

Lecture 2

# Pointers, Arrays and Memory Management

Dr. Yusuf H. Sahin  
Istanbul Technical University

[sahinyu@itu.edu.tr](mailto:sahinyu@itu.edu.tr)

# Structs

- An array is a collection of items, all of which are of the same type.

```
char* weekdays[] = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};
```

- A «struct» is a composite data type that groups together elements of various types, almost without restriction.

```
typedef struct address{  
    char* name;  
    long int number;  
    char* city;  
    char* bloodtype[2];  
} Address;
```

```
myaddress.name = (char*)malloc(6 * sizeof(char));  
strcpy(myaddress.name, "Yusuf");  
  
myaddress.number = 1033;  
  
myaddress.city = (char*)malloc(9 * sizeof(char));  
strcpy(myaddress.city, "Istanbul");  
  
myaddress.bloodtype[0] = (char*)malloc(2 * sizeof(char));  
strcpy(myaddress.bloodtype[0], "0");  
  
myaddress.bloodtype[1] = (char*)malloc(3 * sizeof(char));  
strcpy(myaddress.bloodtype[1], "-");  
  
printf("Size of Address struct: %lu bytes\n", sizeof(myaddress));
```

```
free(myaddress.name);  
free(myaddress.city);  
free(myaddress.bloodtype[0]);  
free(myaddress.bloodtype[1]);
```

Size of Address struct: 40 bytes

long int	8 bytes
char*	8 bytes

# Structs

- We can also dynamically create a struct. The «->» operator is used to access members (either variables or functions) of an object through a pointer.

```
typedef struct address{  
    char* name;  
    long int number;  
    char* city;  
    char* bloodtype[2];  
} Address;
```

```
Address* myaddress = (Address*)malloc(sizeof(Address));  
  
myaddress->name = (char*)malloc(6 * sizeof(char));  
strcpy(myaddress->name, "Yusuf");  
  
myaddress->number = 1033;  
  
myaddress->city = (char*)malloc(8 * sizeof(char));  
strcpy(myaddress->city, "Istanbul");  
  
myaddress->bloodtype[0] = (char*)malloc(2 * sizeof(char));  
strcpy(myaddress->bloodtype[0], "0");  
  
myaddress->bloodtype[1] = (char*)malloc(3 * sizeof(char));  
strcpy(myaddress->bloodtype[1], "-");  
  
printf("Size of Address pointer: %lu bytes\n", sizeof(Address*));
```

Size of Address pointer: 8 bytes

- A structure could be used as a building block for other data structures.

```
Address* all_addresses = (Address*)malloc(20 * sizeof(Address));  
  
all_addresses[0].name = (char*)malloc(6 * sizeof(char));  
strcpy(all_addresses[0].name, "Yusuf");  
  
all_addresses[0].number = 1033;
```

# Functions

- To deal with data structures easily, helper functions could be defined.
  - Especially getters & setters!

```
void set_name(Address* addr, char* new_name) {  
    addr->name = (char*)malloc((strlen(new_name) + 1) * sizeof(char));  
    strcpy(addr->name, new_name);  
}
```

```
char* get_name(Address* addr) {  
    return addr->name;  
}
```

```
void delete_name(Address* addr) {  
    if (addr->name != NULL) {  
        free(addr->name);  
        addr->name = NULL;  
    }  
}
```

- **Function Declaration:** A function declaration, also known as a function prototype, tells the compiler about a function's name, return type, and parameters without the body of the function.

```
void set_name(Address* addr, char* new_name);  
char* get_name(Address* addr);  
void delete_name(Address* addr);
```

Declarations allow functions to be defined in a different file or in a different part of the same file. You can organize your code better by grouping all function declarations in a header file and function definitions in separate source files, improving code readability and maintainability.

