



# **COP 3502 – Computer Science 1**

## **Lecture 02**

**Dr. Sahar Hooshmand**  
sahar.hooshmand@ucf.edu

Department of Computer Science

*Slides modified from Dr. Ahmed, with permission*

# C programming review

- In this lecture, we will review some advance topics in C programming language
- Generally, it is assumed that you already have some idea about these topics and a review will help you to recall them.

# Arrays

# Arrays

- An array is a collection of variables, with:
- 1) *The items are stored in consecutive memory locations*
- 2) *All the variables in an array are of the same type.*

- We define an array as follows: `int data[10];`
- Here the **size of the array is 10.**
- To index a particular variable/element of the array do as follows:

`data[4] = 120;`

- The first index is `data[0]` and last index is `data[9]`
- The items are: `data[0]`, `data[1]`, `data[2]`, ..., `data[9]`
- They contains garbage data if not initialized.
- Another way of declaring and initializing array:


`int data[] = {87, 99, 75, 88, 93, 56, 77, 84, 89, 79};`

# Pointers and Arrays

```
int array[size];
```

```
int* pPtr = array + i; //let's say i is an index of the array
```

```
int* qPtr = array + j; //let's say j is an index of the array
```

- The name `array` is equivalent to `&array[0]`
- It means, name of an array is a pointer
- `pPtr++` increments `pPtr` to point to the next element of `array`.
- `pPtr += n` increments `pPtr` to point to `n` elements beyond where it currently points.
- `pPtr - qPtr` equals `i - j`.
- Thus, we can see that a pointer arithmetic depends on the type.
- The increment or decrement will jump to 4 bytes if the pointer is integer pointer. 

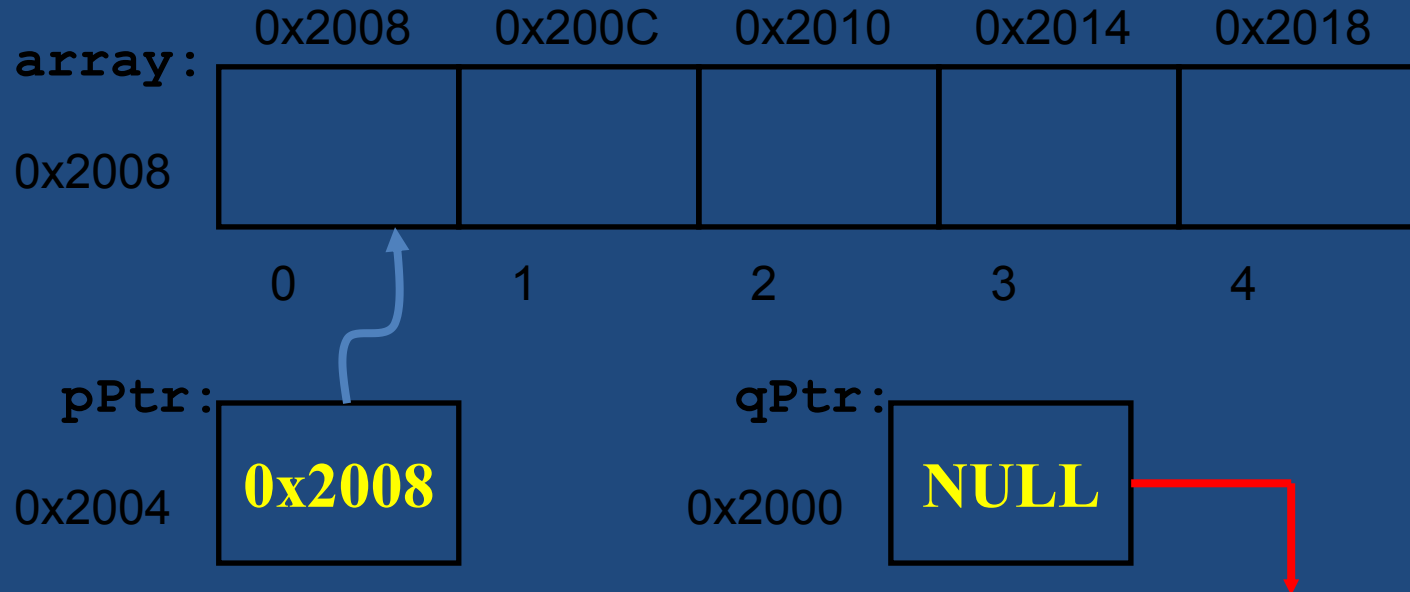
# Pointers and Arrays (cont)

