UEE1302
Introduction to Computers and Programming

C_Lecture 07:

Arrays

C: How to Program 7th ed.
Chapter 6 C Arrays

Agenda

- Introduction to Arrays
- Declaring Arrays
- Passing Arrays to Functions
- Examples using Arrays
  - Partially-Filled Arrays
  - Searching an Array
  - Sorting an Array
- Multidimensional Arrays

Introduction to Arrays

- Array definition:
  - A collection of data of same type
- Used for lists of like items
  - Ex: scores, temperatures, names, etc.
  - Avoids declaring multiple simple variables
  - Can manipulate "list" as one entity

One-Dimension Arrays

- One-Dimension Array (Single-Dimension Array or Vector): a list of related values
  - All items in the list have the same data type
  - All list members are stored using a single group name
- Example: a list of grades

```
98, 87, 92, 79, 85
```

  - All grades are integers and must be declared
  - Can be declared as single unit under a common name (the array name)

Declaring Arrays

- Array declaration statement provides:
  - The array (list) name
  - The data type of array items
  - The number of items in array
- Syntax

```
dataType arrayName[numberOfItems]
```

  - Common programming practice requires defining the number of array items as a constant before declaring the array

Declaring Arrays (cont.)

- Examples of array declaration statements:

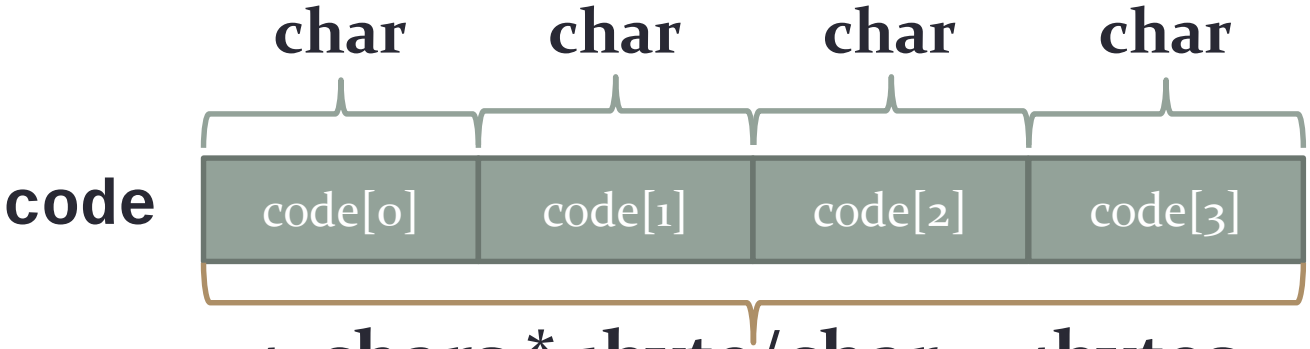
```
// define a constant for the number of  items
const int NUMBER = 5;
int grade[NUMBER]; // declare the array

const int ARRAYSIZE = 4;
char code[ARRAYSIZE];

const int MENUS = 6;
double prices[MENUS];
```

One-Dimension Arrays (cont.)

