

## Course Notes Set 5:

### COMP1200

#### Introduction to Computing for Engineers and Scientists C Programming

## Arrays

Computer Science and Software Engineering  
Auburn University



Let's imagine what a program using technique #2  
would look like:

```
#include <stdio.h>
int main(void)
{
    // One variable for each grade
    double g0, g1, g2, g3, ..., g98, g99, avg;

    // Read in grades
    printf("Enter 100 grades: ");
    scanf("%lf %lf %lf...%lf", &g0, &g1, ..., &g99);

    // Average grades
    avg = (g0 + g1 + g2 + g3 + ... + g99) / 100.0;
    printf("Average is: %f", avg);
    return 0;
}
```



## Arrays: Introduction

What if we had a problem where we needed to average all the grades in a COMP1200 class? How would we store all the grades entered by the user?

Using knowledge we have now we could approach this in a couple ways:

1. We could put all the grades in a data file. Then, open and read the data file each time we want to work with the grades.  
*Advantage:* Code is relatively simple to write  
*Disadvantage:* File I/O is very slow
2. We could declare a variable for each student in the class. We'd then just write an input statement to load each variable.  
*Advantage:* All the variables are in memory, so it's fast  
*Disadvantage:* That's a LOT of variables in a class with 100 students!



## Arrays

Now what happens when we add 20 students to the class? We have to add all those new variables! This is not good.

92	72	87	61	89	55	99	...	86	76
g0	g1	g2	g3	g4	g5	g6	...	g98	g99



## Arrays

A program with this many variables can be long and frustrating to use. What if there was a shorthand notation for the many related variables. It would be nice if we could write:

```
/* Read in grades */  
printf("Enter 100 grades: ");  
for (i=0; i<100; i++)  
    scanf("%lf",&gi);
```

This will not work because C would treat `gi` as a variable and reassign 100 grades to it.

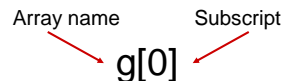
## Arrays

This is where a C data structure called an **array** comes in to help us out.

Think of it as asking the C compiler for “100 double variables.” You can visualize the compiler giving you 100 variables arranged in a row:

92	72	87	61	89	55	99	....	86	76
g0	g1	g2	g3	g4	g5	g6	....	g98	g99
g[0]	g[1]	g[2]	g[3]	g[4]	g[5]	g[6]	....	g[98]	g[99]

## Array element



We call each individual variable in an array an **element** of the array. We give the collection of elements a name, the **array name**. An array can be given any valid variable name.

To get at the individual elements of an array, we number them **starting at zero, incrementing by one**. These numbers are called the **subscripts**.

So to get at the **first element**, we ask for element `g[0]` (remember, we start at zero). To get the **34th element**, we ask for `g[33]`. To get the **100th element**, we ask for `g[99]`.

## Arrays: Defining/Initializing

Arrays are declared just like any other variable in the declaration section of a program:

```
int main(void)
{
    // an array of 3 integers
    int t[3];

    // 2 doubles & an array of 5 doubles
    double a,b,c[5];
    ...
}
```

## Arrays: Defining/Initializing

```
int t[3];
double a,b,c[5];
```

If we were to peek at the memory these declarations allocate, it would look like this:

t[0]	t[1]	t[2]	a	b	c[0]	c[1]	c[2]	c[3]	c[4]

## Arrays

Remember that with normal variables we could initialize them as we declared them? We can do that with arrays also:

```
int main(void)
{
    /* an array of 3 integers */
    int t[3] = {3, 2, 1};

    /* 2 doubles & array of 5 doubles */
    double a=10,b=20,c[5] = {0};

    ...
}
```

## Arrays: Defining/Initializing

```
int t[3] = {3, 2, 1};
```

```
/* two doubles and array of 5 doubles */
```

```
double a=10,b=20,c[5] = {0};
```

Note: If there are fewer initializing values than memory locations allocated, the remainder are initialized to zero.  
Note the array c[].