A normal 1 dimensional array:

```
Type array[size] ;
```

- `array[0]` **is equivalent to** `*array`
- `array[n]` **is equivalent to** `*(array + n)`

# *Basic Pointer Arithmetic*

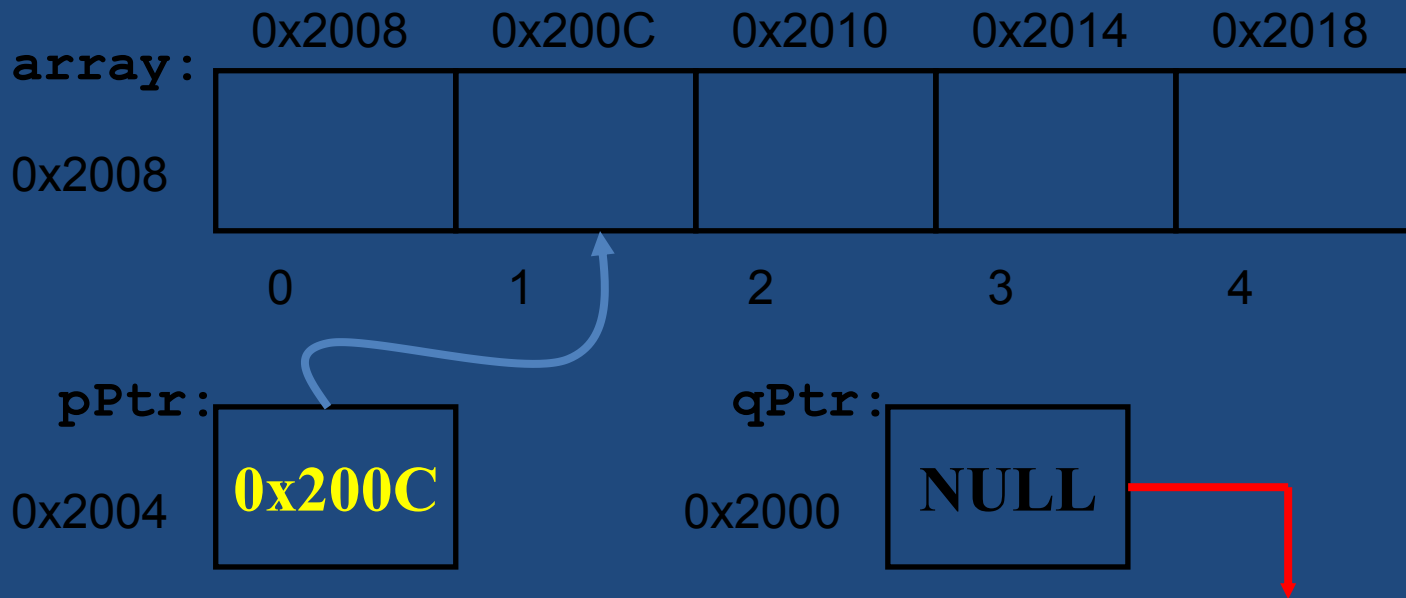


```
float  array[5];
```

```
float* pPtr = array;
```

```
float* qPtr = NULL;
```



