2D Array, Struct, Malloc

Shuai Mu

based on slides from Tiger Wang and Jinyang Li

2D Arrray

2D arrays are stored contiguously in memory in row-major format

Multi-dimensional arrays

Declare a `k` dimensional array

`int arr[`n_1`][`n_2`][`n_3`]…[`n_{k-1}`][`n_k`]`

n_i is the length of the `i`th dimension

Multi-dimensional arrays

Declare a k dimensional array

int arr$[n_1][n_2][n_3]\ldots[n_{k-1}][n_k]$

n_i is the length of the ith dimension

Example: 2D array

int matrix[2][3]

Multi-dimensional arrays

Declare a k dimensional array

int arr[n_1][n_2][n_3]…[n_{k-1}][n_k]

n_i is the length of the ith dimension

Example: 2D array

int matrix[2][3]

	Col 0	Col 1	Col 2
Row 0			
Row 1			

Multi-dimensional arrays

Declare a k dimensional array

int arr$[n_1][n_2][n_3]\ldots[n_{k-1}][n_k]$

n_i is the length of the ith dimension

Example: 2D array

int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};

	Col 0	Col 1	Col 2
Row 0	1	2	3
Row 1	4	5	6

Multi-dimensional arrays

Declare a k dimensional array

int arr[n_1][n_2][n_3]…[n_{k-1}][n_k]

n_i is the length of the ith dimension

Example: 2D array

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

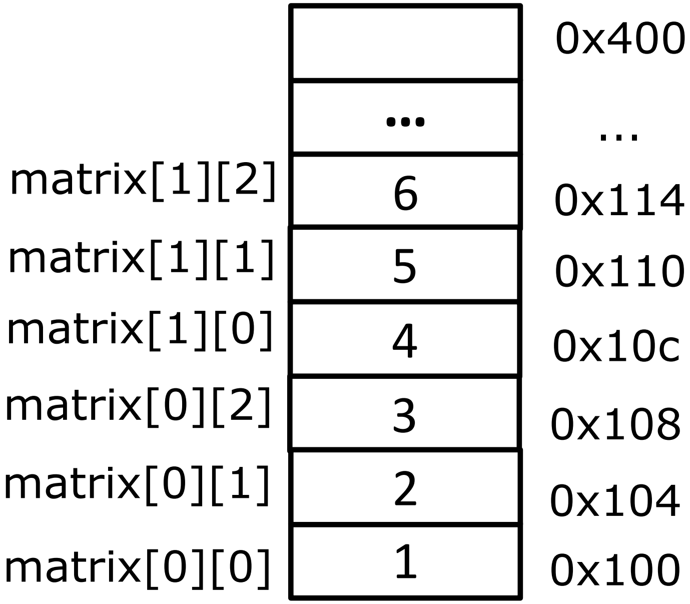
Access an element at second row and third column

```
matrix[1][2] = 10
```

Memory layout

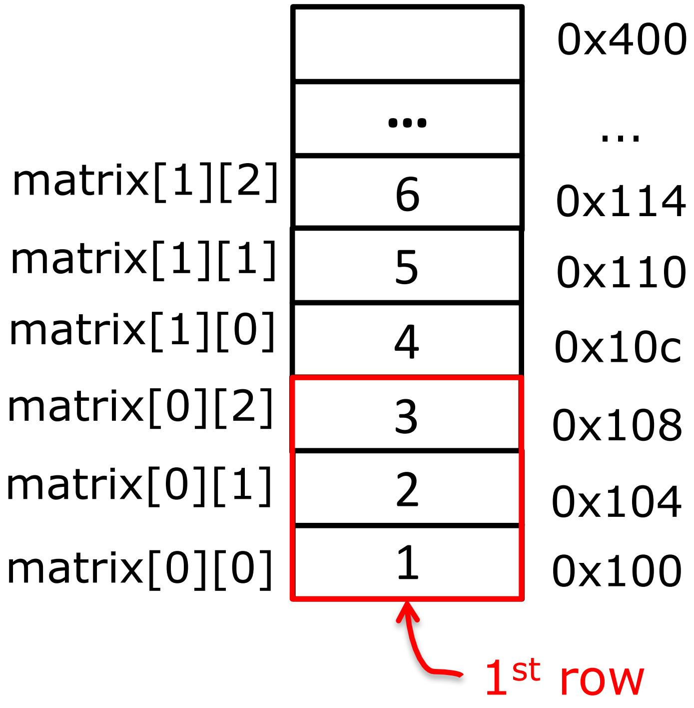
```
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};

for (int i = 0; i < 2; i++) {

    for (int j = 0; j < 3; j++) {

        printf(“%p\n”,&matrix[i][j]);

    }
}
```

Memory layout



		0x400
	…	…
matrix[1][2]	6	0x114
matrix[1][1]	5	0x110
matrix[1][0]	4	0x10c
matrix[0][2]	3	0x108
matrix[0][1]	2	0x104
matrix[0][0]	1	0x100

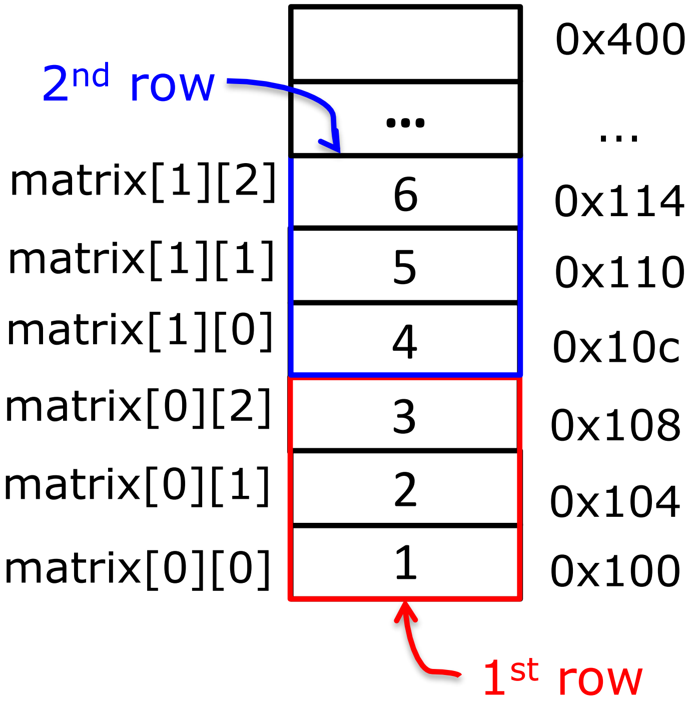
Memory layout



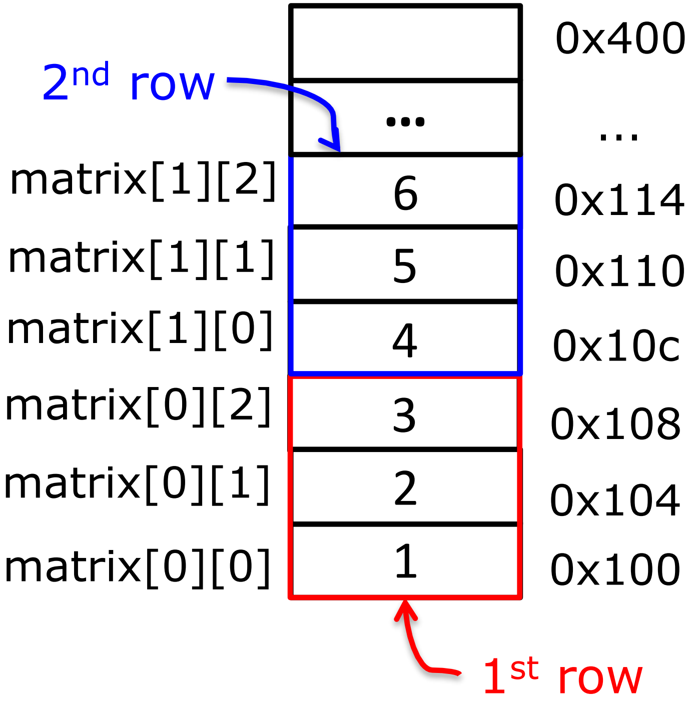
		0x400
	...	...
matrix[1][2]	6	0x114
matrix[1][1]	5	0x110
matrix[1][0]	4	0x10c
matrix[0][2]	3	0x108
matrix[0][1]	2	0x104
matrix[0][0]	1	0x100

1st row

Memory layout



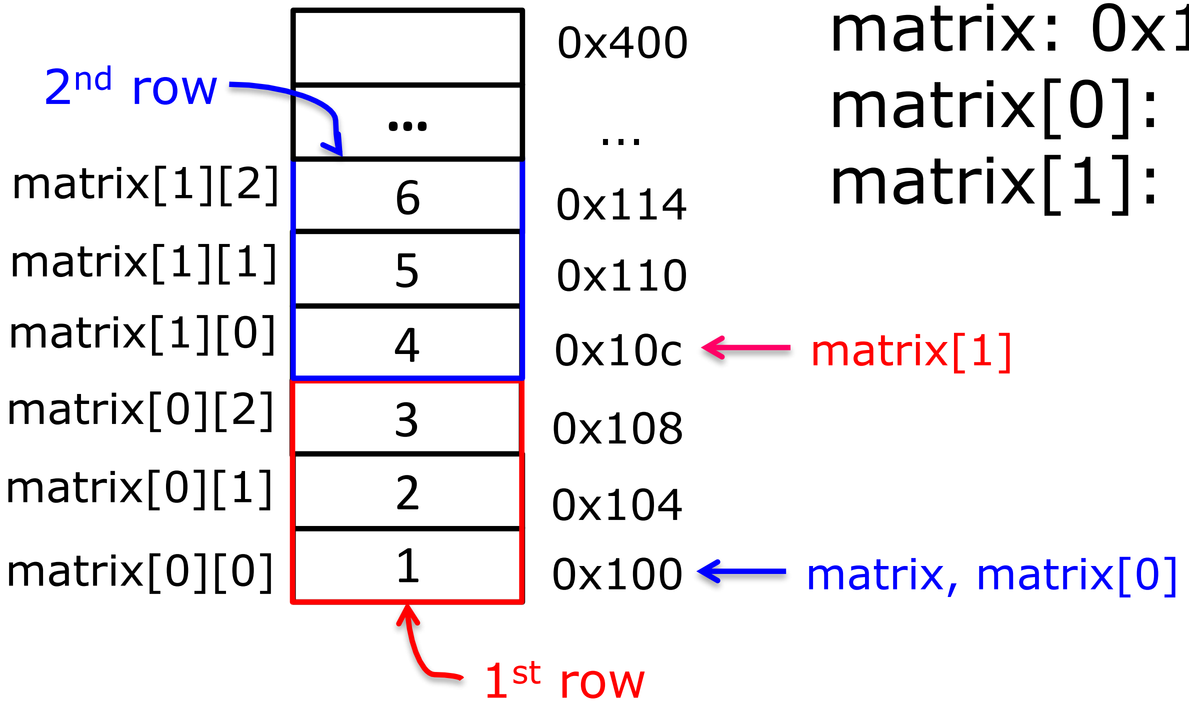
Pointers



What are the values of matrix, matrix[0] and matrix[1]?

```
int *p1, *p2, *p3;
p1 = (int *)matrix;
p2 = matrix[0];
p3 = matrix[1];

printf("matrix:%p matrix[0]:%p\
matrix[1]:%p\n", p1, p2, p3);
```

Pointers



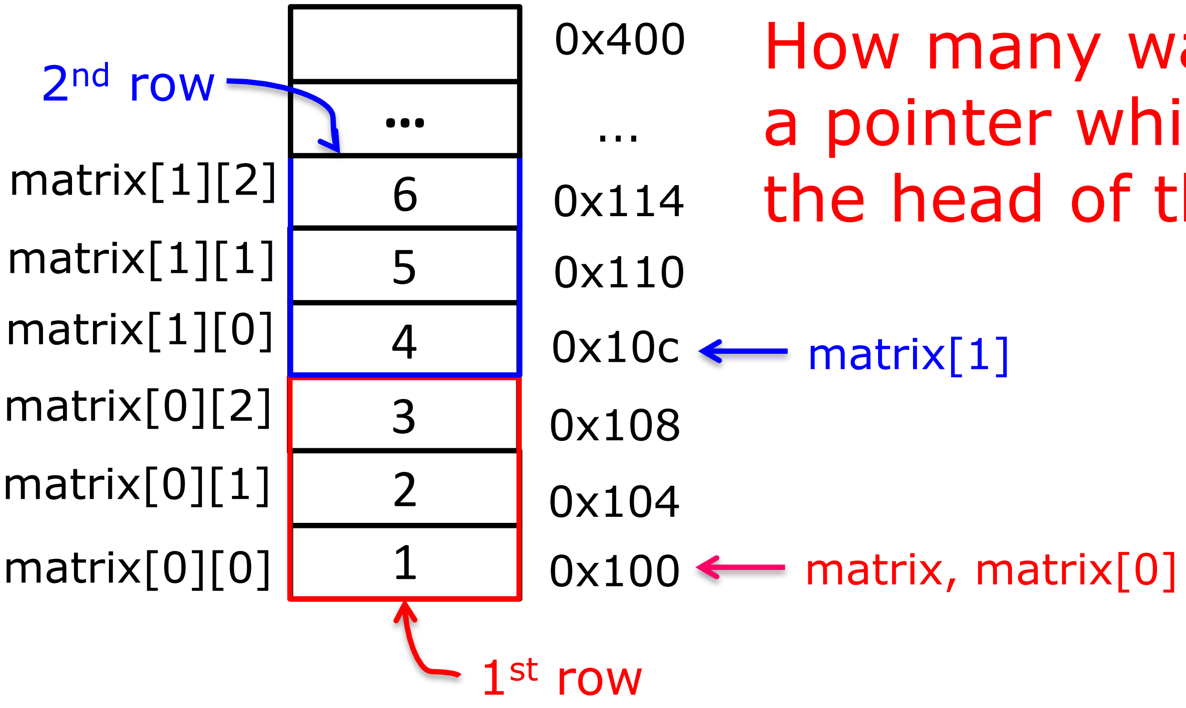
2nd row

		0x400
	...	...
matrix[1][2]	6	0x114
matrix[1][1]	5	0x110
matrix[1][0]	4	0x10c ← matrix[1]
matrix[0][2]	3	0x108
matrix[0][1]	2	0x104
matrix[0][0]	1	0x100 ← matrix, matrix[0]

