

Strings, Structs, malloc, 2D arrays

Jinyang Li

What we have learnt

- Bitwise operations
- Pointers and arrays
- ASCII Characters

Today

- strings
- structs, malloc, 2D array

C Strings

Strings

- String is represented as an array of chars.
 - Array has no space to encode its length.
- How to determine string length?
 - explicitly pass around an integer representing length

```
// tolower_string turns every character in character array s
// into lower case
void tolower_string(char *s, int len) {
    for (int i = 0; i < len; i++) {
        s[i] = tolower(s[i]);
    }
}
```

Strings

- String is represented as an array of chars.
 - Array has no space to encode its length.
- How to determine string length?
 - explicitly pass around an integer representing length
 - C string stores a NULL character to mark the end (by convention)

```
void tolower_string(char *s) {  
  
  
  
  
  
  
}
```

Strings

- String is represented as an array of chars.
 - Array has no space to encode its length.
- How to determine string length?
 - explicitly pass around an integer representing length
 - C string stores a NULL character to mark the end (by convention)

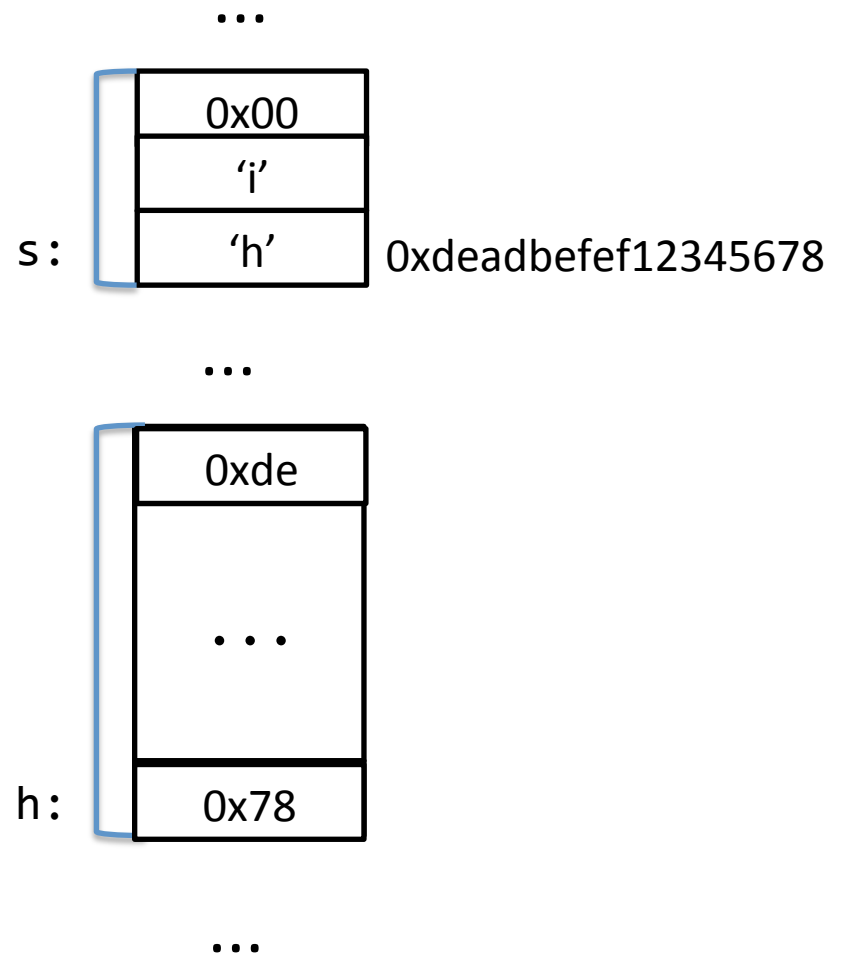
```
void tolower_string(char *s) {  
    int i = 0;  
    while (s[i] != '\0') {  
        s[i] = tolower(s[i]);  
        i++;  
    }  
}
```

Copying string?

does this make a copy of "hi"?

```
char s[3] = {'h', 'i', '\0'};  
char *h;  
h = s;  
h[0] = 'H';
```

```
printf("s=%s h=%s\n", s, h);
```



Copying string?

does this make a copy of "hi"?

```
char s[3] = {'h', 'i', '\0'};
```

```
char h[3];
```

```
h = s;
```

```
h[0] = 'H';
```

```
printf("s=%s h=%s\n", s, h);
```


Copying string

```
void strcpy(char *dst, char *src)
{

}
```

```
int main()
{
    char s[3] = {'h', 'i', '\0'};
    char h[3];
    strcpy(h, s);
    h[0] = 'H';

    printf("s=%s h=%s\n", s, h);
}
```

Copying string

```
void strcpy(char *dst, char *src) {  
    int i = 0;  
    while (src[i] != '\0') {  
        dst[i] = src[i];  
        i++;  
    }  
}
```

strcpy is included in C std library.

```
int main() {  
    char s[3] = {'h', 'i', '\0'};  
    char h[3];  
    strcpy(h, s);  
    h[0] = 'H';  
  
    printf("s=%s h=%s\n", s, h);  
}
```

Copying string

```
void strcpy(char *dst, char *src) {  
    int i = 0;  
    while (src[i] != '\0') {  
        dst[i] = src[i];  
        i++;  
    }  
}
```

```
int main() {  
    char s[3] = {'h', 'i', '\0'};  
    char h[2];  
    strcpy(h, s);  
    h[0] = 'H';
```

Results in out-of-bound write!
Buffer overflow!

```
    printf("s=%s h=%s\n", s, h);  
}
```

Copying string

```
void strncpy(char *dst, char *src, int n) {  
    int i = 0;  
    while (src[i] != '\0' && i < n) {  
        dst[i] = src[i];  
        i++;  
    }  
}
```

strncpy is included in C std library.

```
int main() {  
    char s[3] = {'h', 'i', '\0'};  
    char h[2];  
    strncpy(h, s, 2);  
    h[0] = 'H';  
  
    printf("s=%s h=%s\n", s, h);  
}
```

A different way of initializing string

...

```
char s1[3] = {'h', 'i', '\0'};
```

```
//equivalent to
```

```
//char s1[3] = "hi";
```

```
char *s2 = "bye";
```

```
s1[0] = 'H';
```

← OK

```
s2[0] = 'B';
```

← Segmentation fault (bus error)

```
printf("s1=%s s2=%s\n", s1, s2);
```

A different way of initializing string

```
char s1[3] = {'h', 'i', '\0'};
```

```
//equivalent to
```