After declaring/initializing, the memory looks like:

3	2	1	10	20	0	0	0	0	0
t[0]	t[1]	t[2]	a	b	c[0]	c[1]	c[2]	c[3]	c[4]

## Arrays: Defining/Initializing

Another way to declare/initialize an array is to not specify a size:

```
/* An array of 5 integers */
```

```
int s[] = {10, 20, 30, 40, 50};
```

The compiler counts the number of initializing values and allocates that many variables.

## Arrays: Defining/Initializing

We can also initialize arrays from within our program, this is usually done with a loop:

```
...
/* Declare variables */
int k;
double g[10];
...
/* Init the array g */
for (k=0; k<10; k++)
{
    g[k] = k*0.5;
}
```



## Arrays: Defining/Initializing

We must be VERY careful not to go past the end of the array while initializing it.

You will notice that this loop terminates with **k<10**, NOT **k<=10**.

This is because **g[10]** is not a valid array element ... remember, **we start with g[0]**.

If we didn't observe this rule, the program would overwrite information outside the array and cause hard to find errors.

NOTE: C does NOT give you an error if your array subscript is out of bounds.



## Arrays: Defining/Initializing

```
for (k=0; k<10; k++)
{
    g[k] = k*0.5;
}
```

This program segment takes each element of an array and sets it to the current value of  $\frac{1}{2} k$ . After running, memory looks like:

0	0.5	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5
g[0]	g[1]	g[2]	g[3]	g[4]	g[5]	g[6]	g[7]	g[8]	g[9]



## Arrays: Defining/Initializing

```
int t[3] = {3, 2, 1};
double a=10,b=20,c[5] = {0};
```

```
t[3] = 33;
```

3	2	1	33	20	0	0	0	0	0
t[0]	t[1]	t[2]	a	b	c[0]	c[1]	c[2]	c[3]	c[4]



## Arrays: Using In Programs

Using arrays in a program is no different than using normal variables, you just have to **remember the array subscript**.

This is how the compiler will know which array element you want to work with.

Let's take an example program that averages the elements of an array:

```
#include <stdio.h>
#define N 5
int main(void)
{
    /* Declare Variables */
    int i;
    double y[N] = {76.0, 35.4, 99.2, 81.0, 65.5};
    double average, sum=0;
    /* Total each array element */
    for (i=0; i<N; i++)
    {
        sum += y[i];
    }
    /* Compute the average */
    average = sum / N;
    printf("Average = %f\n", average);
    return 0;
}
```

76.0	35.4	99.2	81.0	65.5
------	------	------	------	------

y[0]   y[1]   y[2]   y[3]   y[4]

## Arrays: Using In Programs

Let's suppose we wanted to print all the values in the array out:

```
for (i=0; i<N; i++)
{
    printf("%f \n", y[i]);
}
```

This prints each element of the array, one per line.

76.0  
35.4  
99.2  
81.0  
65.5

## Arrays: Using In Programs

Similarly, if we wanted to print the numbers on one line:

```
for (i=0; i<N; i++)
{
    printf("%f ", y[i]);
}

/* Add a next line after all numbers */
printf("\n");
```

76.0 35.4 99.2 81.0 65.5

## Arrays: Using In Programs

Similarly, if we wanted to print them to a data file (assume that myfile has been declared and opened):

```
for (i=0; i<N; i++)
{
    fprintf(myfile,"%f\n",y[i]);
}
```

## Arrays: Passing A Single Element To Functions

```
int func1( int a, int b )
{
    ... function body
}
```

Now let's suppose in my main program I want to use `func1`, and I want it to operate on a couple of my array elements. Here's how I would call the function:

```
int result, x[30], i=4, j=8;
result = func1( x[4], x[8] );
```

or

```
result = func1( x[i], x[j] );
```

So here's the rule: When the **formal** arguments are non-array values, the **actual** arguments must be subscripted array elements.

## Arrays: Passing To Functions

We have to consider a couple cases when dealing with arrays and functions.

