C ARRAYS

Deitel 8th Edition, Chapter 6

TOPICS

Review of arrays:

- Definition
- Declaring
- Components
- Accessing Array Elements
- Arrays & Array Elements with Functions

More Examples using Arrays

Sorting and Searching Arrays

Other Types of Arrays

Review of Arrays

Array Definition

A data structure that is a collection of data items of the **same type**, stored in **adjacent memory cells**

Adjacent means right next to each other, one after another

An Array Declaration Has 3 Parts

1. **Data type** of the values that will be stored there
2. A single **identifier** (name) for the group
3. The number of cells (**length or size**) in the group

General Syntax to Declare an Array – Format 1

General format:

```
datatype   array_name [size];
```

Examples:

```
double transactions[8];        //uses numeric constant

int size = 10;
int quiz_grades[size];         //uses numeric variable
```

General Syntax to Declare an Array – Format 2

General format:

datatype array_name[size] = {initialization list};

datatype array_name[] = {initialization list};

Examples:

double discounts[5] = {0.10, 0.12, 0.15, 0.20, 0.25};

double discounts[] = {0.10, 0.12, 0.15, 0.20, 0.25};

Every Array has Six Components

1. Array identifier
2. Length or size
3. Elements
4. Indexes (also called subscripts)
5. Values
6. Location or address in memory

COMPONENTS OF AN ARRAY – ARRAY IDENTIFIER

Array declaration:

int listOfNums[5];

The entire group of five adjacent cells can be referred to using the identifier **listOfNums**:

listOfNums

COMPONENTS OF AN ARRAY – LENGTH

Array declaration:

int listOfNums[5];

The length (or size) of this array is 5. Once an array is declared, its length cannot be changed.

listOfNums

Components of an Array – Elements

Array declaration:

int listOfNums[5];

Each individual memory cell in this array is called an **element** of the array:

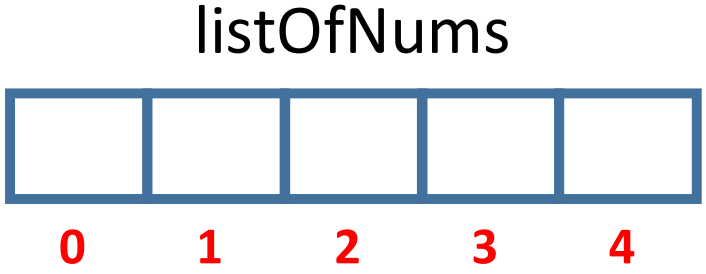
listOfNums

Components of an Array – Indexes or Subscripts

Array declaration:

int listOfNums[5];

Each element of the array is identified with a number that indicates its distance from the beginning of the array:

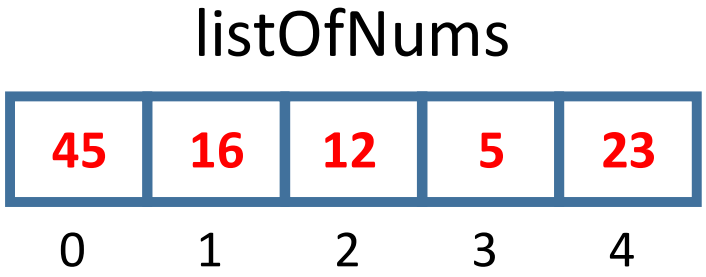


COMPONENTS OF AN ARRAY – VALUES

Array declaration:

int listOfNums[5];

The elements of the array may contain data:

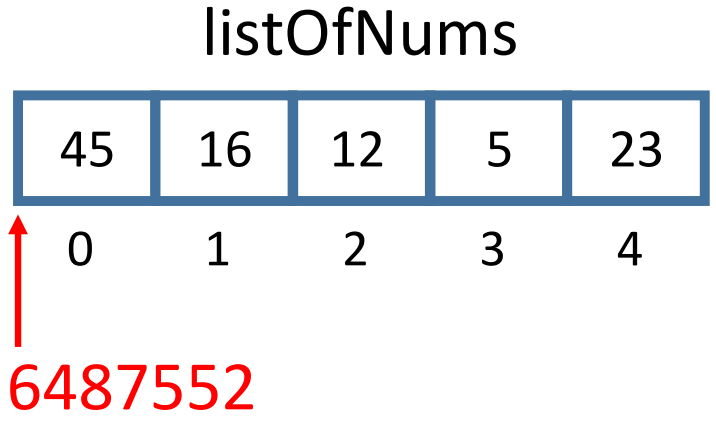


Components of an Array – Location or Address in Memory

Array declaration:

int listOfNums[5];

This array begins at some location – or address – in memory. We'll talk more about this next week.



Array Element Subscripts vs. Array Element Values

Element's **value**

- The **data** stored in that element

Element's **subscript** (or index)

- The number that identifies the element's **location** in the array, as a distance from the beginning

Subscripted Variable = Element of an Array

To identify one element of an array, use a subscripted variable

Subscripted variable

- The array name followed by subscript in square brackets
- Also called **indexed array name**

Can be used anywhere a single variable of that data type can be used

Can be used on either side of an assignment statement

- Variables/values like this are called **lvalues**
- We'll talk more about this another time…

