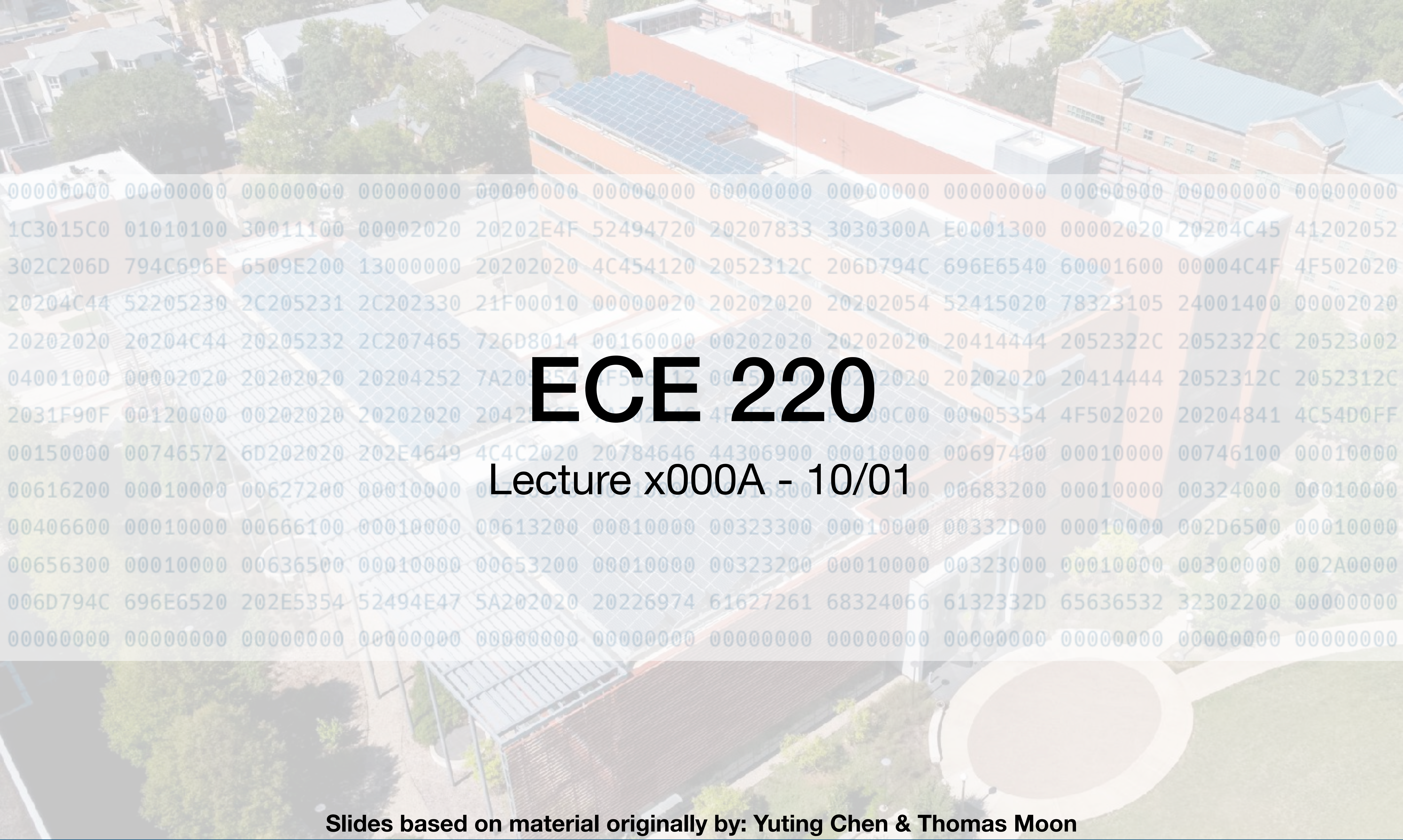


ECE 220

Lecture x000A - 10/01

Slides based on material originally by: Yuting Chen & Thomas Moon

Recap + reminders

- Last week:
  - Introduction to pointers & arrays, `sizeof` function, etc.
  - Briefly touched pointer/array duality
  - Midterm happened

- This week:
  - Wrap up pointer/array duality
  - Strings & multidimensional arrays
  - Problem solving examples
  - Quiz #2

Recap

Last time we wrote this function together in class.

```c
#include <stdio.h>

int my_first_sum(int arr[]){
  int sum=0, i=0;
  for (i=0; i<5; i++)
    sum += arr[i];
  return sum;
}

int main(void){
  int i, arr[5];
  for (i=0; i<5; i++){
    printf("Enter an integer:\t");
    scanf("%d", &arr[i]);
  }
  printf("\nThe sum is %d", my_first_sum(arr));
}
```