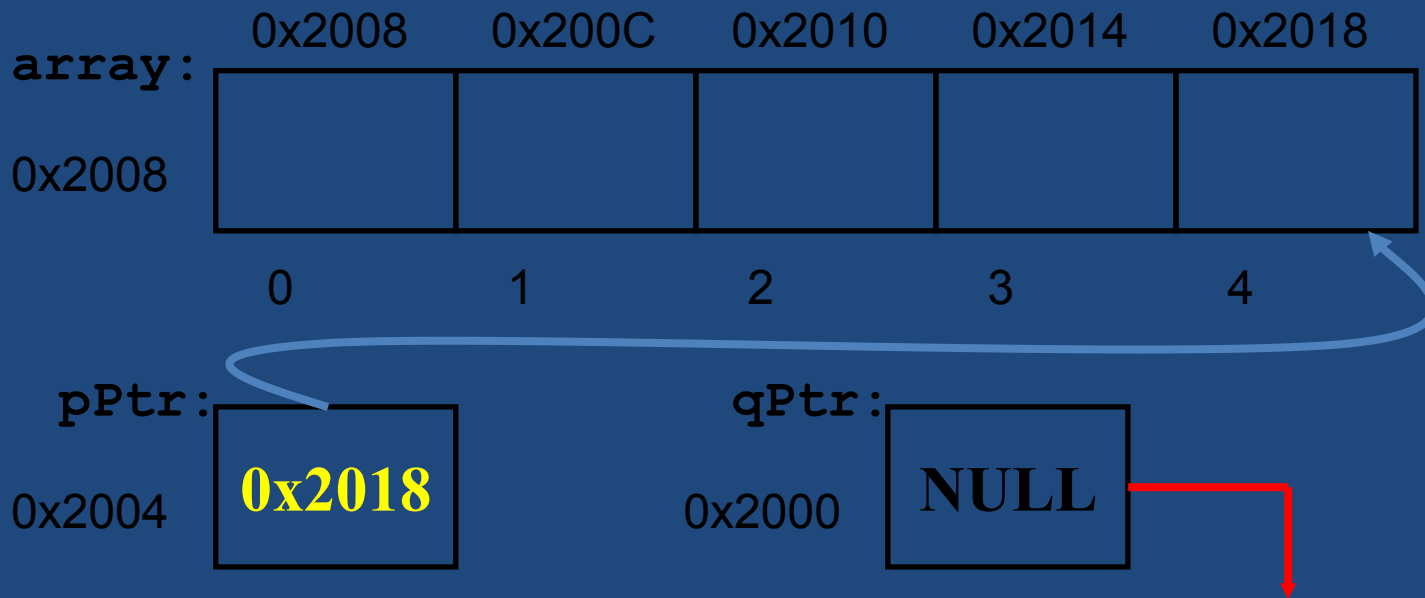


```
float array[5];
```

```
float* pPtr = array;
```

```
float* qPtr = NULL;
```

```
pPtr++; /* pPtr now holds the address: &array[1] */
```



