Programming Language 1 (MT261)

Lecture 5

Dr. Ahmed Fathalla

Allocating Memory with Constants and Variables

- Named Constant: memory location whose content can't change during execution
  - The syntax to declare a named constant is:

```
const dataType identifier = value;
```

  - In C++, `const` is a reserved word
- Variable: memory location whose content may change during execution

The syntax for declaring one variable or multiple variables is:

```
dataType identifier, identifier, . . .;
```

NAMED CONSTANTS

Some data must stay the same throughout a program. For example, the conversion formula that converts inches into centimeters is fixed, because 1 inch is always equal to 2.54 centimeters. When stored in memory, this type of data needs to be protected from accidental changes during program execution. In C++, you can use a **named constant** to instruct a program to mark those memory locations in which data is fixed throughout program execution.

Named constant: A memory location whose content is not allowed to change during program execution.

To allocate memory, we use C++'s declaration statements. The syntax to declare a named constant is:

```
const dataType identifier = value;
```

In C++, `const` is a reserved word. It should be noted that a named constant is initialized and declared all in one statement and that it *must* be initialized when it is declared because from this statement on the compiler will reject any attempt to change the value.

C++ data types

- C++ **data** can be classified into three categories:
  - Simple data type
  - Structured data type
  - Pointers

CHAPTER 8

Arrays and Strings

IN THIS CHAPTER, YOU WILL:

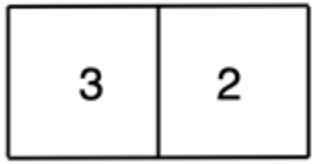
1. Learn the reasons for arrays
2. Explore how to declare and manipulate data into arrays
3. Understand the meaning of "array index out of bounds"
4. Learn how to declare and initialize arrays
5. Become familiar with the restrictions on array processing
6. Discover how to pass an array as a parameter to a function
7. Learn how to search an array
8. Learn how to sort an array
9. Become aware of `auto` declarations
10. Learn about range-based `for` loops
11. Learn about C-strings
12. Examine the use of string functions to process C-strings
13. Discover how to input data into—and output data from—a C-string
14. Learn about parallel arrays
15. Discover how to manipulate data in a two-dimensional array
16. Learn about multidimensional arrays

Simple and **Structured** data types

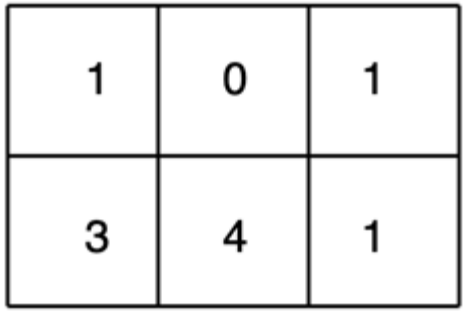
- A data type is called **simple** if variables of that type can store only one value at a time. In contrast, in a **structured** data type, each data item is a collection of other data items.
- **Simple** data types (such as int, char, and float) are building blocks of **structured** data types.
- A **structured data** type is one in which each data item is a collection of other data items.

n-dimensional array

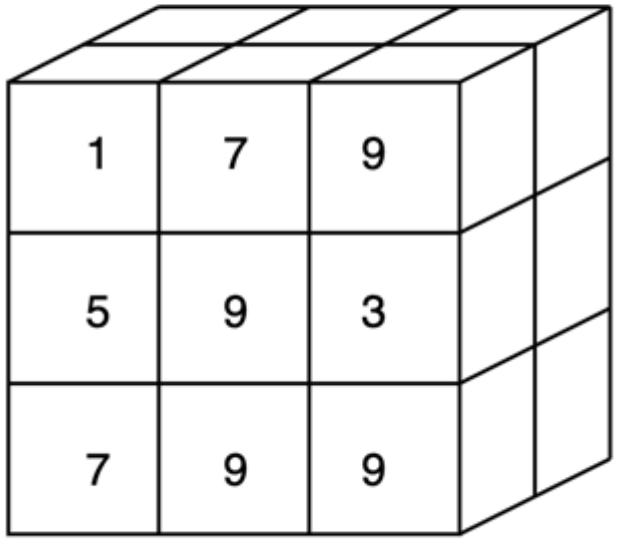
1D Array



2D Array



3D Array



Array

- The first structured data type that we will discuss is an **Array**.
- **Array**: a collection of a fixed number of components wherein all of the components have the same **data type** and in contiguous (that is, adjacent) memory space.

