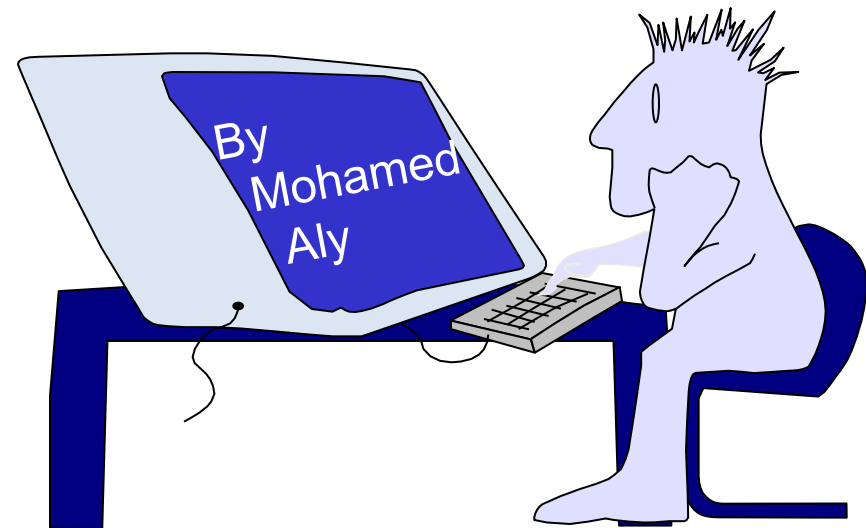


Embedded C

C Programming Part 3

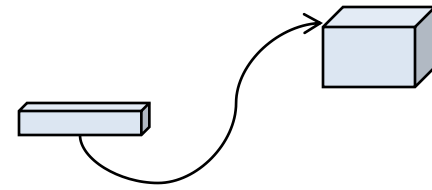


Address vs. Value

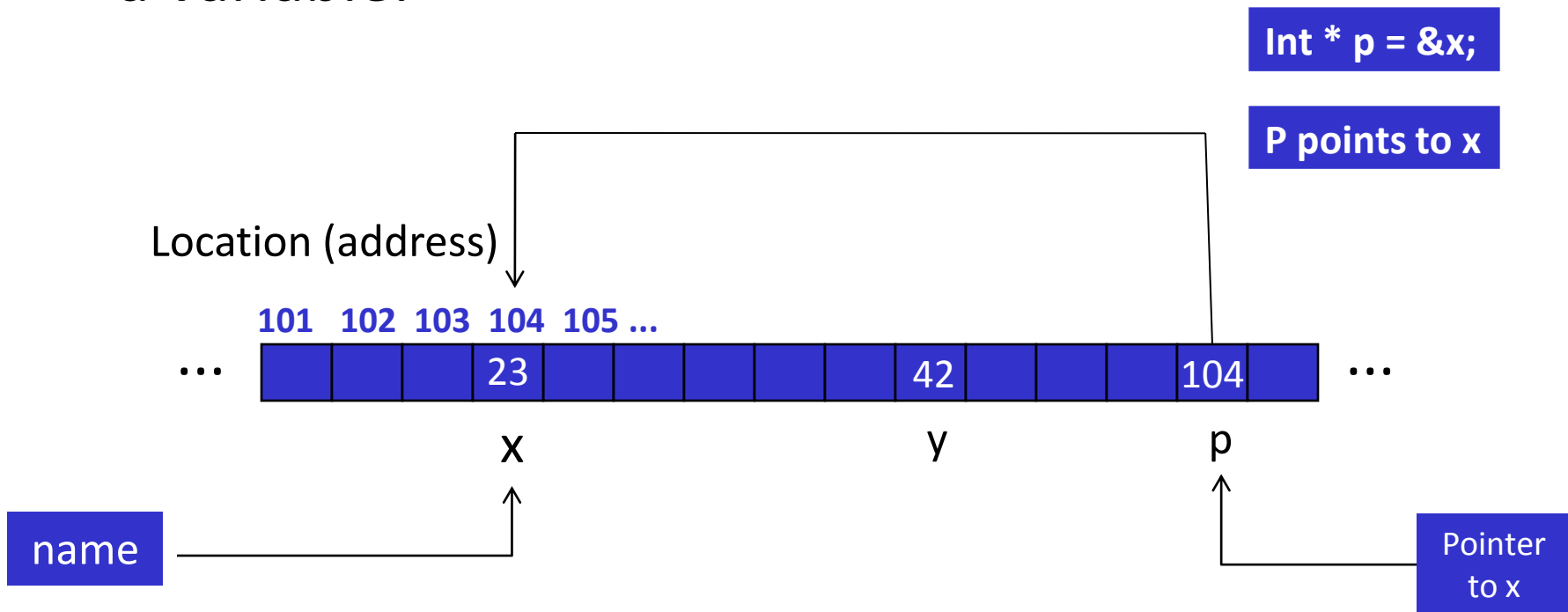
- Consider memory to be a single huge array:
 - Each cell of the array has an address associated with it.
 - Each cell also stores some value.
 - Do you think they use signed or unsigned numbers? Negative address?
- Don't confuse the address referring to a memory location with the value stored in that location.



