

```
#include <stdio.h>
int main(void)
{
    int count;

    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```

AVENIO 10.1.3



© 2009 All rights reserved. | Distributed by Universal Press Books

CSE 30

Lecture 5: C Strings and Multi- dimensional arrays

Outline

I. Strings

II. Arrays

- A. Representation in memory (difference with arrays in Java)
- B. Arrays and Pointers
- C. Fast data access with arrays
- D. Array pitfalls

Promotion and truncation

- Often arithmetic expressions contain mixed integral types
- Promotion: In an expression all variables are 'promoted' to the data type of the largest size

```
char c=4;
```

```
int i= c+ 10; /* c is promoted to an int */
```

```
/*What is the value in i? */
```

- Truncation: If the value of a larger data type is assigned to a smaller one, the compiler drops the most significant bits

```
short i= 0x0104;
```

```
char c =i; /*The value in i is truncated*/
```

```
/* c =0x04 – Least significant byte is assigned */
```

Type casting

- If we want the variables in an expression to be interpreted differently, use casting

- Example:

```
int i=10;
```

```
float f = i/3;
```

- In this case, $f=3.0$
- If we don't want the result of the division to get truncated to an int, use explicit casting

```
int i=10;
```

```
float f = (float)i/3; /* Explicit cast of int to float */
```

Type casting with pointers

- We can cast addresses that are assigned to pointers to a different type

```
char c = 0x04;
```

```
int *p;
```

```
p = (int *)&c; /* Treat &c as address of an int */
```

OR

```
short i = 0x0104;
```

```
char *p;
```

```
p = (char *)&i; /* Treat &i as address of a char */
```

What is the output of the following code?

```
char *p;  
short i = 0x0104;  
p = (char *)&i; /* Treat &i as address of a char */  
printf("Value in p is: %x", *p);
```

- A. It is always the lower byte 0x04
- B. It is always the higher byte 0x01
- C. Depends

What is the output of the following code?

```
char *p;  
short i = 0x0104;  
p = (char *)&i; /* Treat &i as address of a char */  
printf("Value in p is: %x", *p);
```

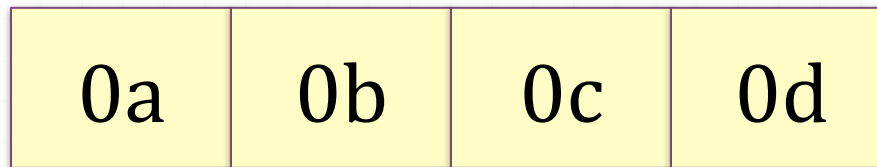
- A. It is always the lower byte 0x04
- B. It is always the higher byte 0x01
- C. Depends: Little endian 0x04, Big endian 0x01

Difference with Java references: Pointers in C have a more generic usage

Byte ordering in memory

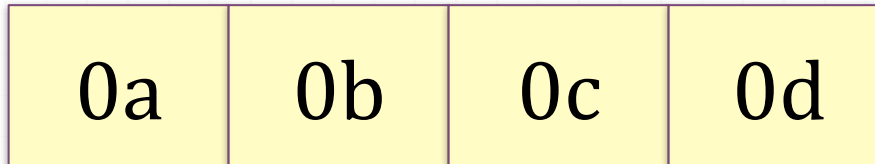
Consider the 4 byte number 0x0a0b0c0d:

- Big Endian: Most significant byte in lowest memory address



Increasing memory addresses

- Little Endian: Least significant byte in lowest memory address



Increasing memory addresses

C Strings

- A string in C is just an array of characters.

```
char string[] = "abc";  
char string[20];
```

C Strings

- A string in C is just an array of characters.

```
char string[] = "abc" ;  
char string[20];
```

- How do you tell how long a string is?
 - Last character is followed by a 0 byte (null terminator)

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

Q: Why can we not copy strings using the function below?

```
void copy (char sTo[], char sFrom[]) {  
    sTo = sFrom;  
}
```

- A.** 'sTo' is an array, therefore its value cannot be changed
- B.** We can use the assignment statement but we have to allocate space for 'sTo' prior to the assignment
- C.** The change in the value of 'sTo' is not reflected in the calling function

Q: Why can we not copy strings using the function below?

```
void copy (char sTo[], char sFrom[]) {  
    sTo = sFrom;  
}
```

- A.** 'sTo' is an array, therefore its value cannot be changed
- B.** We can use the assignment statement but we have to allocate space for 'sTo' prior to the assignment
- C.** The change in the value of 'sTo' is not reflected in the calling function

Q: Why can we not compare strings using the function below?

```
int compare(char sT1[], char sT2[]) {  
    return(sT1 == sTo);  
}
```

- A. We cannot have an arithmetic expression in the return statement
- B. The function only compares the base address of the strings, not their value
- C. The function only compares if the first element of the two strings are the same

Q: Why can we not compare strings using the function below?