Passing A Single Element

Passing An Entire Array

## Arrays: Passing An Entire Array To Functions

When passing an entire array to a function, we usually have to supply two parameters: the name of the array being passed, and the number of values that the array holds.

Consider the following declaration:

```
int g[10] = {10, 20, 30, 40};
```

Memory will look like this:

10	20	30	40	0	0	0	0	0	0
g[0]	g[1]	g[2]	g[3]	g[4]	g[5]	g[6]	g[7]	g[8]	g[9]

## Arrays: Passing An Entire Array To Functions

Now only the **first 4 elements of this 10 element array** hold values that we care about. What we want to do is tell a function to only consider those values when it operates on the array.

Here's how we define the formal parameters for the function:

```
int sumFun(int x[ ], int n)
{
    ...
    for (j=0; j<n; j++)
    {
        sum+=x[j];
    }
    ...
}
```

## Arrays: Passing An Entire Array To Functions

To call the function,

```
int sumFun(int x[ ], int n)
```

we could do this:

```
int result,g[10]={10,20,30,40};
...
result = sumFun(g, 4); // NO subscript
```

```
#include <stdio.h>
#include <stdlib.h>
double max(double x[], int n);
#define FILENAME "lab.dat"
#define N 100
int main(void)
{ int k=0, npts;
  double y[N];

  FILE *lab;
  lab = fopen(FILENAME,"r");
  while ((fscanf(lab,"%lf",&y[k])) == 1)
  { k++; }
  npts = k;

  printf("Maximum value: %f \n",max(y,npts));
  fclose(lab);
  return 0;
}
```

/\* This program  
reads values from  
a data file and  
determines the  
maximum value  
with a function. \*/

```
/* This function returns the maximum */
/* value in the array x with n elements. */
double max(double x[], int n)
{
    int k;
    double max_x;

    max_x = x[0];
    for (k=1; k<=n-1; k++)
    {
        if (x[k] > max_x)
            max_x = x[k];
    }
    return max_x;
}
```

## Arrays In Statistics

Analyzing data gathered from experiments is an important and common task in engineering. Modern equipment is usually capable of being hooked up to networks allowing the automatic collection of sensor data such as temperature, voltage, etc.

What we'll do is take a look at a few of the common statistical measures and see how we can implement them using arrays and functions.

## Arrays In Statistics: Minimum/Maximum

```
double max(double x[], int n)
{
    int k;
    double max_x;
    /* Determine maximum value in the array. */
    max_x = x[0];
    for (k=1; k<=n-1; k++)
    {
        if (x[k] > max_x)
            max_x = x[k];
    }
    return max_x;
}
```

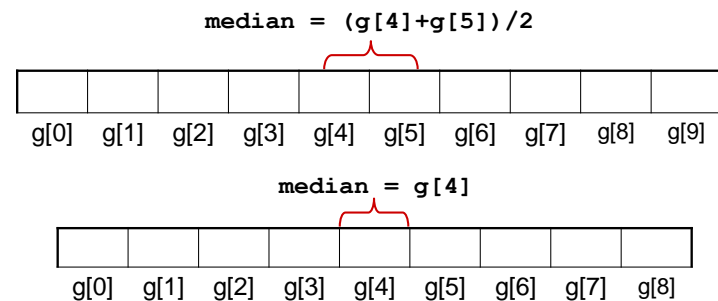
## Arrays In Statistics: Mean (Average)

To calculate the mean of a set of values, we add all the values together, and then divide by the number of values.

```
double mean(double x[], int n)
{
    int k;
    double sum=0;
    /* Determine mean values. */
    for (k=0; k<=n-1; k++)
    {
        sum += x[k];
    }
    return sum/n;
}
```

## Arrays In Statistics: Median

The median of a group of values is the “middle” value.