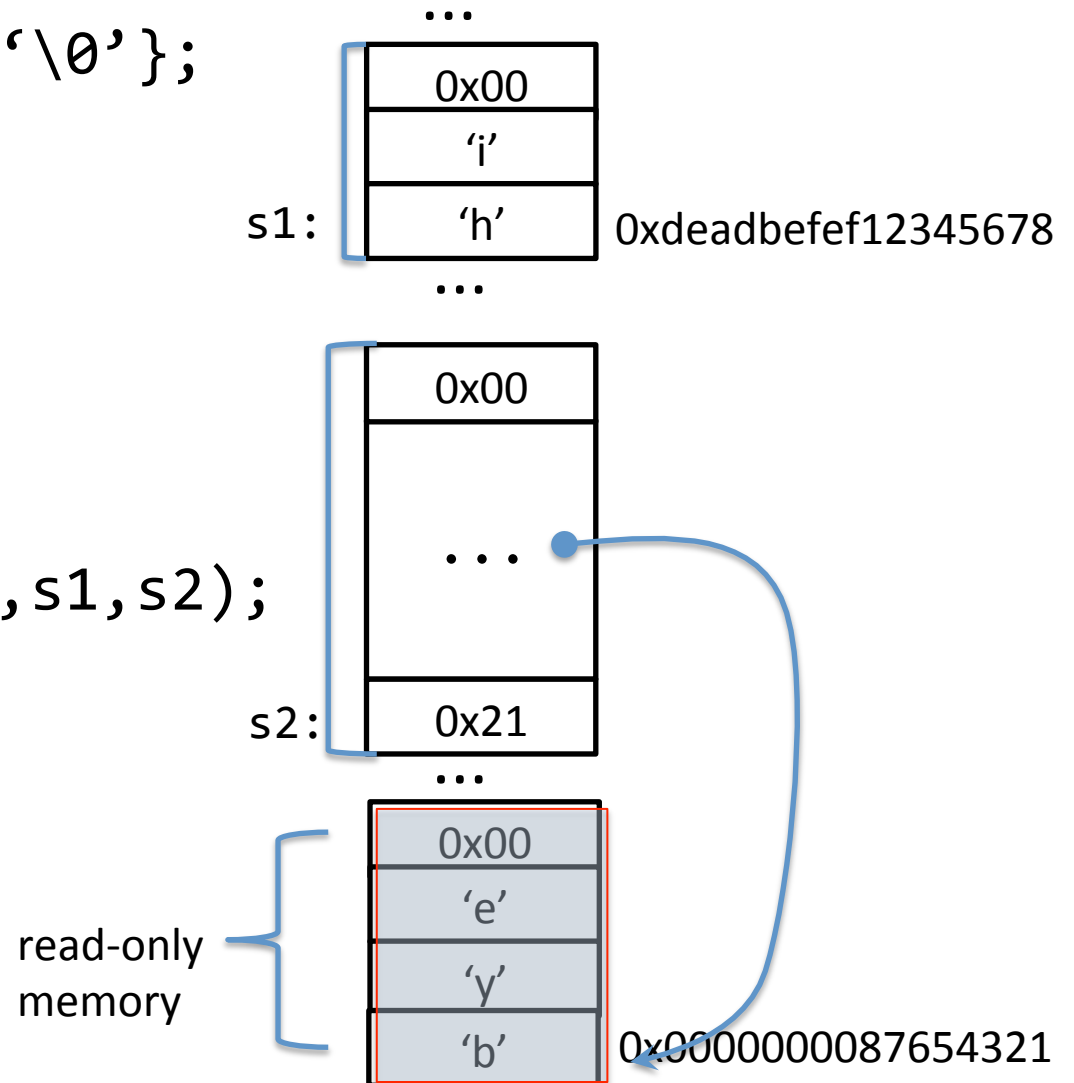
```
//char s1[3] = "hi";
```

```
char *s2 = "bye";
```

```
s1[0] = 'H';
```

```
s2[0] = 'B';
```

```
printf("s1=%s s2=%s\n", s1, s2);
```



The Atoi function

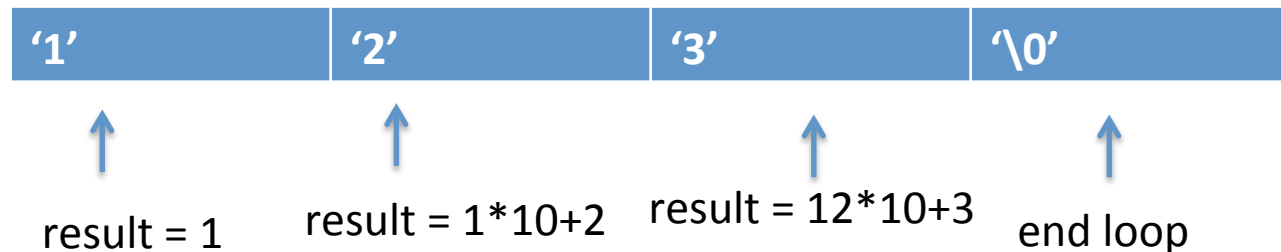
```
// atoi returns the integer  
// corresponding to the string of digits
```

```
int atoi(char *s)  
{  
  
}
```

```
int main()  
{  
    char *s= "123";  
    printf("integer is %d\n", atoi(s));  
}
```

The Atoi function

```
// atoi returns the integer
// corresponding to the string of digits
int atoi(char *s) {
    int result = 0;
    int i = 0;
    while (s[i] >= '0' && s[i] <= '9') {
        result = result * 10 + (s[i] - '0');
        i++;
    }
    return result;
}
```

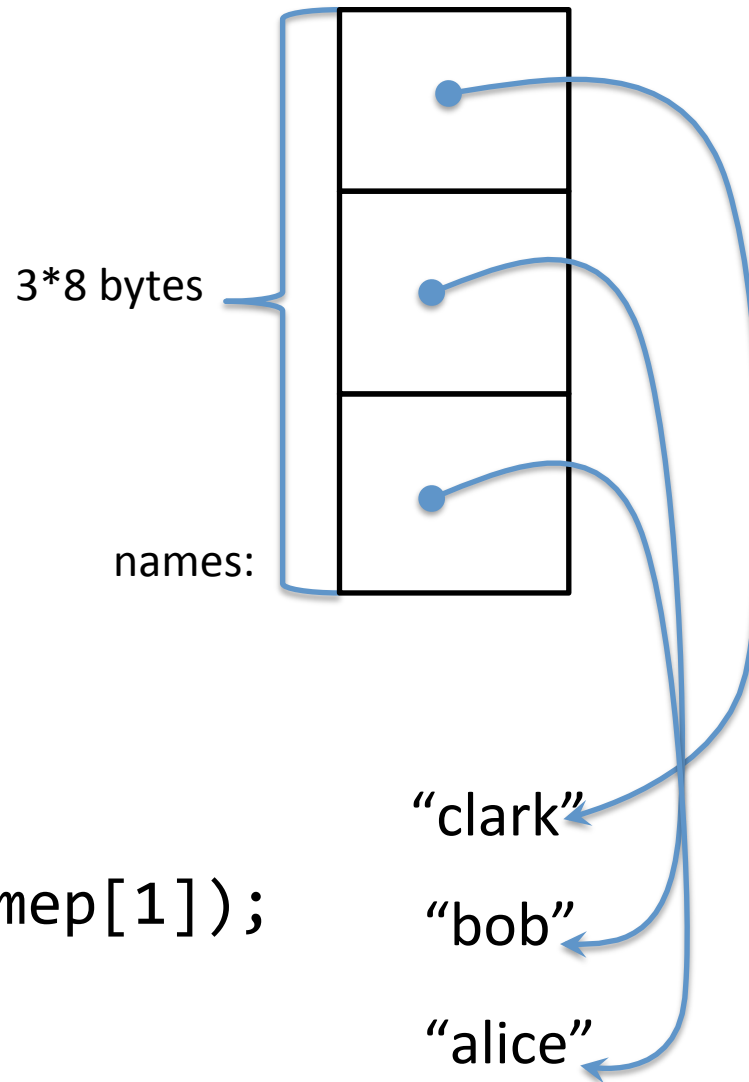


Array of pointers

```
char* names[3] = {  
    "alice",  
    "bob",  
    "clark"  
};
```

```
char **namep;  
namep = names;
```

```
printf("name is %s", namep[1]);
```



The most commonly used array of pointers: argv

```
int main(int argc, char **argv)
{
    for (int i = 0; i < argc; i++) {
        printf("%s\n", argv[i]);
    }
}
```

```
$ ./a.out 1 2 3
./a.out 1 2 3
```



argv[0] is the name of the executable

Structs

Struct stores fields of different types
contiguously in memory

C has no class/object.

Struct is like a class without associated methods

Struct

- Array: a block of n consecutive elements of the same type.
- Struct: a collection of elements of different types.

Structure

```
struct student {  
    int id;  
    char *name;  
};
```

Fields of a struct are allocated next to each other, but there may be gaps (padding) between them.

