CS1010E Lecture 8
Arrays and Matrices

Joxan Jaffar

Block COM1, Room 3-11, +65 6516 7346
www.comp.nus.edu.sg/~joxan
cs1010e@comp.nus.edu.sg

Semester II, 2016/2017

Lecture Outline

- One-Dimensional Arrays

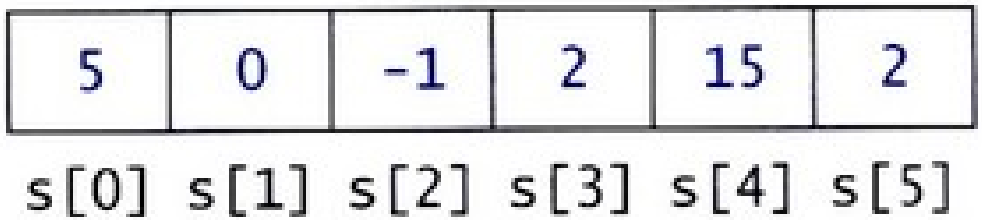
- Sorting Algorithms
  - Inserstion Sort
  - Bubble Sort
  - Quick Sort (non-examinable)

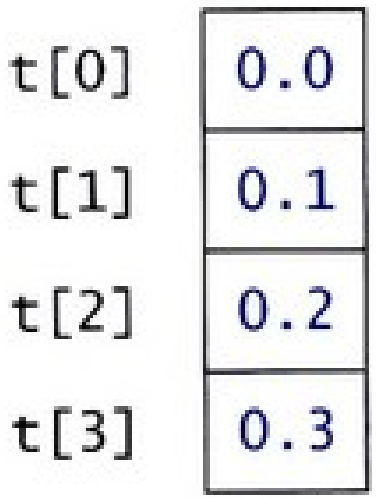
- Search Algorithms
  - Sequential search
  - Binary Search

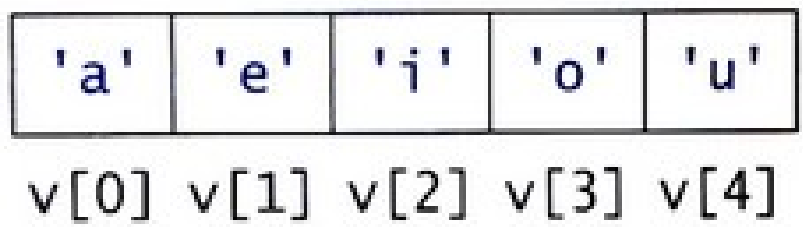
One-Dimensional Arrays

- Sometimes we want to work with a set of similar data values, but we do not want to give each value a separate name.
- Suppose that we have a set of 100 temperature measurements that we want to use to perform several computations.
- Obviously, we do not want to use 100 different variable names.
- We need a method for working with a group of values using a single identifier.

One-Dimensional Arrays







One-Dimensional Arrays

- We assign an identifier to an array, and then we distinguish between **elements** or values in the array using **subscripts**.
- In C, the subscripts always start with 0 and increment by 1.
- On the previous slide, the first value in the `s` array is referenced by `s[0]`.
- The third value in the `t` array is referenced by `t[2]`.
- The last value in the `v` array is referenced by `v[4]`.

One-Dimensional Arrays

- Arrays are convenient for storing and handling large amounts of data, so there is a tendency to use them in algorithms when they are not necessary.
- Arrays are more complicated to use than simple variables, and thus make programs longer and more difficult to debug.
- Therefore, use arrays only when it is necessary to have the complete set of data available in memory.

Definition and Initialization

- An array is defined using declaration statements.
- The identifier is followed by an integer expression in brackets that specifies the maximum number of elements in the array.
- Note that all elements in the array must be the same data type.
- The declaration statements for the three example arrays are as follows:
  - `int s[6];`
  - `char v[5];`
  - `double t[4];`

Definition and Initialization

- An array can be initialized with declaration statements or with program statements.

- To initialize the array with a declaration statement, the values are specified in a sequence that is separated by commas and enclosed in braces.

- To define and initialize the sample arrays `s`, `v`, and `t`, use the following statements:

  - `int s[6]={5,0,-1,2,15,2};`
  - `char v[5]={'a','e','i','o','u'};`
  - `double t[4]={0.0,0.1,0.2,0.3};`

Definition and Initialization

- If the initializing sequence is shorter than the size of the array, then the rest of the values are initialized to zero.