This function returns the median value in an array `x` with `n` elements.

```
double median(double x[], int n)
{
    int k;
    double median_x;
    /* Determine median value. */
    k = n/2;
    if (n%2 != 0)
        median_x = x[k];
    else
        median_x = (x[k-1] + x[k])/2;
    return median_x;
}
```

This function returns the variance of an array with `n` elements.

```
double mean(double x[], int n);
double variance(double x[], int n)
{
    int k;
    double sum=0, mu;
    /* Determine variance. */
    mu = mean(x,n);
    for (k=0; k<=n-1; k++)
    {
        sum += (x[k] - mu)*(x[k] - mu);
    }
    return sum/(n-1);
}
```

## Arrays In Statistics: Variance

Many (if not most) of the time, the average of a set of values doesn't give the "big picture" about the data.

For example, consider the following averages:

`average = (79.0 + 80.0 + 81.0)/3.0; = 80.0`

`average = (60.0 + 80.0 + 100.0)/3.0; = 80.0`

Even though the averages are the same, the data varies very little in the first example, and varies wildly in the second. So one useful measure is variance.

Variance is the sum of the squared differences between a value and the average of the values.

## Arrays In Statistics: Standard Deviation

Since variance gives squared values, we can take the square root of this to give us a better feel for how much the data varied, this is called standard deviation:

```
double std_dev(double x[], int n)
{
    // Return standard deviation.
    return sqrt(variance(x,n));
}
```

## Sorting: Selection Sort

There are many times that we will need to sort information in an array into numerical order. There have been entire books written on this subject. In order to illustrate how a simple sorting algorithm can be implemented, we'll take a look at the selection sort.

The selection sort works on a very simple principle:

You search the array for the minimum value, and put that in the first spot. Then you move to the second spot and search what's left of the array, then move to the third spot, fourth, etc.

## Sorting: Selection Sort

We can visualize the algorithm working like this:

Pass		x[0]	x[1]	x[2]	x[3]	x[4]	x[5]
0	0-5	5	1	3	12	8	9
1	1-5	1	5	3	12	8	9
2	2-5	1	3	5	12	8	9
3	3-5	1	3	5	12	8	9
4	4-5	1	3	5	8	12	9
5		1	3	5	8	9	12

## Sorting: Selection Sort

Here's some pseudo code that describes the action:

```
for elements i in 1 through n-1 in array
    minimum = i
```

```
    for elements j in i through n in array
        if (array[j] < array[minimum])
            minimum = j
```

```
swap array[i], array[minimum]
```

This function sorts an array with n elements in ascending order.

```
void sort(double x[], int numEle)
{
    int pass, nextMin, j;
    double hold;
    for (pass=0; pass<numEle-1; pass++)
    { /* Exchange minimum with next array value. */
        nextMin = pass;
        for (j=pass+1; j<numEle; j++)
        {
            if (x[j] < x[nextMin])
                nextMin = j;
        }
        hold = x[nextMin];
        x[nextMin] = x[pass];
        x[pass] = hold;
    }
}
```

## Arrays: Searching

Consider the problem of having a company with several thousand employees.

You are asked the question: "Do you have an employee with the social security number xxx-xx-xxxx?"

Now let's suppose we have an array with all the employee SS#s in it.

Then we could do this:

```
found = FALSE;
for (i=0; i<n; i++)
{
    if (emp_ss[i] == search_ss)
    {
        found = TRUE;
        break;
    }
}
```

## Arrays: Searching

This little segment of code sets found to TRUE if the employee exists. But there's a problem with searching this way: if we had 10,000 employees, we'd have to search on average 5,000 elements to find a given employee. Not good.

You may remember a game called "high/low." In this game, someone thinks of a number between 1 - 100. You try to guess the number, and the person who knows it responds "high" or "low" to each of your guesses. Person who guesses it in the fewest tries wins the game.

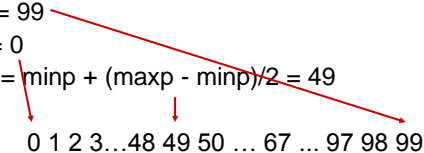
Let's do this, we'll sort the array of employee social security numbers, and then play "high/low" with them:

## Arrays: Searching

To make it simple, let's say that SS#'s in this company are only 4 digits. Someone says, does SS# 2300 work here? Now let's suppose that SS#2300 is in position emp\_ss[67] in an array of 100.