Structure

```
struct student {  
    int id;  
    char *name;  
};
```

```
struct student t; ← define variable t with  
                    type "struct student"
```

Structure

```
struct student {  
    int id;  
    char *name;  
};
```

```
struct student t;
```

```
t.id = 1024;  ← Access the fields of this struct  
t.name = "alice";
```

Typedef

```
typedef struct {  
    int id;  
    char *name;  
} student;
```

```
struct student t;
```


Pointer to struct

```
typedef struct {  
    int id;  
    char *name;  
} student;  
  
student t = {1024, "alice"};  
student *p = &t;  
  
p->id = 1023;  
p->name = "bob";  
printf("%d %s\n", t.id, t.name);
```

Mallocs

Allocates a chunk of memory dynamically

Recall memory allocation for global and local variables


- **Global** variables are allocated space before program execution.
- **Local** variables are allocated when entering a function and de-allocated upon its exit.

Malloc

Allocate space dynamically and flexibly:

- 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

Malloc is implemented as a C library

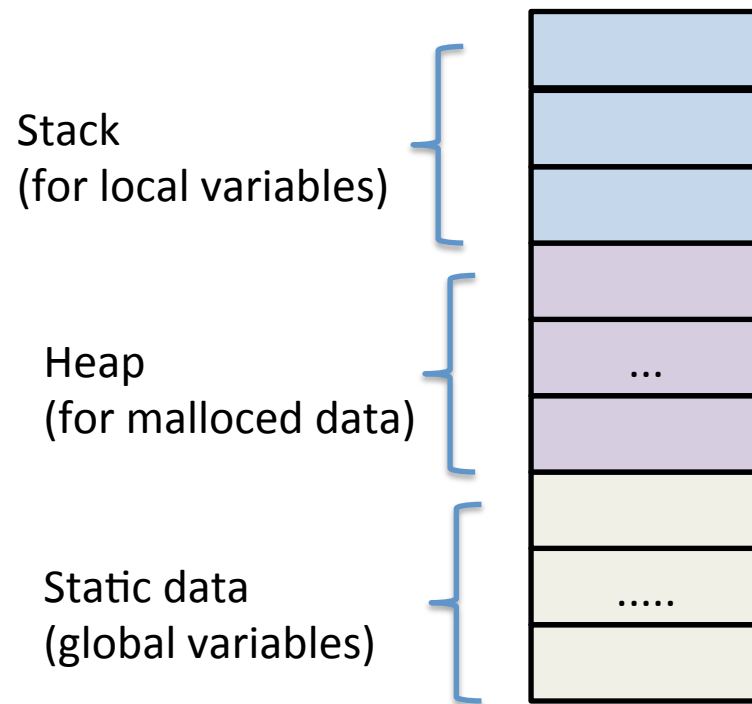


```
#include <stdlib.h>
```

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

Conceptual view of a C program's memory at runtime

- Separate memory regions for global, local, and malloc-ed.



We will refine this simple view in later lectures

Linked list in C: insertion

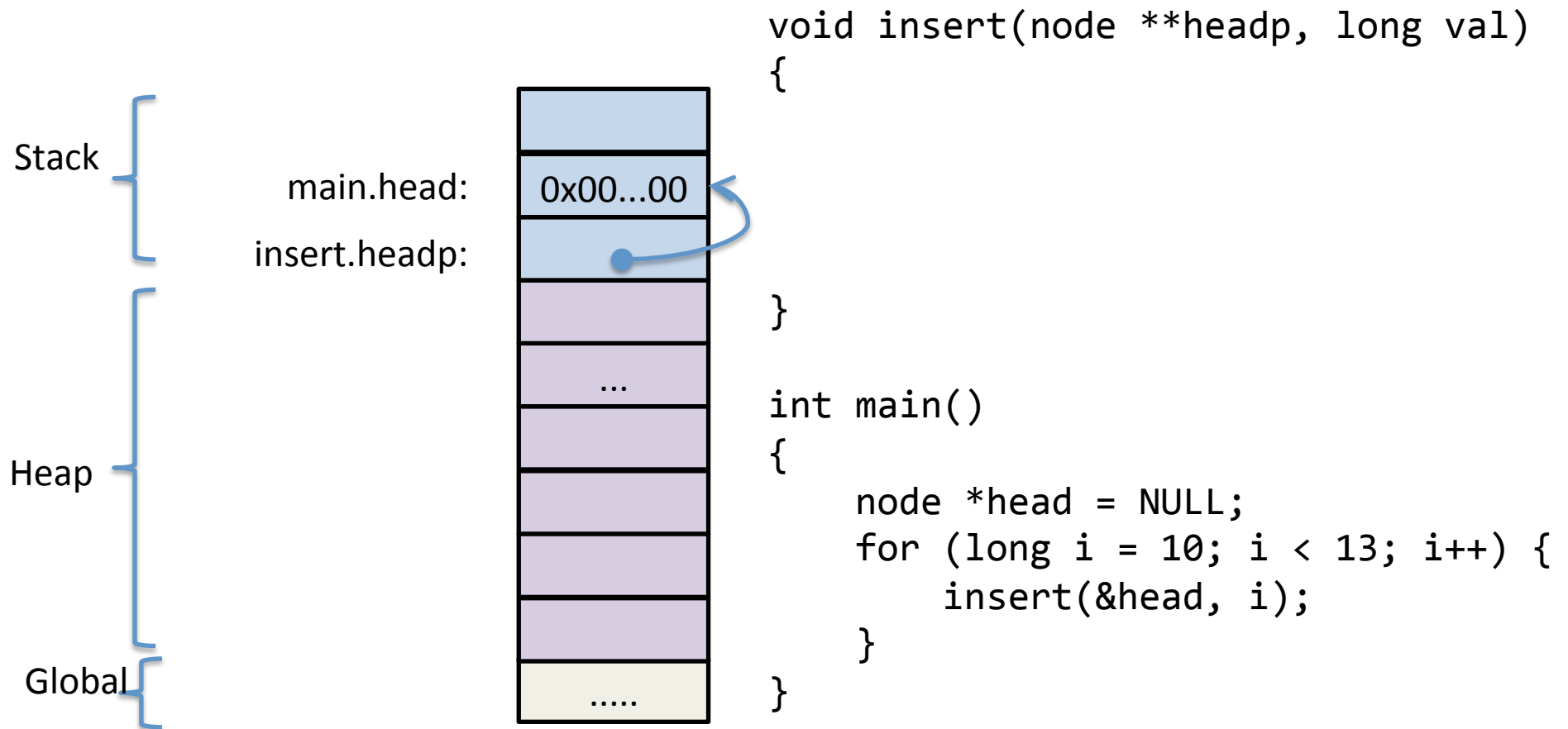
```
typedef struct {
    long val;
    struct node *next;
}node;

// insert val into linked list to the head of the linked
// list and return the new head of the list in *head
void
insert(node **head, long val) {

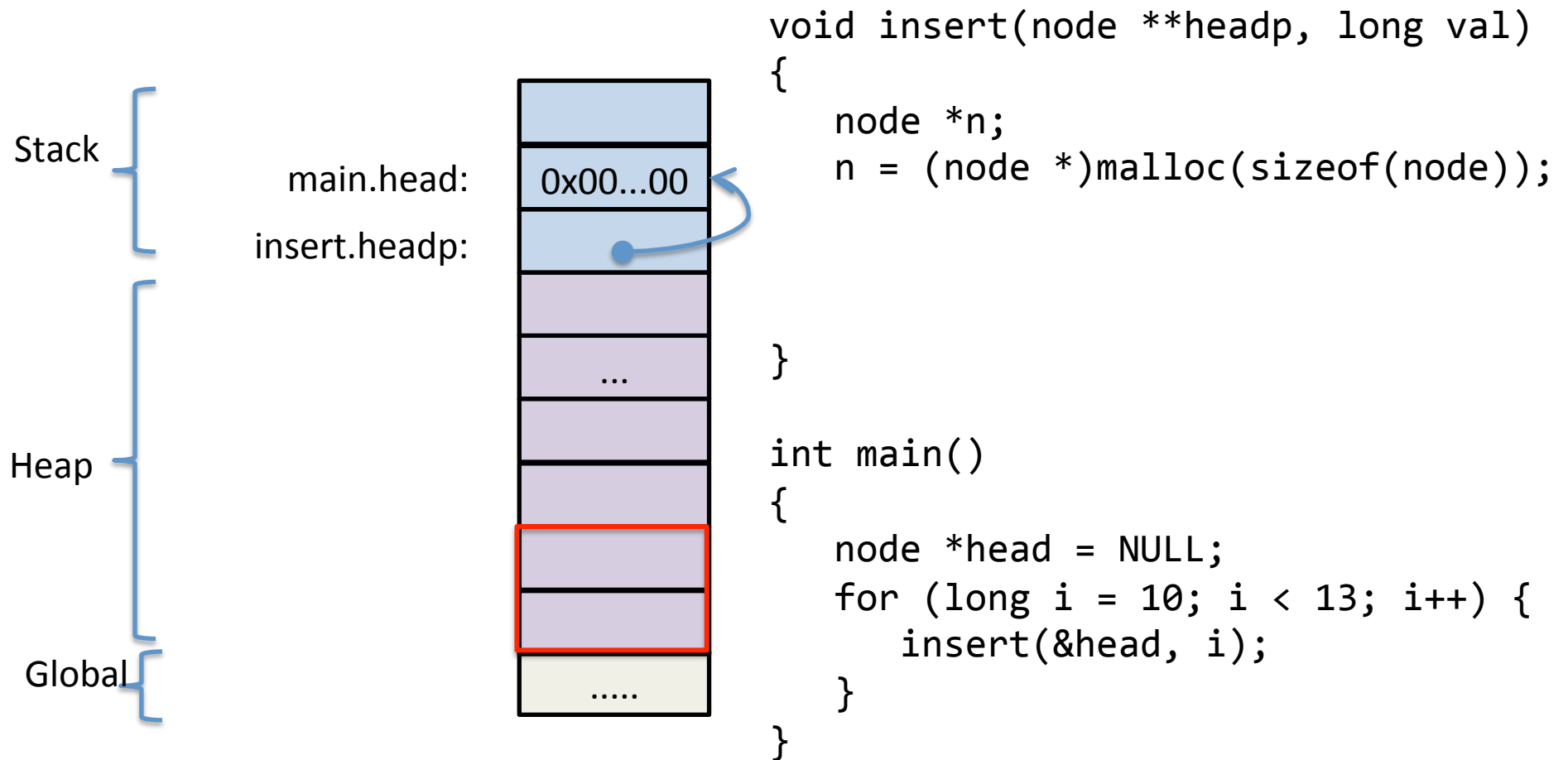
}

int main() {
    node *head = NULL;
    for (long i = 10; i < 13; i++)
        insert(&head, i);
}
```