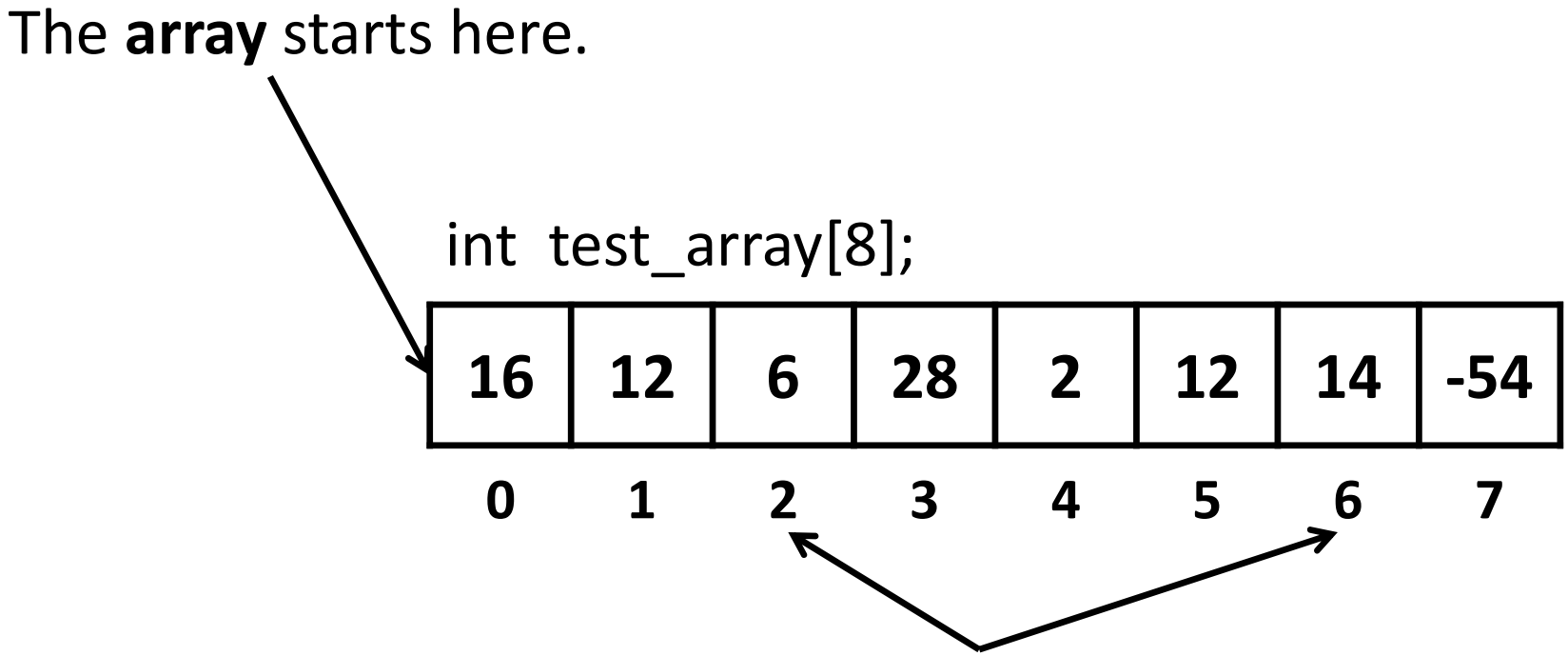
SUBSCRIPTS INDICATE DISTANCE

The subscript of an element is its
distance from the beginning
of the array

Also called the **offset**

Range of valid subscript values **is 0 to array length minus one**

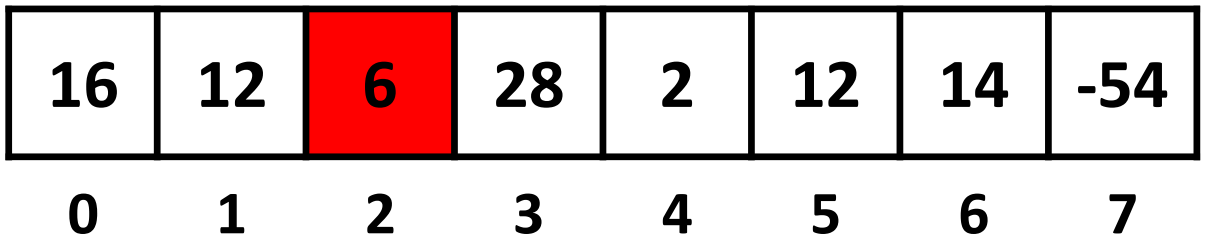
Subscripts Indicate Distance – Diagram of an Array



Subscripts Indicate Distance – Example, pt. 1

Consider this element, which has a value of 6.

int test_array[8];

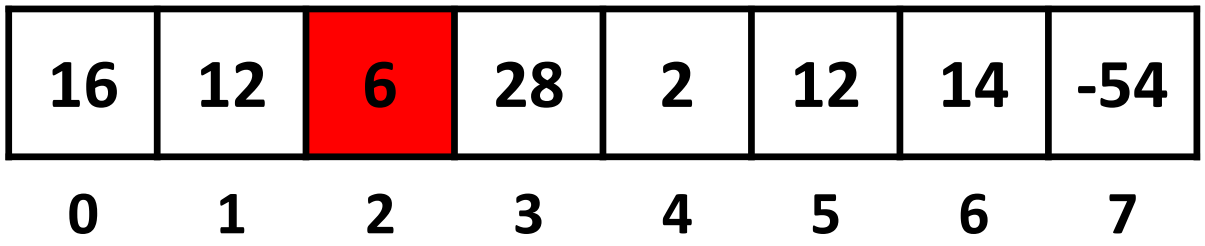


How many other elements are **between this one and the beginning** of the array?

Subscripts Indicate Distance – Example, pt. 2

Consider this element, which has a value of 6.

int test_array[8];



There are **2** elements between this one and the beginning of the array, so this element is at **subscript 2.** It is the **3rd element** in the array.

INITIALIZING ARRAY ELEMENTS WITH CALCULATIONS

Fig06_05.c

- Initializing the elements of array to the even integers from 2 to 20

SUMMING THE ELEMENTS OF AN ARRAY

Fig06_06.c

- Computes the sum of the elements of an array

Can we rewrite this example using functions?