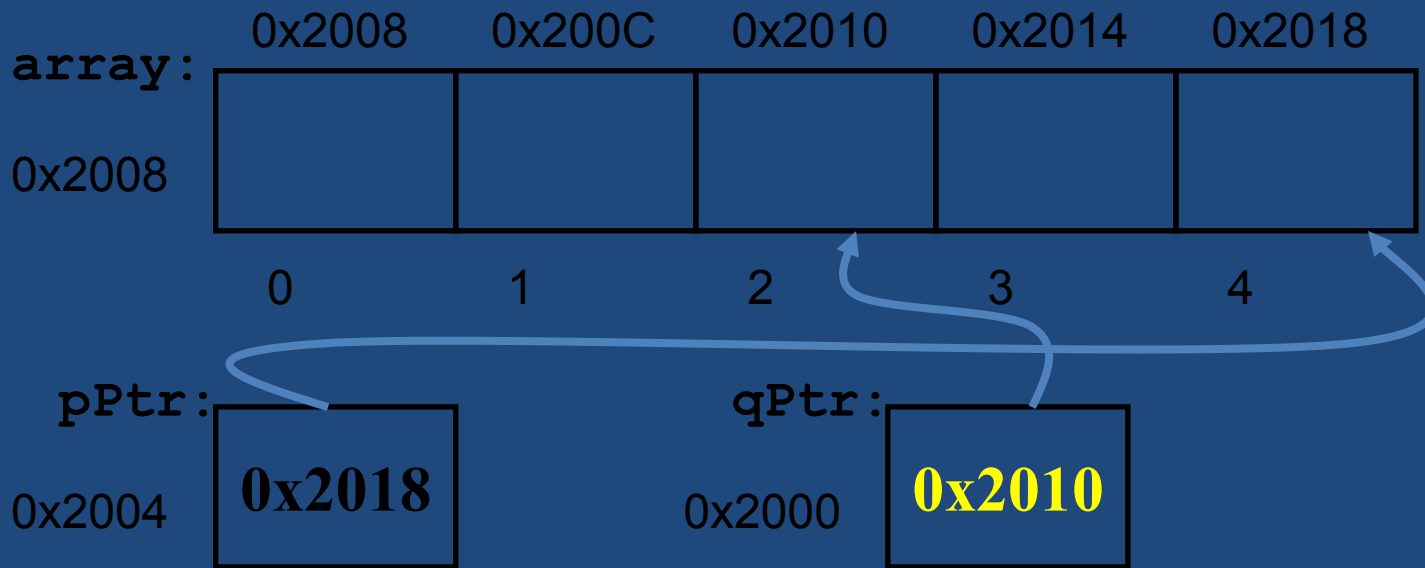
```
float array[5];
```

```
float* pPtr = array;
```

```
float* qPtr = NULL;
```

```
pPtr++; /* pPtr = &array[1] */
```

```
pPtr += 3; /* pPtr now hold the address: &array[4] */
```



```
float array[5];
```

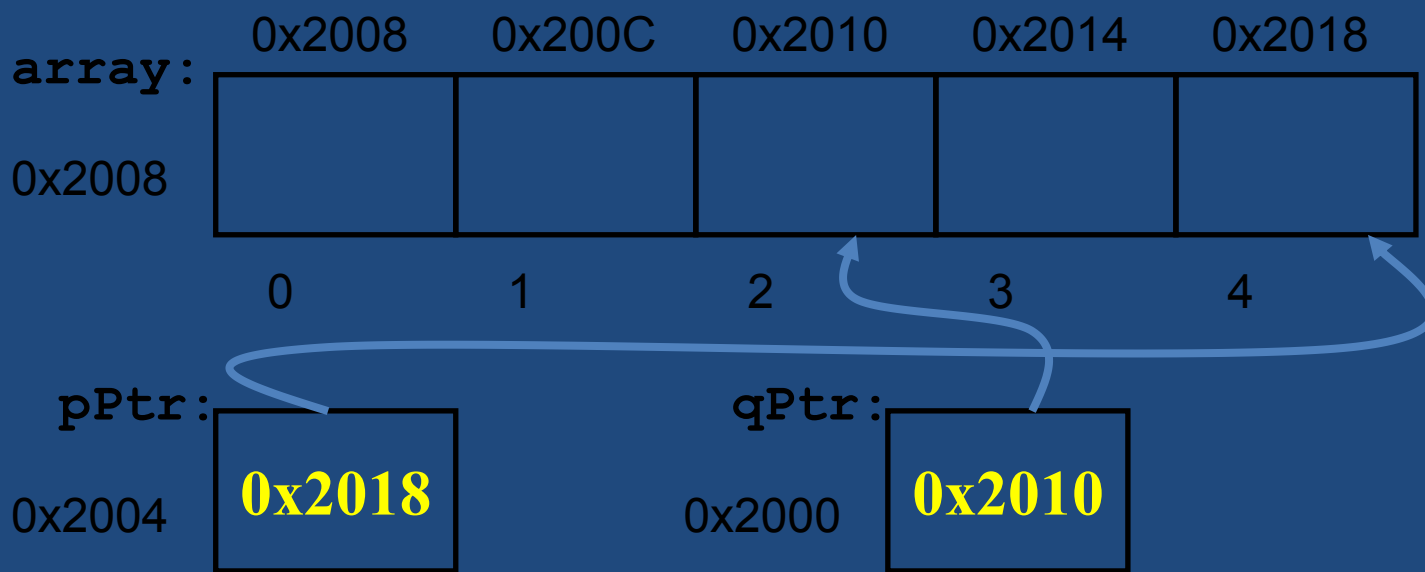
```
float* pPtr = array;
```

```
float* qPtr = NULL;
```