1st row

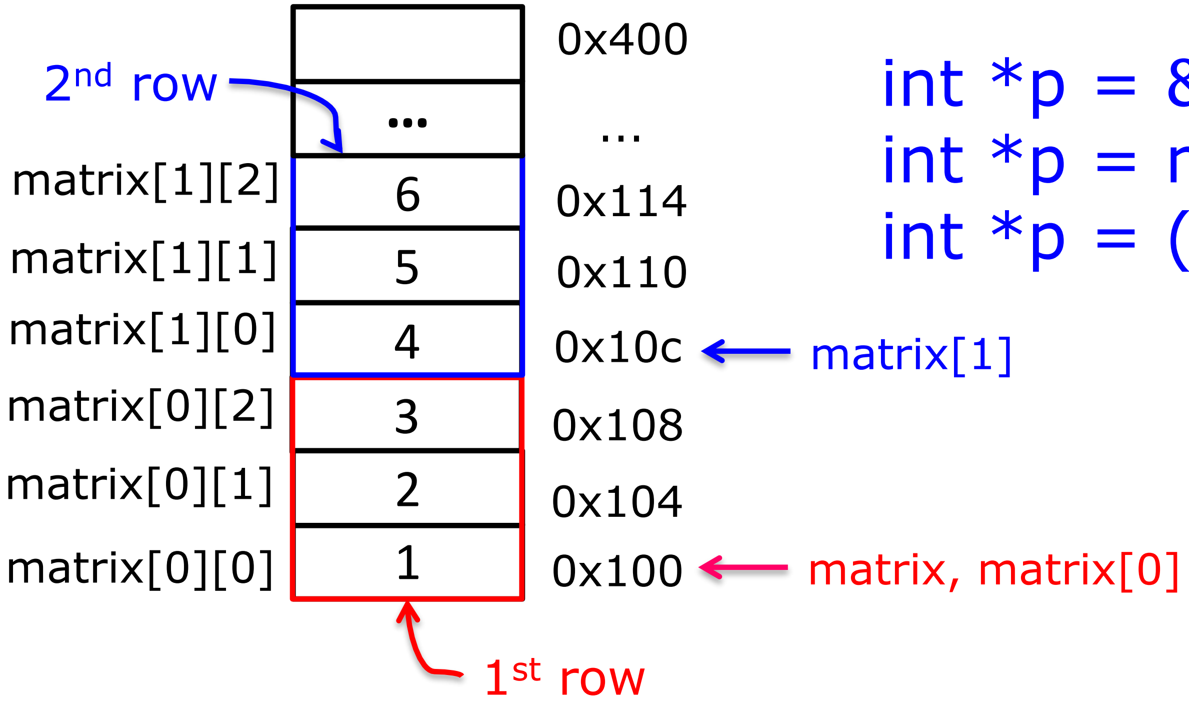
matrix: 0x100
matrix[0]: 0x100
matrix[1]: 0x10c

Pointers



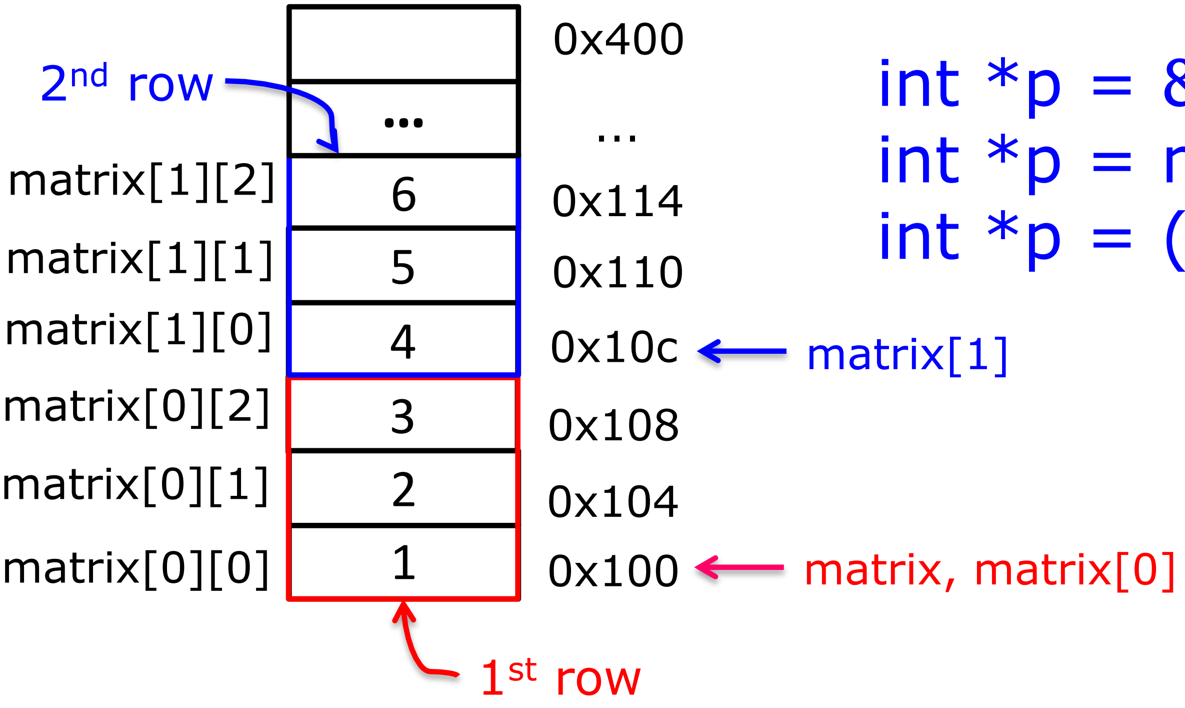
How many ways to define a pointer which points to the head of the array?

Pointers



```
int *p = &matrix[0][0];
int *p = matrix[0];
int *p = (int *)matrix;
```

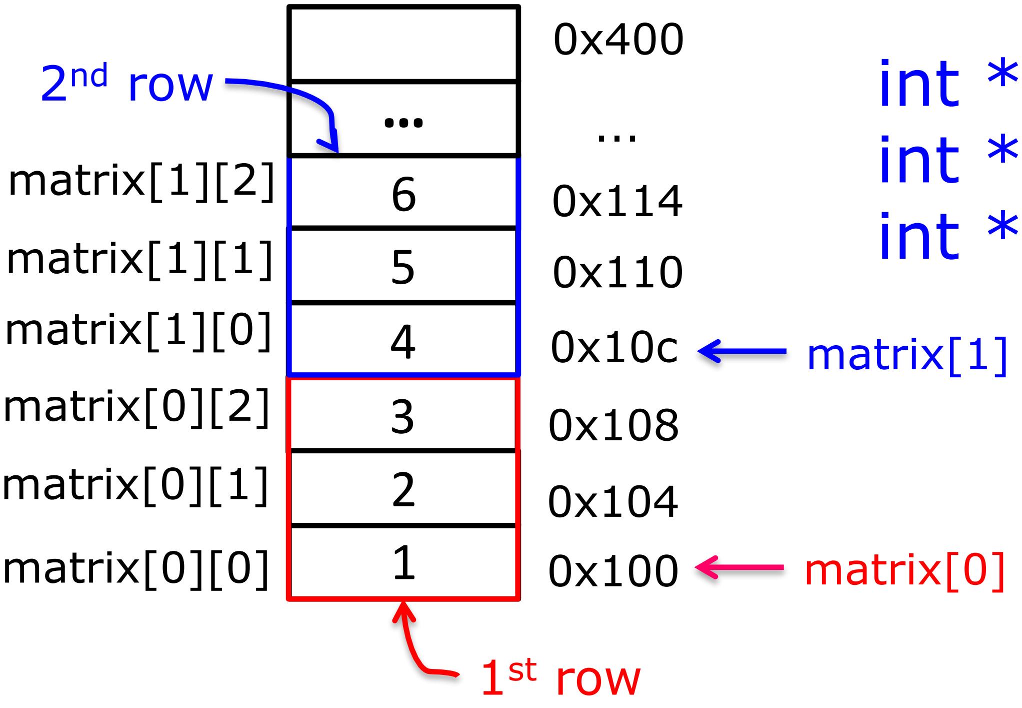
Pointers



```
int *p = &matrix[0][0];
int *p = matrix[0];
int *p = (int *)matrix;
```

How to access matrix[1][0] with p?

Pointers



2nd row

1st row

		0x400
	...	...
matrix[1][2]	6	0x114
matrix[1][1]	5	0x110
matrix[1][0]	4	0x10c ← matrix[1]
matrix[0][2]	3	0x108
matrix[0][1]	2	0x104
matrix[0][0]	1	0x100 ← matrix[0]

```
int *p = &matrix[0][0];
int *p = matrix[0];
int *p = (int *)matrix;
```

matrix[1][0]: *(p + 3)
p[3]

A general question

Given a 2D array matrix[m][n] and a pointer p which points to matrix[0][0], how to use p to access matrix[i][j]?

A general question

Given a 2D array matrix[m][n] and a pointer p which points to matrix[0][0], how to use p to access matrix[i][j]?

address of matrix[i][j]: p + i * n + j

Accessing 2D array using pointer

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

```
for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 3; j++) {
        printf(“%d\n”, matrix[2][3]);
    }
}
```

OR

```
int *p = matrix[0]; // or int *p = (int *)matrix;
for (int i = 0; i < 2*3; i++) {
    printf(“%d\n”, p[i]);
}
```

Structs

Struct stores fields of different types contiguously in memory

Structure

- Array: a block of n consecutive elements of the same type.
- How to define a group of objects, each of which may be of a different type?

Structure

Name of the struct

```
struct student {
    int id;
    char name[100];
};
```

Structure

```
struct student {
    int id;
    char name[100];
};
```

Field 1: a integer

Structure

```
struct student {
    int id;
    char name[100];
};
```

Field 2: an array

Structure

```
struct student {
    int id;
    char name[100];
};

struct student t;
```

← define an object with type student

Structure

```
struct student {
    int id;
    char name[100];
};

struct student t;

t.id = 1024
t.name[0] = ‘z’
t.name[1] = ‘h’
…
```

Access the fields of this object

Structure

```
typedef struct {
    int id;
    char name[100];
} student;
```

~~struct~~ student t;

```
t.id = 1024
t.name[0] = ‘z’
t.name[1] = ‘h’
…
```

Structure

```
typedef struct {
    int id;
    char name[100];
} student;
```

Structure's size

1st question:

What is the size of structure student?

```
typedef struct {
    int id;
    char name[100];
} student;
```

Structure’s size

What is the size of structure A?

```
typedef struct {
    int id;
} A;
```

Structure’s size

What is the size of structure A?

```
typedef struct {
    int id;
} A;
```

Answer: 4

Structure’s size

What is the size of structure B?

```
typedef struct {
    char name[100];
} B;
```

Structure’s size

What is the size of structure B?

```
typedef struct {
    char name[100];
} B;
```

Answer: 100

Structure’s size

1st question:

What is the size of structure student?

```
typedef struct {
    int id;
    char name[100];
} student;
```

Structure's size

1st question:

What is the size of structure student?

```
typedef struct {
    int id;
    char name[100];
} student;
```

Answer: 104

Structure's size

2st question:

What is the size of structure student?

```
typedef struct {
     int id;
     char gender;
} student;
```

Structure’s size

2st question:

What is the size of structure student?

```
typedef struct {
     int id;
     char gender;
} student;
```

Answer: 5 ?

Structure’s size

2st question:

What is the size of structure student?

```
typedef struct {
     int id;
     char gender;
} student;
```

Answer: ~~5~~ ?

Structure’s size

2st question:

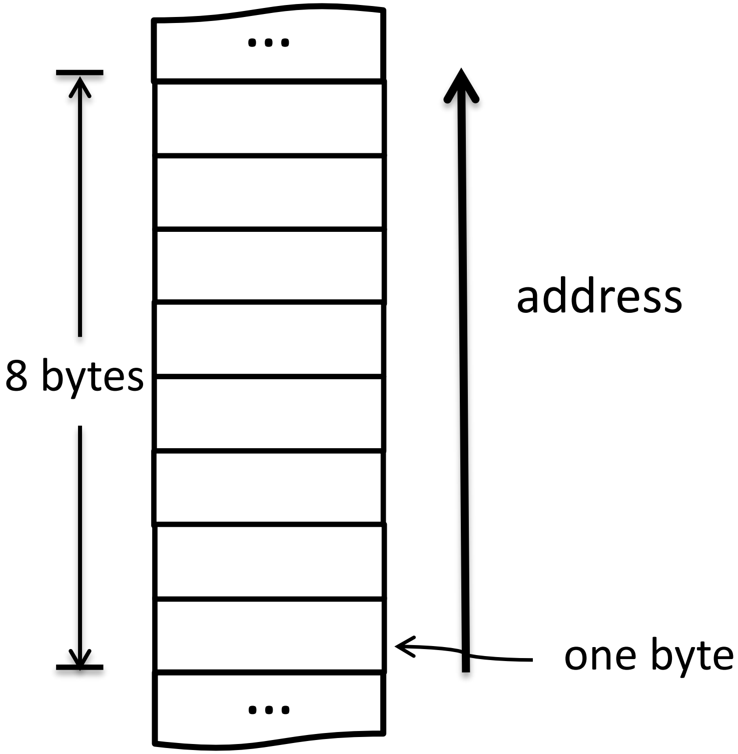
What is the size of structure student?

```
typedef struct {
     int id;
     char gender;
} student;
```

