2: Fill in its value
    strcpy(new_node->value, string);
    return new_node;
}
```

- A. Step 2 is correct.
- B. We should use the operator '.' instead of '->'
- C. Memory is not allocated for 'value'

So far....

```
node *list_add(node* head, char *string)
{
    //Step 1: Create a new node
    node *new_node=(node*) malloc(sizeof(node));
    //Step 2: Fill in its value
    new_node->value = (char*) malloc(strlen(string)+1);
    strcpy(new_node->value, string);
    return new_node;
}
```



What should Step 3 be?

```
node *list_add(node* head, char *string){  
    //Step 1: Create a new node  
    node *new_node=(node*) malloc(sizeof(node));  
    //Step 2: Fill in its value  
    new_node->value = (char*) malloc(strlen(string)+1);  
    strcpy(new_node->value, string);  
    //Step 3:  
    return new_node;  
}
```

A. `new_node->next =head;`

B. `next=head;`

C. `head=new_node;`

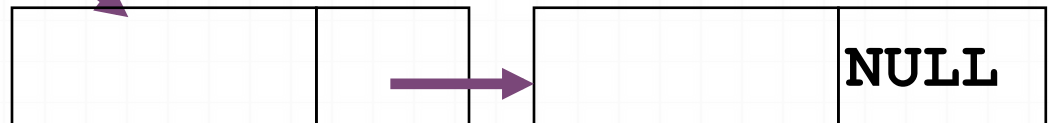
D. `new_node->next =*head;`

new_node

head



"abc"



What should Step 3 be?

```
node *list_add(node* head, char *string){  
    //Step 1: Create a new node  
    node *new_node=(node*) malloc(sizeof(node));  
    //Step 2: Fill in its value  
    new_node->value = (char*) malloc(strlen(string)+1);  
    strcpy(new_node->value, string);  
    //Step 3: Link new_node to the head of the list  
    new_node->next =head;  
    return new_node; }
```

A. `new_node->next =head;`

B. `next=head;`

C. `head=new_node;`

D. `new_node->next =*head;`