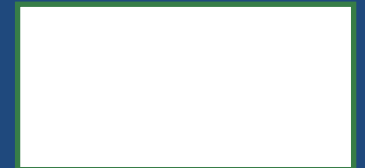
```
pPtr++; /* pPtr = &array[1] */
```

```
pPtr += 3; /* pPtr = &array[4] */
```

```
qPtr = array + 2; /*qPtr now holds the address &array[2]*/
```



```
float array[5];  
float* pPtr = array;  
float* qPtr = NULL;  
  
pPtr++; /* pPtr = &array[1] */  
pPtr += 3; /* pPtr = &array[4] */  
qPtr = array + 2; /* qPtr = &array[2] */  
printf("%d\n", pPtr-qPtr);
```



```
int main()
```

```
{
```

```
    int data[] = {87, 99, 75, 88, 93};
```

```
    int *p;
```

```
    int i;
```

```
    for(i = 0; i<5; i++)
```

```
        printf("address: %p, value: %d, test: %d\n", data+i,  
               *data+i);
```

```
    printf("-----using p-----\n");
```

```
    p = data; // p is now &data[0]
```

```
    for(i = 0; i<5; i++)
```

```
    {
```

```
        printf("address: %p, value: %d \n", p, *p);
```

```
        p++;
```

```
    }
```

```
    return 0;
```

```
}
```

The values of  
the array also  
can be accessed  
using:

`data[i]`  
`*(i+data)`

*Example: traversing through array using pointer*

output:

address: 0x7ffd8ec91e10, value: 87, test: 87

address: 0x7ffd8ec91e14, value: 99, test: 88

address: 0x7ffd8ec91e18, value: 75, test: 89

address: 0x7ffd8ec91e1c, value: 88, test: 90

address: 0x7ffd8ec91e20, value: 93, test: 91

-----using p-----

address: 0x7ffd8ec91e10, value: 87

address: 0x7ffd8ec91e14, value: 99

address: 0x7ffd8ec91e18, value: 75

address: 0x7ffd8ec91e1c, value: 88

address: 0x7ffd8ec91e20, value: 93

*It is just adding i to  
data[0].  
\*data is dereferencing  
data (which is &data[0])*

# What is the difference between pointer and array?

- While **double values[10];** allocates 10 locations to store doubles, **double \*p;** does not.
- Instead **double \*p;** only allocates the memory to store a single location in memory.
- However, we CAN use a pointer to dynamically allocate memory!