# An example

- A formal definition for a 3D mesh
  - A collection of triangles.

```
solid Mesh
facet
  outer loop
    vertex 0.0666225 -0.00713973 -0.0520612
    vertex 0.0695272 -0.00912108 -0.0509354
    vertex 0.0659653 -0.00814601 -0.052367
  endloop
endfacet
facet
  outer loop
    vertex 0.0762163 -0.00201969 -0.0587023
    vertex 0.0769302 -0.00441556 -0.0564184
    vertex 0.0760299 -0.00791856 -0.0610091
  endloop
endfacet
```



# Task: 3D Mesh Manipulation

- Create a data structure to keep point positions:  $x, y, z$
- Create another data structure to store the triangles.
- Create a list of triangles.
- Implement three different manipulation functions:
  - $x\_limit, y\_limit, z\_limit$
  - Delete the triangles according to  $x, y$  and  $z$  axis.



# Pointer of the same type

- In C, a struct can also have a pointer to its own type. This technique is frequently used in data structures such as linked lists.

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

typedef struct UselessStruct {
    int value;
    struct UselessStruct* connected_to;
} UselessStruct;

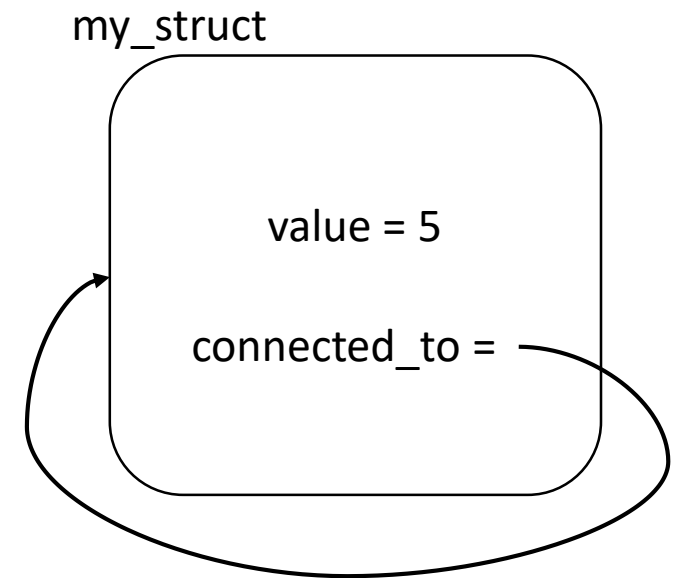
int main() {
    UselessStruct my_struct;
    my_struct.value = 5;
    my_struct.connected_to = &my_struct;

    UselessStruct* ptr = &my_struct;

    for (int i = 1; i < 100; i++) {
        ptr = ptr->connected_to;
    }

    printf("Value: %d\n", ptr->value);

    return 0;
}
```



# Pointer of the same type

- Similarly, we can define a point couple.

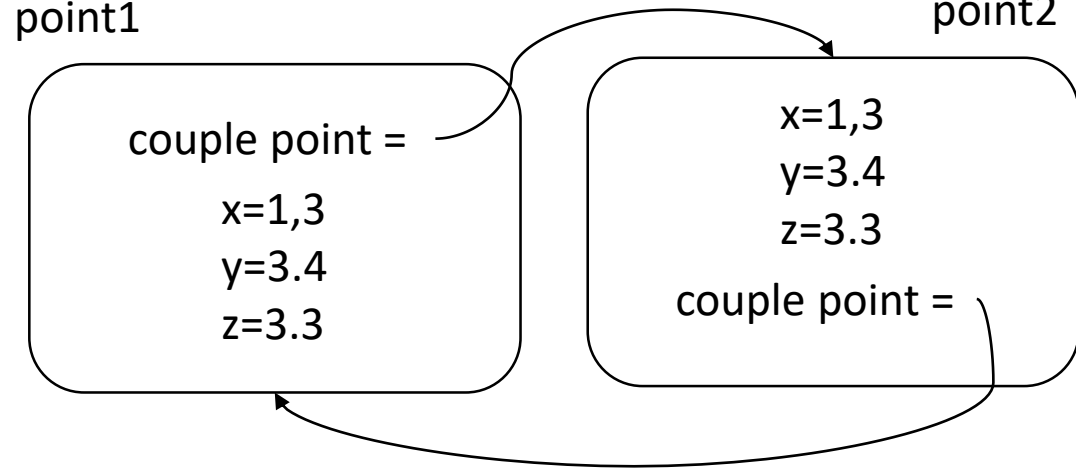
```
typedef struct Point {  
    float x, y, z;  
    struct Point* couple_point;  
} Point;  
  
float calculate_couple_distance(Point p) {  
    return sqrt(pow(p.x - p.couple_point->x, 2) +  
               pow(p.y - p.couple_point->y, 2) +  
               pow(p.z - p.couple_point->z, 2));  
}
```