USING ARRAYS TO SUMMARIZE SURVEY RESULTS

Fig06_07.c

- Uses arrays to summarize the results of data collected in a survey

Note the increment operator in line 25

GRAPHING ARRAY ELEMENT VALUES WITH HISTOGRAMS

Fig06_08.c

- Reads numbers from an array and graphs the information in the form of a bar chart or histogram

Note use of **nested for statements**

Using Array Elements as Function Arguments

Can use **array elements** as **inputs to functions** just like using simple variables

Elements within the array will not have their values changed (same as using simple data type variables)

- Because they are **passed by value**

Array elements passed to printf and scanf are used the same way as non-array variables

Array Elements as Arguments – Example

Example function prototype:

```
void print_score (int score);
```

Example function call:

```
int  array_of_scores[3] = {10, 45, 72};

print_score ( array_of_scores[2] );
```

PASSING ARRAYS TO FUNCTIONS

Can also pass **entire arrays** to functions

Important note!

When an entire array is passed to a function,
the **function has access to the original array, NOT a copy**

Entire arrays passed "by reference"

- It's really being passed by value, but we'll see more about this in chapter 7

Passing Arrays to Functions – Setting it Up

In function **header**:

- To indicate a parameter is an array, use **empty brackets**
- No size inside the brackets!!

In function **prototype**:

- To indicate a parameter is an array, use **empty brackets**
- No size inside the brackets!!

In function **call**:

- Use the array's name **without any brackets at all**

Arrays in C Do Not Know Their Own Size

The size of an array must be indicated somewhere:

- As a value stored in a variable
- Indicated with a symbolic constant
- Hard-coded where needed (as in a loop)

Which technique is best?

Indicate Size of Array with a Separate Parameter

When passing an array to a function, always indicate the size of the array with another parameter

- Unless a symbolic constant is available

Example:

void printArray (int arrayName[], int arraySize)

PASSING ARRAYS TO FUNCTIONS EXAMPLE

Fig06_13.c

- Passes arrays and individual array elements to functions.

Function's parameter list must specify that an array will be received (line 47):

void modifyArray(int b[], size_t size)

Size of the array is **not included between the array brackets**

Size must be **included as a separate parameter**

Returning an Array Result

Cannot use an array as a return type

SORTING ARRAYS

Sorted Arrays Can Increase Efficiency

Some programs run more efficiently if the data is sorted first

There are many sorting algorithms

Bubble sort

- Smaller values gradually “bubble” their way upward to the top of the array like air bubbles rising in water, while the larger values sink to the bottom of the array

Fig06_15.c

BUBBLE SORT

Think of a vertical array with element at index 0 at the top & element at index N at the bottom

- N = 4 in this example

We want the lowest value (Draco) to end up at the top, and the highest value (Ron) at the bottom

Working from the "bottom" up, compare each element to the one above it, and swap them if necessary, until we reach the top

Restart from the bottom, until everything is in place

Bubble Sort First Pass

Last two items at index N – 1 and index N are compared, and swapped if necessary

N – 1 = 3

N = 4

Should Neville be swapped with Hermione?

Bubble Sort First Pass continued

Yes, because Hermione is closer to the beginning of the alphabet

How should we swap these?

We want Hermione to end up at the index where Neville is (3), and Neville to end up at the index where Hermione is (4)

How is Swapping Performed? – Would this work?

Would this work?

names[3] = names[4]

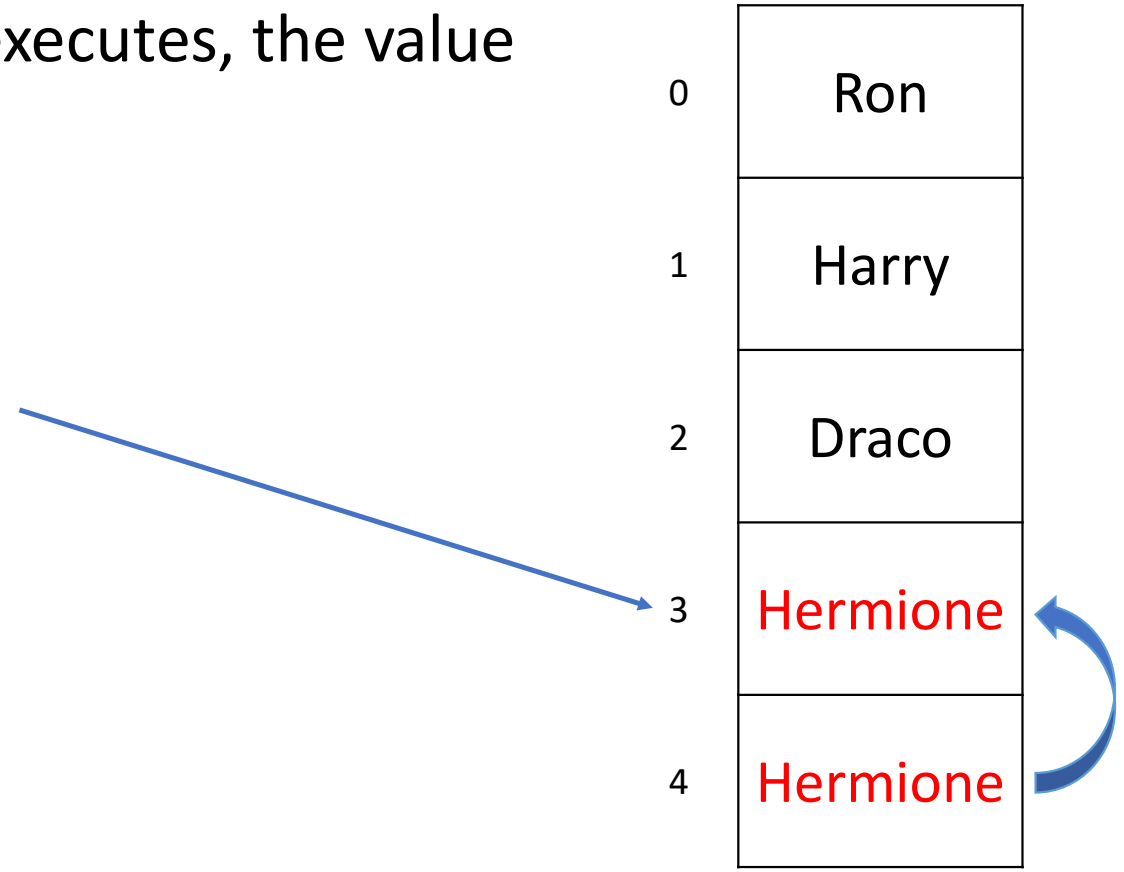
names[4] = names[3]

