C ARRAYS

Deitel 8th Edition, Chapter 6

TOPICS

Data Structures

Declaring Arrays

The Components of an Array

Subscripts, Subscripted Variables & Accessing Array Elements

Examples using Arrays

Arrays & Array Elements with Functions

Secure C Programming

DATA STRUCTURES

Dealing with Data

Up to now, we've only accessed data in a single memory cell at a time

What is one way we've used to locate these memory cells?

Dealing with Data Requires Variables

Up to now, we've only accessed data in a single memory cell at a time

What is one way we've used to locate these memory cells?

We've used variables for each memory cell, even if they are related.

Dealing with Lots of Data

But what if we have a lot of data to store, like 10 quiz grades or 100 quiz grades?

Using single memory cells can become very inefficient for handling a **group of related data**

Data Structures are Groups of Data

Data structure

- Related data is grouped, stored and accessed using one identifier
- Many different types – we'll see another later this semester

Array

- 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

Array element

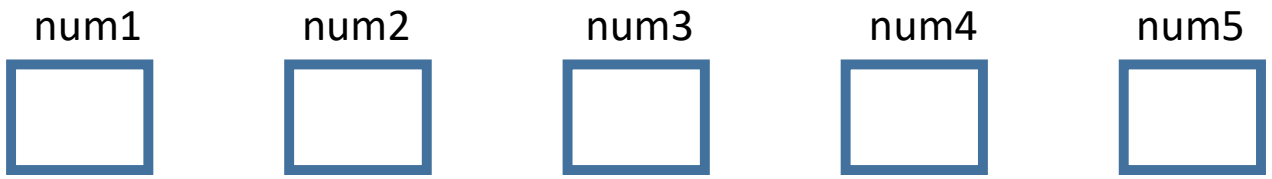
- A single data item that is part of array

EXAMPLE: STORING FIVE INTEGERS IN SEPARATE MEMORY CELLS

Without arrays, we need five separate variables:

int num1, num2, num3, num4, num5;

Each is a separate cell in memory, accessed by a separate identifier:

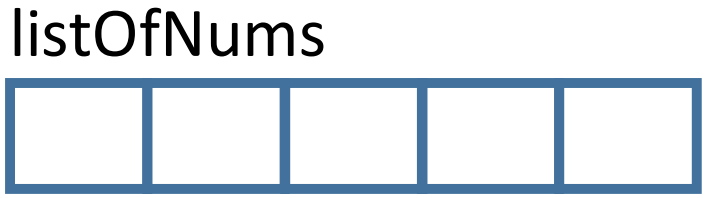


Example: Storing Five Integers in an Array

Using an array, we need one variable that has many parts:

int listOfNums[5];

Which creates cells in memory like this:



DECLARING ARRAYS

Arrays Must Be Declared

Arrays occupy space in memory

Each separate cell in this memory space must be the same data type

We need a name for the group of cells

We need to indicate how many cells there are

Data type of each element and number of elements must be specified so correct amount of memory is allocated

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};

ARRAY INITIALIZATION LIST

Can omit size only when initializing via an **initialization list**

- A list of values within curly braces

Size of the array will be computed automatically, based on the number of items in the list

Examples:

```
int temps[] = {77, 68, 80, 55};

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

ARRAY INITIALIZATION – CHAR ARRAY

When initializing an array of chars, there are two syntax options:

