Introduction to Arrays

An array is a data structure used to store a collection of data items all of the same type.

The name of the array is associated with the collection of data. To access an individual data item, you need to indicate to the computer which array element you want. This is indicated using an array **index** (or **subscript**).

Why are arrays useful? Suppose you want to write a program which accepts 5 integers input by the user, and prints them out in reverse order. You could do it like this:

```
int first, second, third, fourth, fifth;
printf("enter 5 integers, separated by spaces: ");
scanf("%d %d %d %d %d", &first, &second, &third, &fourth, &fifth);
printf("in reverse order: %d, %d, %d, ", fifth, fourth, third);
printf("%d, %d\n", second, first); /* output is all on 1 line */
```

This works as required. But – what if you had 50 inputs? Or 500?! Or… Using integer variables would become very cumbersome…

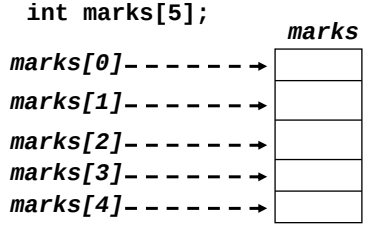
Introduction to Arrays (contd.)

A **variable** holds a **single** value, e.g.

```
int mark;
```

An **array** holds **several** values, all of the same type, e.g.

```
int marks[5];
```



General array declaration:
`type arrayname[size]`
where `size` is an ***integer-valued***
- ***constant***, or
- ***variable***, or
- ***expression***

- The computer provides a **contiguous** block of memory for storing the array.
- All array elements have the same type (`int`, `float`, `char`, …)
- Many algorithms require data to be collected and stored, and later processed in some way(s). If data is ***all of the same type***, an array is the natural way to do this.
- Can access values using `marks[3]`, `marks[i]`, `marks[j*k]`, etc. (index must be integer-valued). An array element can then be treated just like a variable.
- **First** element has index **0** *(not **1**)*. **Last** element has index **size-1** *(not **size**)*.

Introduction to Arrays (contd.)

To solve the reversed-inputs problem using an array:

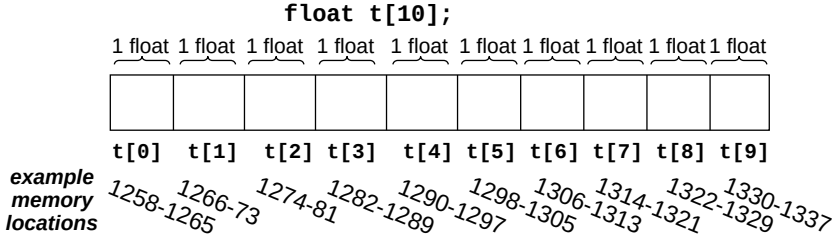
```
int i; /* loop counter */
int table[5];
/* declares an array "table" with storage for 5 integers */
for (i=0; i<=4; i++) {
        printf("enter integer %d: ", i+1);
        /* "i+1" so user sees count starting at 1 */
        scanf("%d", &table[i]);
}
printf("in reverse order: ");
for (i=4; i>=0; i--) {
       printf("%d ", table[i]); /* last element is table[4], etc */
}
```

Produces the screen output *(user inputs in italics)*:

```
enter integer 1: 3
enter integer 2: 4
enter integer 3: 5
enter integer 4: 6
enter integer 5: 7
in reverse order: 7 6 5 4 3
```


More details on arrays

- "Scalar" variables hold a single value: `float temperature;`
- "Array" variables hold a collection of values: `float temperatures[NCITIES];`
- When an integer variable is declared a sufficient amount of memory (e.g. 2 bytes) is reserved for it.
- Similarly when an array is declared a sufficient amount of memory is reserved. If each `float` value requires 8 bytes, an array of 100 `floats` takes up 800 bytes.
- The compiler takes care of these details for you -- but it's important you know them.



- ***C does not check or enforce array bounds*** – it's up to you not to access memory locations that are not part of the array.
  - Get junk/error if you try to access `t[10]`, `t[-1]`, etc.