Pointers



- An address refers to a particular memory location. In other words, it points to a memory location.
- Pointer: A variable that contains the address of a variable.

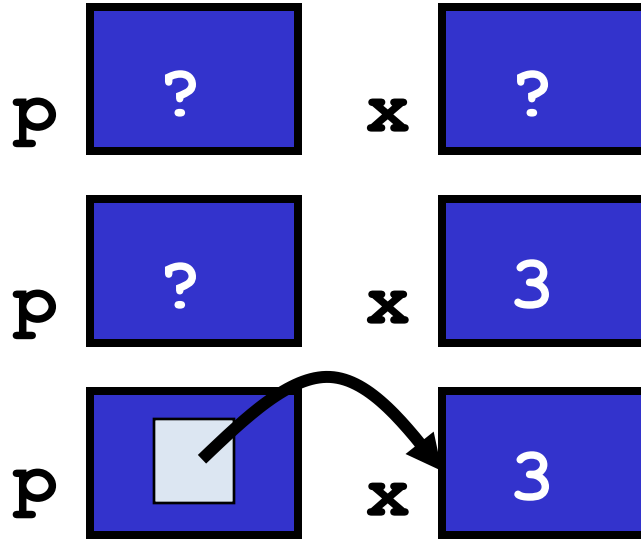


Pointers(Cont.)

- How to create a pointer:

& operator: get address of a variable

```
int *p, x;  
  
x = 3;  
  
p = &x;
```



Note the “*” gets used 2 different ways in this example. In the declaration to indicate that `p` is going to be a pointer, and in the `printf` to get the value pointed to by `p`.

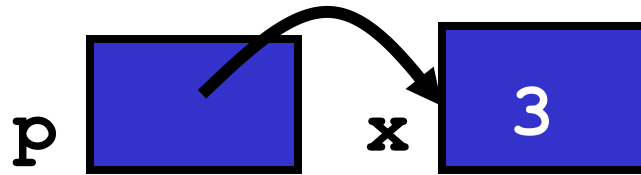
- How get a value pointed to?

* “dereference operator”: get value pointed to

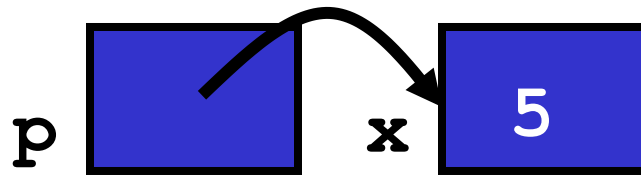
```
printf("p points to %d\n",*p);
```

Pointers(Cont.)

- How to change a variable pointed to?
 - Use dereference `*` operator on left of `=`



```
*p = 5;
```



Pointers(Cont.)

NULL pointer

- The Simplest Pointer in C
- Special constant pointer **NULL**
- Points to no data
- Dereferencing is illegal
- Dereferencing causes *segmentation fault*
- To define, include **<stdlib.h>** or **<stdio.h>**

Pointers(Cont.)

- C pass parameters “by value”

```
void addingOne (int x)
{
    x = x + 1;
}
int main(void)
{
    int y = 3;
    addingOne(y); /*y is still = 3*/
    return 0;
}
```

Pointers(Cont.)

- How to get a function to change a value?

```
void addingOne (int *p)
{
    *p = *p + 1;
}

main
{
    int y = 3;
    addingOne(&y); /*y is now = 4*/
}
```


Pointers(Cont.)

Pointers dangers

- Declaring a pointer just allocates space to hold the pointer – it does not allocate something to be pointed to!
- What does the following code do?

```
void f()  
{  
    int *ptr;  
    *ptr = 5;  
}
```

Pointers(Cont.)

Pointers dangers(Cont.)

- C lets you cast a value of any type to any other type without performing any checking

```
int x = 1000;  
int *p = x;           /* invalid */  
int *q = (int *) x;   /* valid */
```

- The first pointer declaration is invalid since the types do not match. The second declaration is valid C but is almost certainly wrong
 - Is it ever correct?