- In a one-dimensional array, the components are arranged in a list form.
- Syntax for declaring a one-dimensional array:

```
dataType arrayName[intExp];
```

`intExp` evaluates to a positive integer

Define an Array: Method_1

- Example:

```
int list[10];
```

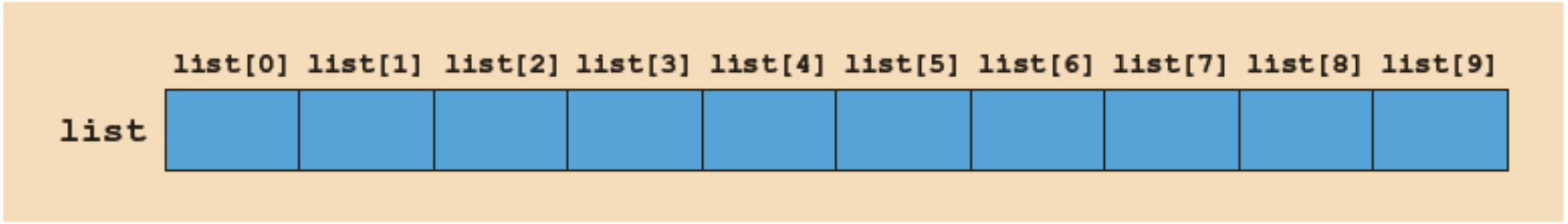


FIGURE 8-3 Array `list`

Accessing Array Components with []

- General syntax:

```
arrayName[indexExp]
```

  where `indexExp`, called an **index**, is any expression whose value is a non-negative integer

- Index value specifies the position of the component in the array
- **`[]`** is the **array subscripting operator**
- The array index always starts at **`0`**

Define an Array: Method_1

- Example:

```
int list[10];
```

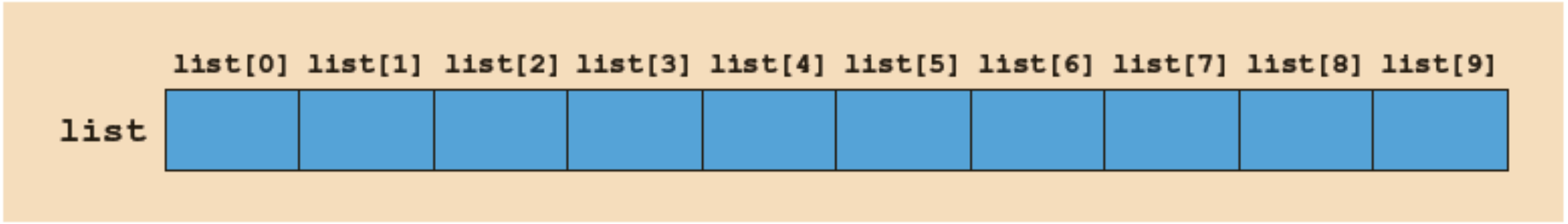


FIGURE 8-3 Array `list`

Accessing Array Components with [] operator

```
list[3] = 10;
list[6] = 35;
list[5] = list[3] + list[6];
```

The first statement stores `10` in `list[3]`, the second statement stores `35` in `list[6]`, and the third statement adds the contents of `list[3]` and `list[6]` and stores the result in `list[5]` (see Figure 8-5).