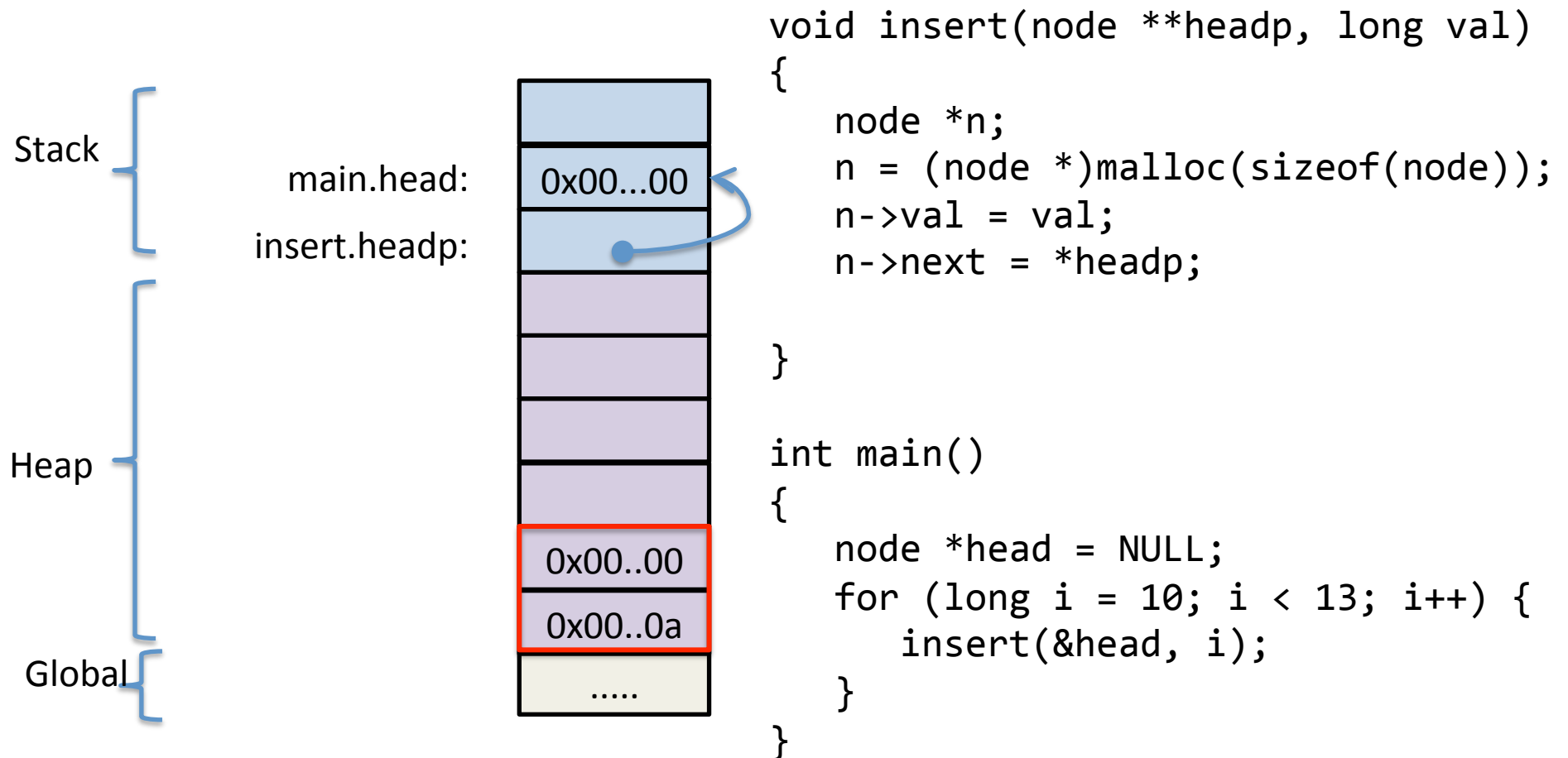
Inserting into a linked list



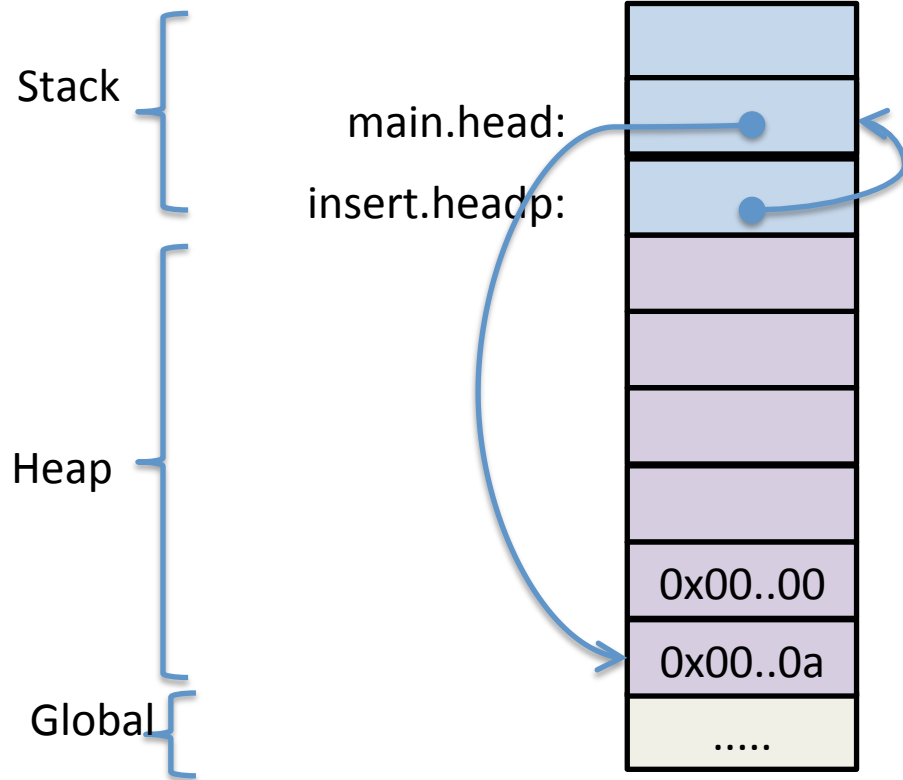
1st insert call



1st insert call



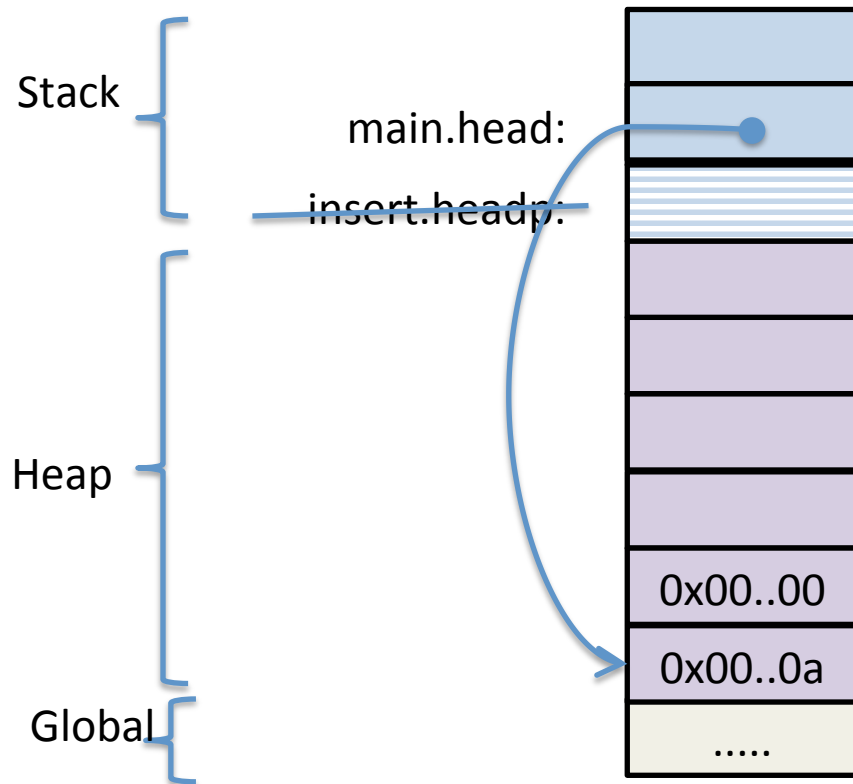
1st insert call



```
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```

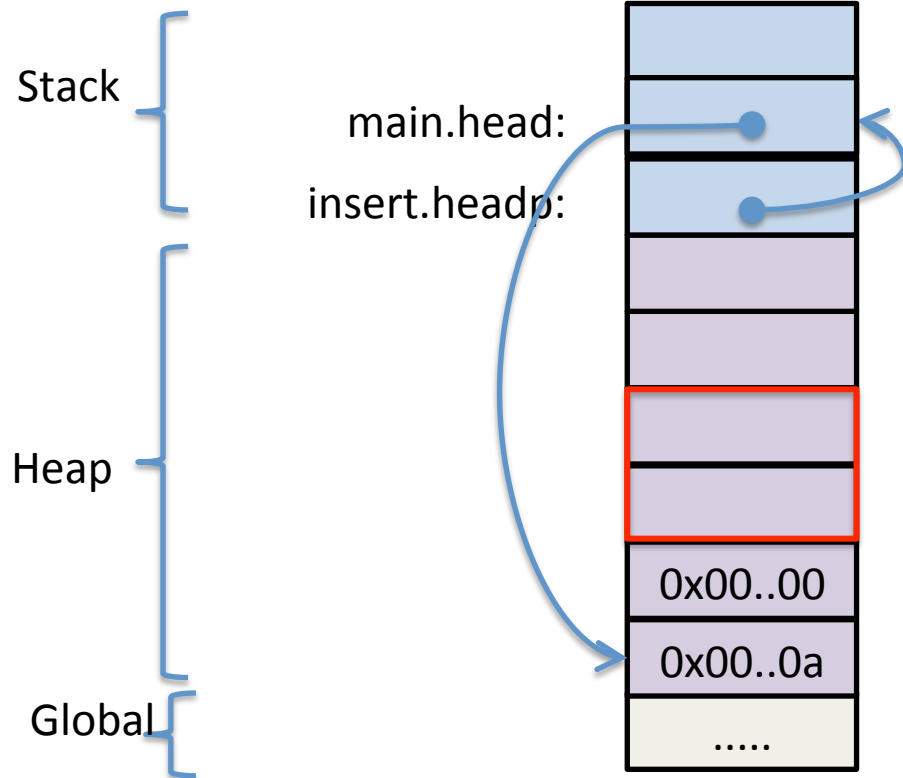