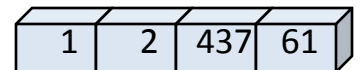
Arrays

- Ordered collection of elements of homogeneous type

- “this string is also a array of chars”



- {1, 2, 437, 61}



Arrays(Cont.)

- Declaration:

```
int ar[2];
```

- declares a 2-element integer array.

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

- If not enough initializers, rightmost elements become 0
- If too many initializers, a syntax error occurs

```
int ar[] = {795, 635};
```

- declares and fills a 2-element integer array.

- Accessing elements:

```
ar[num];
```

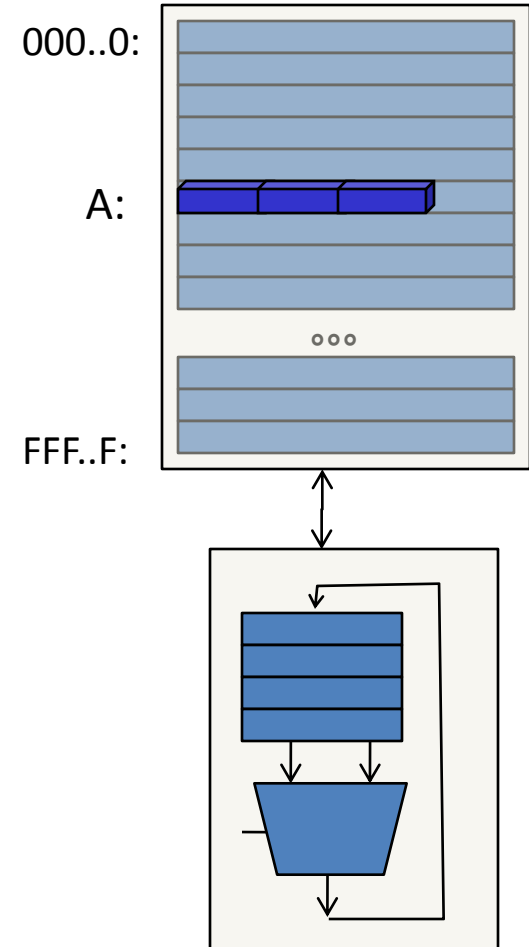
- returns the numth element.

```
int ARRAY_SIZE = 10;
int i;
int a[ARRAY_SIZE];
int result = 0;
for(i = 0; i < ARRAY_SIZE; i++)
{
    result = result + a[i];
}
```

Arrays(Cont.)


Where do arrays reside?

- Arrays are stored in memory
- The variable (i.e., name) is associated with the location (i.e., address) of the collection
 - Just like any basic variable
- Elements are stored consecutively



Arrays(Cont.)

- An array in C does not know its own length, & bounds not checked
 - We can accidentally access off the end of an array
 - So we must pass the array and its size to a procedure which is going to traverse it
 - This may lead to very difficult to find errors like ***segmentation faults*** and ***bus errors***



***Please, Be
careful!!!***

Arrays(Cont.)

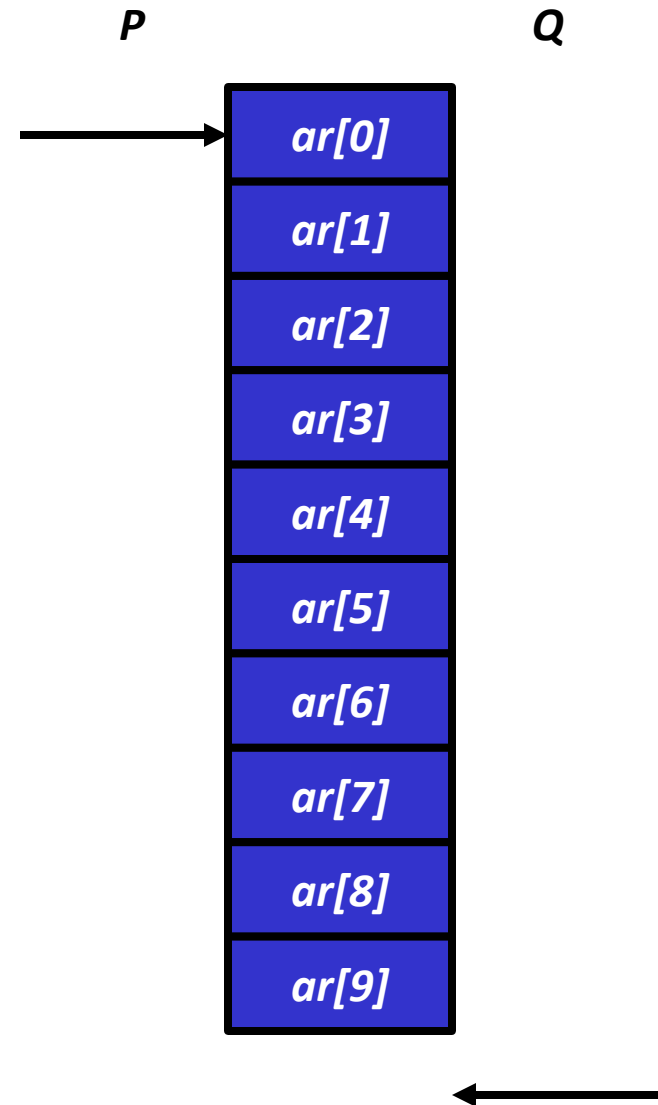
Example

```
int ar[10], *p, *q, sum = 0;  
p = &ar[ 0];  
q = &ar[10];
```