```
char vowels[] = { 'A', 'E', 'I', 'O', 'U' };
```

-or-

```
char vowels[] = "AEIOU";
```

Note that the 2nd version uses double quotes

THE COMPONENTS OF AN ARRAY

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

Array Declaration Example

Remember this array declaration:

int listOfNums[5];

Which reserves five adjacent cells 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:

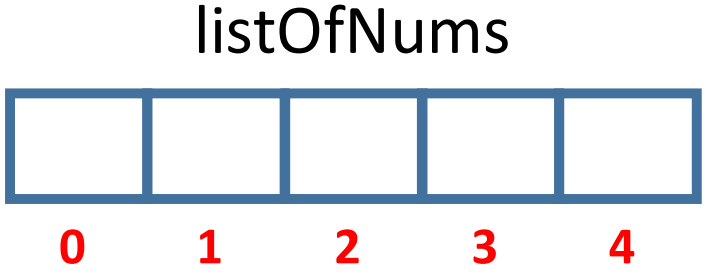
listOfNums

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:

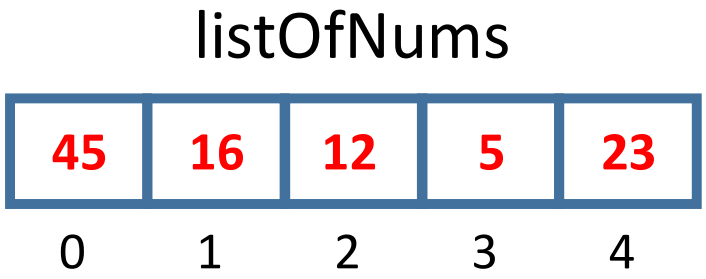


COMPONENTS OF AN ARRAY – VALUES

Array declaration:

int listOfNums[5];

The elements of the array may contain data:

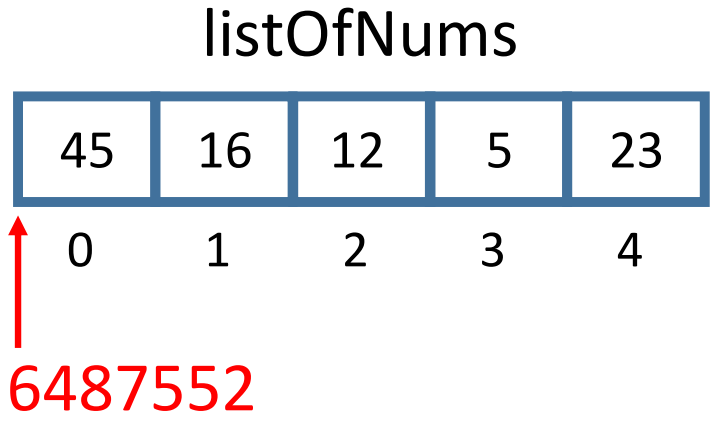


Components of an Array – Location or Address in Memory

Array declaration:

int listOfNums[5];

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



Subscripts, Subscripted Variables & Accessing Array Elements

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…

THE SUBSCRIPT IN A SUBSCRIPTED VARIABLE

Can be any integral value: a **literal**, an **integer variable**, or an **expression** with an integral value

Example of an integral literal: **arrayName[5]**

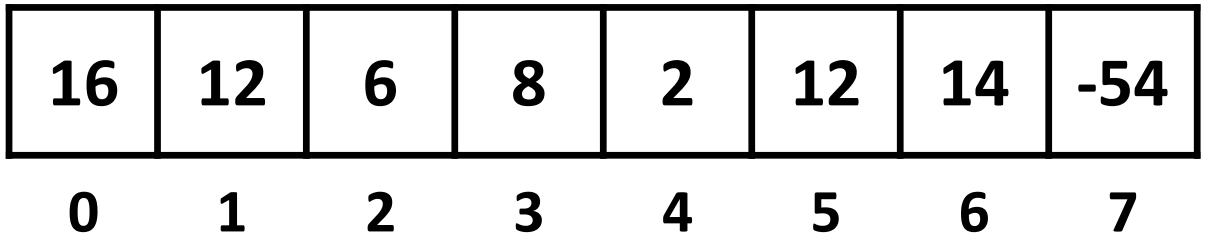
Example of an integer variable: **arrayName[x]**

- x must be an integer

Example of an expression: **arrayName[x + 1]**

USING ARRAY ELEMENTS – EXAMPLES, PT. 1

int test_array[8];



If **i = 4**, what is:

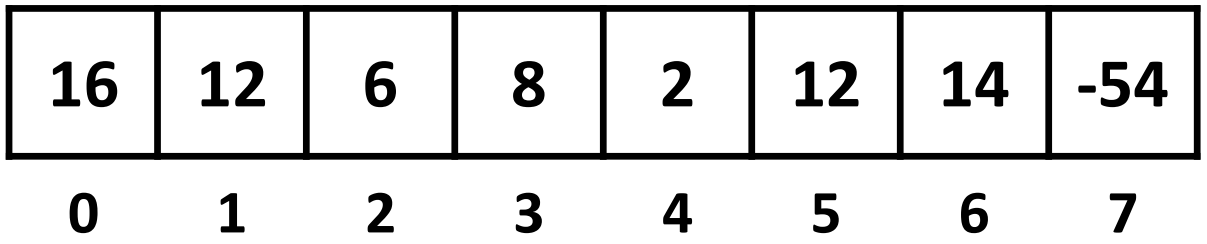
test_array[4]

test_array[i] + 1

test_array[i++]

USING ARRAY ELEMENTS – EXAMPLES, PT. 1 – RESULTS

int test_array[8];



If **i = 4**, what is:

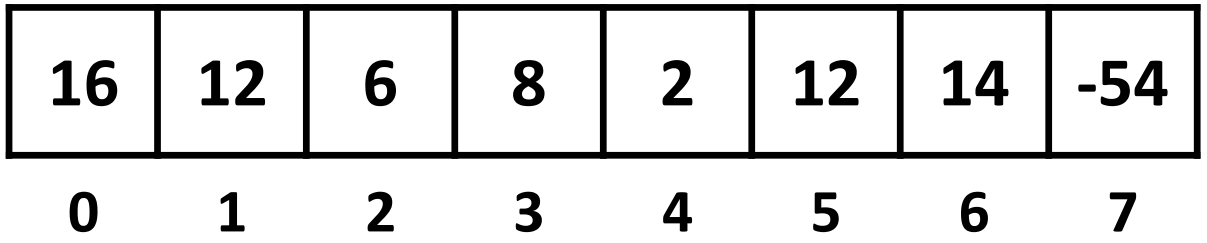
test_array[4] = **2**

test_array[i] + 1 = **3**

test_array[i++] = **2** (and then i is incremented to **5**)

USING ARRAY ELEMENTS – EXAMPLES, PT. 2