after 1st insert call



```
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```

2nd insert call



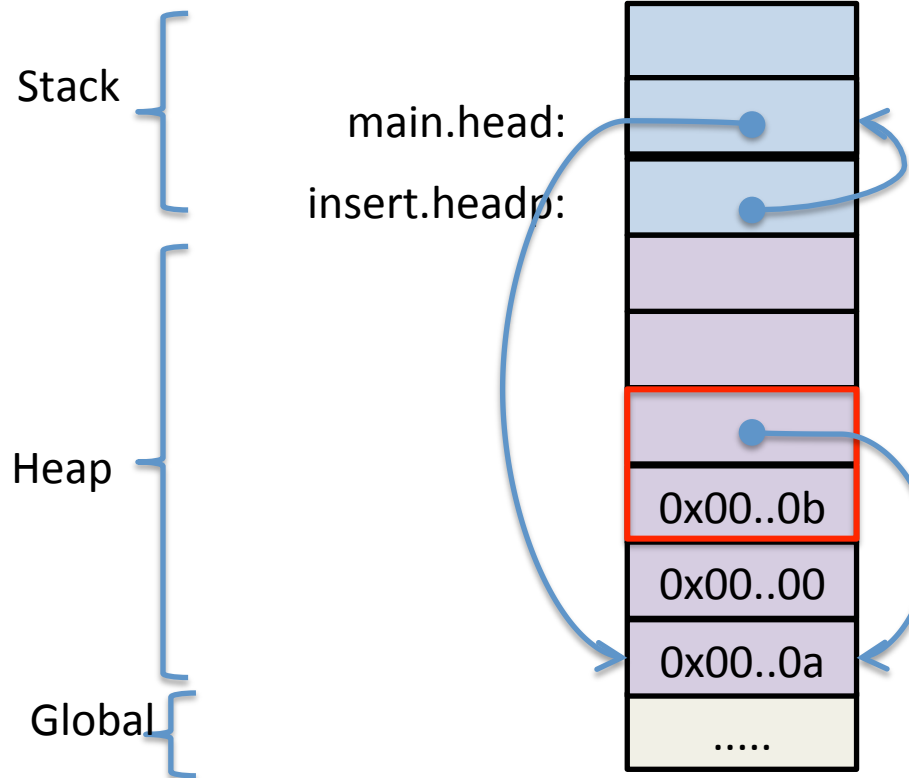
```
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}

int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```

2nd insert call