How is Swapping Performed? No

No. When the first statement executes, the value in names[3] is overwritten

names[3] = names[4]

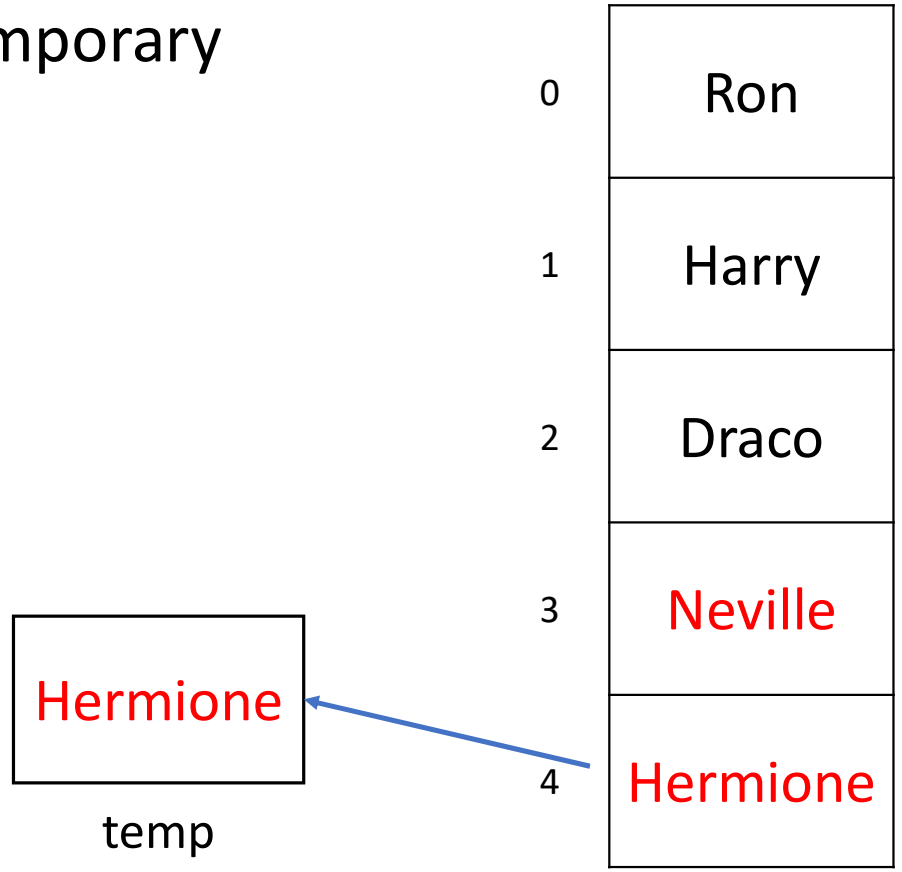
names[4] = names[3]



How Is Swapping Performed? – 3rd Variable

We need a third variable to be a temporary holding area:

temp = names[4]

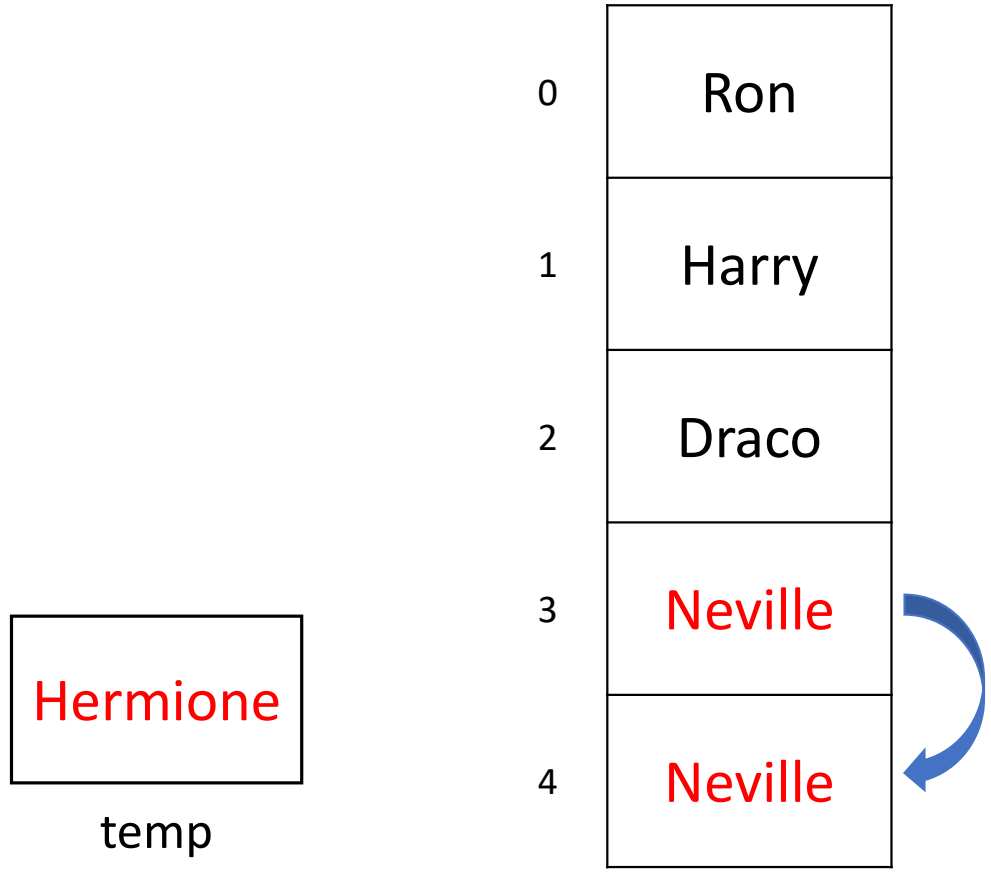


How is Swapping Performed? – Copy One Variable

Then copy one value:

temp = names[4]

names[4] = names[3]