Recap

How did we let the compiler know `my_first_sum` takes an array of integers as a parameter?

```c
#include <stdio.h>

int my_first_sum(int arr[]){
  int sum=0, i=0;
  for (i=0; i<5; i++)
    sum += arr[i];
  return sum;
}

int main(void){
  int i, arr[5];
  for (i=0; i<5; i++){
    printf("Enter an integer:\t");
    scanf("%d", &arr[i]);
  }
  printf("\nThe sum is %d", my_first_sum(arr));
}
```

Recap

How did we pass the parameter `arr` to the function `my_first_sum` ?

```c
#include <stdio.h>

int my_first_sum(int arr[]){
  int sum=0, i=0;
  for (i=0; i<5; i++)
    sum += arr[i];
  return sum;
}

int main(void){
  int i, arr[5];
  for (i=0; i<5; i++){
    printf("Enter an integer:\t");
    scanf("%d", &arr[i]);
  }
  printf("\nThe sum is %d", my_first_sum(arr));
}
```

Fact: The name of the array is *pointer* to the array!

Pointer/array duality

- In fact `arr[3]` is syntactic sugar for `*(arr + 3)` !!

```
int my_third_sum(int *arr){
  int i, sum=0;
  for (i=0; i<5; i++)
    sum += *(arr + i);
  return sum;
}
```

would also work just fine!

Dualities: each row of the table contains equivalent expressions

```
char arr[10];
 char *cptr;
 cptr = arr;
```

Pointer arithmetic	Pointer arithmetic	Array notation

Pointers - subtle points

- Is there a difference between `cptr` and `arr` in the below?

```
 char arr[10];
  char *cptr;
  cptr = arr;
```

`cptr` is defined as a variable. The compiler allows it to be redefined.

- Try doing:

```
cptr = cptr + 1;
arr = arr + 1;
```

`arr` without the `[]` *decays* to a pointer but once declared is not assignable *sans* subscript.

Pointers - subtle points

- What is the difference between `arrp`, `arrpw` in the code snippet on the right?

- **Hint**: Consider the output.

```c
#include <stdio.h>

int main(){
   int *arrp;
   int (*arrpw)[5];
   int arr[5]={5,2,3,1,4};

   arrp = arr;
   arrpw = &arr;

   printf("arrp= %p, arrpw= %p\n", arrp, arrpw);
   arrp++;
   arrpw++;
   printf("arrp= %p, arrpw= %p\n", arrp, arrpw);
}
```

Pointers - subtle points

- What is the difference between `arrpw, parr` in the code snippet on the right?

```c
#include <stdio.h>

int main(void){
  int arr[5] = {1, 2, 3, 4, 5};
  int (*arrpw)[5] ;
  int *parr[5];

  arrpw = &arr;

  for (int i=0; i<5; i++){
    printf("*(*arrpw + %d): %d\n", i, *(*arrpw + i));
    printf("*parr[%d]: %d\n", i, *parr[i]);
  }
}
```

Same as before.

`parr` is now an *array* of five pointers.

More bewares ...

- Pointers can be used to modify *static* variables defined inside functions.

```
#include <stdio.h>

int *printx(void){
    static int x = 0;
    printf("value of x is %d \n",x++);
    return (&x);
}

int main(){
    int *x_ptr;
    x_ptr = printx();
    x_ptr = printx();
    *x_ptr = (*x_ptr) + 1;
    printx();
}
```

- Actually, pointers can also modify `const` variables.

```
int main(void){
    const int var = 10;
    int *ptr = &var;

    *ptr = 12;
    printf("var = %d\n", var);
}
```

Yes there are things called `const` pointers - but we will only go there when we have to.

Summary: pointers & arrays

- Pointer
  - Stores the address of a variable in memory
  - Allows us to indirectly access/change variables
- Arrays
  - A list of values arranged sequentially in memory
  - Array name without index is the same as pointer to the array
  - Therefore in C, all arrays are ***passed by reference,*** i.e., if you change array passed to a function, change will be reflected outside!

Using arrays

- When using arrays we need to know the size or *dimensions* of the arrays.

- **Question:** Write a C *function* that sums an array of integers of given length n.

- Any loops in the function will need to know the size of the array to correctly terminate. Two common strategies:

  - Define the length as a global variable.

  - Write the *function* so that it accepts the array length as a *parameter*.

