Arrays

Chapter 7
Objectives

In this chapter, you will learn about:

- Declaring and Referencing Arrays
- Arrays in Functions
- Programming with Arrays
- Multidimensional Arrays
An array is used to **Process** a **collection** of data **all** of which is of the **same** type

- **Examples:**
  - A List of student **names**
  - A List of test **scores**

- **Why use arrays?**
  - Convenient way to organize related Variables (of the same type) in memory so that the data items can be processed (fetched/stored, algorithm design) efficiently
An array consisting of 5 variables of type `int` is declared as:

```c
int score[5];
```

// This Array Declaration specifies 5 unique variables
// (Array elements):

- `score[0]`
- `score[1]`
- `score[2]`
- `score[3]`
- `score[4]`

Size of Array is associated with array declaration; it specifies number of variables
While index is associated with Array elements
Recall, the declaration: `int score[5];` produces **5 unique indexed variables** (array elements):

- The first indexed value is always 0
- The last indexed value is always **One less than the array size** (5 – 1 = 4)
Introduction to Arrays

Array Variables (Elements)

1. All Array Elements must be of the same type:
   - Example: double, int, char, string, or float

2. An array element can be used anywhere an ordinary variable of the same type is used
To reference an Array element:

- **Use indexing Technique**
  - Array index always start with zero (to reference first element)
  - The last element is always one less than the array size
  - The value of the index can be either:
    - An **integer** or
    - An **expression** that evaluates to an integer value in range: (0, size -1), where size represents the array size

```c
int score[5];     // Assume array declaration
int n = 2;
score [n + 1] = 99; // Same as score[3] = 99;
```
Introduction to Arrays

Looping

- **for** loop is a Convenient way to step through an array: 

  Example:

  ```
  int score[5];  //Assume array declaration
  for (i = 0; i < 5; i++)
      cout << score[i] << " off by " << (100 - score[i]) << endl;
  ```

  Why?

  This is the last index is 5 -1
Use a **Defined Constant** in Array Declarations

Example:

1. const int NUMBER_OF_STUDENTS = 5; // defined constant
2. // Now declare array:
3. int score[NUMBER_OF_STUDENTS];
   ...
   for ( i = 0; i < NUMBER_OF_STUDENTS; i++)
       cout << score[i] << " off by " << (10 - score[i]) << endl;
       .......

**To re-use code for any number of students, all you do is change the constant value in line 1**
You cannot use a variable to declare Array size:

```cpp
Illegal Array declaration

cout << "Enter number of students: ";
cin >> number;
int score[number];
```

Not Portable on most compilers

We will talk about dynamic arrays in subsequent lectures
Introduction to Arrays

Let’s Summarize

So far ….

- **Array Declaration Syntax**
  - *Type-Name* Array-Name[Declared-Size];

- **Examples:**
  - int big-array[99];
  - double x[4];
  - double x[4], tax;
  - char grade[10];

- Each indexed variable is of type *Type-Name*
  - **Examples of index variables:**
    - big-array[0], big-array[1], …, big-array[98] are of type int
    - x[0], x[1], x[2] and x[3] are of type double
    - grade[0], grade[1], …, grade[9] are of type char
Recall, Computer memory is divided into cells (or **bytes**):
- Each byte has a unique number called **address**

An ordinary variable is stored in consecutive bytes:
- The number of bytes depends on the variable's type

The variable's **address** is the address of its **first** byte
When you declare an array `int a[4]`

- You reserve memory for four variables of type `int`.
- The variables are stored in contiguous locations.
- Only the address of `a[0]` is remembered.
- To determine the address of `a[2]`
  - Start at `a[0]`
  - Count `past` enough memory for two integers to find `a[2]`
Assume array declaration:

```
int score[4];  //Specifies a four-element array
```

Then index values for the array are: 0, 1, 2 and 3

All other indexed values will be considered out of range

- 4, 5, 6, 7, ...

Indexing an array with an out of range value does not generate an error message (if the referenced location exists in the computer’s memory)
Introduction to Arrays

Out of Range Indexing Problem

Consider

```c
int a[4];
int i = 5;
a[i] = 238;
```

The computer calculates the address of the illegal `a[5]`
(This address could be where some other variable is stored)
The value 238 is stored at the address calculated for `a[5]`
No warning is given!
Initializing Arrays in a Declaration Statement

Example:

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

Initializes the Array elements as follows:

- `Children[0] = 2;`
- `Children[1] = 12;`
- `Children[2] = 1;`

The values for the indexed variables are enclosed in braces and separated by commas.
What happens if you list few values than the array size?