# Passing Entire Array to Function

```
main( )
{
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int i;
    printf("Original Array: ");
    for(i=0; i<6; i++)
        printf("%d  ", num[i]);

    incBy2( num, 6 ) ;

    printf("\nArray after calling incBy2: ");
    for(i=0; i<6; i++)
        printf("%d  ", num[i]);
}

void incBy2(int *A, int n)//also can be written as: incBy2(int A[], int n) or incBy2(int A[6], int n)
{
    int i ;
    for ( i = 0 ; i < n ; i++ )
    {
        *A = *A + 2;
        A++;
    }
}
```

Output:

Original Array: 24 34 12 44 56 17

Array after calling incBy2: 26 36 14 46 58 19

## Another version-Passing Entire Array to Function (Generally we use this approach)

```
main( )
{
    int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
    int i;
    printf("Original Array: ");
    for(i=0; i<6; i++)
        printf("%d  ", num[i]);

    incBy2( num, 6 ) ;

    printf("\nArray after calling incBy2: ");
    for(i=0; i<6; i++)
        printf("%d  ", num[i]);
}

void incBy2(int *A, int n)//also can be written as: incBy2(int A[], int n) or incBy2(int A[6], int n)
{
    int i ;
    for ( i = 0 ; i < n ; i++ )
    {
        //even A is a pointer, you can write it like an array as you have passed an array here
        A[i] = A[i] + 2;
    }
}
```

Output:

Original Array: 24 34 12 44 56 17

Array after calling incBy2: 26 36 14 46 58 19



# Multidimensional Arrays

- It is also possible to have array with two or more dimensions
- 2 dimensional array also called a matrix and easy to visualize in tabular form.
- An example declaration:
- `int grid[4][2];`
- This array as 4 rows and 2 columns. Can be seen as:

	Column 0	Column 1
Row 0	<code>grid[0][0]</code>	<code>grid[0][1]</code>
Row 1	<code>grid[1][0]</code>	<code>grid[1][1]</code>
Row 2	<code>grid[2][0]</code>	<code>grid[2][1]</code>
Row 3	<code>grid[3][0]</code>	<code>grid[3][1]</code>

# Multidimensional Arrays

- The declaration `int grid[4][2];` can also be interpreted as:
- This is an array of 4 elements
  - Each element is an array of 2 elements
- So, it is kind of 4 one-dimensional arrays.
- Example: `grid[0]` is an array of 2 elements
  - `grid[0][0]`, and `grid[0][1]`
- The elements are laid out sequentially in memory, just like a one-dimensional array

# Multidimensional Arrays

Index	grid[0][0]	grid[0][1]	grid[1][0]	grid[1][1]	grid[2][0]	grid[2][1]	grid[3][0]	grid[3][1]
Value	5	6	8	23	22	9	90	4
address	1000	1004	1008	1012	1016	1020	1024	1028

- The above table shows the memory map for the 2D array we have declared. The address and data are assumed. We consider int takes 4 bytes.
- See, grid[1][0] starts after completing full grid[0] array.

# 2D Array is an array of array

```
main( )
{
    int grid[4][2] = {
        { 5, 6 },
        { 8, 23 },
        { 22, 9 },
        { 90, 4 }
    } ;

    int i ;
    for ( i = 0 ; i <= 3 ; i++ )
        printf ( "\nAddress of %d th 1-D array = %u", i, grid[i] ) ;
}
```

Output:

Address of 0 th 1-D array = 2961709872

Address of 1 th 1-D array = 2961709880

Address of 2 th 1-D array = 2961709888

Address of 3 th 1-D array = 2961709896

# Accessing 2D array using pointer

- The following are equivalent:
- `s[2][1]` //second row, first col
- `*(s[2]+1)`
- `*(*(s+2)+1)` //here 2 indicates the array number

# Accessing 2D array using pointer

```
main( )
{
    int s[4][2] = {
        { 5, 6 },
        { 8, 23 },
        { 22, 9 },
        { 90, 4 }
    } ;

    int i, j ;
    for ( i = 0 ; i <= 3 ; i++ )
    {
        printf ( "\n" ) ;
        for ( j = 0 ; j <= 1 ; j++ )
            printf ( "%d ", *( *( s + i ) + j ) ) ;
    }
}
```

Output:

```
5 6
8 23
22 9
90 4
```

# 2D array as parameter

- Only first subscript may be left unspecified
- `void f1(int grid[][10]) // valid`
- `void f3(int grid[][]); //invalid`
- Generally, array sizes are also passed to functions while dealing with 1D or multi dimensional array
- `void print( int a[ ][4], int row, int col )`
- The sizes allow to iterate properly and access the elements within the ranges.