Using arrays

- When using arrays we need to know the size or *dimensions* of the arrays.

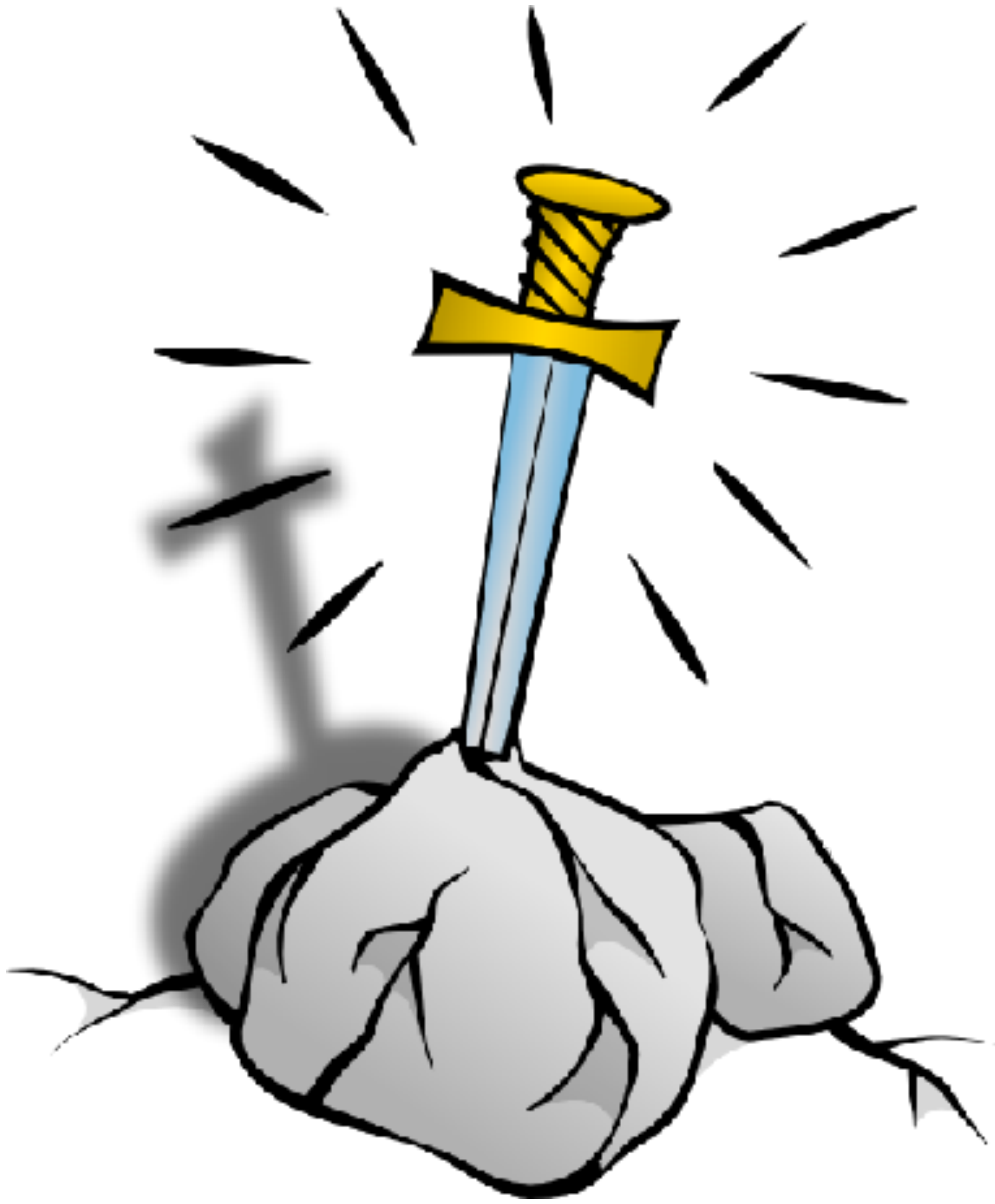
- **Question:** Write a C *function* that sums an array of integers of given length n.

```c
# include<stdio.h>

int any_sum(int arr[], int arr_len){
int i, sum = 0;
  for (i=0; i < arr_len; i++)
    sum += arr[i];
return sum;
}

int main(void){
  int arr1[] = {1, 2, 3, 4, 5};
  int arr2[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

  printf("sum(arr1): %d\n", any_sum(arr1, 5));
  printf("sum(arr2): %d\n", any_sum(arr2, 9));
}
```

Using arrays

- **Challenge:** Can the function be modified so `any_sum` can determine the size of the array *itself* (without passing in the value)?

