

# Computer Science & Engineering 150A

## Problem Solving Using Computers – Laboratory

### Lecture 07 – Arrays

Derrick Stolee

Spring 2009

- Simple data types use a single memory cell to store a variable.
- Sometimes, we want to *group* information together.
- A program that processes exam scores for a class could:
  - 1 Declare 54 variables:  
`double anthony, anthony, anthony, greg, greg, etc, or`
  - 2 Store all of the scores in memory together, and access them as a group.

# Declaring and Referencing Arrays

CSCE150A

Definition

Using arrays

- An array is a single variable that references several data elements in a row.
- To define, declare the *name of the array* and the *number of cells* associated with it.

```
double x[8];
```

- Creates 8 memory cells of type `double` with the name `x`;
- These memory cells will be adjacent to each other in memory.

- Access individual data elements with [index] brackets.
- By specifying the *array name* and identifying the element desired, we can access a particular value.
  - $x[0]$  (read as  $x$  sub zero) references the 0th, element of the array  $x$ ;  $x[1]$  is the next element in the array, followed by  $x[2]$ , ...,  $x[n-1]$ .
- In other words, the integer enclosed in brackets is the array subscript and its value must be in the range from zero to one less than the number of memory cells in the array.

- We can define two arrays and have the  $i$ th element in each array correspond to the same thing, such as the following:

```
int id[50];  
double gpa[50];
```

- $id[i]$  and  $gpa[i]$  correspond to the  $i^{th}$  student.
- You can declare multiple arrays along with regular variables.

```
double cactus[5], needle, pins[7];
```

- We can initialize a simple variable when we declare it:

```
int sum = 0;
```

- Same with arrays:

```
int array[SIZE];  
for(i=0; i < SIZE; i++)  
{  
    array[i] = 0;  
}
```

- We can also initialize an array in its declaration;

```
int prime_lt_100[] = {  
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37,  
    41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,  
    89, 97 };
```

- Subscripts denote the *number of places away from the start*.
- Any expression of type `int` can be used as an array subscript.
  - For Example: `array[2*i+1]`
  - To create a valid reference, the value of this subscript must lie between 0 and one less than the declared size of the array.

# Using for Loops for Sequential Access

CSCE150A

Definition

Using arrays

- Usually, we iterate through element in order from 0..SIZE-1.
- Use a for loop with index variable, starting at 0 and bounded by SIZE.

```
for(i=0; i < SIZE; i++) {  
    printf("%d ",array[i]);  
}
```

- Using the loop counter as an array index (subscript) gives access to each array element in turn.

- To use `scanf` or `printf`.

```
scanf("%d", &x[i]); // or x+i >:-[  
printf("%d\n", x[i]);
```

- Functions can also have entire arrays as arguments.
- The array name without subscripts is the *address* of the first position.

```
int array[10], *pointer;
```

- Set pointer to initial array element (2 forms):

```
pointer = &z[0];
```

```
pointer = z;
```

- Also, `void func(int array[ ]) ≡ void func(int *array)`

```
#include <stdio.h>
void bar(int[], int);
int main(void) {
    int foo[] = {1,2,3,4,5,6,7,8,9,10}, i;
    bar(foo,10);
    for (i = 0; i < 10; i++) { printf("%d\n", foo[i]);
    return 0;
}
void bar(int temp[], int size) {
    int i;
    for (i = 0; i < size; i++) { temp[i] *= temp[i]; }
}
```

# Use of \* in Formal Parameter List

CSCE150A

Definition

Using arrays

- In the declaration for the function bar, we can use either
  - `int list[]`
  - `int *list`
- The first tells us that the actual argument is an array (but passing an address).
- The second declaration would be equally valid for an integer array parameter.

- NOTE: `int *list` does not indicate how many elements are in `list`.
  - The size is declared “out of scope!”
- Unlike passing values, C sends address of arrays.
  - Thus, there is no need for a `&` in the function call.
  - This is *only* for arrays!!
- Without a size requirement, we can pass arrays of *any size!*

In C, it is not legal for a function's return type to be an array; therefore, we need to use pointers!

Arrays are a big topic. More next week!

- Defining: `type name[SIZE];`
- Accessing: `name[index]`
- Iterating:

```
for ( i = 0; i < SIZE; i++ ) {  
    name[i] = 0;  
}
```

It is important to access elements in an array.

- Q: Is 5 one of these values?
- Q: Which position of the array `id` has value 456123?
- Q: What is the GPA of student 456123?
- Q: What is the fifth-best score in the lab?

## Searching Algorithm

1. Assume target has not been found
2. Start with the initial array element
3. Repeat while the target is not found and there are more elements in the array
4.     if the current element matches target
5.         set flag true
6.         remember array index
7.     else
8.         advance to next array element
9. If flag set true
10.     return the array index
11. else
12.     return -1 to indicate not found

```
flag = 0; i = 0;
while ( !flag && (arr[i] != '\0') ) {
    if ( arr[i] == target ) {
        flag = 1;
        index = i;
    } else {
        i++;
    }
}
if ( flag ) {
    return index;
} else {
    return -1;
}
```

# Sorting an Array

CSCE150A

Definition

Using arrays

There are several sorting algorithms. You'll learn more in later classes.

- Selection Sort – Find best/worst so far, add to new list.
- Bubble Sort – Swap values that are in wrong relative position.

Not enough time in lab! Covered in lecture.

We have so far required our arrays to have a fixed size *at compile time!*

```
int array[50]; double stuff[SIZE];
```

However, we can create an array of a *variable size* using dynamic memory allocation!

The function `malloc` takes a number of *bytes* as input and returns an *address* of allocated memory of that size.

Use `sizeof` function to find the number of bytes your type will take!

- 1 `sizeof(int) = 4`
- 2 `sizeof(double) = 8`
- 3 `sizeof(char) = 1`

**Example:**

```
int n, i;
scanf("%d", &n);
int* dynArray = malloc(n * sizeof(int));
for ( i = 0; i < n; i++ )
{
    dynArray[i] = 0;
}
```

We still need to track the size of the array! Even more important, now that it is a variable!

Whenever we use `malloc`, we take memory into our own control. This memory is “locked” in a sense, until we “let it go.”

The function `free` takes a pointer and “deallocates” the memory it points to. This will essentially delete the array we allocated earlier.

```
int* dynArray = malloc(n * sizeof(int));  
free(dynArray);
```

# Common Memory Errors

CSCE150A

Definition

Using arrays

- If you go out of the bounds of your array, you will get “Bus Error” or “Segmentation Fault”
- If you try to access an array that has not been initialized (`malloc`) or has already be deallocated (`free`), you will get a “Segmentation Fault”
- Be VERY CAREFUL when you work with dynamic memory.

- Initializing Dynamic Arrays:

```
type* name = malloc(n * sizeof(type));
```

- Accessing: `name[index]`

- Iterating:

```
for ( i = 0; i < n; i++ ) {  
    name[i] = 0;  
}
```

- Deallocating: `free(name);`