int test_array[8];



If **i = 4**, what is:

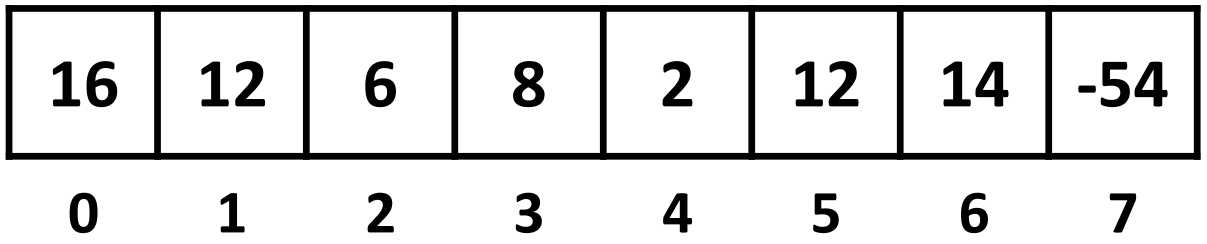
test_array[i]

test_array [i + 2]

test_array[-- i]

USING ARRAY ELEMENTS – EXAMPLES, PT. 2 – RESULTS

int test_array[8];



If **i = 4**, what is:

test_array[i] = **2**

test_array [i + 2] = **14**

test_array[-- i] = **8** (i is decremented to **3** first)

USING ARRAY ELEMENTS – EXAMPLES IN CODE

testingsubscripts.c

USING SUBSCRIPTED VARIABLES EXAMPLES 1

Example	What does it mean?
cost[2]	Refers to the 3rd element in the array **cost**
class[0]	
cost[3] = 12.57;	
class[1] = 23;	

USING SUBSCRIPTED VARIABLES EXAMPLES 2

Example	What does it mean?
cost[2]	Refers to the **3rd element** in the array **cost**
class[0]	Refers to the **1st element** in the array **class**
cost[3] = 12.57;	Sets the **4th element** in the array **cost** to 12.57
class[1] = 23;	Sets the **2nd element** in the array **class** to 23

USING SUBSCRIPTED VARIABLES EXAMPLES 3

Example	What does it mean?
data[3] = data[3] + 20;	
class[1]++;	
printf(“%d”, data[2]);	
scanf(“%lf”, &list[4]);	

USING SUBSCRIPTED VARIABLES EXAMPLES 4

Example	What does it mean?
data[3] = data[3] + 20;	Sets the **4th element** in the array **data** to the sum of the 4th element and 20
class[1]++;	Adds 1 to the value in the **2nd element** in the array **class**
printf("%d", data[2]);	Prints the **3rd element** in the array **data**
scanf("%lf", &list[4]);	Reads a value into the **5th element** of the array **list**

SUBSCRIPTS INDICATE DISTANCE

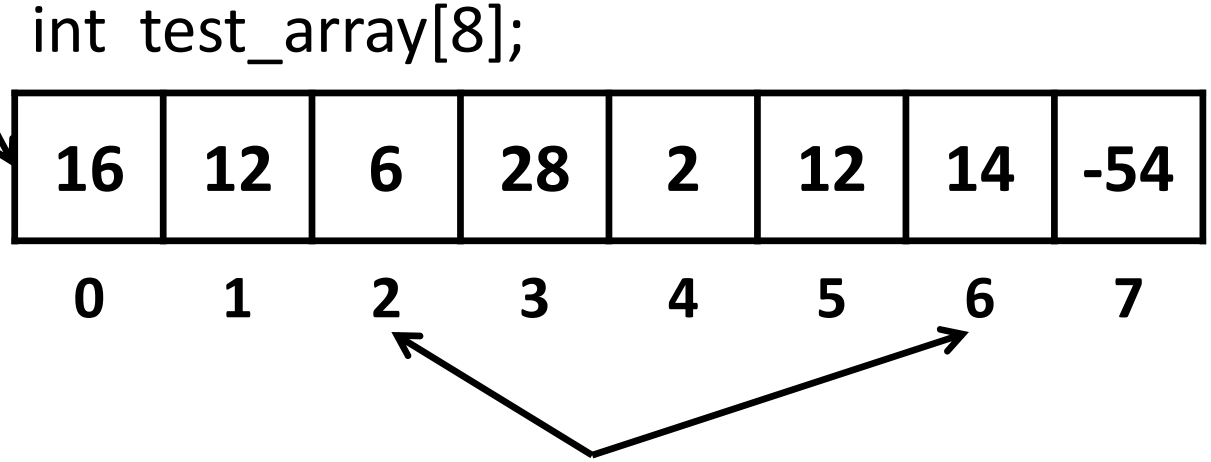
The subscript of an element is its
distance from the beginning
of the array

Also called the **offset**

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

Subscripts Indicate Distance – Diagram of an Array

The **array** starts here.

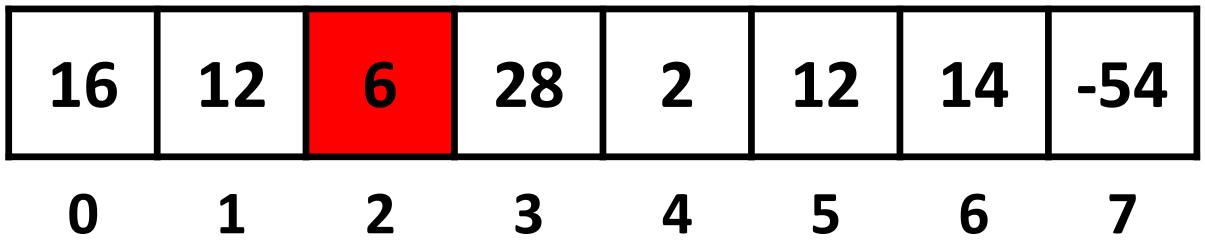