- The listed values are used to initialize the leading indexed variables.
- The trailing indexed variables are initialized to a zero of the base type.
What happens if no values are listed in the array declaration?

Some compilers will set the array elements to zero

Do not depend on a compiler: Always Take full control of your program execution!
What happens if you omit the size during array declaration?

The size is automatically set to the number of values listed in the initialization

```c
int b[ ] = {8, 7, 6};
```

is the same as

```c
int b[3] = {8, 7, 6};
```
Arrays

- Identify the errors in the following declarations:
  - `int x[4] = {8, 7, 6, 5, 4};`
  - `int x[ ] = {8, 7, 6, 4};`
  - `const int SIZE = 4;`
  - `int x[size];`
Arrays

What is the output of the following code?

```cpp
char symbol[3] = {'a', 'b', 'c'};
for (int index = 0; index < 3; index++)
    cout << symbol[index];
```
Arrays

What is the output of the following code?

double a[3] = {1.1, 2.2, 3.3};
cout << a[0] << " " << a[1] << " " << a[2] << endl;
a[1] = a[2];
cout << a[0] << " " << a[1] << " " << a[2] << endl;
What is wrong with the following code?

```c
int sample_array[10];
for (int index =1; index <= 10; index++)
    sample_array[index] = 3*index;
```
Arrays in Functions

Index Variables as Function Arguments

1. Recall, an index variable (array element) can be used anywhere an ordinary variable of the same type is used.
2. An index variable can be an argument to a function call:

Example: Assume

```c
int i, n, a[10];
void my_function(int n) // Function takes one argument of type int

Example Function Call using index variables:

my_function (a[0]); // Index value is of the same type as argument in definition
my_function (a[1]);
my_function (a[i]); // Index expression is evaluated initially to determine value
... my_function (a[9]);
```
Consider the function definition:

```c
void tripler (int& n)
{
    n = 3*n;
}
```

Which of the following are acceptable function calls?

- `tripler(number);`
- `tripler(a[2]);`
- `tripler(a[3]);`
- `tripler(a[number]);`
- `tripler(a);`
Arrays in Functions
Entire Arrays as Function Arguments

- Function Definitions (and Declarations) use a formal parameter for an entire array:
  - So that when function is called the entire array is plugged into the argument
- Formal Parameter is known as **array parameter: a[]**

Array parameter is different from:
- Call-by reference parameter
- Call-by-value parameter
Function with an Array Parameter
Function Declaration and Definition

Function Declaration

```c
void fill_up(int a[], int size);
// size is the declared size of the array, entered by user
// array will contain integer values
```

Function Definition

```c
void fill_up(int a[], int size)
{
    ......
}
```
Function with an Array Parameter

Function Call

Assume

- `int score[5]; number_of_score = 5;`
- `void fill_up(int a[], int size)  // function definition`

Compiler identifies this as array parameter

Calling function `fill_up`

`fill_up(score, number_of_size)`

Array Argument does not use the brackets `[ ]`
Function with an Array Parameter

Example

```cpp
void fill_up(int a[], int size);

void fill_up(int a[], int size)
{
    using namespace std;
    cout << "Enter " << size << " numbers; \n;
    for (int i = 0; i < size; i++)
        cin >> a[i];
    size --;
    cout << "The last array index used is \n"
        << size << endl;
}
```

Assume:

```cpp
int score[5], number_of_score = 5;

fill_up(score, number_of_score);
```

```cpp
using namespace std;
size = 5;
cout << "Enter " << size << " numbers; \n;
for (int i = 0; i < size; i++)
    cin >> score[i];
size --;
cout << "The last array index used is \n"
    << size << endl;
```
The array formal parameter `a[]` is a placeholder for the argument `score`.
- Any action performed on the array parameter is performed on the array argument.
- The values of the indexed variables can be changed by the function.
What does the computer know about an array?
1. The base type
2. The address of the first indexed variable
3. The number of indexed variables (Array size)

What does a function know about an array argument?
1. The base type
2. The address of the first indexed variable

Why?
A function does not know the size of an array argument

Therefore, you should include a formal parameter that specifies the size of the array in the function declaration and definition.