Definitely let me know if you find a way. 😉

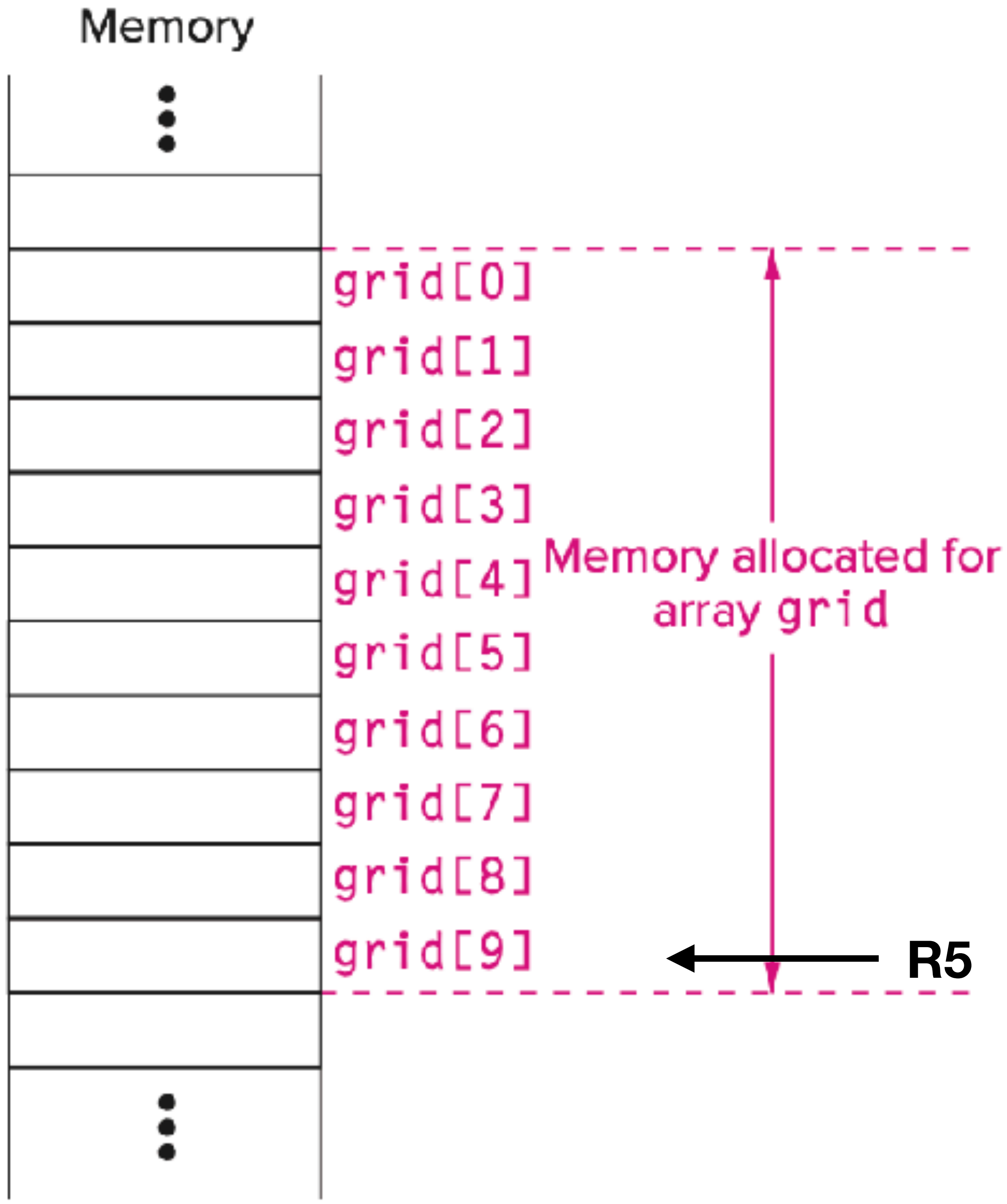


Arrays in LC-3

- The declaration `int grid[10];` allocates 10 integer sized consecutive memory locations on the *stack*.