Answer: 8

Structure's size

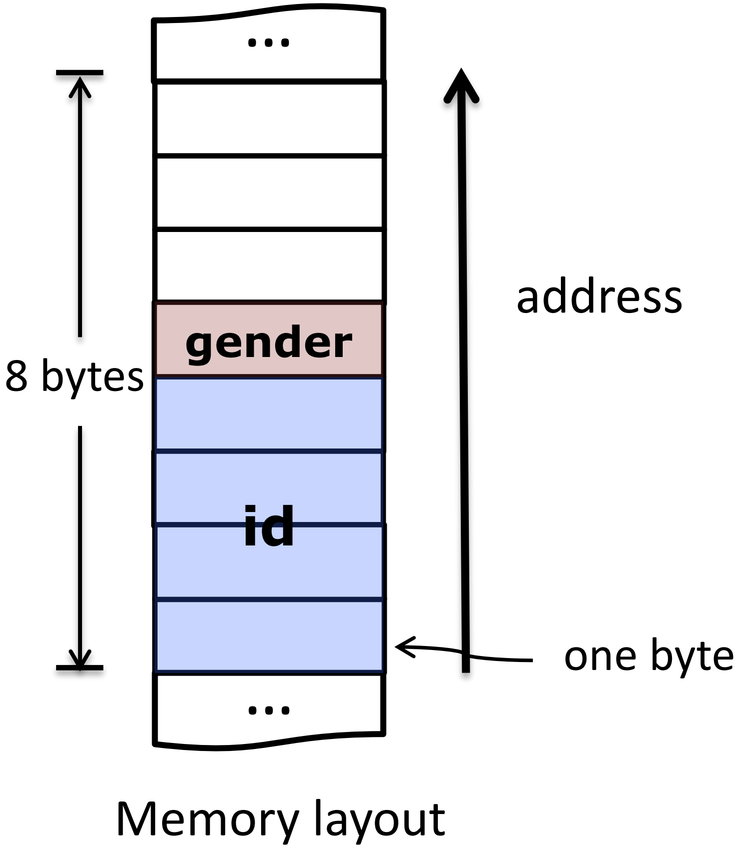
```
typedef struct {
    int id;
    char gender;
} student;
```



Memory layout

Structure's size

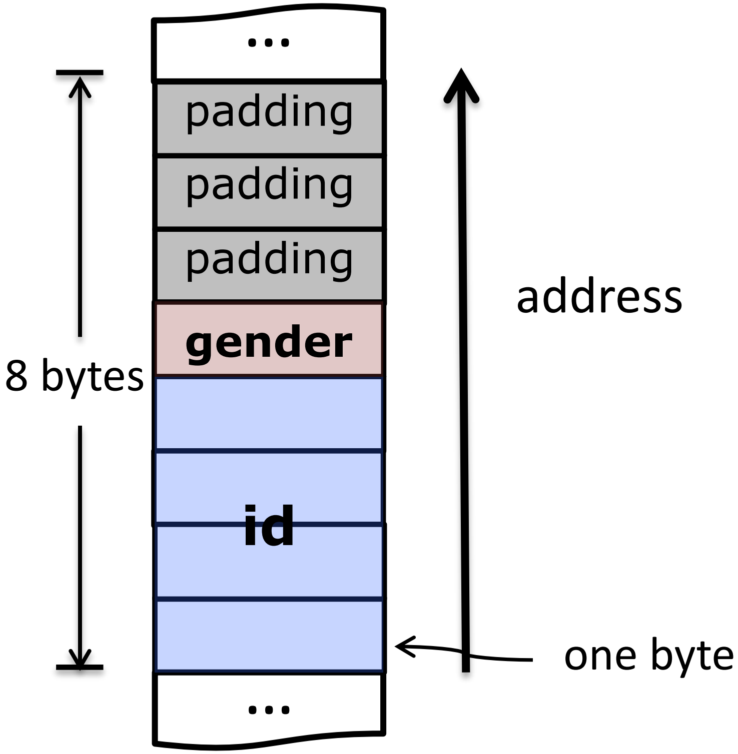
```
typedef struct {
     int id;
     char gender;
} student;
```



Memory layout

Structure's size

```
typedef struct {
    int id;
    char gender;
} student;
```



Memory Layout

Data alignment

Put the data at a memory address equal to some **multiple of the word size** through the **data structure padding**

Data alignment

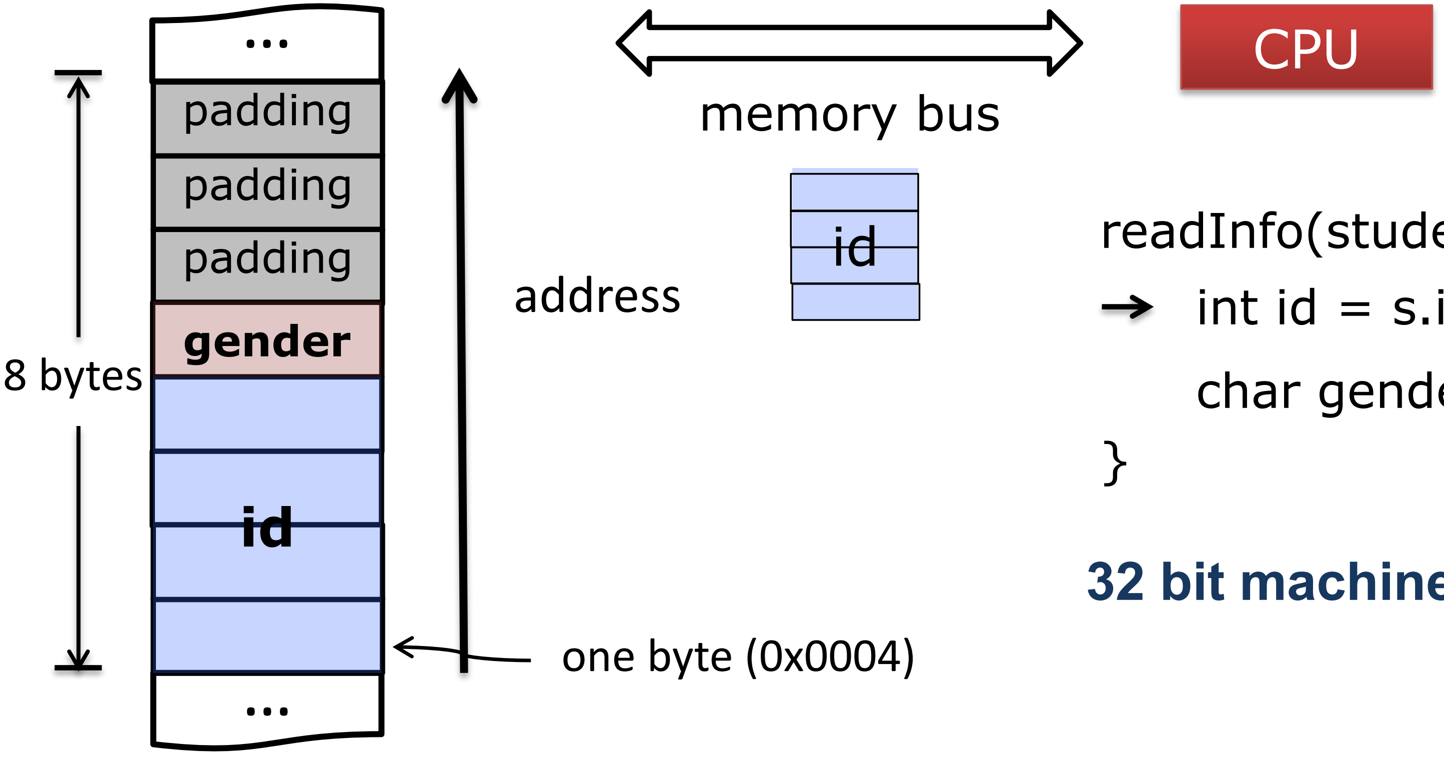
Put the data at a memory address equal to some **multiple of the word size** through the **data structure padding**

CPU reads/writes data from/into memory in word sized chunks.
(e.g., 8 bytes chunks on a 64-bit system)

Ensure read/write each primary type with a single memory access.

Data alignment

Put the data at a memory address equal to some **multiple of the word size** through the **data structure padding**



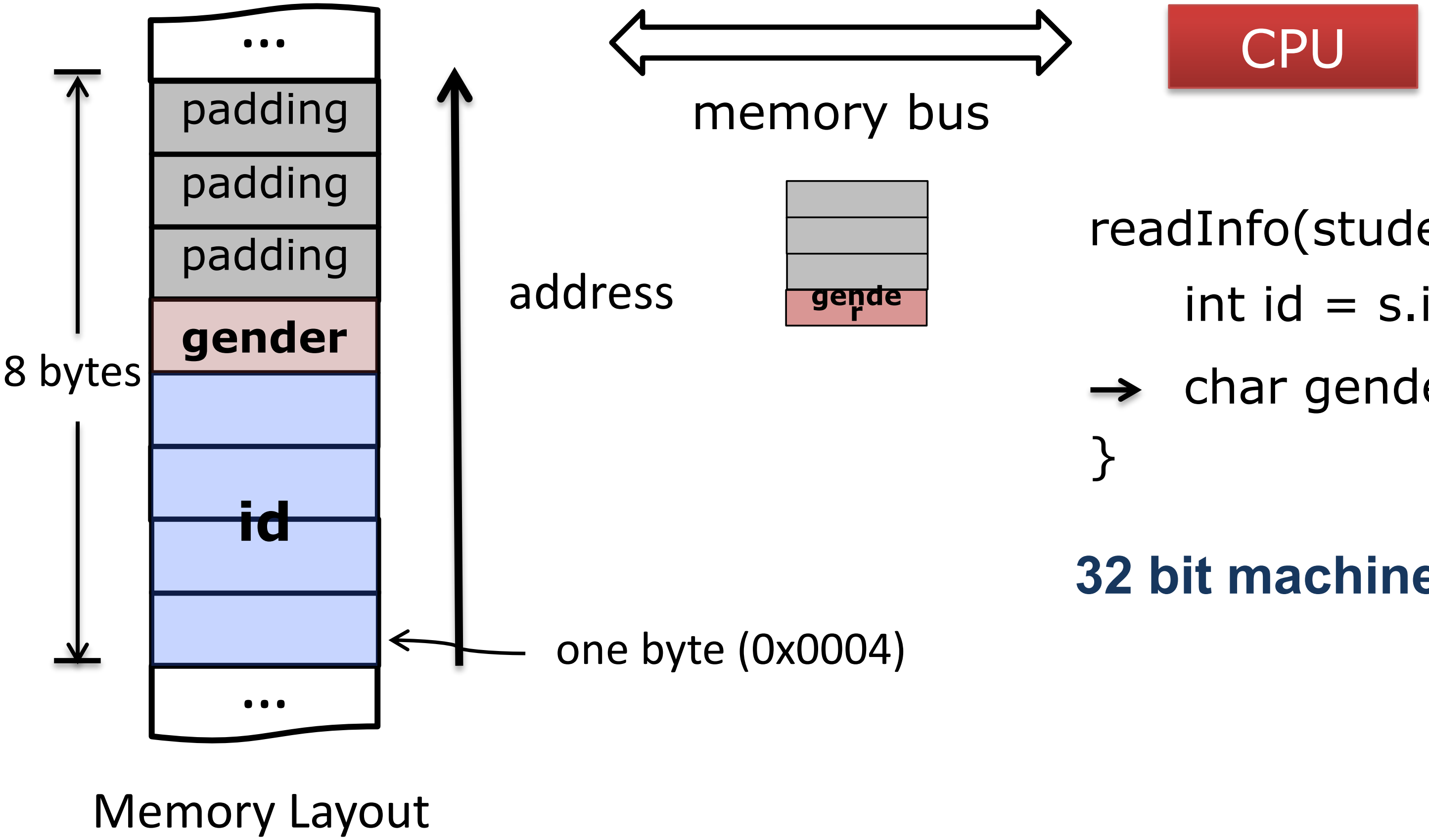
Memory Layout

```
readInfo(student s) {
→   int id = s.id
    char gender = s.gender
}
```

32 bit machine

Data alignment

Put the data at a memory address equal to some **multiple of the word size** through the **data structure padding**



Memory Layout

```
readInfo(student s) {
      int id = s.id
→     char gender = s.gender
}
```

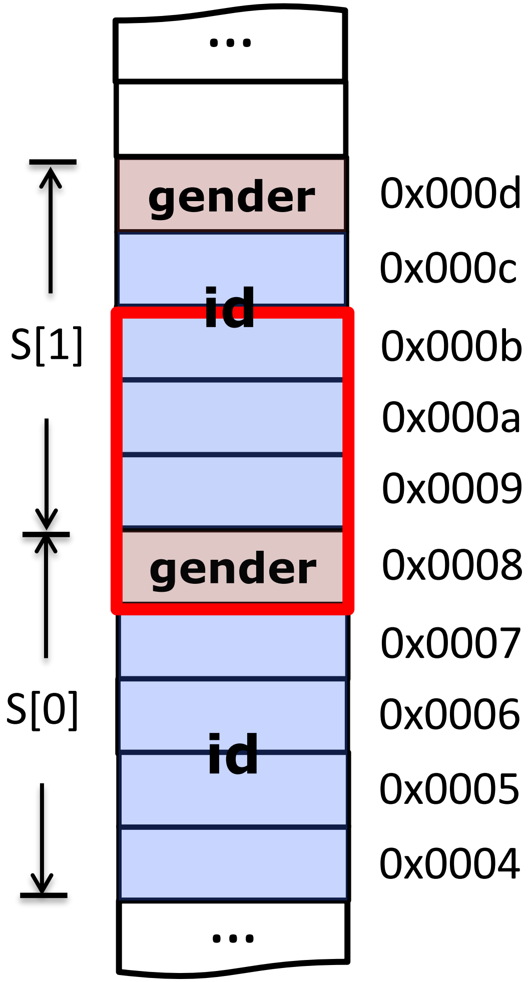
32 bit machine

Problem without data alignment

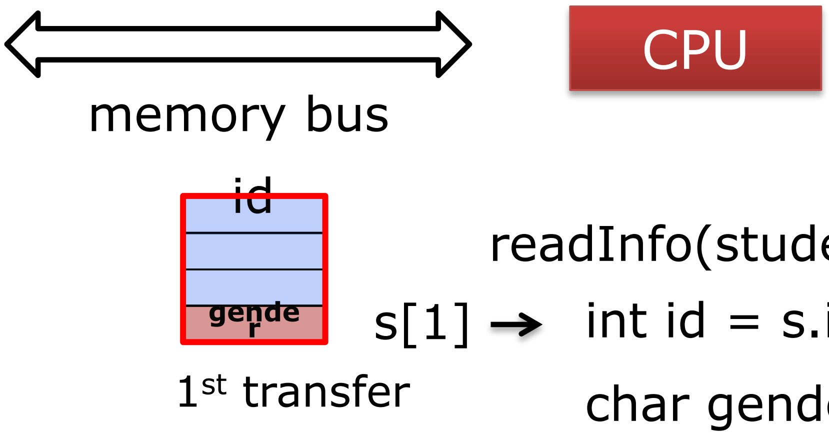
```
student s[2];

for(int i = 0; i < 2; i++) {
    readInfo(s[i])
}
```