```
int compare(char sT1[], char sT2[]) {  
    return(sT1 == sT2);  
}
```

- A. We cannot have an arithmetic expression in the return statement
- B. The function only compares the base address of the strings, not their value
- C. The function only compares if the first element of the two strings are the same

C String Standard Functions

- `int strlen(char *string);`
 - compute the length of `string`
- `int strcmp(char *str1, char *str2);`
 - return 0 if `str1` and `str2` are identical (how is this different from `str1 == str2`?)
- `int strcpy(char *dst, char *src);`
 - copy the contents of string `src` to the memory at `dst`. The caller must ensure that `dst` has enough memory to hold the data to be copied.
- Defined in the header file `string.h`

Multidimensional arrays: 2D

- Declaration

`int a[3][4];` */*Conceptually 2D matrix with 3 rows 4 columns */*

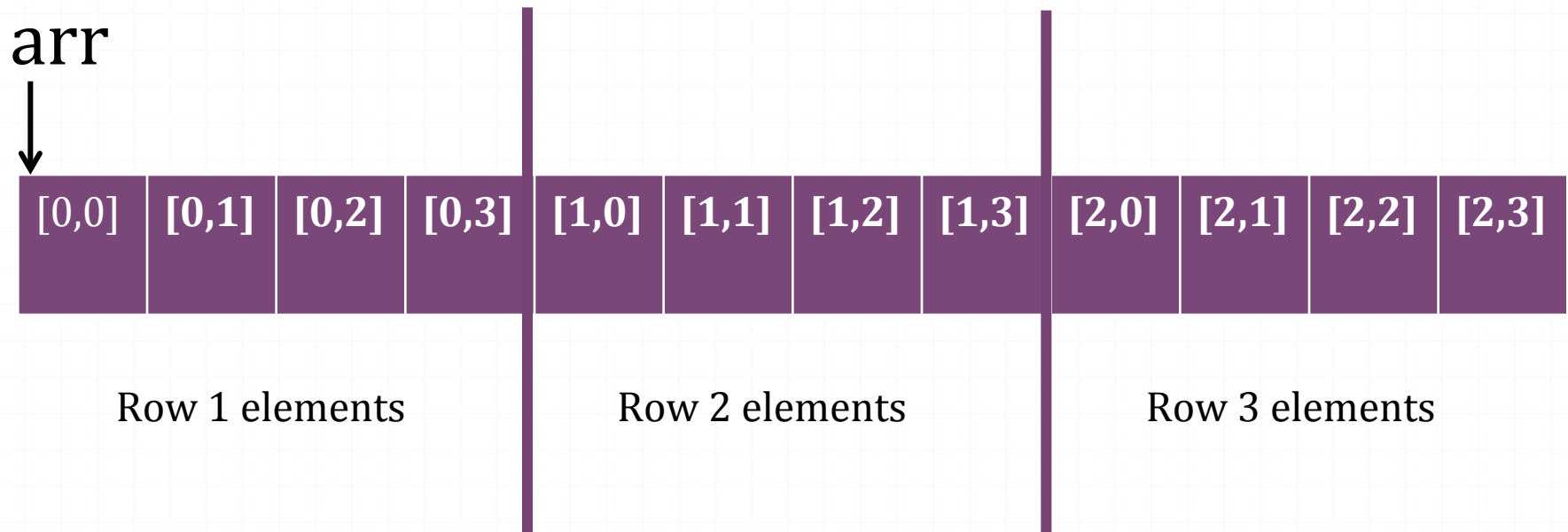
	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Element in row i, column j is retrieved as `a[i][j]`
- 'a' is a pointer to an integer array of size 4

Representation in memory

```
int arr[3][4];    /*Conceptually 2D matrix with 3 rows 4 columns */
```

In memory the elements of 'arr' are stored in a contiguous memory block



(arr+i) increments the address of 'arr' by how many bytes?



- A. $i * \text{sizeof}(\text{int})$
- B. $i * \text{sizeof}(\text{int}^*)$
- C. $i * \text{sizeof}(\text{int}) * \text{number of columns}$

(arr+i) increments the address of 'arr' by how many bytes?



A. $i * \text{sizeof}(\text{int})$

B. $i * \text{sizeof}(\text{int}^*)$

C. $i * \text{sizeof}(\text{int}) * \text{number of columns}$

Express `arr[i][j]` using 'arr' as a pointer



- A. `*((arr+i) +j)`
- B. `*(* (arr+i)+j)`
- C. `*((int*)(arr+i)+j)`
- D. `*(arr+4*i*sizeof(int)+j*sizeof(int))`

Express `arr[i][j]` using 'arr' as a pointer



- A. $*((arr+i) +j)$
- B. $*(* (arr+i)+j)$: This is what is generally used
- C. $*((int*)(arr+i)+j)$
- D. $*(arr+4*i*sizeof(int)+j*sizeof(int))$

Multi-level arrays

- Declaration