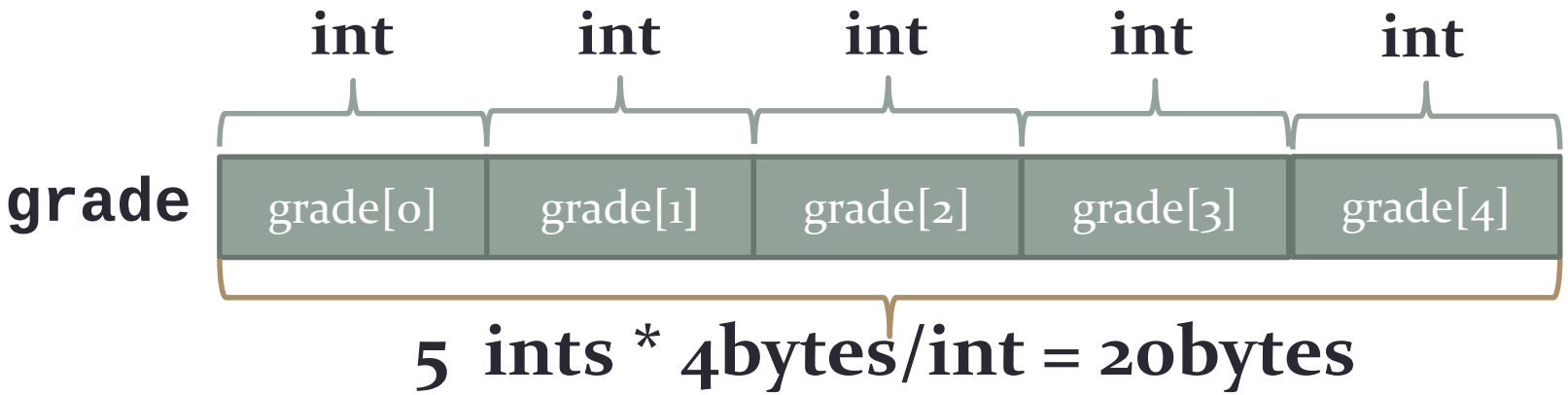
- Each array allocates sufficient memory to hold the number of data items given in the declaration
- Array element (component): an item of the array
- Individual array elements are stored sequentially
  - A key feature of arrays that provides a simple mechanism for easily locating single elements

One-Dimension Arrays (cont.)

```
char code[4];
```



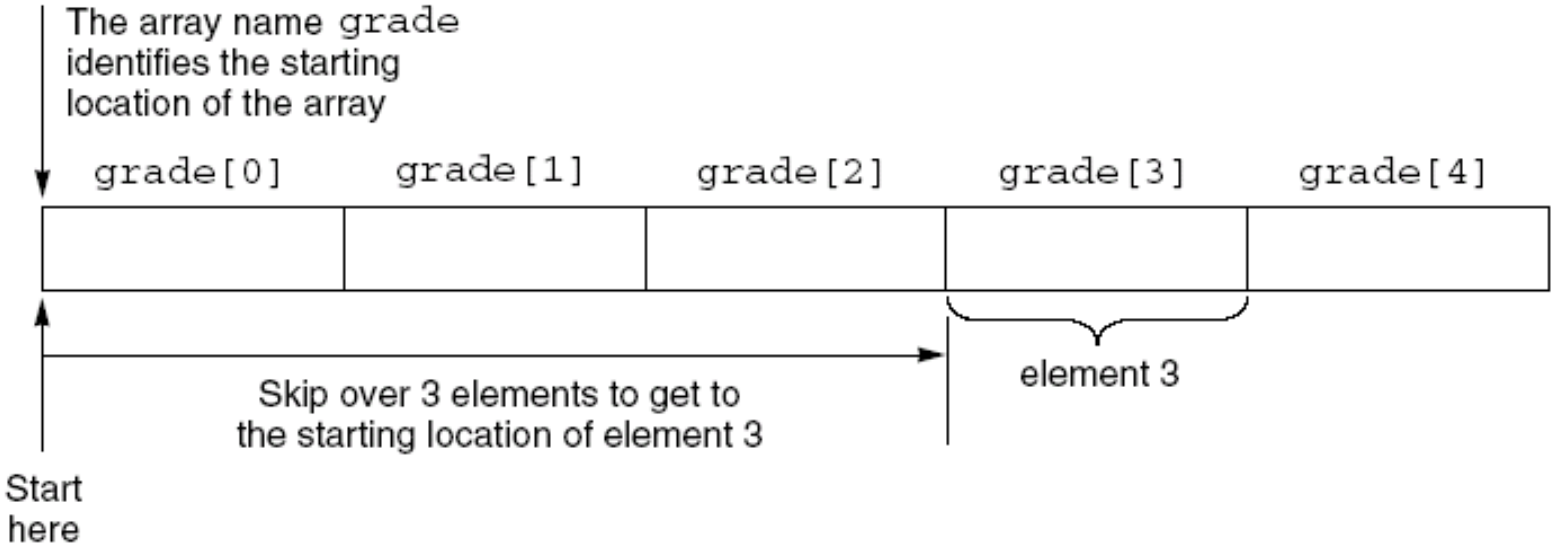

```
int grade[5];
```



Accessing Arrays

- Index (subscript value): position of individual element in an array
- Accessing of array elements: done by giving array name and the element's index
  - `grade[0]` refers to first grade stored in `grade` array
- Subscripted variables can be used anywhere that scalar variables are valid:
  - `grade[0] = 95.75;`
  - `grade[1] = grade[0] - 11.0;`

Accessing Arrays (cont.)