# Accessing 2D array

code available in webcourses

```
main( )
{
    int a[3][4] = {
        1, 2, 3, 4,
        5, 6, 7, 8,
        9, 0, 1, 6
    } ;

    Calling print function
    print ( a, 3, 4 ) ;
}
```

```
print ( int q[ ][4], int row, int col )
{
    int i, j ;
    for ( i = 0 ; i < row ; i++ )
    {
        for ( j = 0 ; j < col ; j++ )
            printf ( "%d ", q[i][j] ) ;
        printf ( "\n" ) ;
    }
    printf ( "\n" ) ;
}
```



## Output of the code in the last slide

Output :

1 2 3 4

5 6 7 8

9 0 1 6

# Strings

# Strings review

- String is a char array
- Sequence of zero or more characters terminated by null ('\0') character
- Note that '\0' terminates the string but it is not part of the string.
- **strlen()** function returns the length of a string. null is not part of the length
- there are many useful functions for strings are available at **string.h**

# Strings review

- `char s[10] = "cat";`
- Here **s** is an string and 10 bytes are allocated for the string or array.

0	1	2	3	4	5	6	7	8	9
c	a	t	\0						

- Also can be declared using pointer:  
`char *s = "cat";` //but you cannot change the content in this case. We have to use dynamic memory allocation to work on it.
- You can access each character in the string like an array.
- You can pass string to a function in the same way you pass an array.

# Strings review

- You can use `scanf` and `gets` method for taking string input
- Be careful while using `scanf` to read string as it will skip anything after space.

```
int main()
{
    char name[20];
    printf("Enter name: ");
    scanf("%s", name);
    printf("Your name is %s.", name);
    return 0;
}
```

Output:

Enter name: Sahar Hooshmand

Your name is. Sahar //Hooshmand does not appear in the output

# Strings review

- `gets` and `puts` methods for string.
- If you use `gets`, it can take string with spaces.

```
#include <stdio.h>
int main()
{
    char name[30];
    printf("Enter name: ");
    gets(name);      // read string
    printf("Name: ");
    puts(name);      // display string
    return 0;
}
```

Output:

Enter name: Sahar Hooshmand

Name: Sahar Hooshmand

# Alternative of gets() method for string input

- gets() method is dangerous as it does not care about the size of the char array and can take more than the size. As a result your program can crash and become unsafe.
- Alternative:

```
char name[30];  
printf("Enter name: ");  
scanf("%[^\\n]s", name);
```

# Some useful methods for string available in string.h

- `char *strcpy(char *dst, char const *src);`
  - It copies src string to the dst string
  - It is programmers responsibility to ensure dst has enough space to copy from src.
- `char *strcat(char *dest, const char *src)`
  - It takes two arguments, i.e, two strings or character arrays, and stores the resultant concatenated string in the first string specified in the argument.
- `size_t strlen(const char *str);`
  - Get the length of the passed string
- `char *strstr(const char *haystack, const char *needle)`
  - Search for needle string inside haystack string and returns a pointer to the first occurrence in haystack. Or a null pointer will be returned if needle is not found in haystack.
- `int strcmp(char const *s1, char const *s2);`

Return Value	Remarks
0	if both strings are identical (equal)
negative	if the ASCII value of first unmatched character is less than second.
positive integer	if the ASCII value of first unmatched character is greater than second.