Problem without data alignment

```
student s[2];

for(int i = 0; i < 2; i++) {
        readInfo(s[i])
}
```



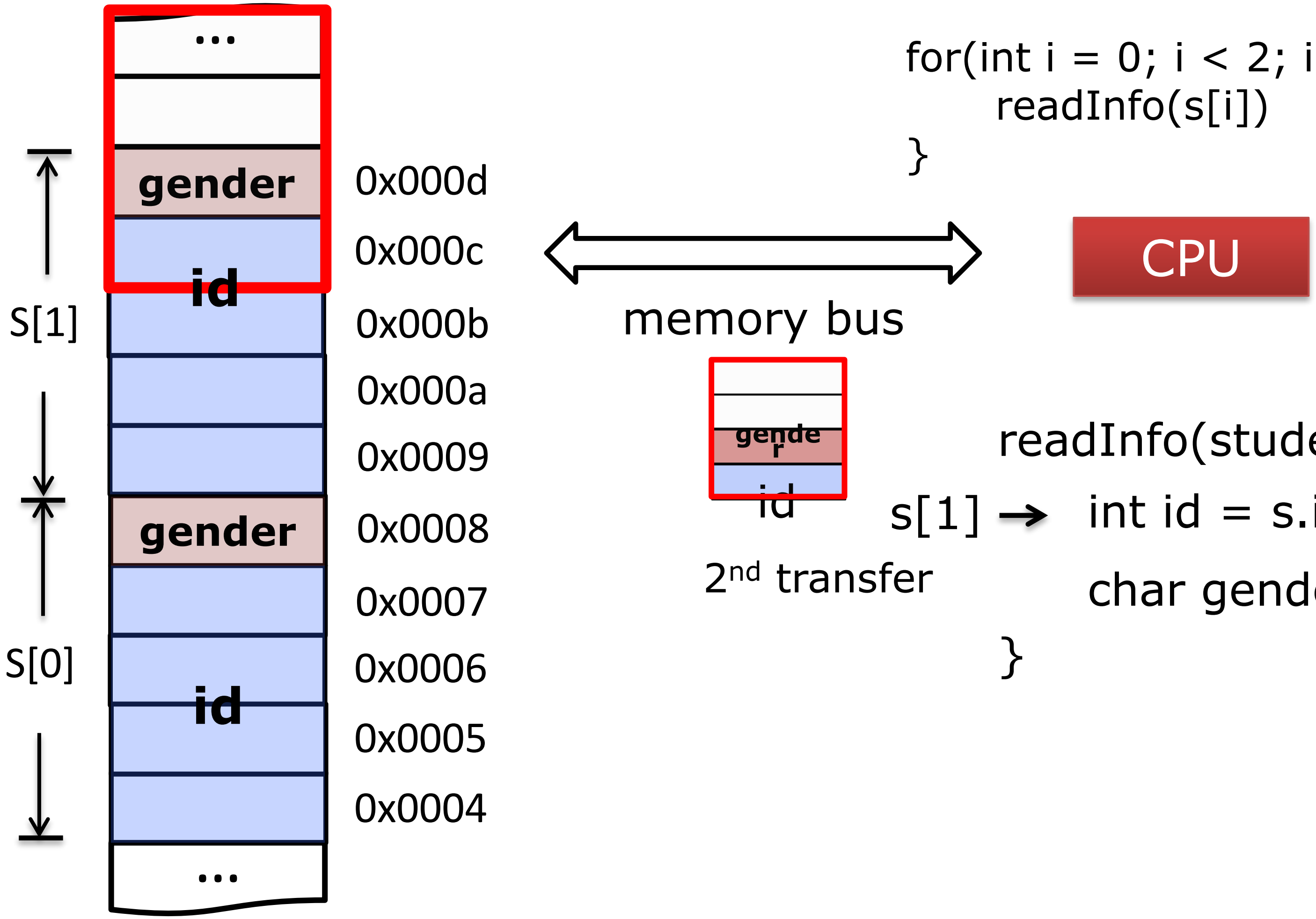
Memory Layout



```
readInfo(student s) {
s[1] →  int id = s.id
        char gender = s.gender
}
```

Problem without data alignment

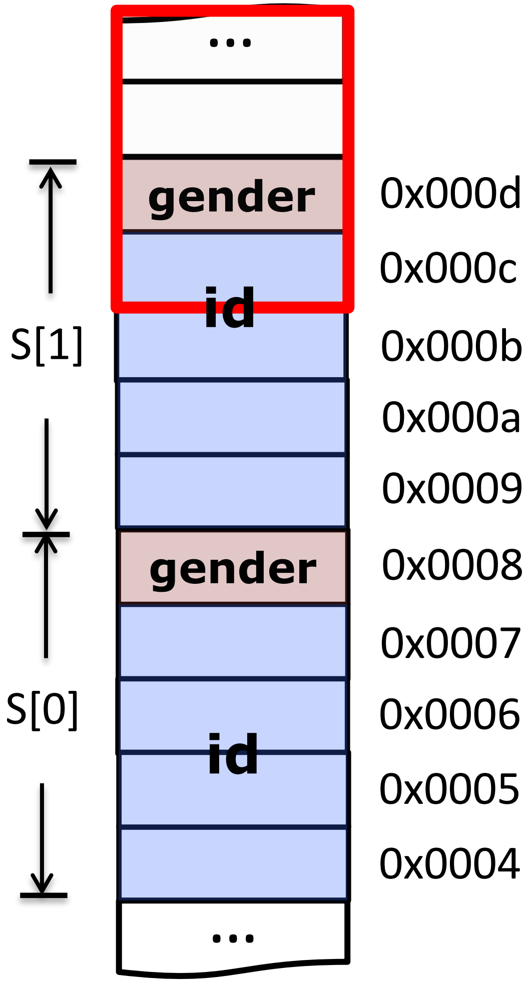
```
student s[2];

for(int i = 0; i < 2; i++) {
        readInfo(s[i])
}
```



```
readInfo(student s) {
    int id = s.id
    char gender = s.gender
}
```

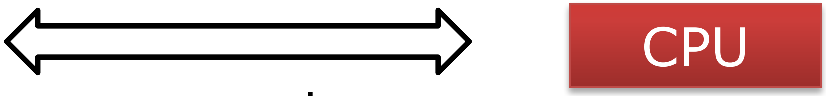
Memory Layout

Problem without data alignment

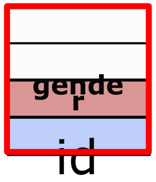


Memory Layout

```
student s[2];

for(int i = 0; i < 2; i++) {
        readInfo(s[i])
}
```



memory bus

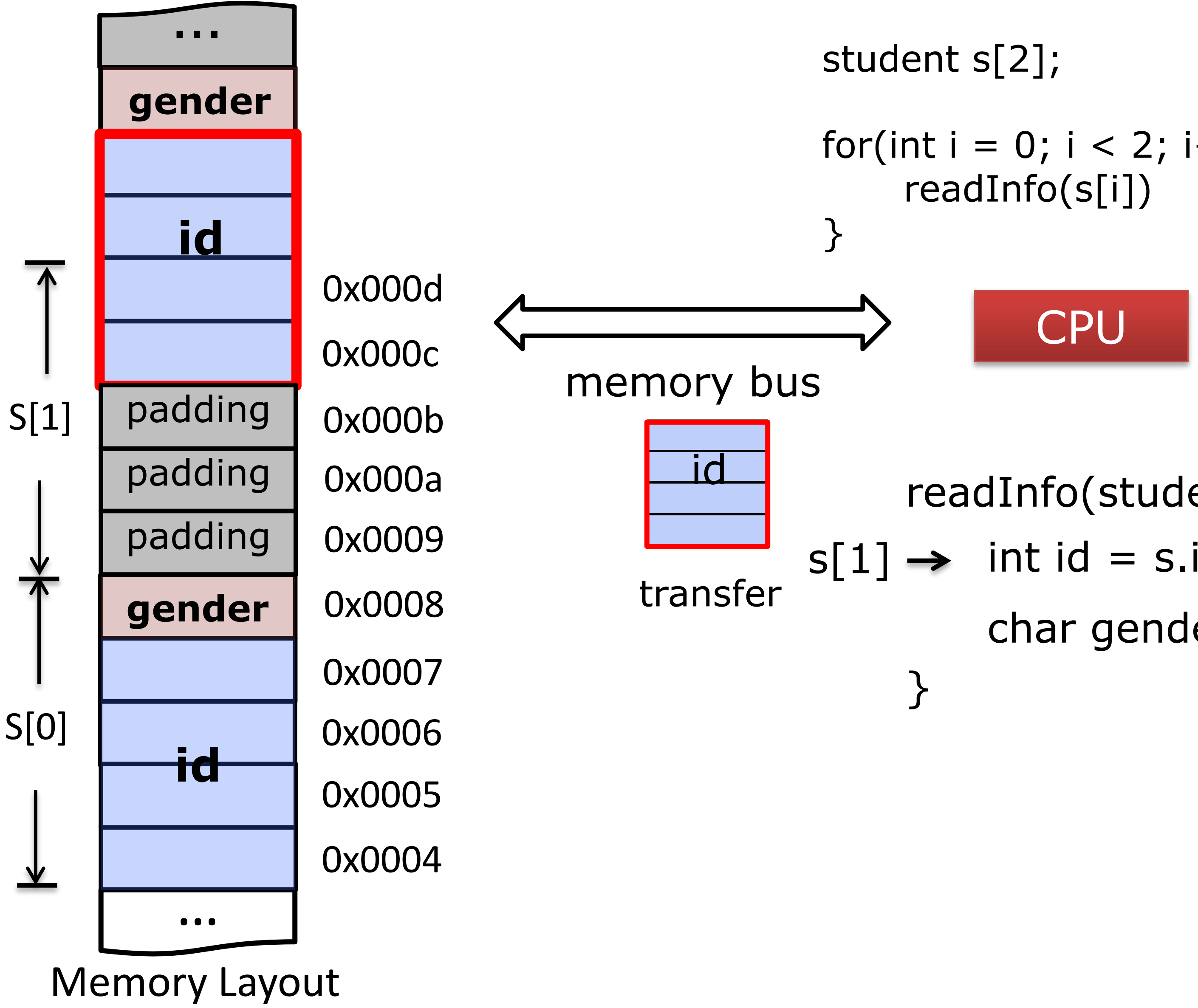


2nd transfer

```
readInfo(student s) {
s[1] →  int id = s.id
        char gender = s.gender
}
```

Performance and correctness issues

Data structure alignment



```
student s[2];

for(int i = 0; i < 2; i++) {
        readInfo(s[i])
}
```

```
        readInfo(student s) {
s[1] →     int id = s.id
           char gender = s.gender
        }
```

Questions

What’s the size/layout of following structs?

```
typedef struct {
    int a;
    char b;
    int c;
    char d;
} S_A;
```

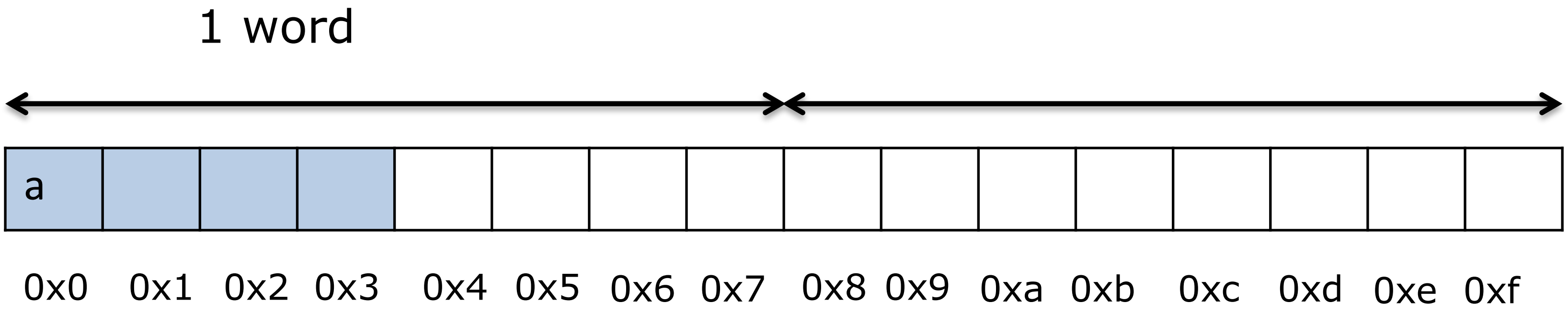
```
typedef struct {
    int a;
    int b;
    char c;
    char d;
} S_B;
```

Alignment rule:

Primitive data type of x bytes → Address must be multiple of x
(so each primary type can be transferred a single read)