- Hence, if we want to define an array of 100 values, where each value is also initialized to zero, we would use the following statement:

  - `int s[100]={0};`

Definition and Initialization

- If an array is specified without a size, but with an initialization sequence, the size is defined to be equal to be equal to the number of values in the sequence, as follows:
  - `int s[]={5,0,-1,2,15,2};`
  - `char v[]={'a','e','i','o','u'};`
  - `double t[]={0.0,0.1,0.2,0.3};`
- The size of an array must be specified in the declaration statement by using either a constant within brackets or by an initialization sequence within braces.

Definition and Initialization

- Arrays can also be initialized with program statements.
- For example, suppose that we want to fill a `double` array `g` with the values $0.0, 0.5, 1.0, 1.5, \ldots, 10.0$.
- Because there are 21 values, listing the values on the declaration statement would be tedious.

Definition and Initialization

- Thus, we use the following statements to define and initialize this array:

```
/* Declare variables. */
int k;
double g[21];
...
/* Initialize the array g. */
for (k=0; k<=20; k++)
   g[k] = k*0.5;
```

- The condition in this `for` statement must specify a final subscript value of 20, and not 21, since the array elements are `g[0]` through `g[20]`.

Definition and Initialization

- It is a common mistake to specify a subscript that is one value more than the largest valid subscript.
- This error can be difficult to find because it accesses values outside the array.
- This can produce execution errors such as “segmentation fault” or “bus error”.
- Often, this error is not detected during program execution, but will cause unpredictable program results, since your program has modified a memory location outside of your array.

Definition and Initialization

- Arrays are often used to store information that is read from data files.

- Suppose that we have a data file named `sensor3.txt` from Lecture 5 that contains 10 time and motion measurements collected from a seismometer.

- To read these values into arrays named `time` and `motion`, we use the statements on the next slide.

Definition and Initialization

```
/* Declare variables. */
int k;
double time[10], motion[10];
FILE *sensor;
...

/* Open file and read data into arrays. */
sensor = fopen("sensor3.txt","r");

for (k=0; k<=9; k++)
    fscanf(sensor,"%lf %lf",&time[k],&motion[k]);

...
```

Computations and Output

- Computations with array elements are specified just like computations with simple variables, but a subscript must be used to specify an individual array element.

- The next program reads an array `y` of 100 floating-point values from a data file, determines the average value of the array, and then stores it in `y_ave`.

- Then, the program will determine the number of values in the array `y` that are greater than the average, count them and print the count.

Computations and Output

- If the purpose of this program had been to determine the average of the values in the data file, an array would not have been necessary.

- The loop to read values could read each value into the same variable and add its value to a sum before the next value is read.

- However, because we needed to compare each value to the average in order to count the number of values greater than the average, an array was needed to access each value again.

Computations and Output

```
/*-------------------------------------------------------*/
/*  Program chapter5_1                                   */
/*                                                       */
/*  This program reads 100 values from a data file and   */
/*  determines the number of values greater than the average.  */

#include <stdio.h>
#define N 10
#define FILENAME "lab1.txt"

int main(void)
{
    /*  Declare and initialize variables.  */
    int k, count=0;
    double y[N], y_ave, sum=0;
    FILE *lab;
```

Code on Next Slide

```
   /* Exit program. */
   return 0; }
```

Body code for Program chapter5_1

```
/*  Open file, read data into an array,  */
/*  and compute a sum of the values.     */
lab = fopen(FILENAME,"r");
if (lab == NULL)
    printf("Error opening input file. \n");
else
{
    /*  Input and process data.  */
    for (k=0; k<=N-1; k++)
    {
        fscanf(lab,"%lf",&y[k]);
        sum += y[k];
    }
   /*  Compute average and count values that  */
    /*  are greater than the average.          */
    y_ave = sum/N;
    for (k=0; k<=N-1; k++)
        if (y[k] > y_ave)
            count++;

    printf("%d values greater than the average \n",count);
    fclose(lab);
}
```

Computations and Output

- Array values are printed using a subscript to specify the individual value desired.

- For example, to print the first and last values of the array `y` from the previous example, we would use the following statement:

  - ```
    printf("first and last array values:\n");
    printf("%f %f\n",y[0],y[N-1]);
    ```

- To print all 100 values of `y`, one per line, we use the following loop:

  - ```
    printf("y values:\n");
    for (k=0; k<=N-1; k++)
       printf("%f\n",y[k]);
    ```

Computations and Output

- Similar statements can be used to write array values to a data file.

- For example, to print the value of `y[k]` on a line in a data file with a file pointer `sensor`, we use the following statement:

  - `fprintf(sensor,"%f\n",y[k]);`

- Since the newline indicator is included, the next value written to the file will be on a new line.