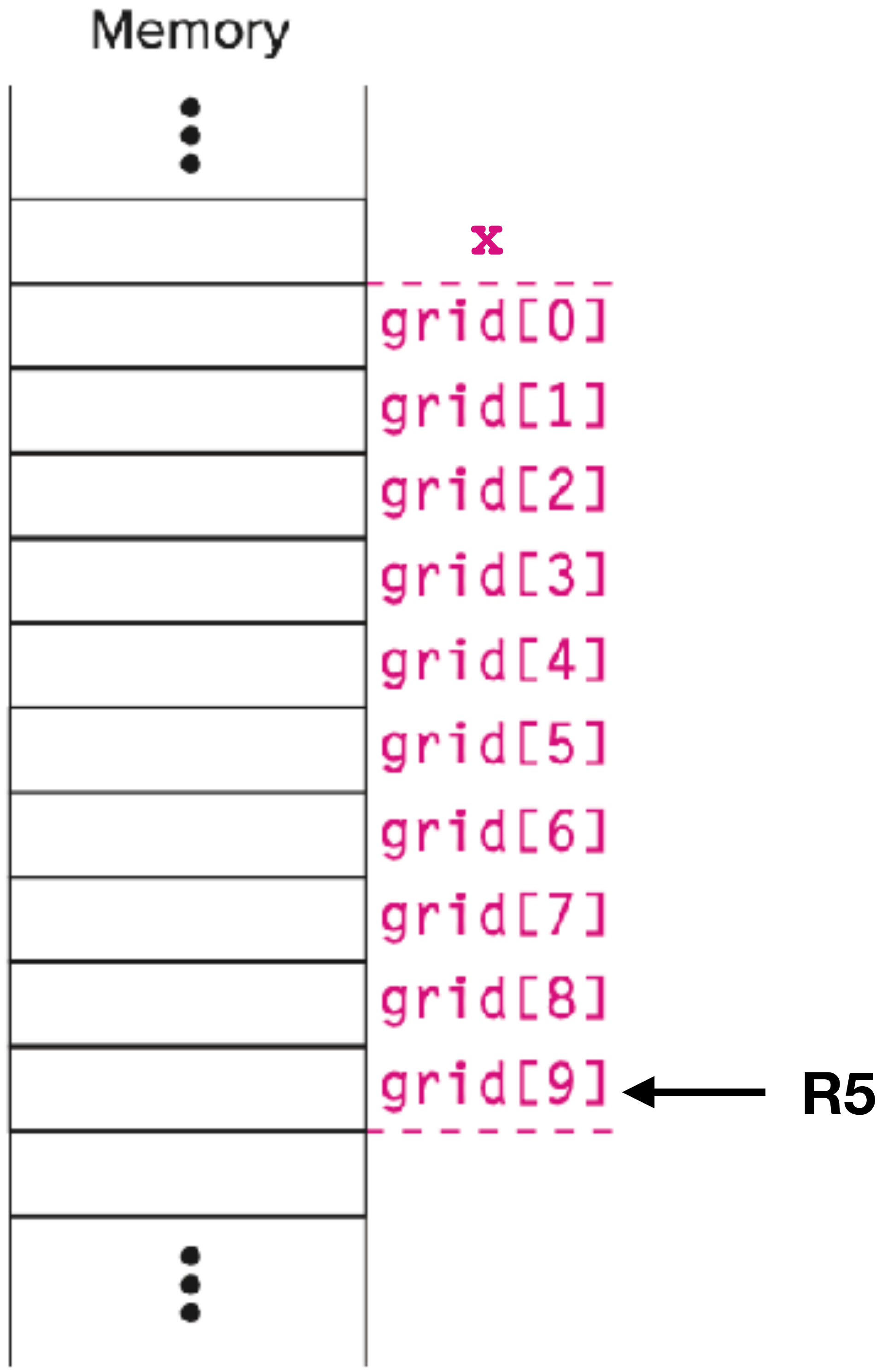
```
grid[6] = grid[3] + 1;
```

```
ADD R0, R5, #-9 ; Base address of grid
LDR R1, R0, #3  ; R1 <-- grid[3]
ADD R1, R1, #1  ; R1 <-- grid[3] + 1
STR R1, R0, #6  ; grid[6] = grid[3] + 1;
```



Arrays in LC-3

```
grid[x+1] = grid[x] + 2;

LDR R0, R5, #-10 ; Load the value of x
ADD R1, R5, #-9  ; Base address of grid
ADD R1, R0, R1   ; Calculate address of grid[x]

LDR R2, R1, #0   ; R2 <-- grid[x]
ADD R2, R2, #2   ; R2 <-- grid[x] + 2

LDR R0, R5, #-10 ; Load the value of x
ADD R0, R0, #1   ; R0 <-- x + 1

ADD R1, R5, #-9  ; Base address of grid
ADD R1, R0, R1   ; Calculate address of grid[x+1]
STR R2, R1, #0   ; grid[x+1] = grid[x] + 2;
```