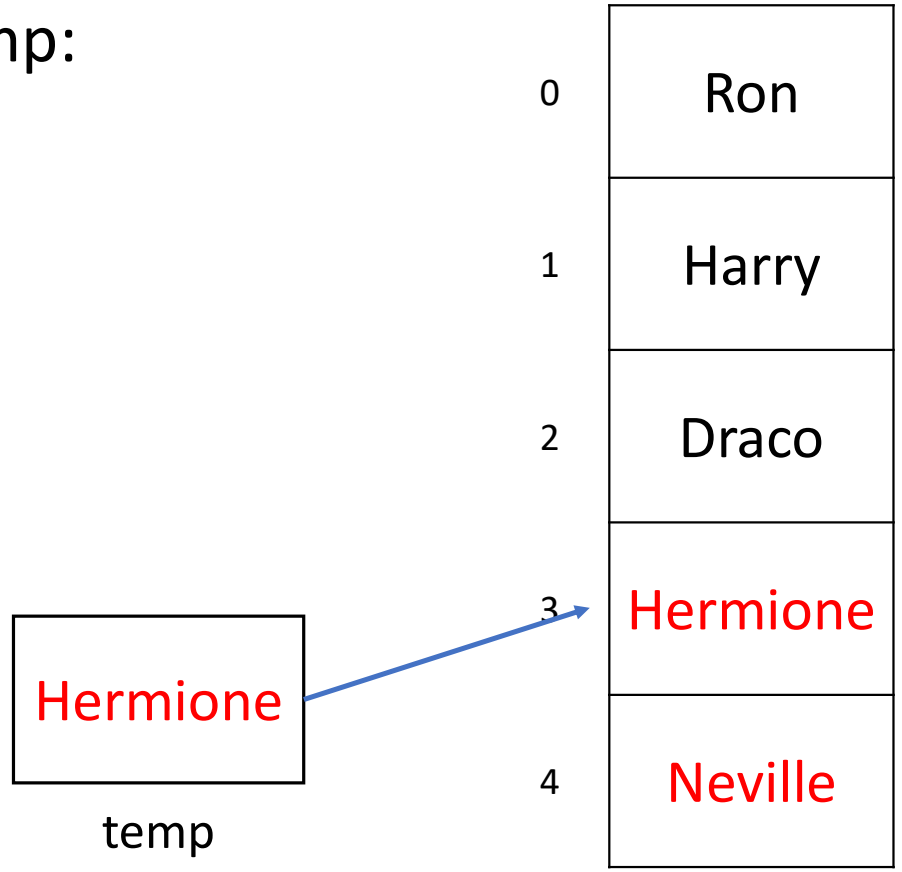
How is Swapping Performed? – Copy the Other Variable

Then copy the other value from temp:

temp = names[4]

names[4] = names[3]

names[3] = temp



Back to Bubble Sort…

Next two items N – 2 and N – 1 are compared, and swapped if necessary

N – 2 = 2

N – 1 = 3

Should Draco be swapped with Hermione?

Bubble Sort First Pass 1

No, so we don't swap, and continue with the algorithm

Bubble Sort First Pass 2

Next two items N – 3 and N – 2 are compared, and swapped if necessary

N – 3 = 1

N – 2 = 2

Should Harry be swapped with Draco?

Bubble Sort First Pass 3

Yes, because Draco is closer to the beginning of the alphabet

Bubble Sort First Pass 4

Next two items N – 4 and N – 3 are compared, and swapped if necessary

N – 4 = 0

N – 3 = 1

Should Ron be swapped with Draco?

Bubble Sort First Pass 5

Yes, because Draco is closer to the beginning of the alphabet

Bubble Sort First Pass 6

Once the top (beginning) of the array is reached, we stop this pass.

The top element in the array now contains the lowest value in the array

It has “bubbled up” to the top

Bubble Sort Second Pass 1

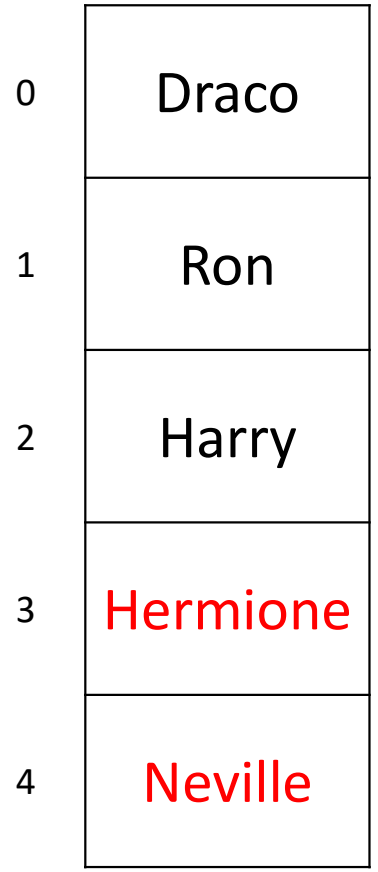
We begin a second pass through the array, repeating the swapping process from the last index N

Last two items at index N – 1 and index N are compared, and swapped if necessary

N – 1 = 3

N = 4

Should Neville be swapped with Hermione?



Bubble Sort Second Pass 2

No, so we don't swap, and continue with the algorithm

Bubble Sort Second Pass 3

Next two items N – 2 and N – 1 are compared, and swapped if necessary

N – 2 = 2

N – 1 = 3

Should Harry be swapped with Hermione?