Accessing Arrays (cont.)

- Subscripts: do not have to be integers
  - Any expression that evaluates to an integer may be used as a subscript
  - Subscript must be within the declared range
- Examples of valid subscripted variables (assumes `i` and `j` are `int` variables):
  - `grade[i]`
  - `grade[2*i]`
  - `grade[j-i]`

Initializing Arrays

- As simple variables can be initialized at declaration:

```
int price = 0;
// 0 is initial value
```

- Arrays can as well:

```
int children[3] = { 2, 12, 1 };
```

- Equivalent to following:

```
int children[3];
children[0] = 2;
children[1] = 12;
children[2] = 1;
```

Auto-Initializing Arrays

- If fewer values than size supplied:
  - Fills from beginning
  - Fills "rest" with zero of array base type
- Example:

```
long vec[15] = {-1};
//vec[0]=-1, vec[1]=0 … vec[14]=0
```

- If array-size is left out
  - Declares array with size required based on number of initialization values
  - Example:

```
int b[] = {5, 12, 11};
```

  - Allocate array b to have the size of 3

Special: `char` Array

- `char` array: an array consists of multiple characters char
  - Example:

```
char name[] = "Win";
printf("%d\n", sizeof(name));
```

  - `\0` in `name[3]` is called `NULL` character and automatically appended in the assignment.
  - Another example:

```
char name[] = {'W','i','n'};
printf("%d\n", sizeof(name));
```

- Will be introduced in details in the later lecture

Input and Output of Array Values

- Individual array elements can be assigned with values interactively using a `scanf` statement

```
scanf("%d", &grade[0]);
scanf("%d%d",&grade[1], &grade[2]);
scanf("%d%d",&grade[3], &grade[4]);
```

- Instead, a `for` loop can be used

```
const int NUM = 5;
for (i = 0; i < NUM; i++)
{
    printf( "Enter a grade: ");
    scanf("%d", &grade[i]);
}
```

Input and Output of Array Values (cont.)

- Bounds checking: C/C++ does not check if value of an index is within the declared bounds

```
int score[4];
score[10] = 5;
```

- If an out-of-bounds index is used, C/C++ will not provide notification
  - Program will attempt to access out-of-bounds element, causing program error or crash

Input and Output of Array Values (cont.)

- Using `printf` to display subscripted variables:
  - Example 1

```
printf("%d", prices[5]);
```

  - Example 2

```
printf("The value of element ");
printf(" %d is %d ", i, grade[i]);
```

  - Example 3

```
const int NUMELS = 20;
for (k = 5; k < NUMELS; k++)
    printf("%d %d", k, amount[k]);
```

Initializing an Array with an Initializer List

```
// Modified from Fig. 6.4: fig06_04.c
// Initializing an array with an initializer list.
#include <stdio.h>

int main ( void )
{
    int n[5] = {32, 27, 64, 18, 95};
    size_t i;

    printf("%s%13s\n", "Element", "Value");

    for (i = 0; i < 5; i++)
        printf("%7u%13d\n", i, n[i]);

    return 0;
}
```

Initializing an Array with an Initializer List (cont.)

- screen output

```
Element      Value
      0         32
      1         27
      2         64
      3         18
      4         95
```

Initializing an Array

```
// Fig. 6.5: fig06_05.c
// Set arrays to the even integers from 2 to 20.
#include <stdio.h>
#define SIZE 10 // maximum size of array

int main ( void )
{
    int s[SIZE]; // array s has 10 elements
    size_t i;
    for (i = 0; i < SIZE; i++)
        s [i] = 2 + 2 * i;
    printf("%s%13s\n", "Element", "Value");
    for (i = 0; i < SIZE; i++)
        printf("%7u%13d\n", i, n[i]);
    return 0;
}
```

Initializing an Array (cont.)