Strings in C

- Strings in C are simply arrays of chars and declared in the same format:

```
char my_name[10];
```

- And can also be initialized like other arrays:

```
char my_name[10] = “is Ivan”
```

Note “ vs. ‘

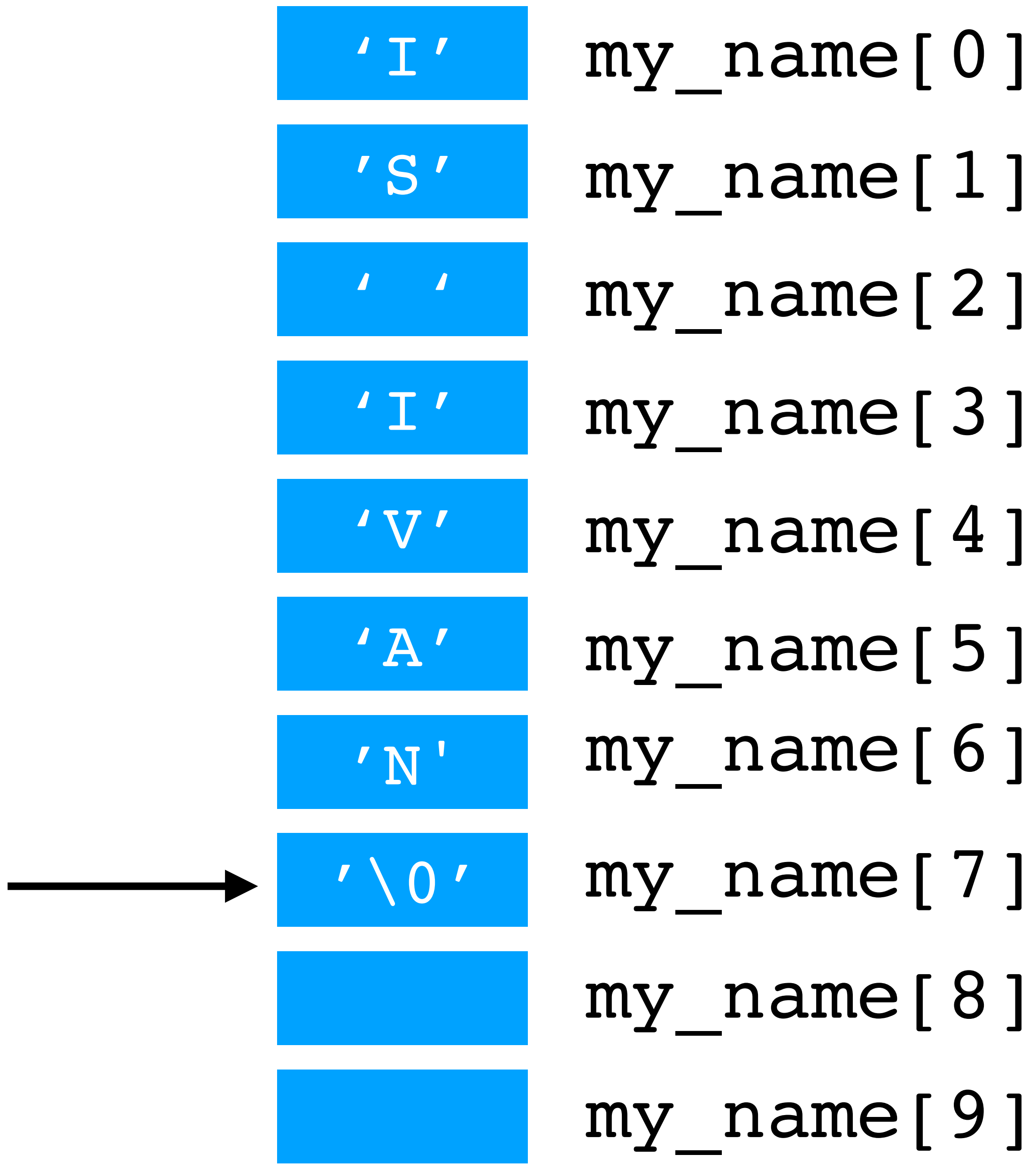
Did not use all 10 characters - some are unused

Strings in C

- To use strings with `printf` use the format specifier `%s`:

```
printf(“My name %s”, my_name);
```

- How does C know not to print garbage from the unused memory locations?
  - *Null-termination* for strings.