```
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}
```

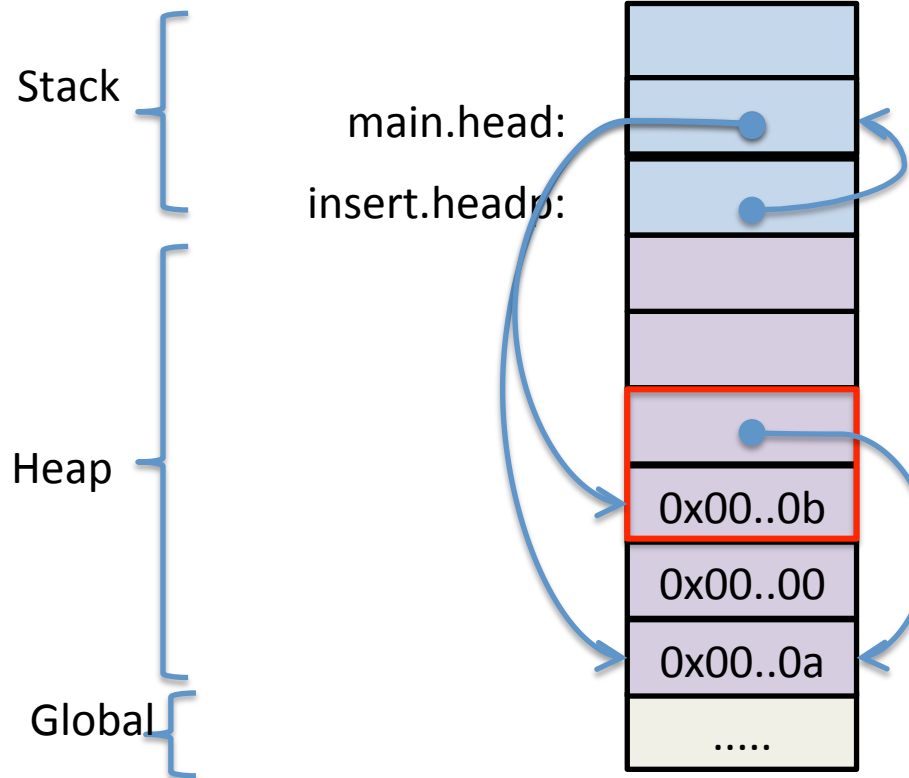
```
int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```



2nd insert call

```
void insert(node **headp, long val)
{
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}
```

```
int main()
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```



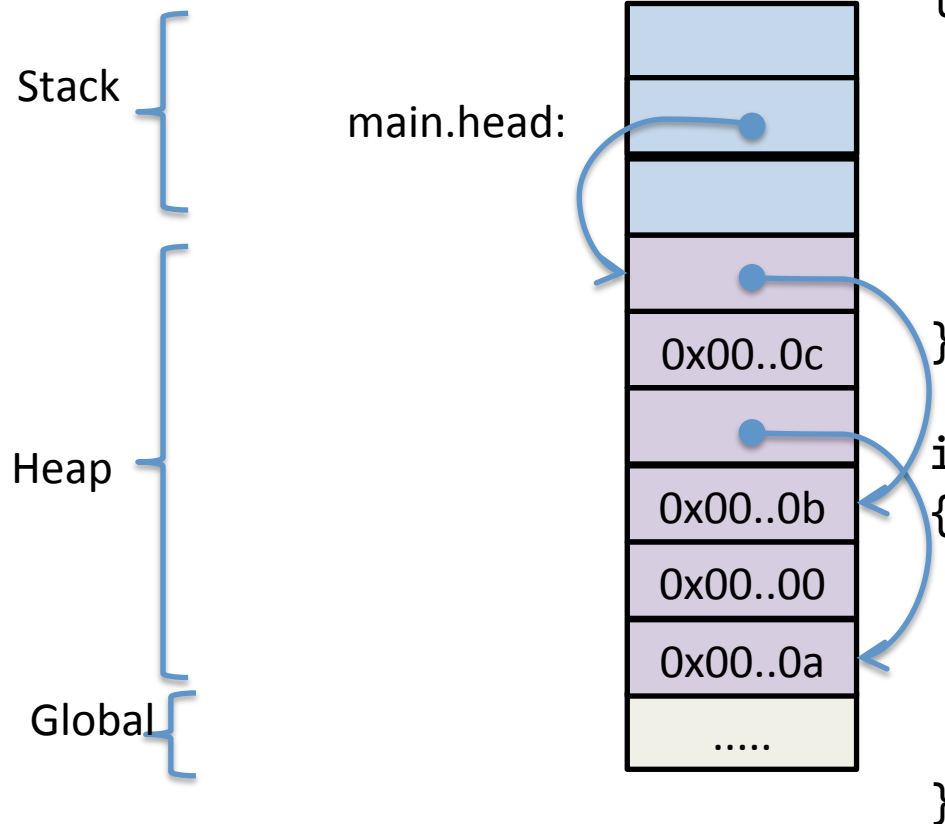
after 3rd call

```
void insert(node **headp, long val)
{
```

```
    node *n;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = *headp;
    *headp = n;
}
```

```
int main()
```

```
{
    node *head = NULL;
    for (long i = 10; i < 13; i++) {
        insert(&head, i);
    }
}
```



2D Array

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

Multi-dimensional arrays

Declare a k dimensional array

```
int arr[n1][n2][n3]...[nk-1][nk]
```

n_i is the length of the i th dimension

Multi-dimensional arrays

Declare a k dimensional array

```
int arr[n1][n2][n3]...[nk-1][nk]
```

n_i is the length of the i th dimension

Example: 2D array

```
int matrix[2][3]
```

Multi-dimensional arrays

Declare a k dimensional array

```
int arr[n1][n2][n3]...[nk-1][nk]
```

n_i is the length of the i th 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[n1][n2][n3]...[nk-1][nk]
```