We start by taking the maximum position (maxp) and subtracting from it the minimum position (minp). We then divide that by two, and give that as our first guess:

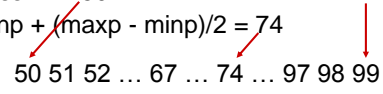
```
maxp = 99
minp = 0
guess = minp + (maxp - minp)/2 = 49
```



## Arrays: Searching

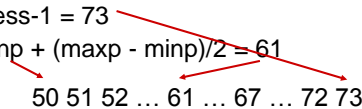
Now emp\_ss[49] is "low" ... so we move our minimum up to guess+1, and try again:

```
minp = guess+1 = 50
guess = minp + (maxp - minp)/2 = 74
```



Now emp\_ss[74] is "high" ... so we move our maximum down to guess-1, and try again:

```
maxp = guess-1 = 73
guess = minp + (maxp - minp)/2 = 61
```



## Arrays: Searching

Now emp\_ss[61] is “low” ... so we move our minimum up to guess+1, and try again:

minp = guess+1 = 62

guess = minp + (maxp - minp)/2 = 67

62 63 64 65 67 68 69 70 71 72 73

emp\_ss[67] is the correct number, so we have found it!

That's a lot better than searching the entire array.

But what if the number hadn't been in the array?

Eventually, minp would become > maxp, at that point, would know the search had failed.

```
int binsearch(int x[], int n, int target)
{
    int minp, maxp, guess;
    int found = 0; /* False */
    minp = 0;
    maxp = n-1;
    while (!found && (minp <= maxp))
    {
        guess = minp + (maxp - minp)/2;
        if (x[guess] == target)
            found = 1;
        else if (x[guess] < target)
            minp = guess+1;
        else
            maxp = guess-1;
    }
    return found;
}
```

## Multi-dimensional Arrays

A 1-D array is like a list.

A 2-D array is like a table or matrix with rows and columns.

A 3-D array is like a stack of tables (a cube).

A 4-D array is like a group of cubes.

Etc....

All are a collection of the **same type** of data, i.e., all float or all integer or all double, etc.

## Multi-dimensional Arrays

Some compilers can handle 7 dimensions and some can handle 12 or more.

Why would you want to use a multi-dimensional array?

Let's suppose...

We have 1200C in an array grades

- for 3 semesters ( 0, 1, 2 )
- each semester had 2 sections ( 0, 1 )
- each section has 50 students ( 0, 1, ..., 49 )
- each student has 8 assignment grades ( 0, 1, ..., 7 )

To access a grade, we need to know

the semester #, the section #, the student ID, and the assignment #.

## Multi-dimensional Arrays

We can declare the array as

```
double grades[3, 2, 50, 8];
```

The grades for student # 30, in section 2, in semester 2 can be printed by

```
for (i=0;i<8;i++)
{
    printf("%4.1f ",grades[1,1,29,i]);
}
```

**Note:** Because subscripts begin with 0,  
I subtracted 1 from each of the numbers.

## Multi-dimensional Arrays

If our example university had multiple campuses, we could add another dimension to represent the campus.

Another example could involve an inventory of boxes of cereal in a grocery store.

Categories, dimensions could be

- the name
- the brand
- the box size
- the store location
- the city
- the warehouse

If a huge array held this information, we could analyze the inventory by one or more categories.

## 2-dimensional Arrays

We will only address 2-D arrays in this course.

A 2-D array uses two subscripts versus one.

- The first subscript represents a row.
- The second subscript represents a column.

The table below describes an array "b" with 3 rows and 4 columns.

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

## Declare a 2-D Array

It is similar to declaring a 1-D array.

```
int b[3][4];
```

Declares

- The name of the array to be **b**.
- The type of the array elements to be **int**.
- The dimensions to be 2 (it has two pairs of [ ]).
- The number of rows to be 3.
- The number of columns to be 4.
- The number of elements of size to be  $3 \times 4 = 12$ .