So that your function can be re-used to process arrays of various sizes:

- Function `fill_up` from can be used in several ways:

  ```plaintext
  fill_up(score, 5);
  fill_up(time, 10);
  fill_up(load, 4);
  ```

The size of score, time, load?
Recall, a function can change the values stored in the array argument.

How do you prevent a function from changing the array argument values?

Use the modifier `const` before the array parameter in both the function declaration and function definition.

```c
void show_the_world(const int a[], int size);
void show_the_world(const int a[], int size)
{
    ....
}
```
What happens if

1. you use the modifier const in both the declaration and definition
2. Your code attempts to change the values of the array argument associated with the constant array parameter?

The compiler will generate an error if you write code that changes the values stored in the array parameter.
Arrays in Function

Function Calls and const

- Suppose a function with a constant array parameter calls another function using the const array parameter as an argument…

Then,

- The called function must use a constant array parameter as a placeholder for the array

Otherwise,

- The compiler will issue an error if a function is called that does not have a const array parameter to accept the array argument
Function Calls and const

Example

- double compute_average(int a[], int size);

  void show_difference(const int a[], int size)
  {
    double average = compute_average(a, size);
    ...
  }

- compute_average has no constant array parameter

- This code generates an error message because compute_average could change the array parameter
Recall that functions can return a value of type int, double, char, …, or a class type

Functions cannot return arrays

We will see shortly how to return a pointer to an array
The exact size needed for an array is unknown
  - Often varies from one run of a program to another
  - Is often not known when the program is written

A common solution to the array size problem
  - Declare the array size to be the largest that could be needed
  - Then Decide how to deal with partially filled arrays
Coping with Array size

Partially filled array

- When using arrays that are partially filled
  - Program must keep track of the array elements used
  - Program must not reference any uninitialized array variables

- Functions dealing with the array may not need to know the declared size of the array, only how many elements are stored in the array
  - A parameter, number_used, may be sufficient to ensure that referenced index values are legal
  - Refer to program on Page 411
Programming with Arrays

Sorting an Array

- Sorting a list of values is very common task
  - Create an alphabetical listing
  - Create a list of values in ascending order
  - Create a list of values in descending order

- Many sorting algorithms exist
  - Some are very efficient
  - Some are easier to understand
Programming with Arrays

Sorting an Array

- When the sort is complete, the elements of the array are ordered such that

\[ a[0] < a[1] < \ldots < a[\text{number}_{used} - 1] \]

- This leads to an outline of an algorithm:
  
  for (int index = 0; index < number_used; index++)
  place the indexth smallest element in a[index]
Programming with Arrays
Selection Sort Algorithm

- One array is sufficient to do our sorting
  - Search for the smallest value in the array
  - Place this value in a[0], and place the value that was in a[0] in the location where the smallest was found
  - Starting at a[1], find the smallest remaining value swap it with the value currently in a[1]
  - Starting at a[2], continue the process until the array is sorted
Programming with Arrays
Selection Sort Algorithm

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>6</td>
<td>10</td>
<td>2</td>
<td>16</td>
<td>4</td>
<td>18</td>
<td>14</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>8</td>
<td>16</td>
<td>4</td>
<td>18</td>
<td>14</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>8</td>
<td>16</td>
<td>4</td>
<td>18</td>
<td>14</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>8</td>
<td>16</td>
<td>4</td>
<td>18</td>
<td>14</td>
<td>12</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>10</td>
<td>8</td>
<td>16</td>
<td>6</td>
<td>18</td>
<td>14</td>
<td>12</td>
<td>20</td>
</tr>
</tbody>
</table>

Refer to Page 419 for sample sort program
Multidimensional Array
Declarations

- You can declare an array with more than one index
  - `char page[30][100];`
    - declares an array of characters named page
  - Each index variable of `page` has two index values:
    - The first ranges from 0 to 29
    - The second ranges from 0 to 99
  - Each index in enclosed in its own brackets
  - `page` can be visualized as an array of
    - 30 rows and 100 columns
The indexed variables for array `page` are

- `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]`

`page` is actually an array of size 30
- `page`'s base type is an array of 100 characters
Recall that the size of an array is not needed when declaring a formal parameter:
`void display_line(const char a[ ], int size);`

The base type of a multi-dimensional array must be completely specified in the parameter declaration
- `void display_page(const char page[ ] [100], int size_dimension_1);`