Array initialization: examples

```
#include "stdio.h"
void main(void){
  int i;
  int somearray[5] = {5,6,7,8,9}; /* initialisation statement */
  for (i = 0; i <= 4; i++){
    printf("element number %d is %d\n", i, somearray[i]);
  }
}
```

Produces the screen output:

```
element number 0 is 5
element number 1 is 6
element number 2 is 7
element number 3 is 8
element number 4 is 9
```

What if initialisation statement is changed to:

```
int j=5, k=9; int somearray[5] = {j,j+1,j+2,k-1,k};

int somearray[5]={5,6,7};

int somearray[] = {5,6,7,8,9};

int somearray[5];
```


A simple use of arrays

```
#include <stdio.h>
#define NSTUDENTS 5
#define BONUS 10
void main(void) {
  int i, marks[NSTUDENTS], total=0;
  float average;
  for (i = 0; i < NSTUDENTS; i++) {
    printf("Enter mark for student %d: ", i);
    scanf("%d", &marks[i]); /* read in all the marks */
  }
  for (i = 0; i < NSTUDENTS; i++) {
    marks[i] = marks[i] + BONUS; /* everyone gets a bonus */
    total += marks[i]; /* get sum of all marks */
  }
  average = total / (float) NSTUDENTS; /* cast to float */
  for (i = 0; i < NSTUDENTS; i++) {
    printf("Student %d's mark deviates from average", i);
    printf(" by %.2f\n", marks[i]-average);
  }
}
```

- note **`for`** loop – very useful for dealing with arrays
  - how would you write this with **`while`** loops instead of **`for`**?
- note that **`arrayname[index]`** can be treated just like a regular variable
  - in this case, **`marks[i]`** treated just like a variable of type **`int`**

A simple use of arrays – running the program

```
Enter mark for student 0: 40
Enter mark for student 1: 35
Enter mark for student 2: 61
Enter mark for student 3: 77
Enter mark for student 4: 28
Student 0's mark deviates from average by -8.20
Student 1's mark deviates from average by -13.20
Student 2's mark deviates from average by 12.80
Student 3's mark deviates from average by 28.80
Student 4's mark deviates from average by -20.20
```

(with the above marks, the adjusted marks – when the bonus 10 marks are given to all students – are 50, 45, 71, 87, and 38. The average of these adjusted marks is 58.2, so the students' deviations from this average are –8.2, –13.2, 12.8, 28.8, and –20.2, which agrees with the above program output)

Yet Another Example

A program to read in an array of 5 exam marks and calculate the highest mark:

```
#include <stdio.h>
void main(void) {
  int index, marks[5];
  int high_index = 0;
  for (index = 0; index < 5; index++) {
    printf("enter mark no. %d: ", index);
    scanf("%d", &marks[index]);
  }
  for (index = 1; index < 5; index++) { /* NOTE 1 */
    if (marks[index] > marks[high_index]) {
      high_index = index;
    }
  }
  printf("The highest mark was %d, ", marks[high_index]);
  printf("obtained by student %d\n", high_index);
}
```

Things for you to think about:

NOTE 1: why **`index = 1`** and not **`index = 0`** ?
Also: what happens here if 2 or more students get the same "highest" mark?

Yet Another Example – running the program

```
enter mark no. 0: 70
enter mark no. 1: 55
enter mark no. 2: 71
enter mark no. 3: 29
enter mark no. 4: 44
The highest mark was 71, obtained by student 2


enter mark no. 0: 60
enter mark no. 1: 60
enter mark no. 2: 60
enter mark no. 3: 60
enter mark no. 4: 60
The highest mark was 60, obtained by student 0
```


Linear Search

A common task: you have an array, and you need to find a specific element.
Technique: "search" -- systematically examine the array to find the element
Simplest kind of search: linear search

Example: two arrays: **`patientIDs[]`** and **`ages[]`**

```
int patientIDs[5000];
int ages[5000];
/* fill up these arrays somehow (read from file, keyboard,...) */
```

Question: Is there a 100-year-old patient?

```
int i;
for (i = 0; i < 5000; i++) {
     if (ages[i] == 100) {
            printf("yes\n");
     }
}
```


Linear Search (contd.)