- screen output

```
Element      Value
      0          2
      1          4
      2          6
      3          8
      4         10
      5         12
      6         14
      7         16
      8         18
      9         20
```

Die-rolling Program

```
// Fig. 6.9: fig06_09.c
// Roll a six-sided die 6,000 times.
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 7
int main( void )
{
    int face; // random die value 1 – 6
    unsigned int roll; // roll counter 1 - 6000
    unsigned int frequency[SIZE] = {0}; // initialize to 0
    srand( time(NULL) ); //seed random number generator

    for (roll = 1; roll <= 6000; roll++) {
        face = 1 + rand()%6;
        ++frequency[face];
    }
```

Die-rolling Program (cont.)

```
    printf("%s%17s\n", "Face", "Frequency");
    for (face = 1; face < SIZE; face++)
        printf("%4d%17d\n", face, frequency[face]);

    return 0;
}
```

- screen output

```
   Face   Frequency
      1        1029
      2         951
      3         987
      4        1033
      5        1010
      6         990
```

Defined Constant as Array Size

- Always use defined/named constant for array size, not one number
  - improves readability
  - improves versatility
  - improves maintainability
- Example:

```
const int NUM_STUDENTS = 4;
int score[NUM_STUDENTS];
```

Uses of Defined Constant

- Use everywhere size of array is needed
  - In for loop for traversal:

```
for ( i = 0; i < NUM_STUDENTS; i++ )
{
// Manipulate array
}
```

  - In calculations involving size:

```
lastIndex = (NUM_STUDENTS – 1);
```

  - When passing array to functions
- If the size changes => requires only ONE change in your program!

Arrays in Functions

- Pass indexed variables to functions
  - array elements as arguments
  - Ex: `func(c[0],…)`

    => an individual "element" of an array can be function parameter
- Pass entire arrays to functions
  - array names as arguments
  - Ex: `func(arrayName,24)`

    => all array elements can be passed as one entity
- As return value from function
  - can be done and discussed in later lecture

Indexed Variables as Arguments

- Indexed variable handled same as simple variable of array base type
- Given this function declaration:

```
void myFunction(double par1);
```

- And these declarations:

```
int i; double n, a[10];
```

- Can make these function calls:

```
myFunction(i); // i is converted to double
myFunction(a[3]); // a[3] is double
myFunction(n); // n is double
```

Subtlety of Indexing

- Consider the example:

```
myFunction(a[i]);
```

  - Value of `i` is determined first => `i` determines which indexed variable is sent

- More example

```
myFunction(a[i*5]);
```

  - Perfectly legal, from compiler's view
  - Programmer responsible for staying "in-bounds" of array

  => mean checking if `i` falls in the legal range

Entire Arrays as Arguments

- Formal parameter can be entire array
  - Argument then passed in function call is array name
  - Called "array parameter"

- Send size of array as well
  - Typically done as second parameter
  - Simple `int` type formal parameter

Example of Arrays as Arguments

```
void fillUp(int a[], int size)
{
    int i;
    printf("Enter %d numbers:\n", size);
    for (i = 0; i < size; i++)
        scanf("%d", &a[i]);
    printf("The last array index is %d\n",
                                    size-1);
}
```

- In main() function definition, consider the following calls:

```
int score[4], numberOfScores = 4;
fillUp(score, numberOfScores);
```

  - 1st argument score is entire array
  - 2nd argument numberOfScores is an integer
  - Note no brackets in array argument!

Arrays as Arguments: How?

