

# Week 5 Lecture 1

## Arrays

# Arrays

- Arrays are a sequence of identical data items.
  - e.g. `int numbers[80];`
- They are accessed through an integer index.
  - e.g. `number[5] = 35;`
- String are an example of arrays.

# Array Syntax

- Declaration:  $\langle \text{type} \rangle \langle \text{name} \rangle [\langle \text{int} \rangle]$ 
  - $\langle \text{type} \rangle$  is the kind of identical things stored
  - $\langle \text{name} \rangle$  is the name of the things
  - $\langle \text{int} \rangle$  is a constant int expression
- Access:  $\langle \text{name} \rangle [\langle \text{int expr} \rangle]$ 
  - $\langle \text{name} \rangle$  is the name of the array
  - $\langle \text{int expr} \rangle$  is any int expression (e.g. constant, variable, function ...)
- $[]$  is an operator.

# Arrays are pointers

- The name of the array is a pointer to the beginning of the array.
- E.g., `float a[3]`: three floating point numbers
  - `*a` is the same as `a[0]`
  - `*(a+1)` is the same as `a[1]`
- When passing an array parameter, you can use empty brackets to refer to the array
  - Parameter `float a[]` is the same as `float *a`

# Example: Define Sum

- Array parameter: float numbers[]
  - Need to know length of array: int length
- Add each element of the array to the total.
- Return total

```
float sum(float numbers[], int length)
{
    float total = 0;

    for (int i = 0; i < length; i++) {
        total = total + numbers[i];
    }
    return total;
}
```

# Call Sum

- Declare and initialize array
  - float nums[3] = {1.2, 2.3, 3.4}
- Call function
  - sum(nums, 3)

```
int main()
{
    float nums[3] = {1.2, 2.3, 3.4};

    printf("sum = %f\n", sum(nums, 3));
}
```

# Printing an Array

# Define print\_array

- Array to print
- Length of array to print
- Open brace
- First length-1 elements
- Last element with closing brace and newline.

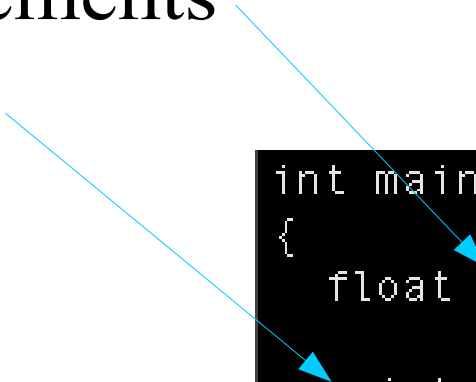
```
void print_array(float a[], int length)
{
    printf("{");
    for (int i = 0; i < length-1; i++) {
        printf("%f, ", a[i]);
    }
    printf("%f}\n", a[length-1]);
}
```



# Call print\_array

- Array with three elements
- Calling print\_array
- Output:

```
examples> gcc -std=c11 -o swap swap.c
examples> ./swap
{1.000000, 2.000000, 3.000000}
```



```
int main(void)
{
    float a[3] = {1.0, 2.0, 3.0};
    print_array(a, 3);
}
```

Swapping two numbers

# Define swap\_indices

- Temporary variable
- Save one element
- Copy value
- Replace saved value

```
void swap_indices(float a[], int index1, int index2)
{
    float temp;

    temp = a[index1];
    a[index1] = a[index2];
    a[index2] = temp;
}
```

# Call Swap\_indices

- Show array
- Swap second and third items
- Show array

```
int main(void)
{
    float a[3] = {1.0, 2.0, 3.0};

    print_array(a, 3);
    swap_indices(a, 1, 2);
    print_array(a, 3);
}
```

- Output:

```
examples> gcc -std=c11 -o swap swap.c
examples> ./swap
{1.000000, 2.000000, 3.000000}
{1.000000, 3.000000, 2.000000}
```

# Need a temporary variable

- When you move one variable to the other, the value in the receiving variable is lost.
- You need to save the value of the variable you overwrite before you move.
- You then move the temporarily saved value to the variable from which you moved.
- Local variables are good temporary variables because they vanish when the function ends.

# Define swap\_values

- Pass pointers
  - Parameters go away after the function returns
- Swap values pointed to

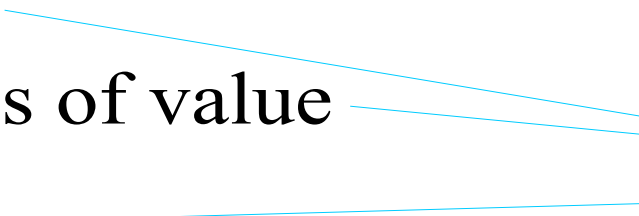
```
void swap_values(float *a, float *b)
{
    float temp;

    temp = *a;
    *a = *b;
    *b = temp;
}
```

# Call Swap\_values

- Show array
- Pass address of value
- Show array

```
int main(void)
{
    float a[3] = {1.0, 2.0, 3.0};
    print_array(a, 3);
    swap_values(&a[1], &a[2]);
    print_array(a, 3);
}
```



- Output:

```
examples> gcc -std=c11 -o swap swap.c
examples> ./swap
{1.000000, 2.000000, 3.000000}
{1.000000, 3.000000, 2.000000}
```

# Call Swap\_values (alternative)

- Pass address of value
  - Variable a is an address
  - Adding 1 move the address forward one space

```
int main(void)
{
    float a[3] = {1.0, 2.0, 3.0};

    print_array(a, 3);
    swap_value(a+1, a+2);
    print_array(a, 3);
}
```

- Output:

```
examples> gcc -std=c11 -o swap swap.c
examples> ./swap
{1.000000, 2.000000, 3.000000}
{1.000000, 3.000000, 2.000000}
```



# Sorting

- Conceptually simple; computationally hard.
- An array,  $a_0 \dots a_n$ , is sorted if for each  $a_i$ ,  $a_i \leq a_{i+1}$
- It is computationally difficult because you have to look at all pairs.
  - Naive implementations are usually  $O(n^2)$ .

# Implementing Bubble Sort

```
void bubble_sort(float a[], int length)
{
    for (int pass = 1; pass < length - 1; pass++) {
        for (int i = 0; i < (length - pass - 1); i++) {
            if (a[i] > a[i + 1]) {
                swap(&(a[i]), &(a[i+1]));
            }
        }
    }
}
```

# Bubble Sort is $O(n^2)$

- For each pass through the array
  - One needs to pass through most of the array.
- $N$  passes through the array for  $n$  comparisons  $\rightarrow O(n^2)$

# Median

# Median

- The median is the number where half of the data values are greater than the number and half are lower.
  - Given a sorted array, the median is just the value in the middle of the array.
  - If the array is even, it is the average of the two middle values