```
char name_1[]="John";
```

```
char name_2[]="Paul";
```

```
char name_3[]="Rose";
```

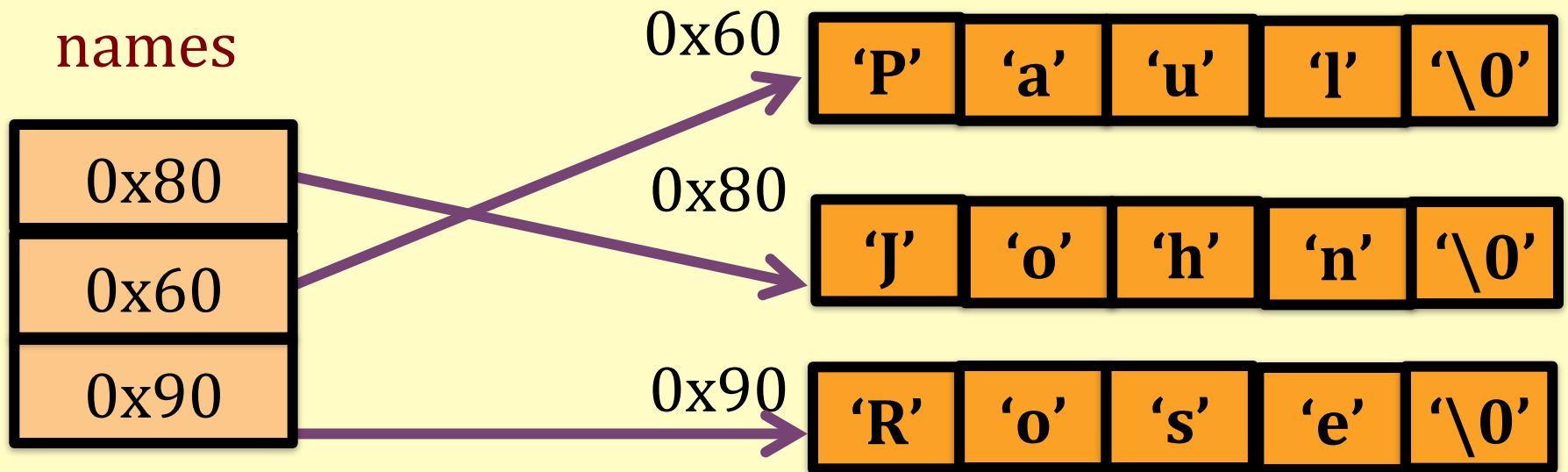
```
char * names[]={name_1, name_2, name_3};
```

What does the following statement print?

```
printf("\n Lettter: %c\n", names[1][2]);
```

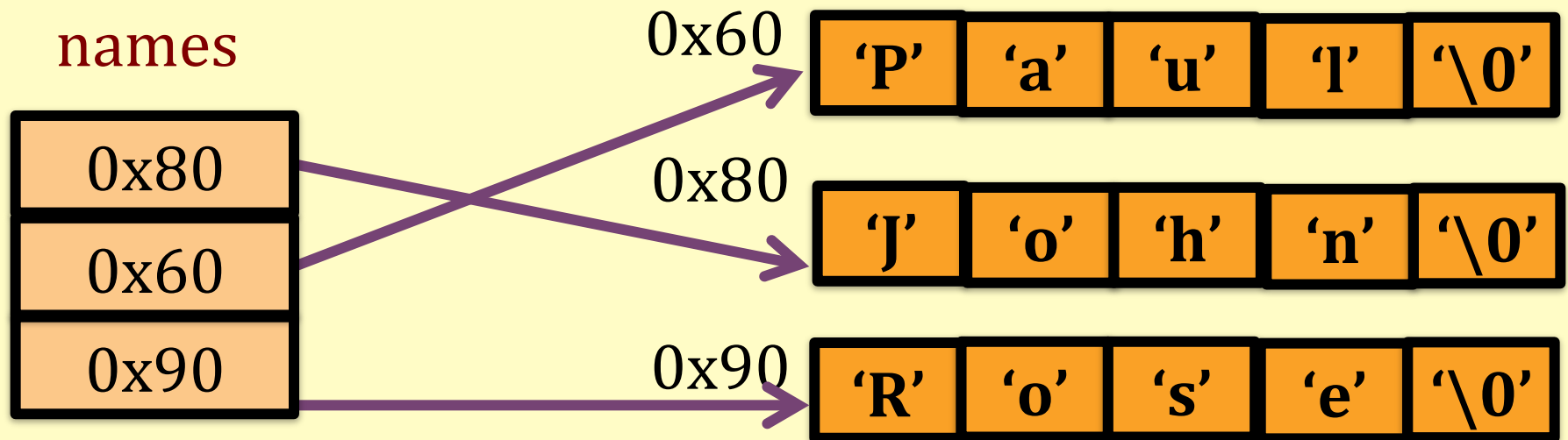
Multi-level arrays: Representation in memory

```
char name_1[]="John";  
char name_2[]="Paul";  
char name_3[]="Rose";  
char * names[]={name_1, name_2, name_3};
```



Multi-level vs multi-dimensional arrays

1. What does `names[1][2]` give in each case?
2. Which one needs more memory accesses?
3. When would we prefer multi-level arrays?



names

`'J' 'o' 'h' 'n' \0 'P' 'a' 'u' '1' \0 'R' 'o' 's' 'e' \0`