Computations and Output

- The number of elements in an array is used in the array declaration and is used in loops to access the elements in the array.
- If the number of elements is changed, then there are several places in the program that need to be modified.
- Changing the size of an array is simplified if a symbolic constant is used to specify the size of the array.
- To change the size, only the pre-processor directive needs to be changed.

Operator Precendence

Precedence	Operation	Associativity
1	`( ) [ ]`	Innermost first
2	`++ -- + - ! (type)`	Right to left (unary)
3	`* / %`	Left to right
4	`+ -`	Left to right
5	`< <= > >=`	Left to right
6	`== !=`	Left to right
7	`&&`	Left to right
8	`||`	Left to right
9	`?:`	Right to left
10	`= += -= *= /= %=`	Right to left
11	`,`	Left to right

Function Arguments

- When the information in an array is passed to a function, two parameters are usually used; one parameter specifies a particular array, and the other parameter specifies the number of elements used in the array.

- By specifying the number of elements of the array, the function becomes more flexible.

Function Arguments

- The next program reads an array from a data file and then references a function to determine the maximum value in the array.
- The variable `npts` is used to specify the number of values in the array.
- The value of `npts` can be less than or equal to the defined size of the array, which is 100.
- The function has two arguments—the name of the array and the number of points in the array, as indicated in the function prototype statement.

Function Arguments

- This program assumes that there will not be more than 100 values in the file; otherwise, this program will not work correctly.
- Arrays must be specified to be as large as, or larger than, the maximum number of values to be read into them.
- The purpose of program `chapter5_2` is to illustrate the use of an array as a function argument.
- If the purpose of this program was to determine the maximum of the data values in the file, an array would not have been necessary; the maximum could have been determined as the data values are read.

Function Arguments

```
/*-------------------------------------------------------------*/
/*  Program chapter5_2                                         */
/*                                                             */
/*  This program reads values from a data file and             */
/*  determines the maximum value with a function.              */

#include <stdio.h>
#define N 10
#define FILENAME "lab2.txt"

double max(double x[],int n);

int main(void)
{
    /*  Declare variables.  */
    int k=0, npts;
    double y[N];
    FILE *lab;
```

Code on Next Slide

```
   /* Exit program. */
   return 0; }
```

Body code for Program chapter5_2

```
/*  Open file, read data into an array.  */
lab = fopen(FILENAME,"r");
if (lab == NULL)
    printf("Error opening input file. \n");
else
{
    while ((fscanf(lab,"%lf",&y[k])) == 1)
        k++;
    npts = k;

    /*  Find and print the maximum value.  */
    printf("Maximum value: %f \n",max(y,npts));

    /*  Close file and exit program.  */
    fclose(lab);
}
```

Function Arguments

```
/*-------------------------------------------------------*/
/*  This function returns the maximum value in an array x      */
/*  with n elements.                                           */
double max(double x[],int n)
{
    /*  Declare variables.  */
    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 maximum value.  */
    return max_x;
}
```

Function Arguments

- There is a very significant difference between using simple values as function parameters and using arrays as parameters.

- When a simple variable is used as a parameter, the value is passed to the formal argument in the function, and thus the value of the original variable cannot be changed; this is a **call-by-value** reference.

- When an array is used as a parameter, the memory address of the array is passed to the function instead of the entire set of values in the array.

Function Arguments

- Therefore, the function references values in the original array; this is a **call-by-address** reference.
- Because a function accesses the original array values, we must be very careful that we do not inadvertently change values in an array within a function.
- Of course, there may be occasions where we wish to change the values in the array.

Sorting Algorithms

- Sorting a group of data values is another operation that is routinely used when analyzing data.
- One of the reasons that there are so many sorting algorithms is that there is not one “best” sorting algorithm.
- Some algorithms are faster if the data are already close to the correct order, but these algorithms may be very inefficient if the order is random or close to the opposite order.

Sorting Algorithms

- Therefore, to choose the best sorting algorithm for a particular application, you usually need to know something about the order of the original data.

- We will present two simple sorting algorithms: **selection sort** and **bubble sort**.

Selection Sort

- The **selection sort** algorithm begins by finding the minimum value and exchanging it with the value in the first position in the array.
- Then the algorithm finds the minimum value beginning with the second element, and it exchanges this minimum with the second element.
- This process continues until reaching the next-to-last element, which is compared with the last element; the values are exchanged if they are out of order.
- At this point, the entire array of values will be in ascending order.

