# Function Arguments

pass by value means when we call a function, we pass a copy of the value of arguments to the function, not the actual "address" of the arguments. The default behavior/sematic of the C language is pass by value. later on we will learn about "pass by address"

- Recall we saw that functions pass arguments “by value” – a copy is made and assigned to the parameter variable local to the callee
- Changes made to local variables and parameters in callee are not visible to caller

SIDE NOTE: Parameter is a variable in a function definition.
Argument is the value that we pass into the function.

```
int abs (int x) { … }

int main() {
   int a = 5;
   abs(a);
   ….
}
```

parameter

argument

# Passing Arrays to Functions

- Extra care is required when passing arrays to functions, or returning them from functions
- ✱ Arrays are *not* passed by value ✱
  - Copying could be excessive
- Instead, passing an array amounts to passing a *pointer to its first element*
  - A pointer is a variable which holds an address (we'll discuss these more next week)
- ✱ Callee *can modify* the array ✱

# Passing Arrays to Functions: Example 1

```c
// function_arrpass_eg1.c:
#include <stdio.h>
// No need to specify a length for array parameter itself. The same amount of info is passed
// whether array is size 6 or size 600 -- an 8-byte address.
// So we feed in 2nd parameter to tell function the array's length.
int total(int n[], int len) {

    int tot = 0;
    for(int i = 0; i < len; i++) {
        tot += n[i];
    }
    return tot;
}

int main() {
    int evens[6] = {0, 2, 4, 6, 8, 10};
    printf("%d\n", total(evens, 6));
    return 0;
}
```

length of the array

no need to specify the size of the array

```
$ gcc function_arrpass_eg1.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
30
```

# Passing Arrays to Functions: Example 2

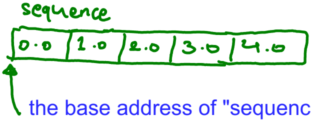
```c
// function_arrpass_eg2.c:
#include <stdio.h>

// Multiply each array element by factor, modifying the array
void scale_array(float arr[], int len, float factor) {
    for(int i = 0; i < len; i++) {
        arr[i] *= factor;
    }
}

int main() {
    float sequence[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
    scale_array(sequence, 5, 2.0);
    for(int i = 0; i < 5; i++) {
        printf("sequence[%d] = %.1f\n", i, sequence[i]);
    }
    return 0;
}
```

modifies arr
this is going to modify the array "sequence" down in the main function

we are passing the base address (i.e., the memory address of the first element of the array "sequence" into the "scale_array" function



the base address of "sequence" will be passed into "scale_array"

# Passing Arrays to Functions: Example 2 Output

```
$ gcc function_arrpass_eg2.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
sequence[0] = 0.0
sequence[1] = 2.0
sequence[2] = 4.0
sequence[3] = 6.0
sequence[4] = 8.0
```

← modified

# Clicker quiz!

What is the output of the following program?

```
#include <stdio.h>
void myFunc(int x, int a[]) {
  x += 3;
  a[0] = 42;
}
int main(void) {
  int y = 4;
  int r[] = { 1, 2, 3 };
  myFunc(y, r);
  printf("y=%d, r[0]=%d\n",
         y, r[0]);
  return 0;
}
```

A. `y=4, r[0]=1`
B. `y=7, r[0]=1`
C. `y=4, r[0]=42`
D. `y=7, r[0]=42`
E. None of the above

# Returning an Array from a Function

- When returning an array, the return type is the array's base type with * added
  - It's technically a *pointer*
- *However*, we don't yet know the correct way to return arrays

# Returning an Array from a Function: Bad Example

this is how we declare the return type for a function that is supposed to return an array (e.g., in this case an array of double numbers)

```c
// function_arrpass_eg3.c:
#include <stdio.h>

double* scale_array(double arr[], double factor) {
    double scaled_arr[5];  //suppose we just know array's size is 5
    for(int i = 0; i < 5; i++) {
        scaled_arr[i] = arr[i] * factor;
    }
    return scaled_arr;
}
int main() {
    double array[] = {1.0, 4.5, 8.4, 2.5, 8.3};
    double* scaled_array = scale_array(array, 2.0);
    printf("%0.2f %0.2f\n", scaled_array[0], scaled_array[4]);
    return 0;
}
```

this does not work!

```
$ gcc function_arrpass_eg3.c -std=c99 -pedantic -Wall
-Wextra function_arrpass_eg3.c: In function ‘scale_array’:
function_arrpass_eg3.c:8:12: warning: function returns
address of local variable [-Wreturn-local-addr]
return scaled_arr;                ^~~~~~~~~~
```

- error message says: *function returns address of local variable*

# For Now, We Can Pass In An Empty Array To Fill

- Instead of returning a local array, caller should pass in “destination” array to modify, as we did here:

```
void scale_array(float arr[], int len, float factor) {
    for(int i = 0; i < len; i++) {
        arr[i] *= factor;
    }
}
```

# Array Parameters That Shouldn't Be Modified

- When an array parameter *should not* be modified by the function, add const before the type
- Compiler gives an error if you try to modify a const variable

Use when you want to protect the passed array from any possible modifications (e.g., in this case, the "scale_array" function is not allowed to make any changes to the passed array "sequence"; note that both "arr" and "sequence" essentially are the same array)

```
// function_arrpass_eg4.c:
#include <stdio.h>

void scale_array(const float arr[], int len, float factor) {
//               ^^^^^
    for(int i = 0; i < len; i++) {
        arr[i] *= factor;
    }
}

int main() {
    float sequence[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
    scale_array(sequence, 5, 2.0);
    return 0;
}
```

# Array Parameters That Shouldn’t Be Modified

```
$ gcc function_arrpass_eg4.c -std=c99 -pedantic -Wall -Wextra
function_arrpass_eg4.c: In function ‘scale_array’:
function_arrpass_eg4.c:6:16: error: assignment of read-only location ‘*
         arr[i] *= factor;
                ^~
```

- arr is “read-only” because of const in its type
    - Similar to final in Java
- We’ll see an example of a const array parameter in today’s exercise