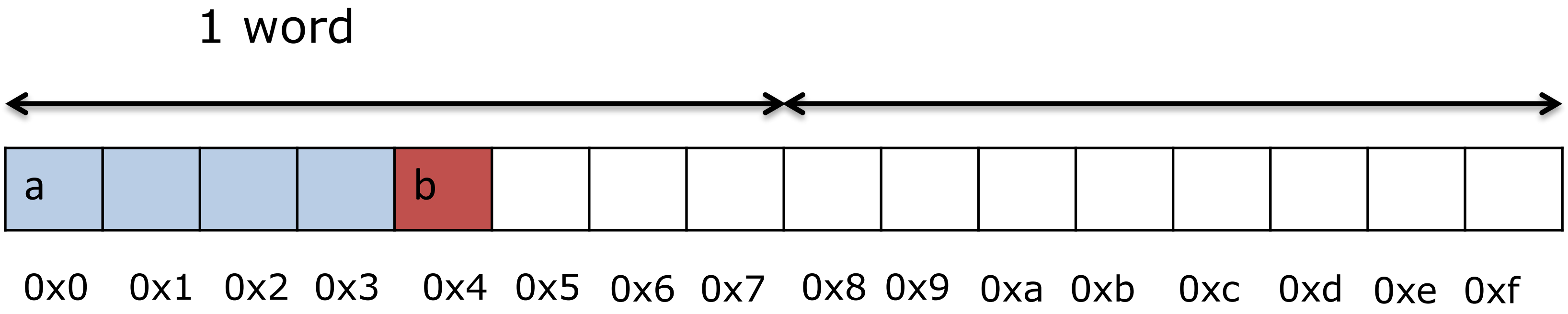
Questions

```
typedef struct {
    int a;
    char b;
    int c;
    char d;
} S_A;
```



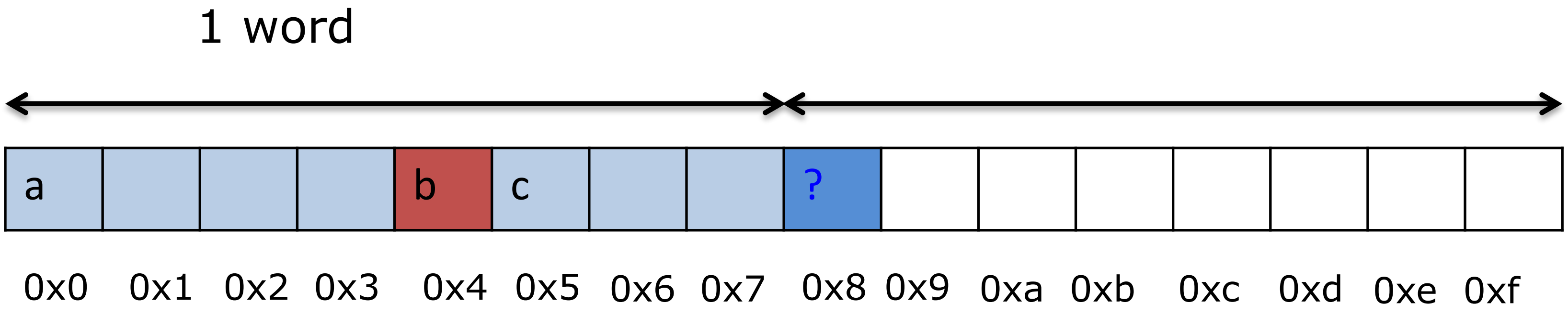
Questions

```
typedef struct {
    int a;
    char b;
    int c;
    char d;
} S_A;
```



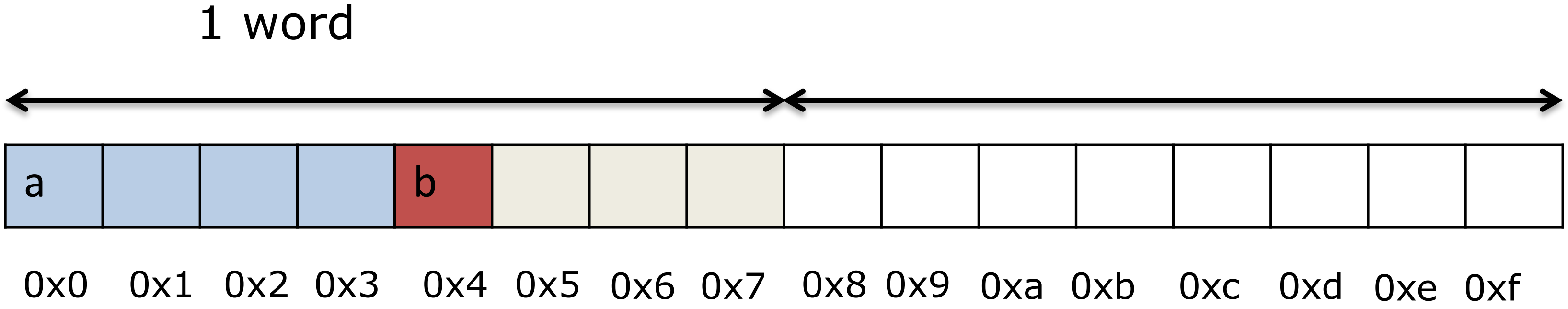
Questions

```
typedef struct {
    int a;
    char b;
    int c;
    char d;
} S_A;
```



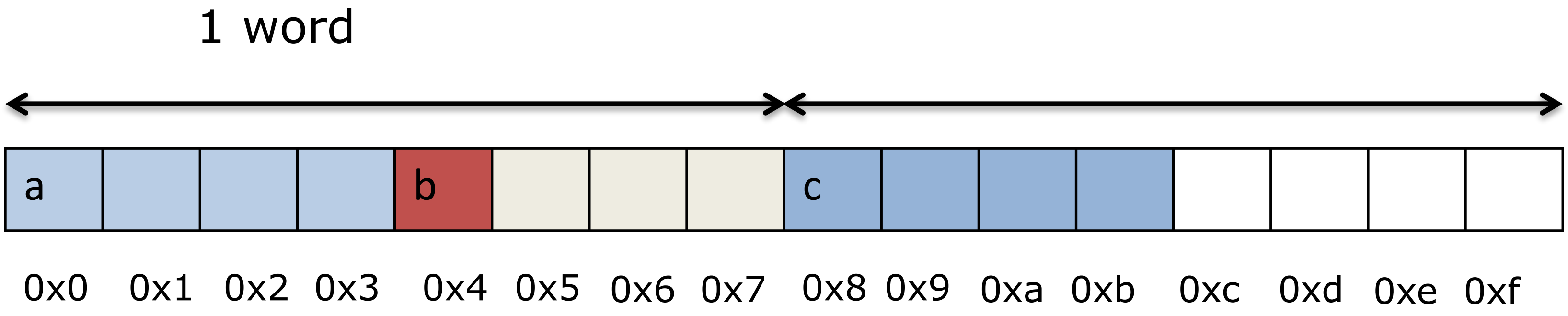
Questions

```
typedef struct {
    int a;
    char b;
    int c;
    char d;
} S_A;
```



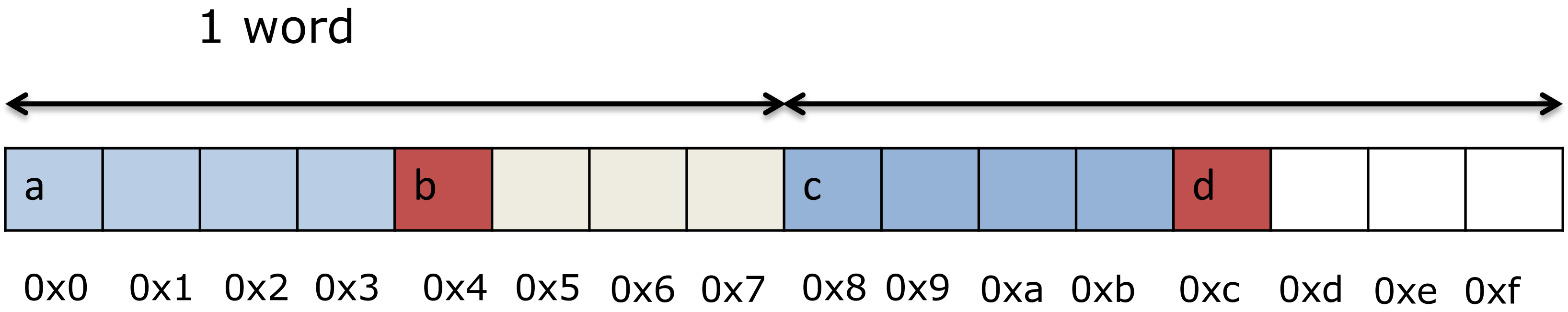
Questions

```
typedef struct {
    int a;
    char b;
    int c;
    char d;
} S_A;
```



Questions

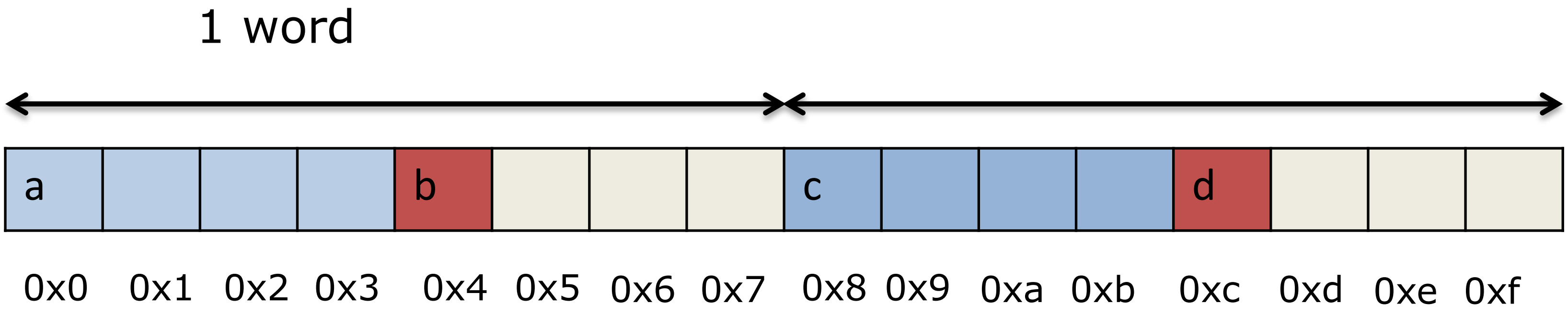
```
typedef struct {
    int a;
    char b;
    int c;
    char d;
} S_A;
```



0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
a				b				c				d			

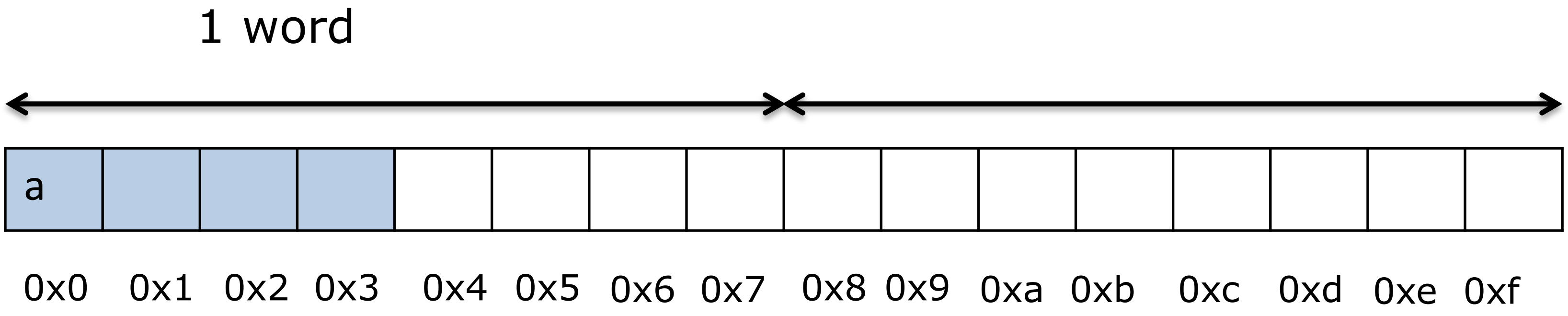
Questions

```
typedef struct {
    int a;
    char b;
    int c;
    char d;
} S_A;
```



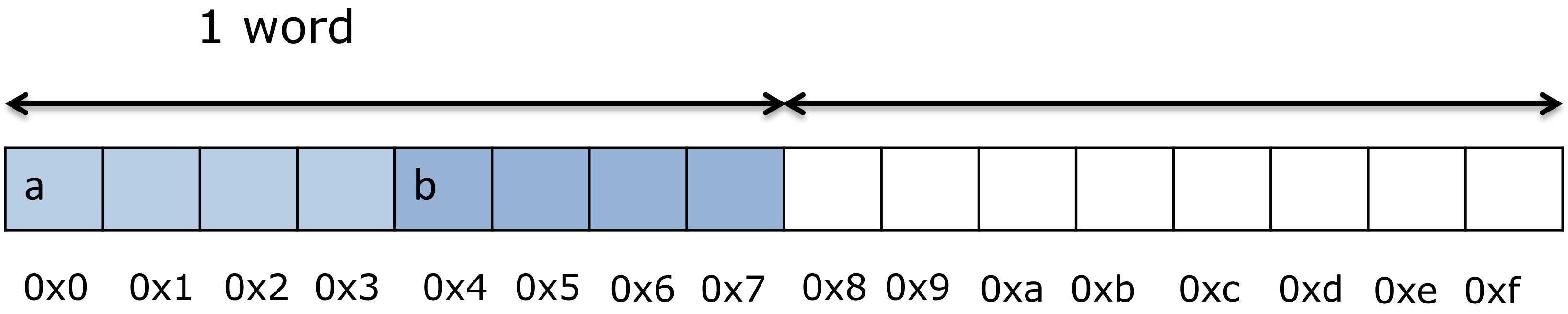
Questions

```
typedef struct {
    int a;
    int b;
    char c;
    char d;
} S_A;
```