‘I’	my_name[0]
’S’	my_name[1]
‘ ‘	my_name[2]
‘I’	my_name[3]
‘V’	my_name[4]
‘A’	my_name[5]
’N'	my_name[6]
’\0’	my_name[7]
	my_name[8]
	my_name[9]

Strings in C

- Thus, the *length* of a string need not be the same as the size of the memory allocated to its identifier.
    - Food for thought: Write a function to determine the *length* of a string.
- **Note**: To replace the space in `my_name[2]` with an underscore do:

```
my_name[2] = ‘_’;
```

Single quote

‘I’	my_name[0]
’S’	my_name[1]
‘ ‘	my_name[2]
‘I’	my_name[3]
‘V’	my_name[4]
‘A’	my_name[5]
’N'	my_name[6]
’\0’	my_name[7]
	my_name[8]
	my_name[9]

Accepting keyboard input

- So far we used `scanf` to accept keyboard input.

- Run code on right with input “ECE 220” typed in from the console.

- What happened?

```
#include <stdio.h>

int main(void){
  char mystr[10];
  char mychar;
  printf("Enter a string:\t");
  scanf("%s", mystr);
  printf("\nYou entered: %s", mystr);
  printf("\nEnter a character:\t");
  scanf("%c", &mychar);
  printf("\nYou entered: %c\n", mychar);
  return 0;
}
```

Accepting keyboard input

- We can avoid that using the `fgets` function.

- Is that the only way to fix the issue?

  - **Answer**: No. Could use regexes:

```
scanf("%10[0-9a-zA-Z ]", mystr);
```

```
#include <stdio.h>

int main(void){
  char mystr[10];
  char mychar;
  printf("Enter a string:\t");
  fgets(mystr, 10, stdin);
  printf("\nYou entered: %s", mystr);
  printf("\nEnter a character:\t");
  scanf("%c", &mychar);
  printf("\nYou entered: %c\n", mychar);
  return 0;
}
```

Syntax: `fgets(charbuf, buf_size, source)`

Parsing string inputs

- Often we want to parse user input in a certain way.
- For example if the user enters: `217-333-2300` we may want to store it as three integer variables: `area_code, prefix, pnum`.
- We use the `sscanf` function.

```
sscanf(char_buffer, format_string, variables…)
```

Example

- Write a C program that will parse user input of a sequence of digits in the format `xxx-xxx-xxxx` as 10 digit phone number. In other words into an area code, prefix and a local identifying number. Print each out to the console separately.

Example

Why 13?

What if input did not fit given format?

Need to check return or exit codes.

```c
#include <stdio.h>

int main(void){
  int area_code, prefix, pnum;
  char mystr[13];

  printf("Enter a 10-digit phone number.\n");
  printf("Format: xxx-xxx-xxxx\n");

  fgets(mystr, 13, stdin);
  sscanf(mystr, "%d-%d-%d", &area_code, &prefix, &pnum);

  printf("\nArea code: %d", area_code);
  printf("\nPrefix: %d", prefix);
  printf("\nLocal: %d", pnum);

  return 0;
}
```

`sscanf` will return number of values correctly parsed

Entering multiple strings?

```c
#include <stdio.h>

int main(void){
char arr[][6] = {"cat",
                 "horse",
                 "golf"};


int i;
printf("Elements are:\n");
for (i = 0; i < 3; i++)
   printf("%s\n", arr[i]);
}

arr[1] = "cat";
}
```

→ Compiler error! Cannot assign to array.

Memory allocation

arr[0]	c	a	t	\0		
arr[1]	h	o	r	s	e	\0
arr[2]	g	o	l	f	\0	

To modify character arrays after declaration use `strcpy` from `<string.h>` (which also houses a `strlen` function just FYI).