- What's really passed?
- Think of array as 3 "pieces"
  - address of first indexed variable (`arrName[0]`)
  - array base type
  - size of array
- Only 1st piece is passed!
  - Just the beginning address of array
  - Very similar to "pass-by-reference"

Array Name and Address

```
// Fig. 6.12: fig06_12.c
// The name of an array is the same as &array[0].
#include <stdio.h>

int main( void )
{
    char array[5]; // define an array of size 5

    printf("    array = %p\n&array[0] = %p\n   &array =
                %p\n", array, &array[0], &array);
    return 0;
}
```

- screen output

```
    array = 0012FF78
&array[0] = 0012FF78
   &array = 0012FF78
```

Array Parameters

- May seem strange
  - No brackets in array argument when being called
  - Must send size separately!!
- One nice property:
  - Can use SAME function to fill any size array (of same type)!
  - Exemplifies "re-use" properties of functions
  - Example:

```
int score[4], time[10];
fillUp(score, 4);
fillUp(time, 10);
```

`const` Parameter Modifier

- Recall: array parameter actually passes address of 1st element
  - similar to pass-by-reference
- Function can then modify array!
  - Often desirable, sometimes not!
- Protect array contents from modification
  - use "`const`" modifier before array parameter
  - called "constant array parameter"
  - Tells compiler to "not allow" modifications
  - Ex: `int fillUp(const int a[], int size)`

Functions that Return an Array

- Functions cannot return arrays as the same way simple types are returned
- Requires use of a "pointer"
- Will be discussed in later lectures…

Programming with Arrays

- Plenty of uses
  - Partially-filled arrays

    => must be declared some "max size"
  - Sorting
  - Searching

Partially-Filled Arrays

- Difficult to know exact array size needed
  - Ex: The maximum number of students in one class is 50 but may be 45 or 47 in fact
- Must declare to be largest possible size
  - Must then keep "track" of valid data in array
  - Additional "tracking" variable needed

```
int numberUsed;
```

  - Tracks current number of elements in array

Example of Partially-Filled Arrays

```
#include <stdio.h>
const int MAX_NUM_SCORES = 100;

void initializeArray(int a[], int size);
int fillArray(int a[], int size);
double computeAverage(const int a[],int usedSize);

int main()
{
    int score[MAX_NUM_SCORES], numUsed = 0;
    double avg = 0;
    initializeArray(score, MAX_NUM_SCORES);
    numUsed = fillArray(score, MAX_NUM_SCORES);
    avg = computeAverage(score, numUsed);
    printf("Average = %.2f\n", avg);
    return 0;
}
```

Example of Partially-Filled Arrays (cont.)

```
void initializeArray(int a[], int size)
{
    // fill up cells a[0] … a[size-1] with -1
    int i;
    for (i = 0; i < size; i++)
    {
        a[i] = -1;
    }
}
```

Example of Partially-Filled Arrays (cont.)

```
//called from fillArray(score, MAX_NUM_SCORES,  numUsed);
int fillArray(int a[], int size)
{
    int next = -1, i = 0;
    int usedSize;
    printf("Enter up to %d numbers: ", size);
    scanf("%d", &next);
    while ((next >= 0) && ( i < size))
    {
        a[i] = next;
        i++;
        scanf("%d", &next);
    }
    usedSize = i;
    return usedSize;
}
```

Example of Partially-Filled Arrays (cont.)

```
//called from computeAverage(score, numUsed);
double computeAverage(const int a[], int usedSize)
{
    double total = 0.0;
    int i;
    for (i = 0; i < usedSize; i++)
        total = total + a[i];
    if (usedSize>0)
    {
        return (total/usedSize);
    }
    else
    {
        printf("ERROR: no element in array\n");
        return 0;
    }
}
```

Searching Arrays with Linear Search

```
// Fig. 6.18: fig06_18.c
// Linear search of an array.
#include <stdio.h>
#define SIZE 100
int linearSearch( const int array[], int key, int size);

int main( void )
{
    int a[SIZE];
    int i; // counter
    int searchKey;
    int element;
    // create data
    for ( i = 0; i < SIZE; i++)
        a[ i ] = 2 * i;

    printf("Enter integer search key:\n");
    scanf("%d", &searchKey);
```

Searching Arrays with Linear Search (cont.)

```
    // attempt to locate searchKey in array a
    element = linearSearch( a, searchKey, SIZE );

    if ( element != -1 )
        printf("Found value in element %d\n", element);
    else
        printf("Value not found\n");

    return 0;
}
```

Searching Arrays with Linear Search (cont.)

```
// compare key to every element of array until location is
// found or until end of array is reached; return
// subscript of element if key is found of -1 if key not
// found
int linearSearch( const int array[], int key, int size)
{
    int j; // counter
    for ( j = 0; j < size; j++)
        if ( array[j] == key ) // if found,
            return j;
    return -1; // key not found
}
```

Searching Arrays with Linear Search (cont.)