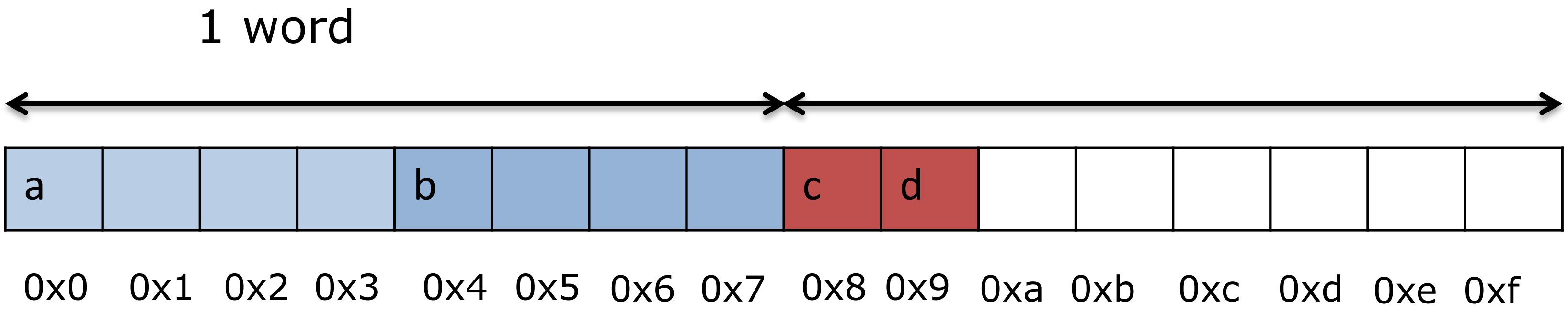
Questions

```
typedef struct {
    int a;
    int b;
    char c;
    char d;
} S_A;
```



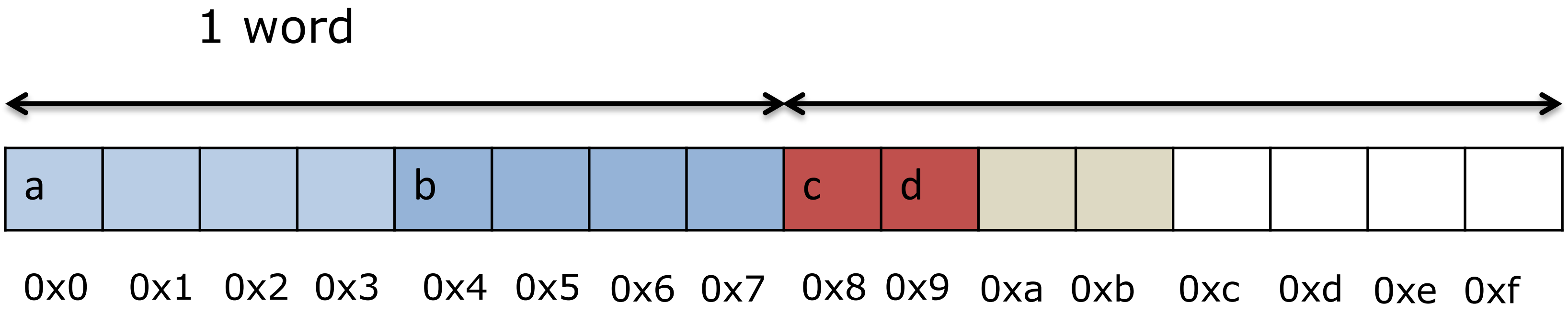
Questions

```
typedef struct {
    int a;
    int b;
    char c;
    char d;
} S_A;
```



Questions

```
typedef struct {
    int a;
    int b;
    char c;
    char d;
} S_A;
```



Pointer & Structure

```
typedef struct {
   int id;
   char gender;
} student;

student t = student{1, ‘m’};
student *p = &t;
p->id = 2;
```

Mallocs

Allocates a chunk of memory dynamically

Malloc

```
int a[10];
```

- Global variables are allocated space before program execution.
- Local variables are allocated at the entrance of a function (or a block) and de-allocated upon the exit of the function (or the block)

Malloc

Dynamically allocate a space

- malloc: allocate storage of a given size
- free: de-allocate previously malloc-ed storage

```
void *malloc(size_t size);
```

A void pointer is a pointer that has no associated data type with it. A void pointer can hold address of any type and can be casted to any type.

```
void free(void *ptr);
```

Malloc

Dynamically allocate a space

- malloc: allocate storage of a given size
- free: de-allocate previously malloc-ed storage

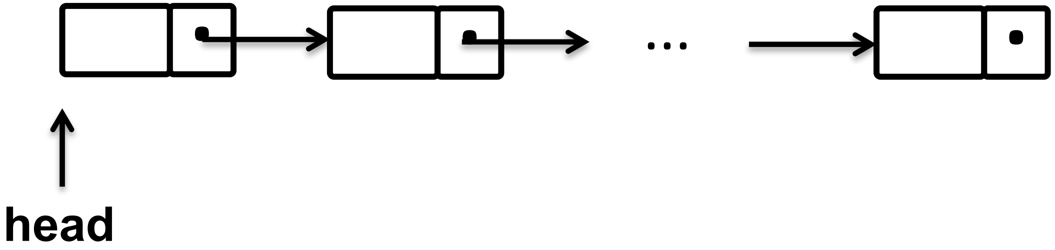
```
#include <stdlib.h>

int *newArr(int n) {
   int *p = (int*)malloc(sizeof(int) * n);
   return p;
}
```

Linked list in C: insertion

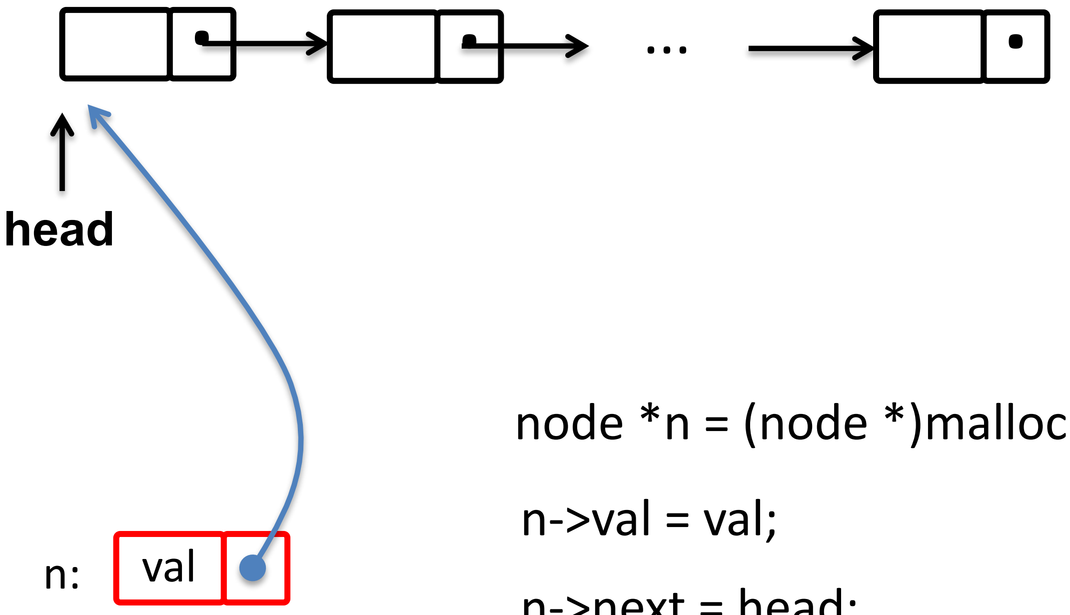
```
typedef struct {
     int val;
     struct node *next;
 }node;

// insert val into linked list to the head
// of the linked list and return the new
// head of the list.
node* insert(node* head, int val) {

}

int main() {
    node *head = NULL;
    for (int i = 0; i < 3; i++)
       head = insert(head, i);
}
```

* this linked list implementation is different from Lab1

Inserting into a linked list

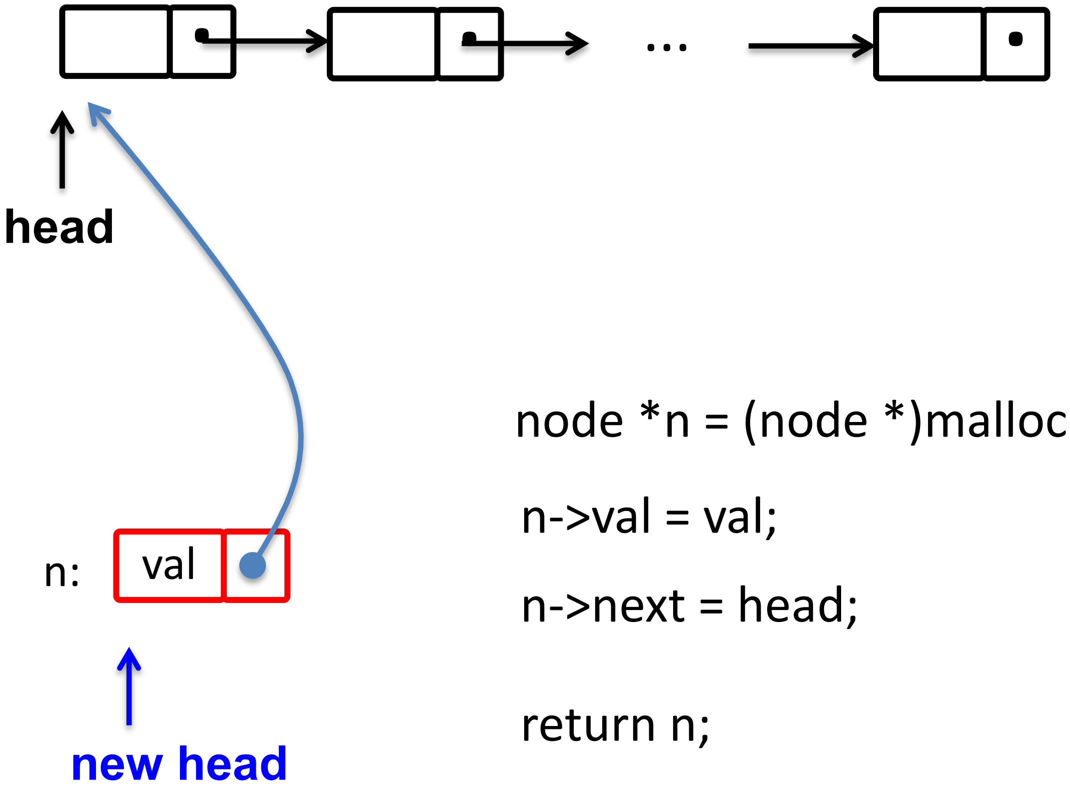


Inserting into a linked list



```
node *n = (node *)malloc(sizeof(node));

n->val = val;

n->next = head;
```

Inserting into a linked list

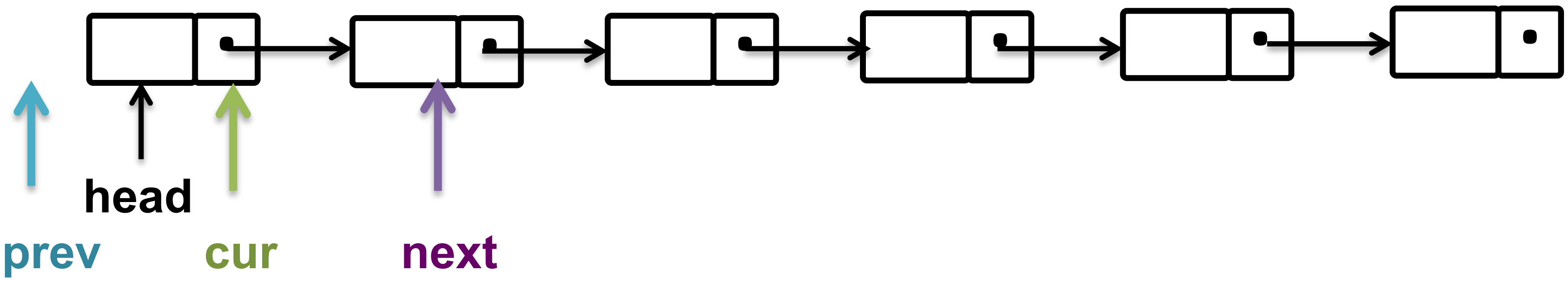


```
node *n = (node *)malloc(sizeof(node));

n->val = val;

n->next = head;

return n;
```

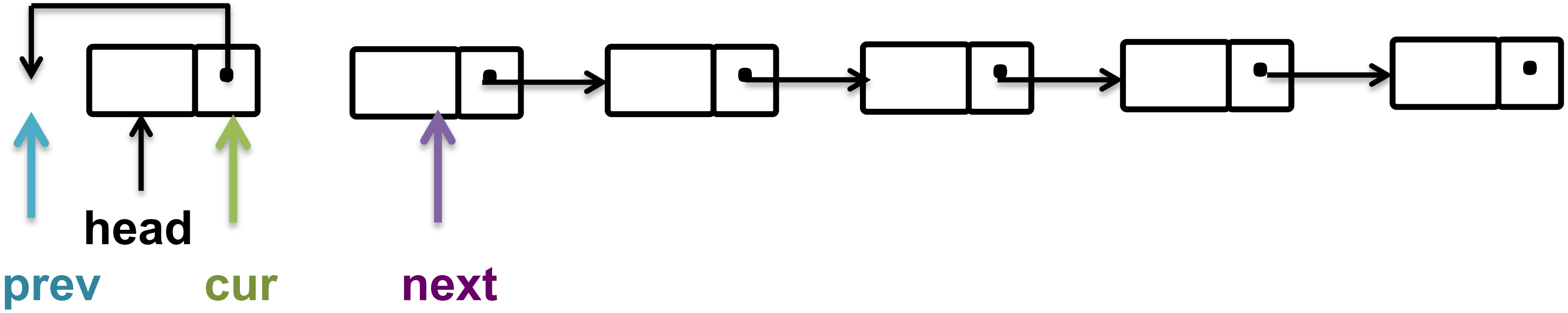
Exercise 1: Reverse a linked list

```
struct node {
     int val;
     struct node *next;
 };

struct node*
reverseList(struct node* head) {
    // your code here
}
```