Selection Sort

- Original order:
  5 3 12 8 **1** 9
- Exchange the minimum with the value in the first position:
  1 **3** 12 8 5 9
- Exchange the minimum with the value in the second position:
  1 3 **12** 8 **5** 9
- Exchange the minimum with the value in the third position:
  1 3 5 **8** 12 9

Selection Sort

- Exchange the minimum with the value in the fourth position:
  1 3 5 8 **12 9**
- Exchange the minimum with the value in the fifth position:
  1 3 5 8 9 12
- Array values are now in ascending order!

Selection Sort

```
void selection_sort(int x[],int npts)
{
    /*  Declare variables.  */
    int k, j, m, hold;

    for (k=0; k<=npts-2; k++)
    {
        /*  Exchange minimum with next array value.  */
        m = k;
        for (j=k+1; j<=npts-1; j++)
            if (x[j] < x[m])
                m = j;
        hold = x[m];
        x[m] = x[k];
        x[k] = hold;
    }
    /*  Void return.  */
    return;
}
```

Bubble Sort

- The **bubble sort** algorithm checks adjacent elements. If they are out of order, swap them.
- For each pass, checks the first and second element, then checks the second and third element, all the way until the second last and last element.
- Keep repeating as long as there is at least one swap in the previous pass.

Bubble Sort

- Original order:
  5 3 12 8 1 9
- The first and second element are out of order:
  3 **5 12** 8 1 9
- The second and third element are in order:
  3 5 **12 8** 1 9
- The third and fourth element are out of order:
  3 5 8 **12 1** 9
- The fourth and fifth element are out of order:
  3 5 8 1 **12 9**

Bubble Sort

- The fifth and sixth element are out of order:

```
3   5   8   1   9 12
```

- The first pass has been completed. There has been at least one swap, so we do another pass.

Bubble Sort

- After the second pass:
  `3 5 1 8 9 12`
- After the third pass:
  `3 1 5 8 9 12`
- After the fourth pass:
  `1 3 5 8 9 12`
- After the fifth pass:
  `1 3 5 8 9 12`
- There are no swaps in the fifth pass, so we end and the array is sorted!

Bubble Sort

```
void bubble_sort(int x[],int npts)
{
    /*  Declare variables.  */
    int done, hold, k;

    do {
        done = 1;
        for (k=0; k<=npts-2; k++)
            if (x[k] > x[k+1]) {
                /* Swap adjacent elements only if they are
                 * out of order. */
                hold = x[k];
                x[k] = x[k+1];
                x[k+1] = hold;
                done = 0;
            }
    } while (done==0);
    /*  Void return.  */
    return;
}
```

Quicksort (Non-Examinable)

- An important sorting algorithm
- C programming practice on pointers for array traversal, element exchanges and recursion.
- Illustration of the *Divide-and-Conquer* method
- Basic idea: *separate*

  To rearrange the array around one chosen *pivot* element. Elements that are smaller than the pivot are arranged to be on the left, and elements that are larger than the pivot are arranged to be on the right.

Quicksort

Original list:

4 10 3 6 −1 0 2 5

Separate into groups of values smaller and larger than the pivot value:

[3 −1 0 2] 4 [10 6 5]

Separate each remaining group into groups of values smaller and larger than the pivot value of each group:

[−1 0 2] 3 4 [6 5] 10

Separate each remaining group into groups of values smaller and larger than the pivot of each group:

−1 [0 2] 3 4 5 6 10

Separate each remaining group into groups of values smaller and larger than the pivot of each group:

−1 0 2 3 4 5 6 10

Quicksort

```
#include <stdio.h>
#define N 8

void swap(int [], int, int);
void quicksort(int [], int, int);
int separate(int [], int low, int high);

int main(void) {
    int a[N] = {4, 10, 3, 6, -1, 0, 2, 55}, k;
quicksort(a, 0, N-1);
    for (k = 0; k < N; k++) printf(" %d ", a[k]);
    printf("\n");
    return 0;
}

void swap(int a[], int i, int j) {
    int tmp;
    tmp = a[i]; a[i] = a[j]; a[j] = tmp;
}
```

Quicksort - Main Function

```
void quicksort(int a[], int left, int right) {
    int break_pt;
    if (right - left < 2) return;
    if (right == left + 1) {
        if (a[left] > a[right])
            swap(a, left, right);
        return;
    }
    break_pt = separate(a, left, right);
    quicksort(a, left, break_pt - 1);
    quicksort(a, break_pt + 1, right);
}
```

Separate (or “Partition”)

- Most implementations of quick sort make use of the fact that you can partition in place by keeping two pointers: one moving in from the left and a second moving in from the right.
- They are moved towards the centre until the left pointer finds an element greater than the pivot and the right one finds an element less than the pivot. These two elements are then swapped.
- The pointers are then moved inward again until they "cross over".
- The pivot is then swapped into the slot to which the right pointer points and the partition is complete.
- Note that code below does not check that left does not exceed the array bound. You need to add this check, before performing the swaps - both the one in the loop and the final one outside the loop.