- screen output

```
Enter integer search key:
36
Found value in element 18
```

```
Enter integer search key:
37
Value not found
```

Sorting an Array

- Input:
  - an array `c = {c[0],c[1],…,c[n-1]}` of `n` numbers
- Output:
  - a permutation `c'[0],c'[1],…,c'[n-1]` of the input sequence such that .

    `c'[0] ≦ c'[1] ≦ ... ≦ c'[n-1]`
  - The number that we wish to sort are known as the keys.
- Example:
  - Input: `8 2 4 9 3 6`
  - Output: `2 3 4 6 8 9`

Example of Insertion Sort

- IEDA: sort your poker cards

8	2	4	9	3	6
2	8	4	9	3	6
2	4	8	9	3	6
2	4	8	9	3	6
2	3	4	8	9	6
2	3	4	6	8	9

Pseudocode of Insertion Sort

```
Insertion-Sort(A)
1 for j ←1 to length[A]-1
2     do key ← A[j]
3     //Insert A[j] into the sorted sequence A[0..j-1]
4     i ← j
5     while i>0 and A[i]>key do
6         A[i] ← A[i-1]
7         i ← i - 1
8      A[i] ← key
```

i	j			Key: 2	
8	2	4	9	3	6
8	8	4	9	3	6
2	8	4	9	3	6

Sorting Arrays with Insertion Sort

```
// This program sorts an array's values in ascending order.
#include <stdio.h>

int main( void )
{
    const int arraySize = 10; // size of array a
    int i, next;
    int data[arraySize] = {34, 56, 4, 10, 77, 51, 93, 30,
5, 52};
    int insert; // temporary variable to hold element to
insert

    printf( "Unsorted array:\n");
    // output original array
    for (i = 0; i < arraySize; i++)
        printf("%4d", data[ i ]);
```

Sorting Arrays with Insertion Sort (cont.)

```
    // insertion sort
    // loop over the elements of the array
    for ( next = 1; next < arraySize; next++)
    {
        insert = data[ next ];
        int moveItem = next;
        while ( ( moveItem > 0) &&
                ( data[moveItem-1] > insert ) )
        {
            // shift element one slot to the right
            data[ moveItem ] = data[ moveItem – 1 ];
            moveItem--;
        }
        data[ moveItem ] = insert;
    }
```

Sorting Arrays with Insertion Sort(cont.)

```
    printf("\nSorted array:\n");
    // output sorted array
    for (i = 0; i < arraySize; i++)
        printf("%4d", data[ i ]);
    printf("\n");

    return 0;
}
```

Sorting Arrays with Insertion Sort (cont.)

- screen output

```
Unsorted array:
  34  56   4  10  77  51  93  30   5  52
Sorted array:
   4   5  10  30  34  51  52  56  77  93
```

Selection Sort

12	16	22	14	8	17

6	16	22	14	12	17

- Given n numbers to sort:
- Repeat the following n-1 times:
  - Mark the first unsorted number
  - Find the smallest unsorted number
  - Swap the marked and smallest numbers

Selection Sort

6	8	22	14	12	17

6	8	12	14	22	17

- Given n numbers to sort:
- Repeat the following n-1 times:
  - Mark the first unsorted number
  - Find the smallest unsorted number
  - Swap the marked and smallest numbers