```
int main() {  
    Point point1, point2;  
  
    point1.x = 1.3;  
    point1.y = 3.4;  
    point1.z = 3.3;  
    point1.couple_point = &point2;  
  
    point2.x = 5.0;  
    point2.y = 5.0;  
    point2.z = 3.3;  
    point2.couple_point = &point1;  
  
    printf("Distance between point1 and point2: %f\n", calculate_couple_distance(point1));  
    printf("Distance between point2 and point1: %f\n", calculate_couple_distance(point2));  
  
    return 0;  
}
```



point1

point2



```
Distance between point1 and point2: 4.031129  
Distance between point2 and point1: 4.031129
```



# Function Calls


- In the context of function arguments, there are two typical ways:

## Call by value

- A copy of the actual argument (value) is passed to the function.
- Changes made to the parameter inside the function have no effect on the actual argument in the caller.

```
void add20(int param)
{
    param = param + 20;
}
```

```
int x = 0;
add20(x);
printf("%d\n", x);
```




## Call by reference

- Instead of a copy of the value, a reference to the actual argument is passed to the function.
- The original data can be modified by the function.

```
void add20(int* param)
{
    *param = *param + 20;
}
```

```
int x = 0;
add20(&x);
printf("%d\n", x);
```




- Especially if a struct is large, call by reference is often more useful due to efficiency.

# Swap Function

- A function which swaps its two inputs should be written in «call by reference» logic.

```
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

```
int x=5, y =3;
swap(&x, &y);
printf("%d %d\n", x, y);
```



3 5

# Arrays as Arguments

- Arrays are sent to a function as a pointer containing the address of the first element.
- For strings (char\*), end of the array could be found according to the '\0' character. For other types of arrays, it is a rule of thumb to also pass the array size as an argument.

**Example:** Find the length of a string

```
int strlen(char* s)
{
    int len = 0;
    for(char* ptr = s; *ptr != '\0'; ptr++)
    {
        len++;
    }
    return len;
}
```

**Example:** Print the characters at the even indices of a string.

```
void even_chars(char* s) {
    int current_ind = 0;
    while (*s != '\0') {
        if (current_ind % 2 == 1) {
            printf("%c", *s);
        }
        current_ind++;
        s++;
    }
    printf("\n");
}
```

```
char text[] = "characterarray";
even_chars(text);
return 0;
```

→ hrceary

even\_chars(&text[5]); → trra

# Foreshadowing: Singly Linked Lists

- Using pointer mechanism, we can directly chain the nodes to obtain a linked list.

```
typedef struct intNode {  
    int data;  
    struct intNode* next;  
} intNode;  
  
typedef struct intList {  
    intNode* head;  
    int elemcount;  
} intList;
```

```
intNode* Node1 = (intNode*)malloc(sizeof(intNode));  
intNode* Node2 = (intNode*)malloc(sizeof(intNode));  
intNode* Node3 = (intNode*)malloc(sizeof(intNode));  
  
Node1->data = 1;  
Node2->data = 2;  
Node3->data = 3;  
  
Node1->next = Node2;  
Node2->next = Node3;  
Node3->next = NULL;  
  
intList myList;  
myList.head = Node1;  
myList.elemcount = 3;
```

```
intNode* ptr = myList.head;  
while (ptr != NULL) {  
    printf("%d\n", ptr->data);  
    ptr = ptr->next;  
}  
  
free(Node1);  
free(Node2);  
free(Node3);
```

1  
2  
3