Quicksort

The code here is different from Etter, p319.

```
int separate(int a[], int left, int right) {
    int i, j, pivot;
    pivot = a[left];
    i = left + 1;
    j = right;
    while ( i < j ) {
        while(a[i] <= pivot) i++; /* Move i rightwards */
        while(a[j] > pivot) j--;  /* Move j leftwards */
        if (i < j) swap(a, i, j);
    }
    /* j is final position for the pivot */
    a[left] = a[j];
    a[j] = pivot;
    return j;
}
```

Search Algorithms

- Another very common operation performed with arrays is searching the array for a specific value.
- We may want to know if a particular value is in the array, how many times it occurs in the array, or where it first occurs in the array.
- We will develop several functions for searching an array; then, when we need to perform a search in a program, we can probably use one of these functions with little or no modification.
- Searching algorithms fall into two groups: those for searching an unordered list and those for searching an ordered list.

Unordered List

- In an unordered list, the elements are not sorted into any order.
- The algorithm to search an unordered array is just a simple **sequential search**: it will check the first element, check the second element, and so on.
- We could develop it as an integer function that either returns the position of the desired value in the array or returns a value of -1 if the desired value is not in the array.

Unordered List

```
int sequential_search1(int x[], int npts, int value)
{
    /*  Declare variables.  */
    int k=0, index=-1;

    /*  Search for value.  */
    while (k<=npts-1 && x[k]!=value)
        k++;
    if (k != npts)
        index = k;

    /*  Return index.  */
    return index;
}
```

Ordered List

- We now consider searching an ordered or sorted list of values.
- Assume that we have a list of ordered values, and we are searching for the value 25:
  `-7 2 14 38 52 77 105`
- As soon as we reach the value 38, we will know that 25 is not in the list because we know that the list is ordered in ascending numerical order.
- Therefore, we do not always have to search the entire list if the value we are searching for is not in the list.

Ordered List

```
int sequential_search2(int x[], int npts, int value)
{
    /*  Declare variables.  */
    int k=0, index=-1;

    /*  Search for value.  */
    while (k<=npts-1 && x[k]<value)
        k++;
    if (k <= npts-1)
        if (x[k] == value)
            index = k;

    /*  Return index value.  */
    return index;
}
```

Binary Search

- Another popular and more efficient algorithm for searching an ordered list is a technique called **binary search**.
- We first check the middle of the array and determine whether our desired value is in the first half of the array or the second half of the array.
- If it is in the first half, we then check the middle of the first half and determine whether our desired value is in the first quarter of the array or the second quarter of the array.
- The process of dividing the array into smaller and smaller pieces continues until we find the element or find the position where it should have been.

Binary Search

- Search for value 14 in the following list:

```
-7   2   14   38   52   77   105
 t             m              b
```

- 38 $>$ 14 so choose top half:

```
-7   2   14   38   52   77   105
 t   m    b
```

- 2 $<$ 14:

```
-7   2   14   38   52   77   105
         tb
          m
```

- 14 = 14 so it is found!

Binary Search

- Search for value 25 in the following list:

```
-7   2   14   38   52   77   105
 t             m              b
```

- $38 > 25$ so choose top half:

```
-7   2   14   38   52   77   105
 t   m    b
```

- $2 < 25$:

```
-7   2   14   38   52   77   105
         tb
          m
```

- $14 < 25$:

```
-7   2   14   38   52   77   105
          b    t
          m
```

- $t > b$ so item 25 is not found!

Binary Search

```
int binary_search(int x[], int npts, int value)
{
    int done=0, top=0, bottom=npts-1, mid;
    int index=-1;

    while (top<=bottom && done==0) {
        /*  Determine middle.  */
        mid = (top + bottom)/2;
        /*  Check value in middle. in an array */
        if (x[mid] == value)
            done = 1;
        else
            /*  Is value in top or bottom half?  */
            if (x[mid] > value)
                bottom = mid - 1;
            else
                top = mid + 1;
    }
    /*  Return index value.  */
    if (done == 1)
        index = mid;
    return index;
}
```

References

Etter Sections 5.1 to 5.7

Next Lecture

Arrays and Matrices (Part II)