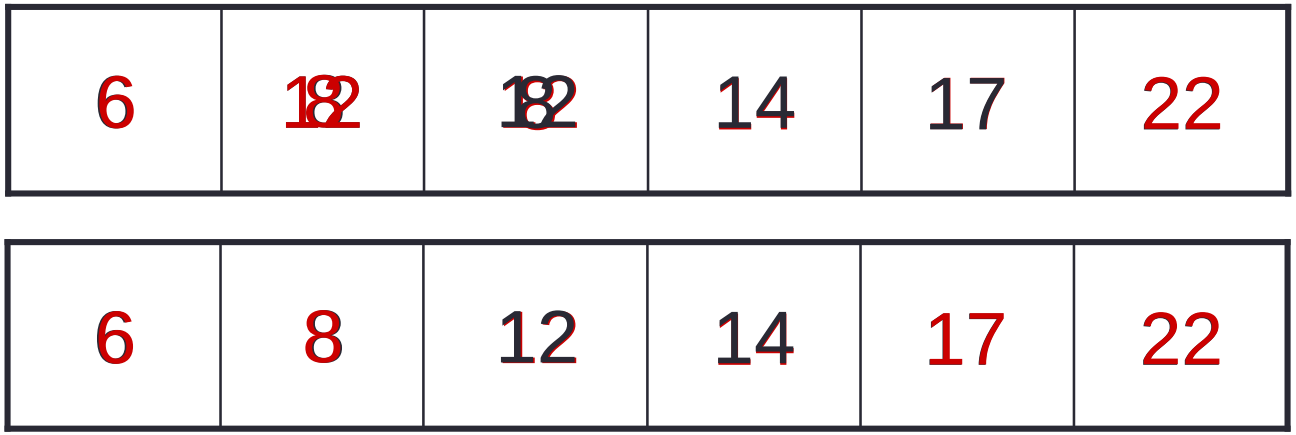
Selection Sort Code

```
// selection sort
void selectSort( int array[], const int size)
{
    int i, j, min;
    int hold;
    for ( i = 0; i < size - 1; i++)
    {
        min = i;
        for ( j = i + 1;  j < size; j++)
        {
            if (array[j] < array[min])
                min = j;
        }
        hold = array[i];
        array[i] = array[min];
        array[min] = hold;
    }
}
```

Bubble Sort

- Given n numbers to sort:
- Repeat the following n-1 times:
  - For each pair of adjacent numbers:
    - If the number on the left is greater than the number on the right, swap them.

Bubble Sort



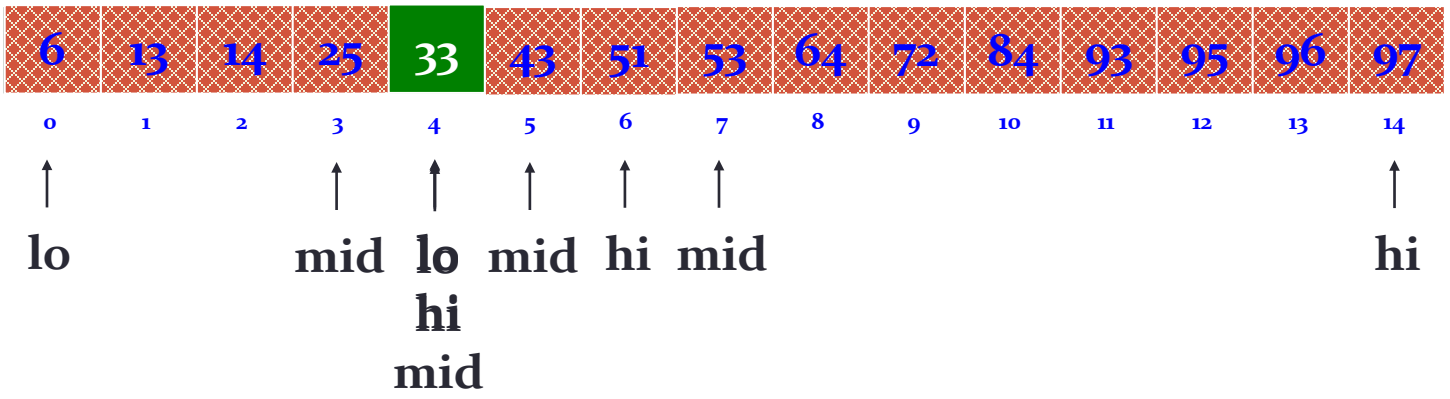
- Given n numbers to sort:
- Repeat the following n-1 times:
  - For each pair of adjacent numbers:
    - If the number on the left is greater than the number on the right, swap them.