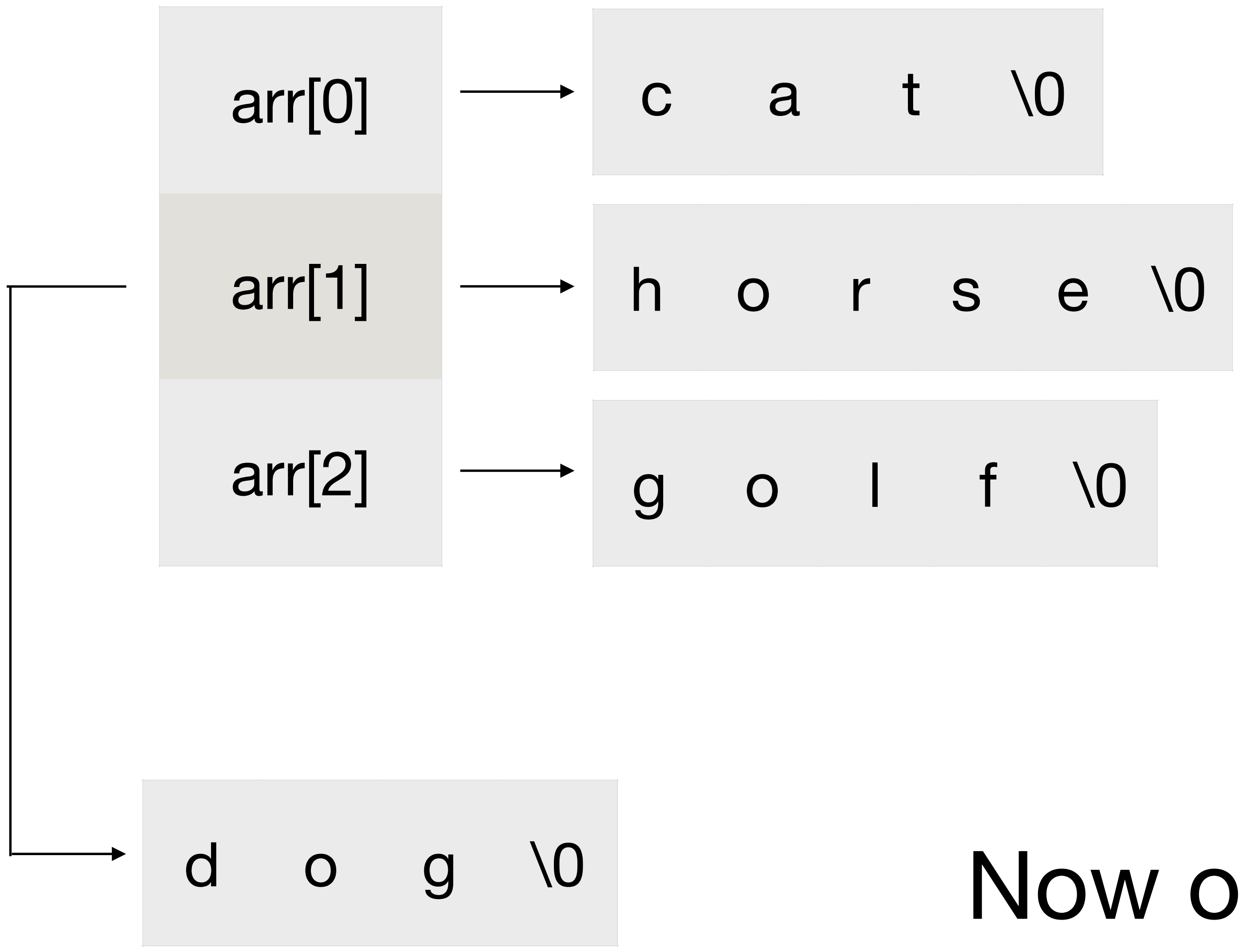
Strings - subtle points

- Common point of confusion responsible for much frustration is conflating *character arrays* with *string literals*.
- You will often see the code from the previous slide written this way.
- But they are **NOT** equivalent.

```c
#include <stdio.h>

int main(void){
char *arr[3] = {"cat",
                "horse",
                "golf"};
int i;

printf("Elements are:\n");
for (i = 0; i < 3; i++)
   printf("%s\n", arr[i]);

arr[1] = "dog";
}
```

Strings - subtle points

Memory allocation



arr[0] → c a t \0

arr[1] → h o r s e \0

arr[2] → g o l f \0

d o g \0

```
#include <stdio.h>

int main(void){
char *arr[3] = {"cat",
                "horse",
                "golf"};

printf("Elements are:\n");
for (int i = 0; i < 3; i++)
   printf("%s\n", arr[i]);

arr[1] = "dog";
}
```

Now okay!

Strings - subtle points

```
#include <stdio.h>

int main(void){
char arr[3][6] = {"cat",
                  "horse",
                  "golf"};

printf("Elements are:\n");
for (int i = 0; i < 3; i++)
   printf("%s\n", arr[i]);
}

arr[0][1] = 'o';
}
```

These are allocated on the stack and so `arr` remains modifiable.

Okay.

```
#include <stdio.h>

int main(void){
char *arr[3] = {"cat",
                "horse",
                "golf"};

printf("Elements are:\n");
for (int i = 0; i < 3; i++)
   printf("%s\n", arr[i]);

arr[0][1] = 'o';
}
```

These are stored as *string literals*, often in read-only memory. `arr` just points to them.

Undefined behavior!