Each **subscript** indicates the
distance of that element from the beginning of the array.

Subscripts Indicate Distance – Example, pt. 1

Consider this element, which has a value of 6.

int test_array[8];

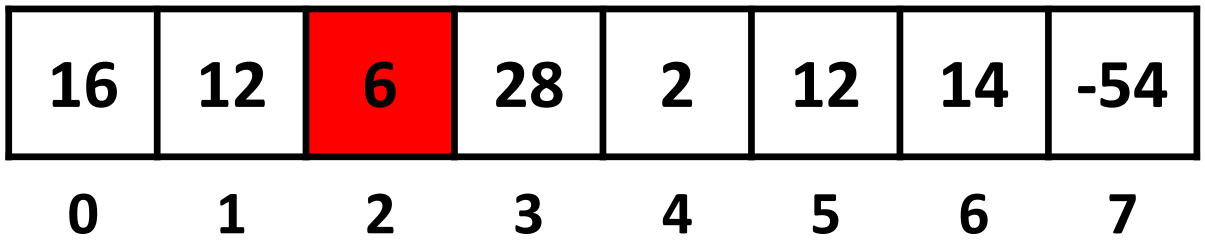


How many other elements are **between this one and the beginning** of the array?

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.

EXAMPLES USING ARRAYS

Sequential Array Access

Sequential access

- Start at one end, and access each element one after another, repeatedly
- So we need a loop

To access each element sequentially in an array, use a **counter-controlled loop**

Why use counter-controlled and not one of the other types?

USE A COUNTER-CONTROLLED LOOP FOR SEQUENTIAL ARRAY ACCESS

We know how many elements are in an array, so we know how many times the loop should run

Common to use an **indexed for** loop

- A counting loop where loop control variable starts at 0 and increases up to one less than the array size

Use the **loop control variable as the array subscript**

USING A LOOP TO INITIALIZE AN ARRAY'S ELEMENT VALUES

Fig06_03.c

- Initializes the elements of an array to zeros

Uninitialized array elements may contain garbage values!

NOTE USE OF SIZE_T IN THE FOR LOOPS

unsigned integral type

Recommended for variable that represents an array's size or indices

- Why use an unsigned data type here?

Defined in library **stddef.h**

- Often included by other libraries, such as stdio.h
- If you attempt to compile Fig. 6.3 and receive errors, include stddef.h in your program

On some compilers, **size_t** represents **unsigned int** and on others it represents **unsigned long**

FORMATTED OUTPUT

Note the output lines in previous example:

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

printf("%7u%13d\n", i, n[i]);
```