Bubble Sort Code

```
// Bubble sort
void bubbleSort( int array[], const int size)
{
    int i, j;
    int hold;
    for ( i = 0; i < size - 1; i++)
    {
        for ( j = 0;  j < size - 1; j++)
        {
            if (array[j] < array[j+1])
            {
                hold = array[j];
                array[j] = array[j+1];
                array[j+1] = hold;
            }
        }
    }
}
```

Binary Search

- Binary search is used to find the position of a key in a sorted array.
  - We compare the key with value in the middle of array.
    - If the key is less than the value of the middle element of the array, we repeat the above step on the sub-array on the left.
    - If the key is greater than the value of the middle element of the array, we repeat the above step on the sub-array on the right.



Binary Search Code

```
// Binary search
int binarySearch(const int b[], int key, int low, int high)
{
    int middle;
    while ( low <= high) {
        middle = (low + high) / 2;
        if ( key == b[middle] )
            return middle;
        else if ( key < b[middle] )
            high = middle – 1;
        else
            low = middle + 1;
    }
}
```

Multi-dimensional Arrays

- Declare arrays with more than one index
  - `char page[30][100];`
    `//fixed sizes for two dimensions`
  - Two indexes: An "array of arrays"
  - Visualize as:

Page[0][0]	Page[0][1]	…	Page[0][99]
Page[1][0]	Page[1][1]	…	Page[1][99]
…	…	…	…
Page[29][0]	Page[29][1]	…	Page[29][99]

- C allows any number of indexes
  - Typically no more than two levels

Initializing Multi-dimensional Arrays

- Unless specified, all initial values of arrays are garbage.
- You can specify initial values by enclosing each row in curly braces like this

```
char ticTacToeBoard[3][3] =
            { {'O', 'X', 'X'},
              {'O', 'O', 'X'},
              {' ', 'X', ' '} };
```

Multi-dimensional Array Parameters

- Similar to one-dimensional array
  - 1st dimension size not given: provided as second parameter
  - 2nd dimension size IS given => remember
- Example:

```
void DisplayPage(const char p[][100], int sizeDim1)
{
    for ( i = 0; i < sizeDim1; i++)
    {
        for ( j = 0; j < 100; j++)
            printf( "%3c", p[ i ][ j ]);
        printf("\n");
    }
}
```

Initializing a Multi-dimensional Array

```
// Fig. 6.21: fig06_21.c
// initializing multi-dimensional array.
#include <stdio.h>
void printArray( const int a[][3] );

int main ( void )
{
    int array1[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int array2[2][3] = {1, 2, 3, 4, 5};
    int array3[2][3] = {{1, 2}, {4}};
    printf("Values in array 1 by row are:\n");
    printArray(array1);
    printf("Values in array 2 by row are:\n");
    printArray(array2);
    printf("Values in array 3 by row are:\n");
    printArray(array3);
    return 0;
}
```

Initializing a Multi-dimensional Array (cont.)

```
void printArray( const int a[][3] )
{
    int i;  // row counter
    int j;  // column counter

    for ( i = 0; i <= 1; i++) {
        for ( j = 0; j <= 2; j++)
            printf("%d ", a[i][j]);
        printf("\n");
    }
}
```

Initializing a Multi-dimensional Array (cont.)

- screen output

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Summary

- Array is collection of "same type" data
- Indexed variables of array used just like any other simple variables
- `for`-loop "natural" way to traverse arrays
- Programmer responsible for staying "in bounds" of array
- Array parameter is "new" kind
  - Similar to call-by-reference

Summary (cont.)

- Array elements stored sequentially
  - "Contiguous" portion of memory
  - Only address of 1st element is passed to functions
- Partially-filled arrays => more tracking
- Constant array parameters
  - Prevent modification of array contents
- Multi-dimensional arrays
  - Create "array of arrays"