Bubble Sort Second Pass 4

No, so we don't swap, and continue with the algorithm

Bubble Sort Second Pass 5

Next two items N – 3 and N – 2 are compared, and swapped if necessary

N – 3 = 1

N – 2 = 2

Should Ron be swapped with Harry?

Bubble Sort Second Pass 6

Yes, because Harry is closer to the beginning of the alphabet

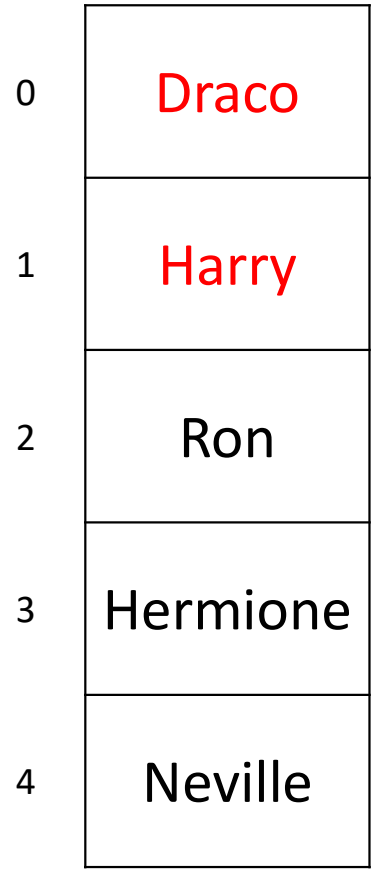
Bubble Sort Second Pass 7

Next two items N – 4 and N – 3 are compared, and swapped if necessary

N – 4 = 0

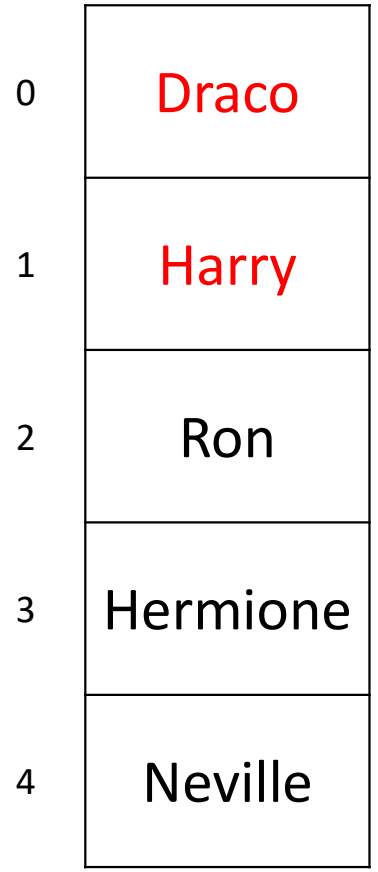
N – 3 = 1

Should Draco be swapped with Harry?



Bubble Sort Second Pass 8

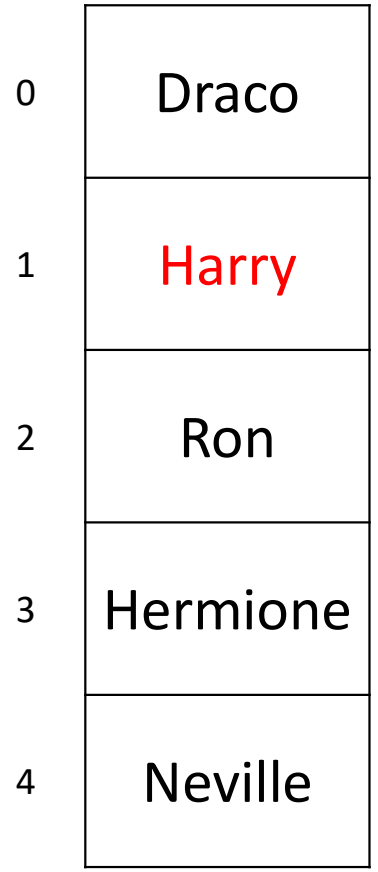
No, so we don't swap, and continue with the algorithm



Bubble Sort Second Pass 9

We've reached the end of the second pass

The second pass through the algorithm gets the second-lowest value into its correct position



Bubble Sort Third Pass 1

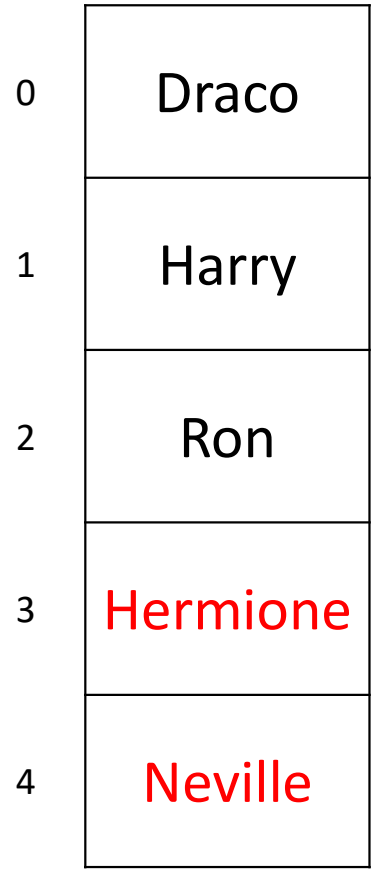
We begin a third pass through the array, repeating the swapping process from the last index N

Last two items at index N – 1 and index N are compared, and swapped if necessary

N – 1 = 3

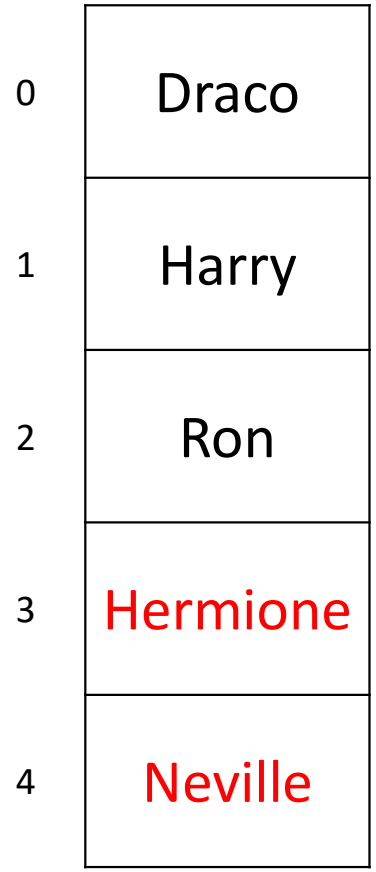
N = 4

Should Hermione be swapped with Neville?