n_i is the length of the i th 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[n1][n2][n3]...[nk-1][nk]
```

n_i is the length of the i th 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

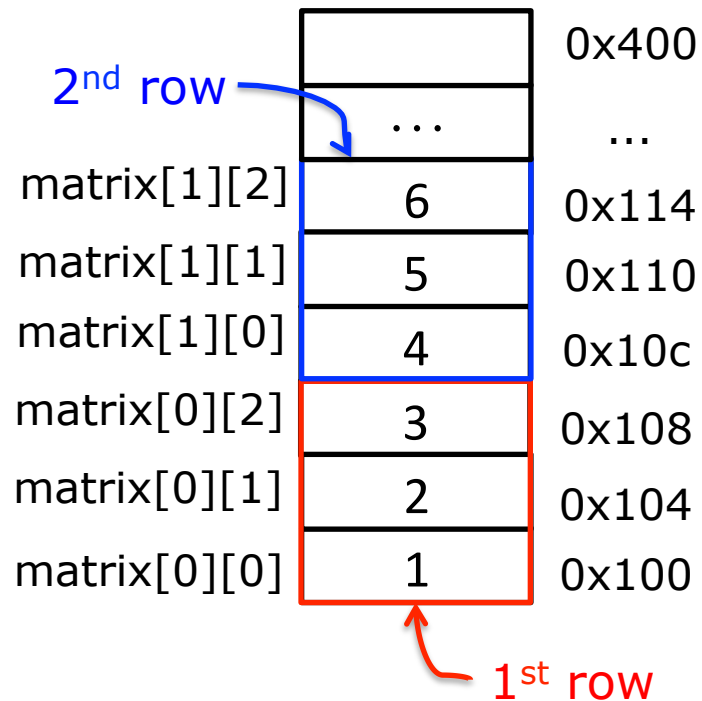
Memory layout

		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

		0x400

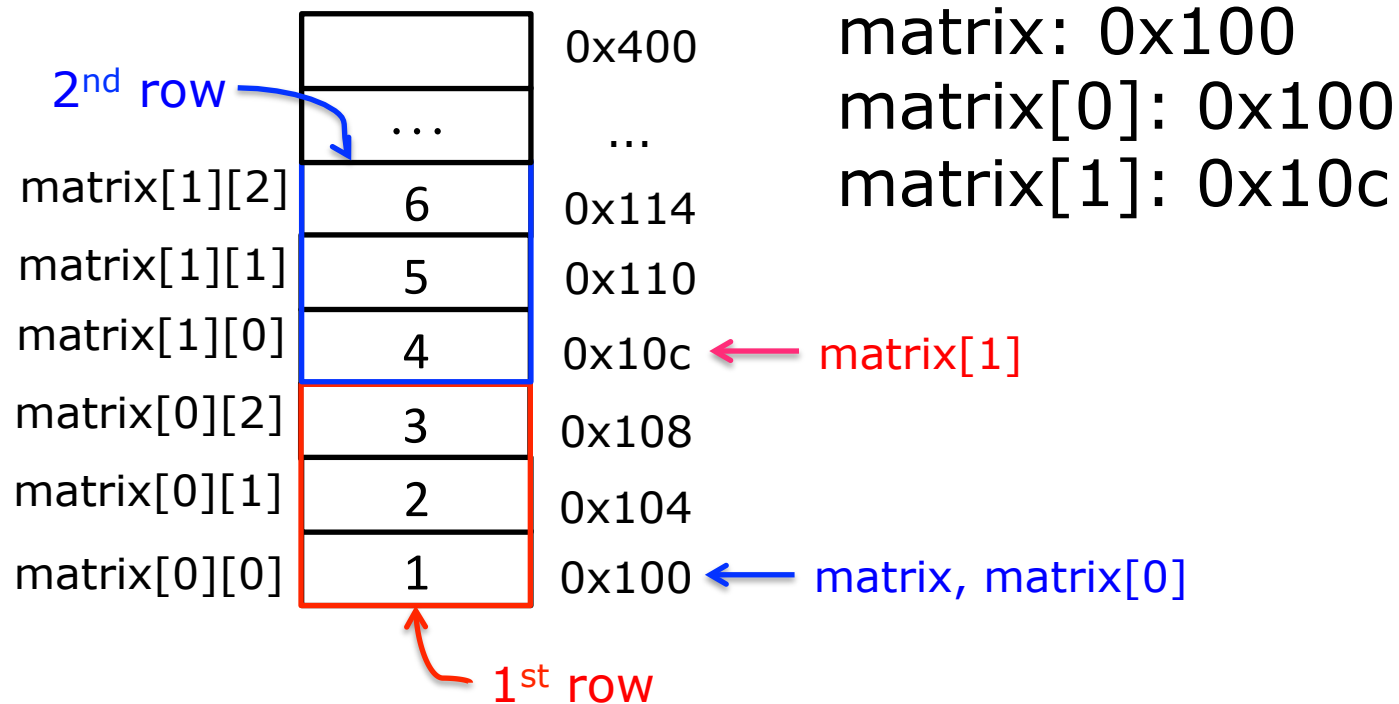
matrix[1][2]	6	0x114
matrix[1][1]	5	0x110
matrix[1][0]	4	0x10c
matrix[0][2]	3	0x108
matrix[0][1]	2	0x104
matrix[0][0]	1	0x100

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

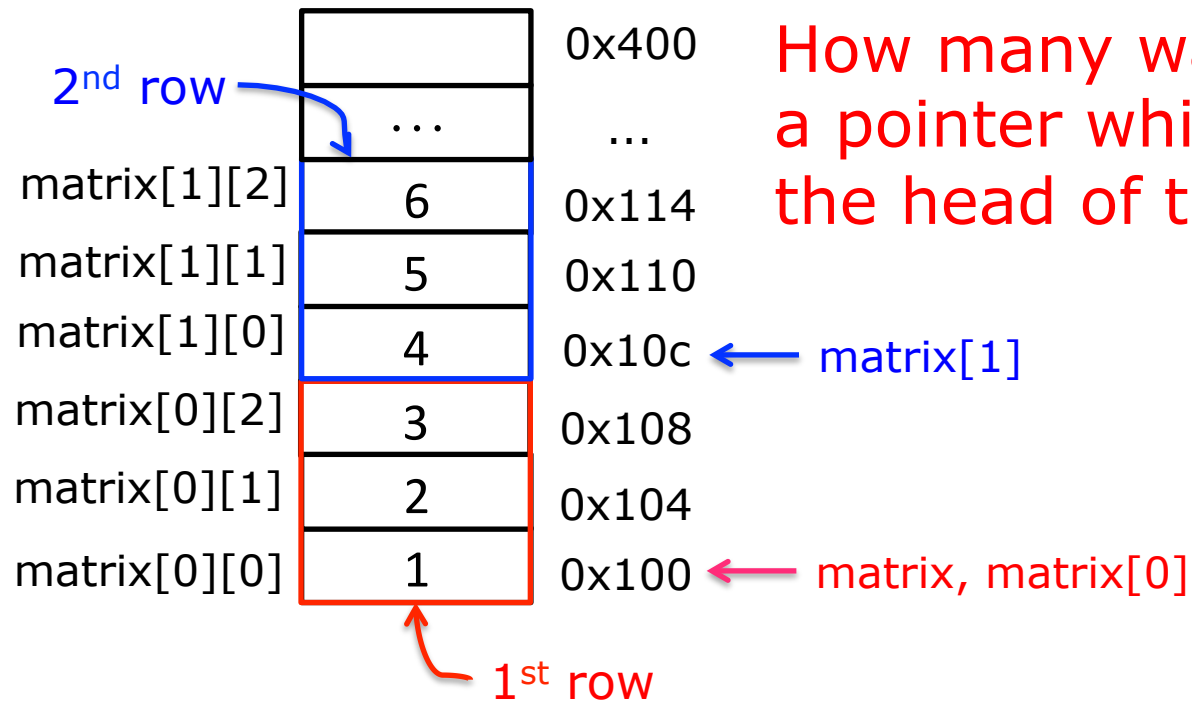
```
