Structs, malloc, 2D arrays

Jinyang Li
What we have learnt

- Pointers
- Pointers and arrays
- Characters and strings

Today

- Finish string leftovers
- structs, malloc, 2D array
A different way of initializing string

... 

cchar 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);

OK
Segmentation fault (bus error)
A different way of initializing string

```c
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);
```

```
read-only memory
```
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;
}

<table>
<thead>
<tr>
<th>'1'</th>
<th>'2'</th>
<th>'3'</th>
<th>\0</th>
</tr>
</thead>
<tbody>
<tr>
<td>result = 1</td>
<td>result = 1*10+2</td>
<td>result = 12*10+3</td>
<td>end loop</td>
</tr>
</tbody>
</table>
char* names[3] = {
  "alice",
  "bob",
  "clark"
};

char **namep;
namep = names;

printf("name is %s", namep[1]);
The most commonly used array of pointers: argv

```c
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.
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”
struct student {
    int id;
    char *name;
};

struct student t;

int id = 1024;  // Access the fields of this struct
char *name = "alice";
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

```c
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.

```c
void free(void *ptr);
```
#include <stdlib.h>

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

Malloc

Malloc is implemented as a C library
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 {
    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

![Diagram of linked list with an arrow indicating the head]
Inserting into a linked list

```c
node *insert(node *head, int val) {
    node nn;
    node *n = &nn;
    n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = head;
}
```
Inserting into a linked list

```
node *insert(node *head, int val) {
    node *n = (node *)malloc(sizeof(node));
    n->val = val;
    n->next = head;
    return n;
}
```
2D Array

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

\[ \text{int arr}[n_1][n_2][n_3]...[n_{k-1}][n_k] \]

\( n_i \) is the length of the \( i \)th dimension

Example: 2D array

\[ \text{int matrix}[2][3] \]
Multi-dimensional arrays

Declare a $k$ dimensional array

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

$n_i$ is the length of the $i$th dimension

Example: 2D array

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

<table>
<thead>
<tr>
<th></th>
<th>Col 0</th>
<th>Col 1</th>
<th>Col 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Row 1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Multi-dimensional arrays

Declare a k dimensional array

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

\( n_i \) is the length of the i\(^{th} \) dimension

Example: 2D array

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

<table>
<thead>
<tr>
<th></th>
<th>Col 0</th>
<th>Col 1</th>
<th>Col 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Row 1</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
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
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

<table>
<thead>
<tr>
<th></th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix[1][2]</td>
<td>0x100</td>
</tr>
<tr>
<td>matrix[1][1]</td>
<td>0x104</td>
</tr>
<tr>
<td>matrix[1][0]</td>
<td>0x108</td>
</tr>
<tr>
<td>matrix[0][2]</td>
<td>0x110</td>
</tr>
<tr>
<td>matrix[0][1]</td>
<td>0x114</td>
</tr>
<tr>
<td>matrix[0][0]</td>
<td>0x118</td>
</tr>
<tr>
<td>...</td>
<td>0x144</td>
</tr>
</tbody>
</table>

...
Memory layout

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

1st row
Memory layout

2nd row

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

<table>
<thead>
<tr>
<th>1st row</th>
<th>2nd row</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix[0][0]</td>
<td>1 0x100</td>
</tr>
<tr>
<td>matrix[0][1]</td>
<td>2 0x104</td>
</tr>
<tr>
<td>matrix[0][2]</td>
<td>3 0x108</td>
</tr>
<tr>
<td>matrix[1][0]</td>
<td>4 0x10c</td>
</tr>
<tr>
<td>matrix[1][1]</td>
<td>5 0x110</td>
</tr>
<tr>
<td>matrix[1][2]</td>
<td>6 0x114</td>
</tr>
</tbody>
</table>

```c
int *p1, *p2, *p3;
p1 = (int *)matrix;
p2 = matrix[0];
p3 = matrix[1];
printf("matrix:%p matrix[0]:%p\n matrix[1]:%p\n", p1, p2, p3);
```
## Pointers

<table>
<thead>
<tr>
<th>1st row</th>
<th>2nd row</th>
<th>matrix: 0x100</th>
<th>matrix[0]: 0x100</th>
<th>matrix[1]: 0x10c</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix[0][0]</td>
<td>matrix[1][2]</td>
<td>1</td>
<td>0x100</td>
<td></td>
</tr>
<tr>
<td>matrix[0][1]</td>
<td>matrix[1][1]</td>
<td>2</td>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>matrix[0][2]</td>
<td>matrix[1][0]</td>
<td>3</td>
<td>0x108</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>2nd row</th>
<th>1st row</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>matrix[1][2]</code></td>
<td>6</td>
</tr>
<tr>
<td><code>matrix[1][1]</code></td>
<td>5</td>
</tr>
<tr>
<td><code>matrix[1][0]</code></td>
<td>4</td>
</tr>
<tr>
<td><code>matrix[0][2]</code></td>
<td>3</td>
</tr>
<tr>
<td><code>matrix[0][1]</code></td>
<td>2</td>
</tr>
<tr>
<td><code>matrix[0][0]</code></td>
<td>1</td>
</tr>
</tbody>
</table>

- `matrix[1]` = 0x10c
- `matrix[0]` = 0x100
- `matrix[0][0]` = 0x100
- `matrix[1][2]` = 0x400
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**

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix[1][2]</td>
<td>matrix[1][1]</td>
<td>matrix[1][0]</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>matrix[0][2]</td>
<td>matrix[0][1]</td>
<td>matrix[0][0]</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

```
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

```c
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

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