Bubble Sort Third Pass 2

No, so we don't swap, and continue with the algorithm



Bubble Sort Third Pass 3

Next two items N – 2 and N – 1 are compared, and swapped if necessary

N – 2 = 2

N – 1 = 3

Should Ron be swapped with Hermione?

Bubble Sort Third Pass 4

Yes, because Hermione is closer to the beginning of the alphabet

Bubble Sort Third Pass 5

Next two items N – 3 and N – 2 are compared, and swapped if necessary

N – 3 = 1

N – 2 = 2

Should Harry be swapped with Hermione?

Bubble Sort Third Pass 6

No, so we don't swap, and continue with the algorithm

Bubble Sort Third Pass 7

Next two items N – 4 and N – 3 are compared, and swapped if necessary

N – 4 = 0

N – 3 = 1

Should Draco be swapped with Harry?

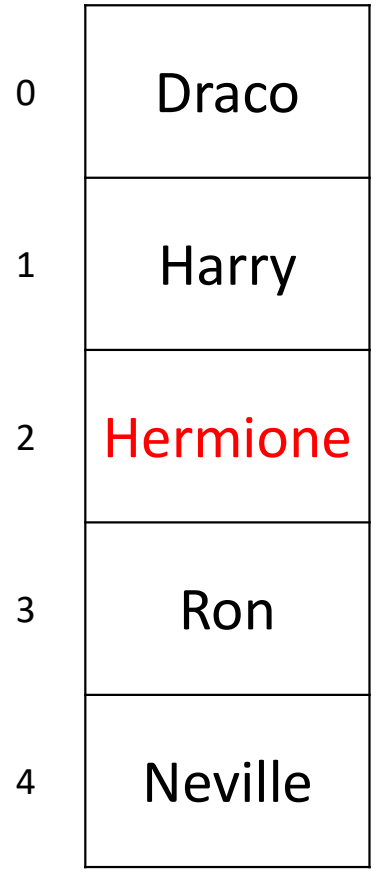
Bubble Sort Third Pass 8

No, so we don't swap, and continue with the algorithm

Bubble Sort Third Pass 9

We've reached the end of the third pass

The third pass gets the third-lowest value into its correct position



Bubble Sort Fourth Pass 1

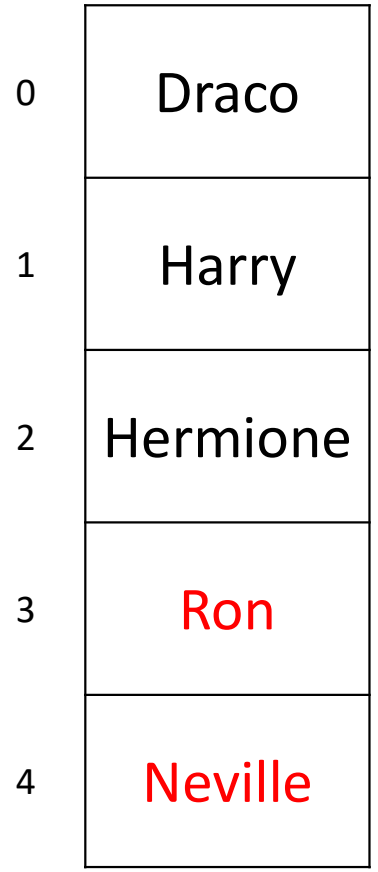
We begin a fourth pass through the array, repeating the swapping process from the last index N

Last two items at index N – 1 and index N are compared, and swapped if necessary

N – 1 = 3

N = 4

Should Ron be swapped with Neville?



Bubble Sort Fourth Pass 2

Yes, because Neville is closer to the beginning of the alphabet

Bubble Sort Fourth Pass 3

Next two items N – 2 and N – 1 are compared, and swapped if necessary

N – 2 = 2

N – 1 = 3

Should Hermione be swapped with Neville?

Bubble Sort Fourth Pass 4

No, so we don't swap, and continue with the algorithm

Bubble Sort Fourth Pass 5

Next two items N – 3 and N – 2 are compared, and swapped if necessary

N – 3 = 1

N – 2 = 2

Should Harry be swapped with Hermione?

Bubble Sort Fourth Pass 6

No, so we don't swap, and continue with the algorithm

Bubble Sort Fourth Pass 7

Next two items N – 4 and N – 3 are compared, and swapped if necessary

N – 4 = 0

N – 3 = 1

Should Draco be swapped with Harry?

Bubble Sort Fourth Pass 8

No, so we don't swap, and continue with the algorithm

Bubble Sort Fourth Pass 9

We've reached the end of the fourth pass

The fourth pass gets the fourth-lowest value into its correct position

Bubble Sort Fifth Pass 1

We begin a fifth pass through the array, repeating the swapping process from the last index N

We proceed through every step as before, but this time, no swaps occur, because these items are in order

How do we know no swaps have occurred?

In Bubble Sort, a Swap Means We Need to Keep Sorting

In the code for bubble sort, every time we make a swap between two values, we know that we need to check the sort order again, by performing another pass through the array

If no swaps have occurred, we know the values are in their sorted order, and we can stop the algorithm

How do we know if a swap has occurred or not?