Problems with previous program: never print "no"; don't know which patient is 100, if there is one; check every element even after we know the answer!

Solution: use a **`while`** loop, stop loop early if find what you're looking for; then test value of counter **`i`** after program execution

```
int i = 0;
while ((i < 5000) && (ages[i] != 100)) {
     i++; /* keep going while unsuccessful */
}
if (i == 5000) { /* did we search the whole array? */
     printf("no hundred-year-old patient\n");
 } else { /* stopped early, therefore success */
     printf("patient number %d", i);
     printf(" (ID %d) is 100\n", patientIDs[i]);
}
```


Linear Search: another example

Find the age of the patient whose ID is 347822

```
int targetPatientID, targetAge, i;
printf("What patient ID are you looking for: ");
scanf("%d", &targetPatientID);
for (i = 0; i < 5000; i++) {
  if (patientIDs[i] == targetPatientID) {
    targetAge = ages[i]; /* found target patient */
    i = 5000; /* force loop to exit early */
  }
} /* if target not found, loop runs to completion */
printf("Patient %d's age is %d\n", targetPatientID, targetAge);
```

MORAL of the story: the loop control variables can be used to do something besides merely control the number of iterations, e.g. to control the number of iterations in a "special" way.

Problem with the above program: if target patient not found, print out result (with rubbish value for **`targetAge`***) anyway. How would you fix this?*

Passing 1D arrays to functions

```
/* a general-purpose averaging function */
float average(float array[], int size) {
 int i;
 float sum = 0;
 for (i = 0; i < size; i++) {
   sum += array[i];
 }
 return sum/size;
}

void main(void) {
 float temperatures[20];
 /* somehow fill in temperatures array */
 float marks[50];
 /* somehow fill in marks array */
 printf("avg. temp.=%.2f", average(temperatures, 20));
 printf("avg. mark=%.2f", average(marks, 50));
}
```

leave size unspecified in formal parameter; just give array name as actual parameter…

*… but **can** also pass in size as separate argument, to make sure the function only accesses valid indices*

Q: under what assumption could you omit the size of the array as a parameter of the called function?

More Examples -- Passing 1D arrays to functions

```
/* returns sum of elements A[0] … A[size-1] */
int sum(int A[], int size) {
  int i, tot = 0;
  for (i=0; i<size; i++) {tot += A[i];}
  return tot;
}

/* returns the maximum value in A */
int max(int A[], int size) {
  int m = A[0], i;
  for (i=1; i<size; i++) {if (A[i]>m) {m=A[i];}}
  return m;
}

/* count the number of times a given "target" occurs in A */
int count(int A[], int size, int target) {
  int n = 0, i;
  for (i=0; i<size; i++) {if (A[i]==target) {n++;}}
  return n;
}

/* return index of the first occurrence in A, or -1 if not found */
int find(int A[], int size, int target) {
  int i;
  for (i=0; i<size; i++) {if (A[i]==target) {return i;}}
  return -1; /* target not found, since for loop completed */
}
```


Arrays & function calls

By default, "simple" data types (`int`, `char`, `double`, …) are passed by value.

In contrast, arrays are passed by address. For example:

```
#include <stdio.h>
void zap(int array1[], int size) {
  int i;
  for (i = 0; i < size; i++) {
    array1[i] = 0; /* sets all elements to 0 */
  }
}
void main(void) {
  int i, charges[5]={7,8,7,6,7};
  printf("elements of charges[] are ");
  for (i=0;i<5;i++){printf("%d ", charges[i]);}
  zap(charges, 5); /* function call will affect charges */
  printf("\nnow elements of charges[] are ");
  for (i=0;i<5;i++){printf("%d ", charges[i]);}
}
```

`array1[]` in **`zap()`** is another name for **`charges[]`** in **`main()`**

Produces the screen output:

```
elements of charges[] are 7 8 7 6 7
now elements of charges[] are 0 0 0 0 0
```