int *p1, *p2, *p3;  
p1 = (int *)matrix;  
p2 = matrix[0];  
p3 = matrix[1];
```

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

Pointers

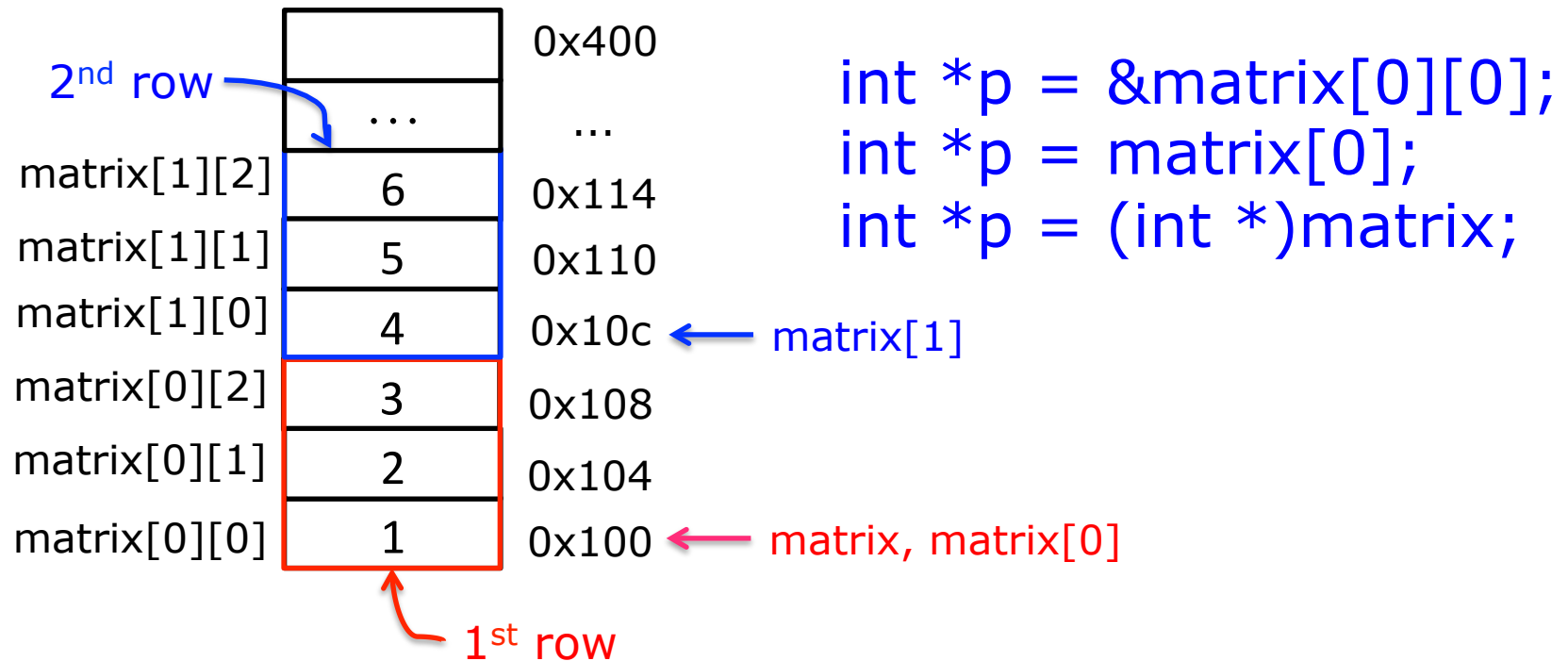


Pointers

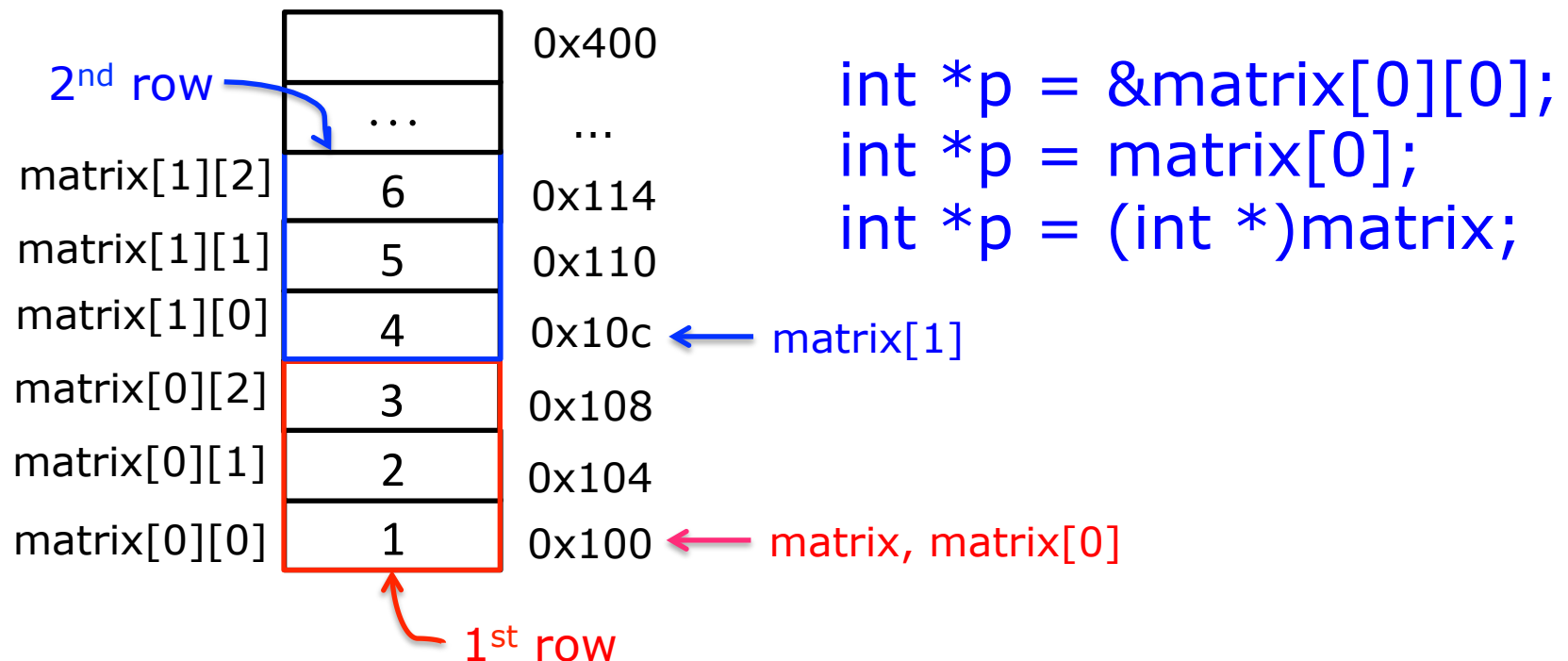


How many ways to define a pointer which points to the head of the array?

Pointers

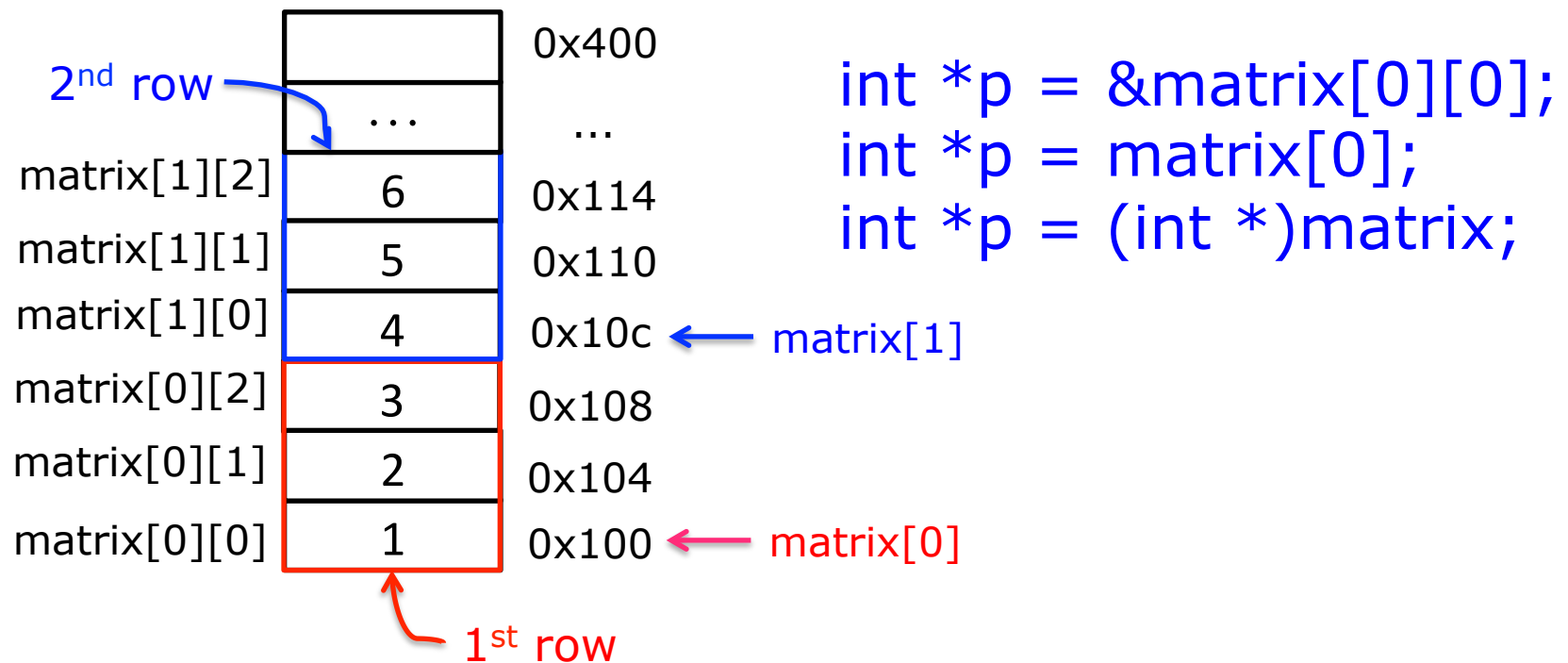


Pointers



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

Pointers



```
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[i][j]);  
    }  
}
```

OR

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