## Initialize a 2-D Array

```
int b[3][4]={52,34,23,54,63,36,98,79,89,85,35,67};
```

Or

```
int b[3][4] = {52,34,23,54,  
               63,36,98,79,  
               89,85,35,67};
```

Or

```
int b[ ][4] = {52,34,23,54,  
              63,36,98,79,  
              89,85,35,67};
```

The 3<sup>rd</sup> example implicitly declare the number of rows as 3 by dividing 12 (the number of elements) by 4 (the number of columns).

## Loop to print 2-D Arrays

```
for (r=0;r<3;r++)  
{  
    for (c=0;c<4;c++)  
    {  
        printf("b[%1d][%1d]=%2d ",r,c,b[r][c]);  
    }  
    printf("\n");  
}  
b[0][0]=52 b[0][1]=34 b[0][2]=23 b[0][3]=54  
b[1][0]=63 b[1][1]=36 b[1][2]=98 b[1][3]=79  
b[2][0]=89 b[2][1]=85 b[2][2]=35 b[2][3]=67
```

## Loop to print 2-D Arrays

Notice what happens when the inner and outer loops are switched.

```
for (c=0;c<4;c++)  
{  
    for (r=0;r<3;r++)  
    {  
        printf("b[%1d][%1d]=%2d",r,c,b[r][c]);  
    }  
    printf("\n");  
}
```

```
b[0][0]=52 b[1][0]=63 b[2][0]=89  
b[0][1]=34 b[1][1]=36 b[2][1]=85  
b[0][2]=23 b[1][2]=98 b[2][2]=35  
b[0][3]=54 b[1][3]=79 b[2][3]=67
```

## Loop to print 2-D Arrays

```
#include <stdio.h>  
int main()  
{  
    int r=3, c=4;  
    int b[3][4] = {52,34,23,54,  
                   63,36,98,79,  
                   89,85,35,67};  
    for (r=0;r<3;r++)  
    {  
        for (c=0;c<4;c++)  
        {  
            printf("b[%1d][%1d]=%2d ",r,c,b[r][c]);  
        }  
        printf("\n");  
    }  
    printf("\n");  
    for (c=0;c<4;c++)  
    {  
        for (r=0;r<3;r++)  
        {  
            printf("b[%1d][%1d]=%2d ",r,c,b[r][c]);  
        }  
        printf("\n");  
    }  
    return 0;  
}
```

## Functions and 2-D Arrays

```
    print2D(r,c,b);  
    printf("\n");  
    print2D(c,r,b); } in the caller  
  
void print2D(int num1, int num2, int b[][num2])  
{ int n1, n2;  
  for (n1=0;n1<num1;n1++)  
  {  
    for (n2=0;n2<num2;n2++)  
    {  
      printf("%2d ",b[n1][n2]);  
    }  
    printf("\n");  
  }  
}
```

## Functions and 2-D Arrays

```
print2D(r,c,b);  
printf("\n");  
print2D(c,r,b);
```

```
52 34 23 54  
63 36 98 79  
89 85 35 67  
  
52 34 23  
54 63 36  
98 79 89  
85 35 67
```

## Functions and 2-D Arrays

```
#include <stdio.h>  
void print2D(int num1, int num2, int b[][num2]);  
int main()  
{  
    int r=3, c=4;  
    int b[3][4] = {52,34,23,54,  
                  63,36,98,79,  
                  89,85,35,67};  
  
    print2D(r,c,b);  
    printf("\n");  
    print2D(c,r,b);  
    return 0;  
}  
  
void print2D(int num1, int num2, int b[][num2])  
{  
    int n1, n2;  
    for (n1=0;n1<num1;n1++)  
    {  
        for (n2=0;n2<num2;n2++)  
        {  
            printf("%2d ",b[n1][n2]);  
        }  
        printf("\n");  
    }  
}
```