//What if we write ar[-1]

```
while (p != q)  
    sum += *p++;  
/* sum = sum + *p; */  
/* p = p + 1; */  
/*WILL THIS LEAD TO ERROR?*/
```

- **No**, this will not lead to error
- C defines that one element past end of array must be a valid address



Segmentation fault and bus error

■ ***Segmentation fault***

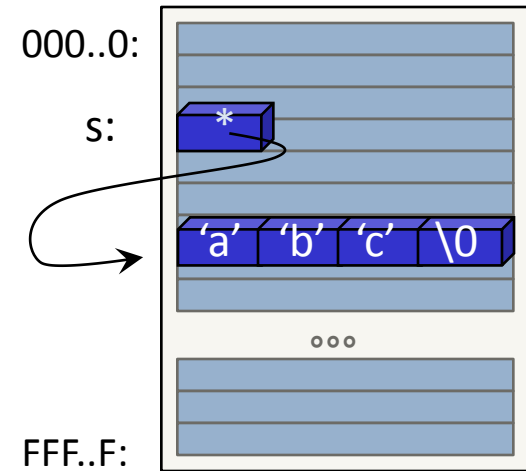
- Running program attempts to access memory not allocated to it
- Running program terminates with a segmentation violation error

■ ***Bus error***

- Processor detecting an anomalous condition on its bus
- Such conditions include invalid address alignment (accessing a multi-byte number at an odd address), accessing a physical address that does not correspond to any device, or some other device-specific hardware error.
- A bus error triggers a processor-level exception which may lead to terminate the running program

Arrays(Cont.)

```
char *s;  
s = "abc";
```

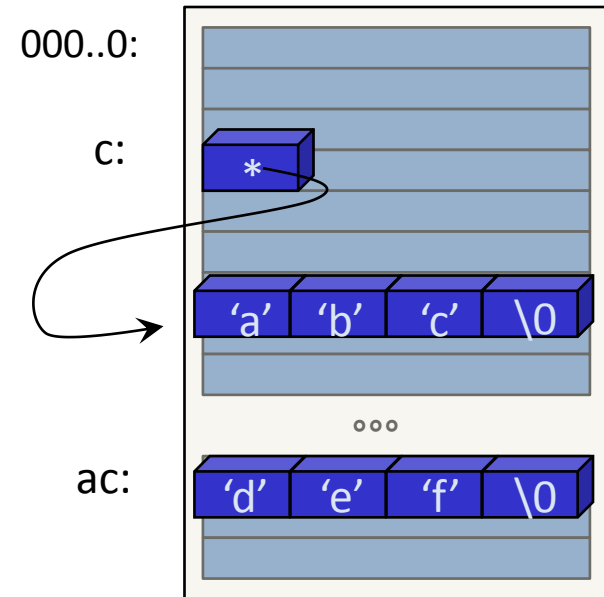


s is a reference to the string “abc”

s is a pointer to the string “abc”

Arrays(Cont.)

```
int main( void )
{
    char * c      = "abc";
    char  ac [4] = "def";
    printf("c[1]=%c\n",c[1] );
    printf("ac[1]=%c\n",ac[1] );
    return 0;
}
```