# strcmp test

```
int main()
{
    char str1[] = "abcd", str2[] = "abde", str3[]="abCd",
    str4[]="abcd";
    int result;
    result = strcmp(str1, str2);
    printf("strcmp(str1, str2) = %d\n", result);

    result = strcmp(str2, str1);
    printf("strcmp(str2, str1) = %d\n", result);

    result = strcmp(str1, str3);
    printf("strcmp(str1, str3) = %d\n", result);

    result = strcmp(str3, str1);
    printf("strcmp(str3, str1) = %d\n", result);

    result = strcmp(str1, str4);
    printf("strcmp(str1, str4) = %d\n", result);

    result = strcmp(str4, str1);
    printf("strcmp(str4, str1) = %d\n", result);

    return 0;
}
```

/\*output

strcmp(str1, str2) = -1

strcmp(str2, str1) = 1

strcmp(str1, str3) = 32

strcmp(str3, str1) = -32

strcmp(str1, str4) = 0

strcmp(str4, str1) = 0

\*/

STRUCTS

# Struct revision

- When the data you want to store doesn't fit neatly into a predefined C type, you can create your own. Here is an example:

```
struct employee {  
    char[20] name;  
    double salary;  
    int empID;  
};
```

- Now, to declare a variable of this new data type, we write:

```
struct employee officeworker;
```

- To access the fields of a struct employee variable do as follows:

```
officeworker.empID = 1;
```

# Struct and Pointer

```
struct employee {  
    char[20] name;  
    double salary;  
    int empID;  
};
```

- Now, to declare a variable of this declared type, we write:

```
struct employee officeworker;
```

- Often times, for efficiency purposes, it's easier to declare pointers to structs and use those to manipulate the structures formed.
- Consider the following:

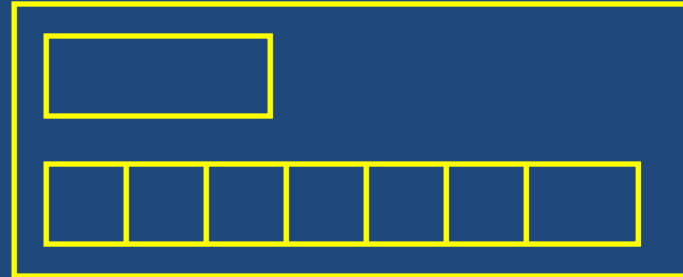
```
struct employee *temp; // declare a pointer of type employee  
temp = &officeworker; //store the address of office worker to temp  
(*temp).salary = 50000; //dereference temp to access salary of officeworker
```

- Since expressions similar to the last one are used so often, there is a shorthand to access a field/member of a record through a pointer to that record. The last statement can also be written as:

```
temp->salary = 50000 // -> is mostly used to dereference structure pointer.
```

- Another reason to use a pointer to a struct is to dynamically allocate the memory for the struct.

# Typedef and Pointers and Structures



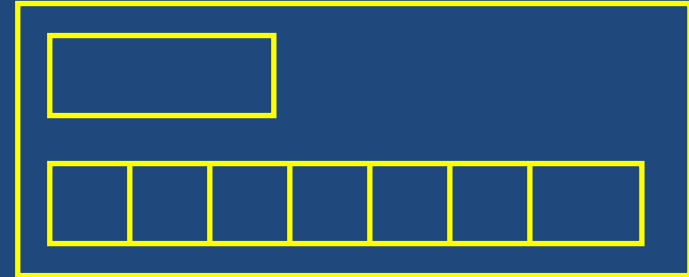
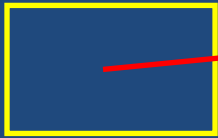
temp

```
typedef struct bookRec{  
    float price;  
    char name[7];  
}Book;
```

//if you use typedef, you don't have to write struct every time you declare a variable of your sturcture

```
Book temp; //this line is possible as we have used typedef  
  
scanf("%d %s", &temp.price, temp.name);
```

# Pointers and Structures



aPtr

typedef must be used

temp

```
struct bookRec{  
    float price;  
    char name[7];  
}Book;
```

```
Book *aPtr;  
Book temp;  
aPtr = &temp;
```

```
scanf("%d %s", &(aPtr->price), aPtr->name);
```