What is going on with these?

USING AN INITIALIZER LIST

fig06_04_rewritten.c

- Initializes the elements of an array with an initializer list

If fewer initializers than elements in the array:

- Remaining elements initialized to zero

If more initializers than elements in the array:

- Syntax error

If array size is omitted with an initializer list:

- Array size will be set to the number of elements in the initializer list

INITIALIZING ARRAY ELEMENTS WITH CALCULATIONS

Fig06_05.c

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

#DEFINE PREPROCESSOR DIRECTIVE DEFINES A SYMBOLIC CONSTANT

Symbolic constant

- Identifier that's **replaced with replacement text** by the C preprocessor before the program is compiled

NOTE – not the same as a variable!

Compiler **does not reserve space for symbolic constants** – so a new value cannot be assigned to it via the assignment operator

Can make programs more readable and easier to change

Use only uppercase letters and underscores for symbolic constant names

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**

ARRAYS & ARRAY ELEMENTS WITH FUNCTIONS

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?

How To Store the Size of an Array Depends on the Program

Use a value stored in a variable

- When the size of the array is not known until the program executes
- For example, the program might ask the user how much data to store

Use a symbolic constant

- When the size is known before the program executes
- #define MONTHS 12 (Remember, this isn't really storing anything)

Avoid hard-coding sizes

- In a declaration or in a loop
- Anything hard-coded reduces reusability and readability

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**

CONST QUALIFIER

If we only need to **read** an array, we should be careful that we don't change the array accidentally

Use qualifier **const** in function header and prototype, immediately to left of array name

- Provides information to the compiler
- Compiler will flag attempts to modify the array

Example of the **principle of least privilege**

Fig06_14.c

- function named tryToModifyArray

Returning an Array Result

Cannot use an entire array as a return type

Can use individual elements as return types

SECURE C PROGRAMMING

SUBSCRIPTS OUTSIDE OF ARRAY

Attempting to reference array elements "past the end" of the array is undetermined

Very important to check for a valid array reference before attempting to use it

Example:

int scores_array[10];

What is the range of valid subscripts?

No Automatic Bounds Checking for Array Indices in C

Accessing elements outside the bounds of arrays is a common security flaw

Reading from out-of-bounds array elements is undetermined

- Can cause a program to crash or appear to execute correctly while using bad data.

Writing to an out-of-bounds element can cause a **buffer overflow**

- Can corrupt a program's data in memory, crash a program and allow attackers to exploit the system and execute their own code.