SWAP FLAG

We use a variable called a **flag**

- Also sometimes called a switch
- Used to indicate if some situation has occurred

Often we use two values for this flag: **1** and **0** -or- **True** and **False**

- Whatever makes sense for the algorithm

Set the flag to False at the beginning, and if the situation occurs that we need to check for, we change the flag's value to True

SWAP FLAG IN BUBBLE SORT

The algorithm repeatedly makes passes through the array, and stops when no swaps have occurred

So at the beginning of the algorithm, we set the flag to 1, so that the algorithm will start

Before each sort pass begins, we set this variable to 0

If there is a swap, change the value back to 1

- So that another sorting pass will be made

If there hasn't been a swap, the value of the flag is 0, which tells the algorithm we don't need any more passes

CASE STUDY: COMPUTING MEAN, MEDIAN, MODE

Fig06_16.c

- Uses array initialized with 99 responses to a survey (a number from 1 to 9)
- Program computes the mean, median and mode of the 99 values

Mean

- arithmetic average

Median

- Middle value, so array must be sorted first

Mode

- Value that occurs most frequently

SEARCHING ARRAYS

TWO TYPES OF SEARCH ALGORITHMS

Linear search

Binary search

SEARCHING AN ARRAY

Search target

- The value we are looking for

Linear search

- Examine each array element in order until search target is found
- Uses a loop
- Array does not have to be sorted first

LINEAR SEARCH

Fig06_18.c

- Compares each element of the array with the search key

Because the array is not in any order, it's just as likely that the value will be found in the first element as in the last.

On average, will have to compare the search key with half the elements of the array.

Works well for **small** or **unsorted** arrays

BINARY SEARCH

Linear searching inefficient for large arrays

If array is **sorted**, can use high-speed **binary search**

Eliminates from consideration one-half of the elements in a sorted array after each comparison

BINARY SEARCH ALGORITHM

Locate the middle element of the sorted array and compare it to the search key

If equal

- Search key is found and the array index of that element is returned

If not equal

- If the search key is less than the middle element of the array
  - Search the first half of the array
- Otherwise
  - Search the second half

Fig06_19.c

Other Types of Arrays

Three Other Types of Arrays

Parallel

Multi-dimensional

Variable-length

PARALLEL ARRAYS

A collection of arrays of the same size that hold related data for some group of items

```
int id[NUM_STUDENTS];

double gpa[NUM_STUDENTS];
```

id and gpa are **parallel arrays**

There is no special syntax to declare parallel arrays

PARALLEL ARRAYS – EXAMPLES

int id[10];

index	0	1	2	3	4	5	6	7	8	9
Id	**123**	**451**	**256**	**815**	**578**	**987**	**654**	**258**	**159**	**357**

double gpa[10];

index	0	1	2	3	4	5	6	7	8	9
gpa	**2.6**	**3.2**	**4.0**	**4.0**	**2.8**	**1.5**	**0.5**	**1.9**	**2.7**	**3.7**

Which element has the gpa of the student with ID #123?

What is the gpa of the student at index 4?

Which student ids have the same gpa?

PARALLEL ARRAYS – EXAMPLES – ANSWERS

int id[10];

index	0	1	2	3	4	5	6	7	8	9
Id	**123**	**451**	**256**	**815**	**578**	**987**	**654**	**258**	**159**	**357**

double gpa[10];

index	0	1	2	3	4	5	6	7	8	9
gpa	**2.6**	**3.2**	**4.0**	**4.0**	**2.8**	**1.5**	**0.5**	**1.9**	**2.7**	**3.7**

Which element has the gpa of the student with ID #123? **gpa[0]**

What is the gpa of the student at index 4? **2.8**

Which student ids have the same gpa? **256**, **815**

MULTIDIMENSIONAL ARRAYS

Useful for data that can be pictured as a table with rows and columns

Needs two indices

The first identifies the element's **row**

- Still following the rules of subscripts

The second identifies the element's **column**

- Still following the rules of subscripts

DECLARING MULTIDIMENSIONAL ARRAYS

char tictactoe_board[3][4];

Creates an array with 3 rows and 4 columns

Accessing Elements of a 2-Dimensional Array

Access each element with two subscripts, using two sets of brackets:

tictactoe_board[0][2]

Logic error: **a[x, y]** instead of **a[x][y]**

C interprets **a[x, y]** as **a[y]**

- Comma in this context is treated as a **comma operator**
- Not a syntax error

INITIALIZATION OF MULTIDIMENSIONAL ARRAYS

Data grouped by rows using {}

```
char init_soduku [3][3] = {
                              {'7', ' ', '   '},
                              {'6', '2', '   '},
                              {'   ', '   ', '3'}
                          };
```

Initializing a Double-Subscripted Array

Fig06_21.c

If there are not enough initializers for a given row:

- Remaining elements of that row are initialized to 0

```
int b[2][2] = {{1}, {3, 4}};
```

Initializes:

b[0][0] to 1
b[0][1] to 0
b[1][0] to 3
b[1][1] to 4

1	0
3	4

Multidimensional Arrays Passed to Functions

In a parameter, the first element (only) can be omitted:

- Where are parameters located?

```
char tictactoe_board[][3];

char tictactoe_board[3][3];

char tictactoe_board[][3][3];
```

Arrays with Several Dimensions

Use lots of memory

To access in order, need a loop for each dimension

VARIABLE-LENGTH ARRAYS

What if you cannot determine an array's size until execution time?

Variable-length arrays (VLAs)

- Arrays whose lengths are defined in terms of expressions evaluated at execution time

Valid if the variables representing the array sizes are of an integral type

Fig06_23.c

- Declares and prints several VLAs

SIZEOF **OPERATOR**

Line 34: **sizeof(array);**

Gives the size of the array in **bytes**

Can be used for any data type