Reverse a linked list

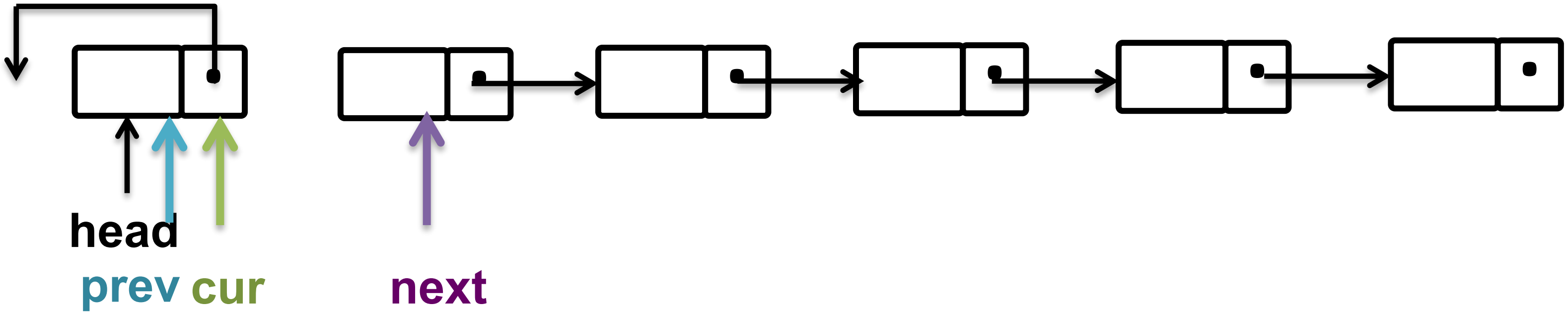


Reverse a linked list



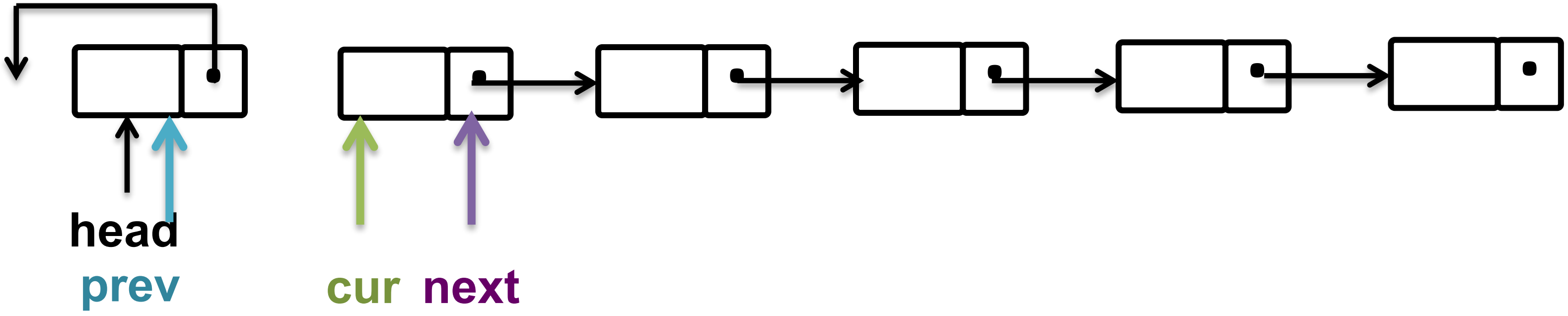
cur->next = prev

Reverse a linked list



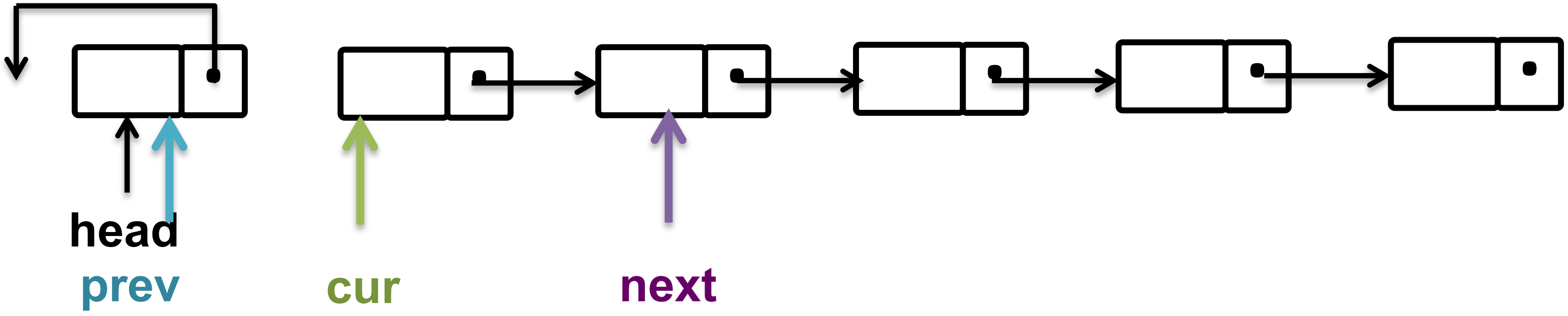
```
cur->next = prev
prev = cur
```

Reverse a linked list



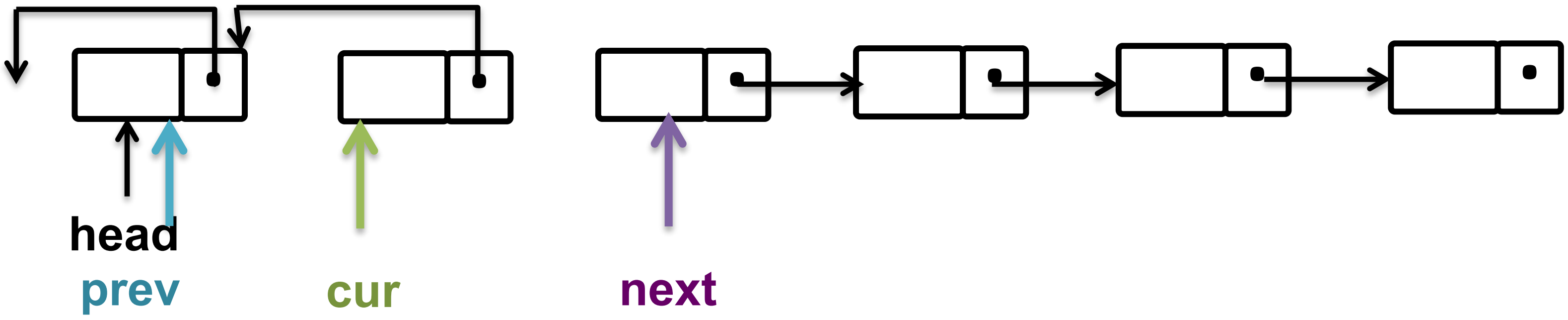
```
cur->next = prev
prev = cur
cur = next
```

Reverse a linked list



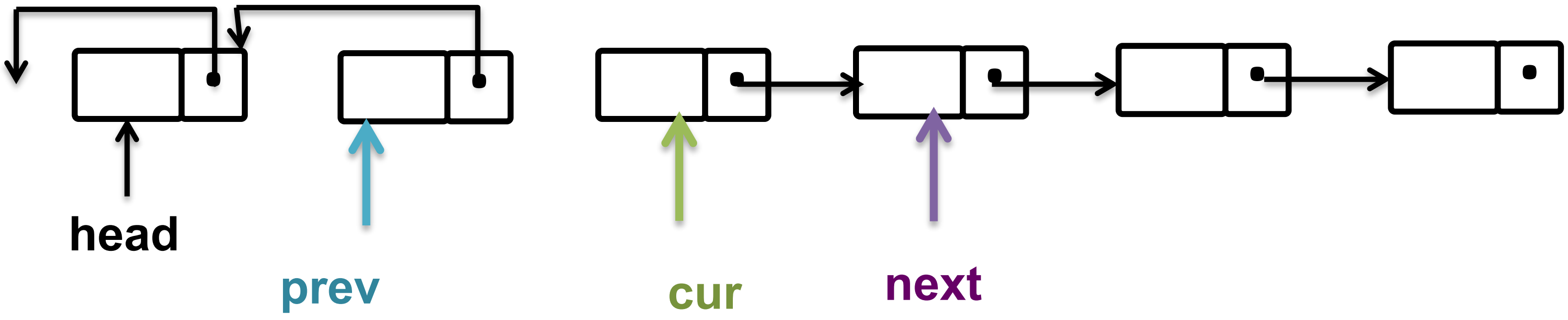
```
cur->next = prev
prev = cur
cur = next
next = cur->next
```

Reverse a linked list



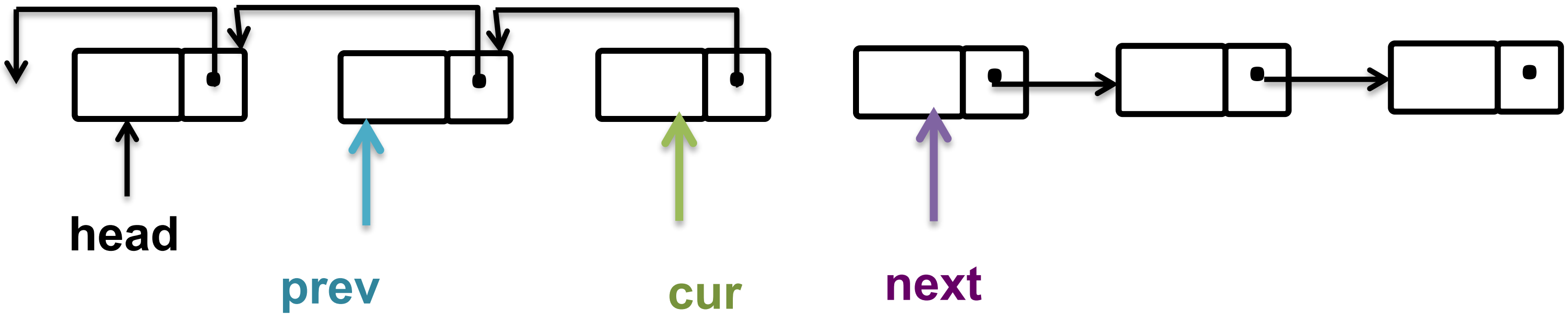
cur->next = prev
prev = cur
cur = next
next = cur->next

Reverse a linked list



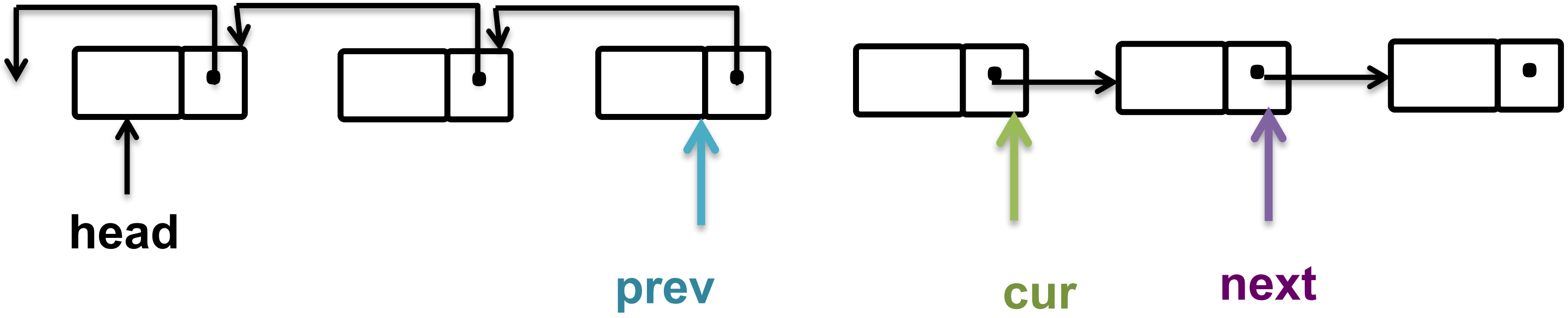
```
cur->next = prev
prev = cur
cur = next
next = cur->next
```

Reverse a linked list



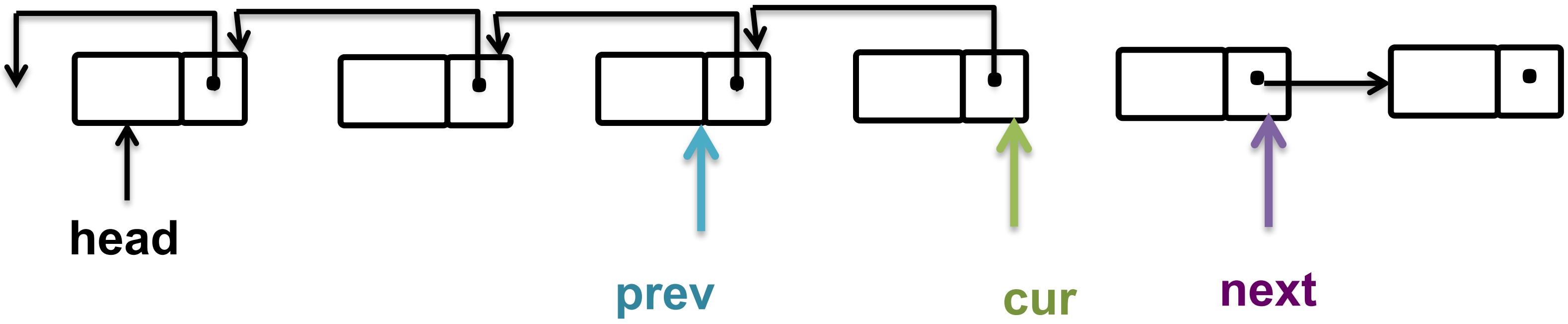
```
cur->next = prev
prev = cur
cur = next
next = cur->next
```