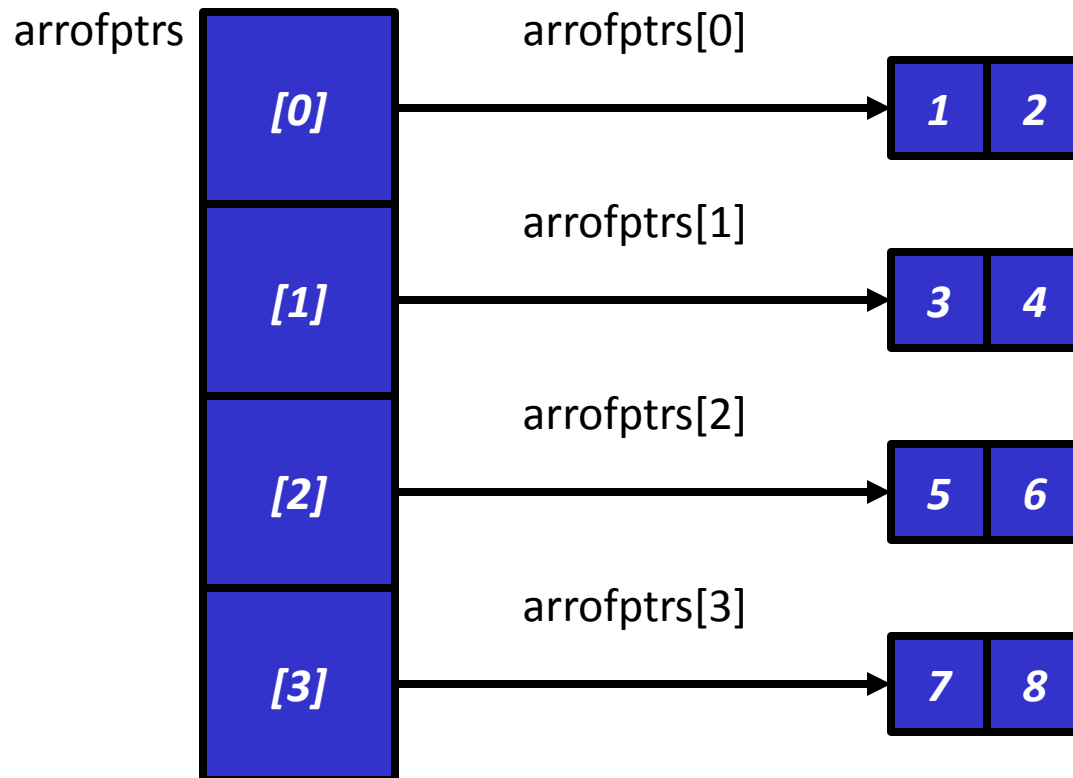
- Array name is essentially the address of (**read-only** pointer to) the zeroth element in the array
- Arrays are (almost) identical to pointers
- There are a few subtle differences
 - Can change what `c` refers to, but not what `ac` refers to

Arrays(Cont.)

Array of pointers

- Arrays can contain pointers

```
int *arrofptrs[ 4 ] = { {1,2}, {3,4}, {5,6}, {7,8}};
```



Multiple-subscripted arrays

- Multiple subscripted arrays are used to describe tables/matrices with rows and columns (m by n matrix)

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

b[0][0]

b[0][1]

b[1][0]

b[1][1]

- Initializers grouped by row in braces
- If not enough, unspecified elements set to zero

```
printf( "%d", b[ 0 ][ 1 ] );
```

- C interprets a[x, y] as a[y], and as such it does not cause a syntax error.

Arrays(Cont.)

Passing arrays to functions

- An array is passed to a function as a pointer
 - **The array size is lost!**
- Function prototype
 - void modifyArray(int arrayName[], int arraySize);**
 - Parameter names optional in prototype
- Passing arrays
 - To pass an array argument to a function, specify the name of the array without any brackets
 - int myArray[24];**
 - modifyArray(myArray, 24);**
 - Arrays passed call-by-reference
- Passing array elements
 - Passed by call-by-value
 - Pass subscripted name (i.e., myArray[3]) to function

Arrays(Cont.)

Passing arrays to functions(Cont.)

```
int fun(int array[], unsigned int size)
{
    printf(" %d\n", sizeof(array));
}
```

What does this print? **4**

... because **array** is really
a pointer

```
int main(void)
{
    int a[10], b[5];
    fun(a, 10);
    fun(b, 5);
    printf("%d\n", sizeof(a));
}
```

What does this print? **40**

Arrays(Cont.)

Passing arrays to functions(Cont.)

- An array parameter can be declared as an array or a pointer; an array argument can be passed as a pointer.

```
int strlen(char s[])  
{  
    int n = 0;  
    while (s[n] != 0)  
        n++;  
    return n;  
}
```

Can be written:
`while (s[n])`

```
int strlen (char *s)  
{  
    int n = 0;  
    while (s[n] != 0)  
        n++;  
    return n;  
}
```

Can be written:
`while (s[n])`

Arrays(Cont.)

Passing arrays to functions(Cont.)

- Passing a multidimensional array as an argument requires all but the first dimension
- **int** array[10][3][2];
- **void** examine(array[][3][2]) {....}

Thanks

Embedded C