Reverse a linked list



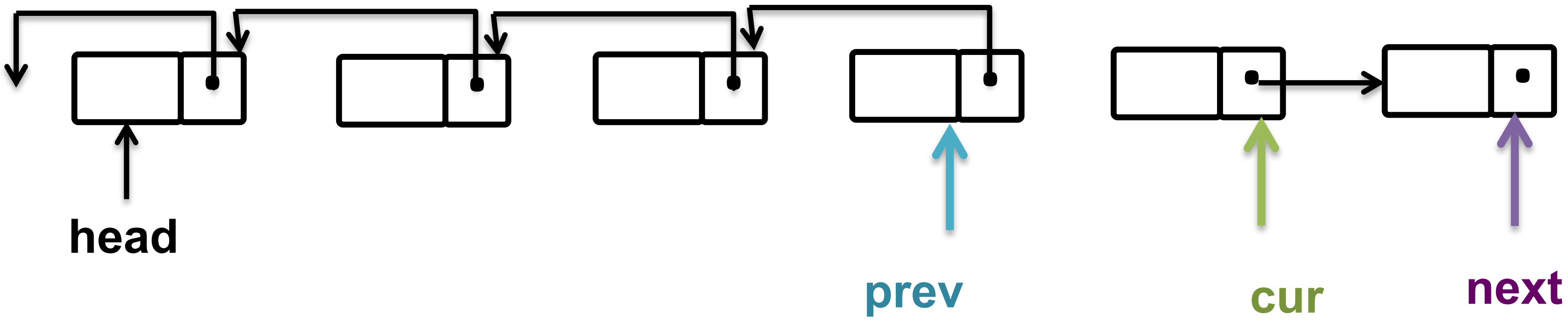
cur->next = prev
prev = cur
cur = next
next = cur->next

Reverse a linked list



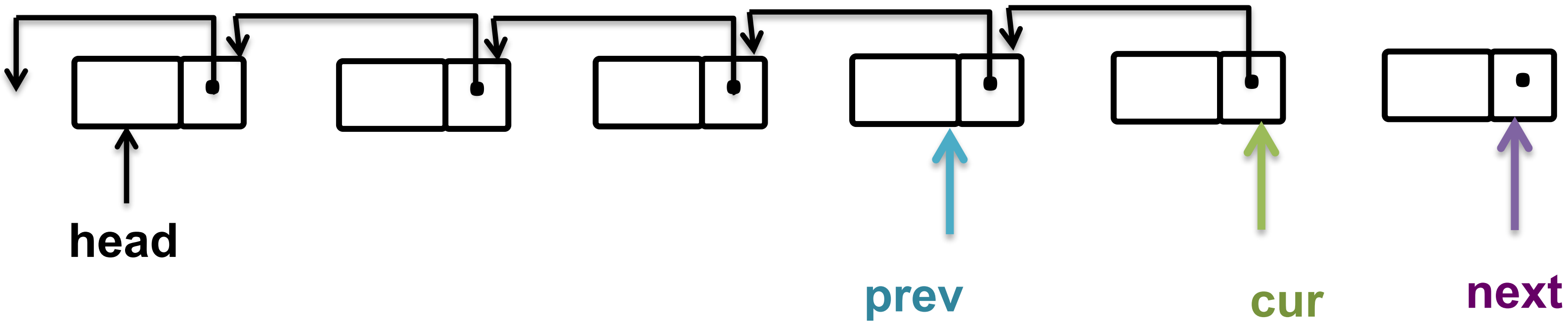
```
cur->next = prev
prev = cur
cur = next
next = cur->next
```

Reverse a linked list



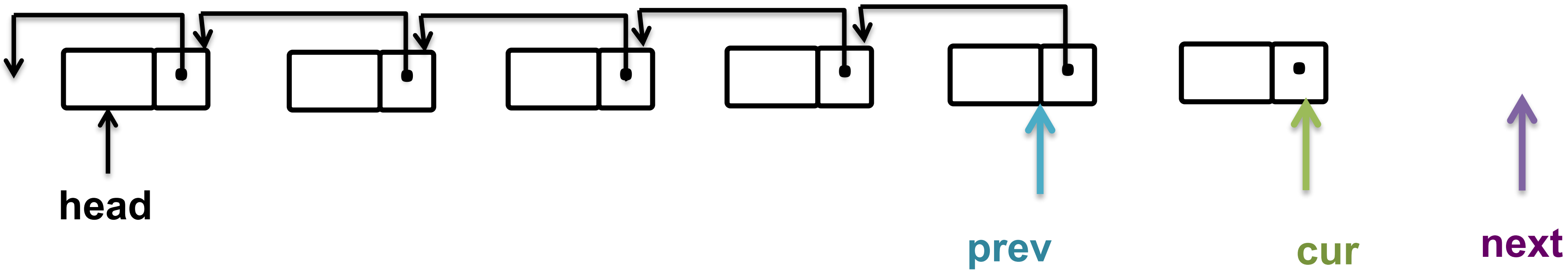
cur->next = prev
prev = cur
cur = next
next = cur->next

Reverse a linked list



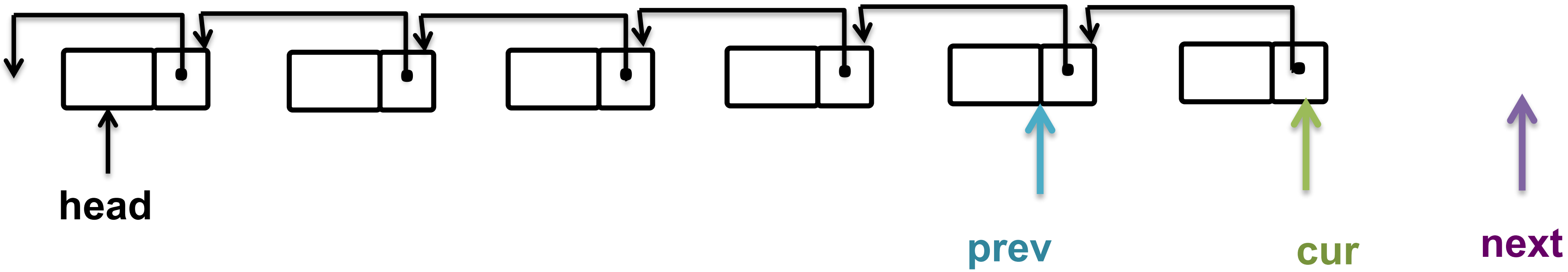
```
cur->next = prev
prev = cur
cur = next
next = cur->next
```

Reverse a linked list



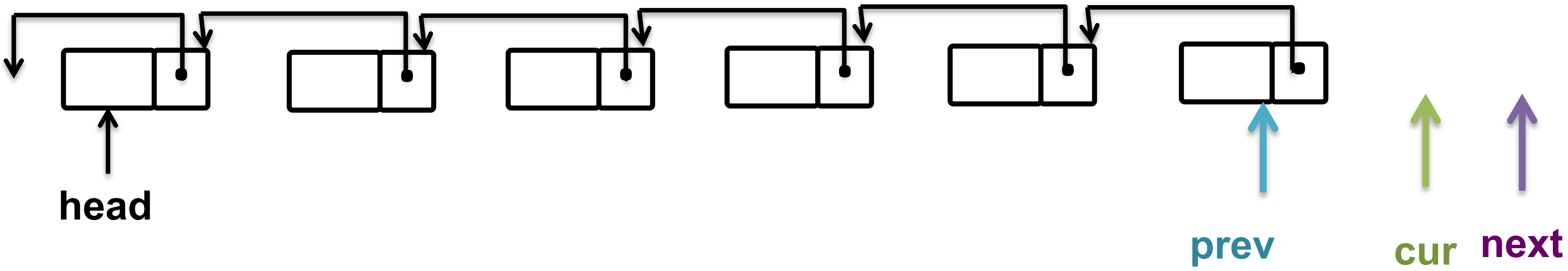
```
cur->next = prev
prev = cur
cur = next
next = cur->next
```

Reverse a linked list



```
cur->next = prev
prev = cur
cur = next
next = cur->next
```

Reverse a linked list



```
cur->next = prev
prev = cur
cur = next
next = cur->next
```

Reverse a linked list

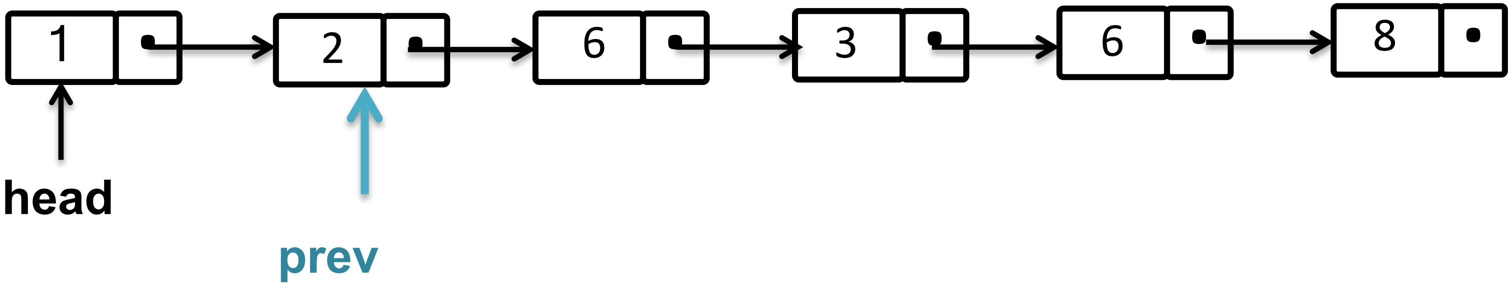
```
struct node {
     int val;
     struct node *next;
 };

struct node*
reverseList(struct node* head) {

  node *prev = null;
  node *curr = head;
  while (curr != null) {
    node *next = curr->next;
    curr->next = prev;
    prev = curr;
    curr = next;
  }
  return prev;
}
```

Exercise 2: Remove an element

```
struct node {
     int val;
     struct node *next;
 };

struct node*
removeElements(struct node* head, int val)
{
    // your code here
}
```

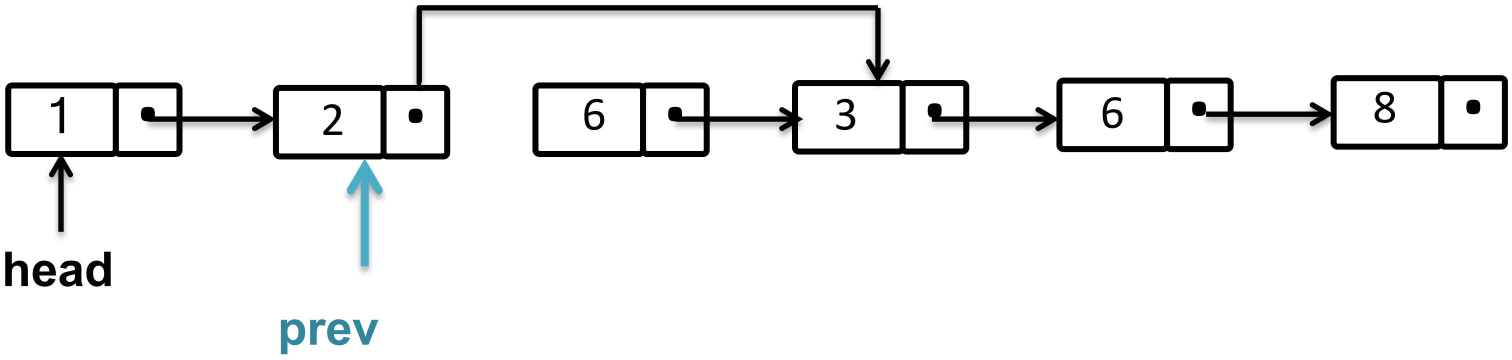
Example
Given: 1 → 2→ 6 → 3 →6 →8, val =
6
Return: 1 → 2 →3 →8

Remove linked list element



check prev->next->val

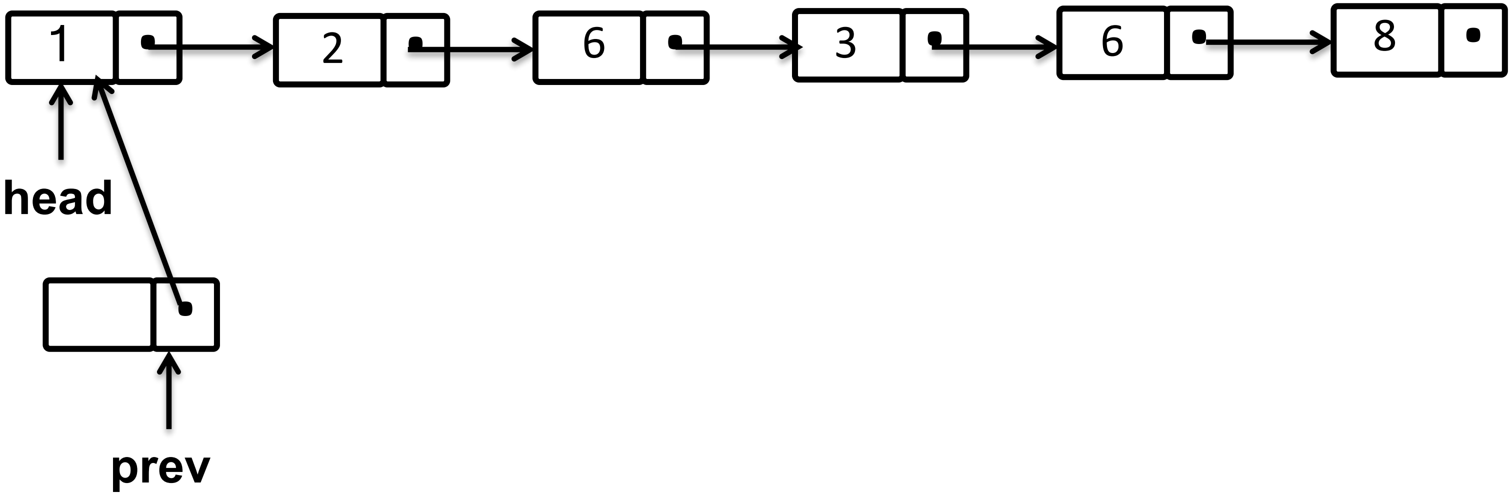
Remove linked list element



```
check prev->next->val
if prev->next->val == val {
    prev->next->next = prev->next
}
```

But how to remove the first element?

Remove linked list element



Basic idea: add a fake node at beginning

```
struct node {
     int val;
     struct node *next;
 };

struct node*
removeElements(struct node* head, int val) {
    struct node *n = (struct node *)malloc(sizeof(struct node));
    struct node *r = n;

    n->next = head;
    while(n->next != NULL) {
        if (n->next->val == val) {
            n->next = n->next->next;
        } else {
            n = n->next;
        }
    }

    return r->next;
}
```