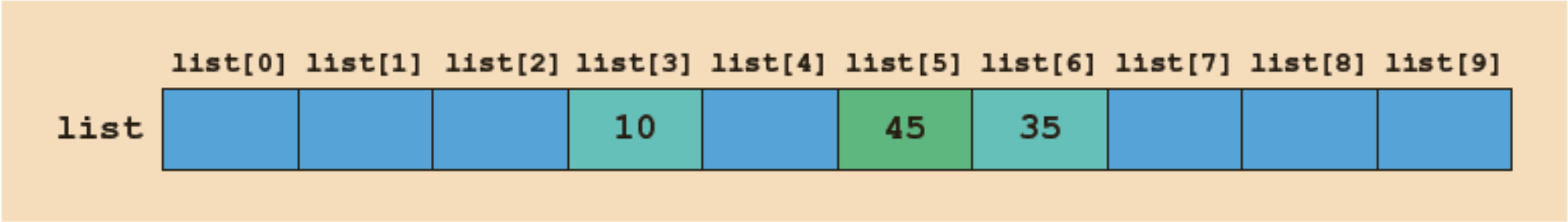


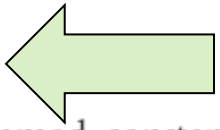
FIGURE 8-5 Array `list` after execution of the statements `list[3] = 10;`, `list[6] = 35;`, and `list[5] = list[3] + list[6];`

Define an Array: Method_2

EXAMPLE 9-2

You can also declare arrays as follows:

```
const int ARRAY_SIZE = 10;
int list[ARRAY_SIZE];
```



That is, you can first declare a named constant and then use the value of the named constant to declare an array and specify its size.

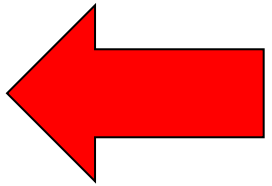
NOTE

When you declare an array, its size must be known. For example, you cannot do the following:

```
int arraySize;                                  //Line 1

cout << "Enter the size of the array: ";  //Line 2
cin >> arraySize;                               //Line 3
cout << endl;                                   //Line 4

int list[arraySize];                            //Line 5; not allowed
```



Example (8-3)

- Some basic operations performed on a one-dimensional array are:
  - Initializing.
  - Inputting data.
  - Outputting data stored in an array.
  - Finding the sum, average, and largest.
- Each operation requires ability to step through the elements of the array
- Easily accomplished by a loop

Processing One-Dimensional Arrays

- Consider the declaration

```
int list[100];  //array of size 100
int i;
```

- Using `for` loops to access array elements:

```
for (i = 0; i < 100; i++) //Line 1
        //process list[i]  //Line 2
```

  - Example:

```
for (i = 0; i < 100; i++)    //Line 1
     cin >> list[i];   //Line 2
```

Example (8-3)

Define a variable

```
double sales[10];
int index;
double largestSale, sum, average;
```

Initializing an array:

```
for (index = 0; index < 10; index++)
    sales[index] = 0.0;
```

Reading data into an array:

```
for (index = 0; index < 10; index++)
    cin >> sales[index];
```

Printing an array:

```
for (index = 0; index < 10; index++)
    cout << sales[index] << " ";
```

Finding the sum and average of an array:

```
sum = 0;
for (index = 0; index < 10; index++)
    sum = sum + sales[index];

average = sum / 10;
```

Largest element in the array:

```
maxIndex = 0;
for (index = 1; index < 10; index++)
    if (sales[maxIndex] < sales[index])
        maxIndex = index;
largestSale = sales[maxIndex];
```

Array Index Out of Bounds

- If we have the statements:

```
double num[10];
int i;
```

- The component `num[i]` is valid if `i = 0, 1, 2, 3, 4, 5, 6, 7, 8,` or `9`
- The index of an array is in bounds if the `index >=0` and the `index <= ARRAY_SIZE-1`
  - Otherwise, we say the `index` is out of bounds
- **In C++, there is no guard against indices that are out of bounds**

Exercise_1: Search in an array for a specific item

```
int main()
{
    int arr[5];

    // reading array items:
    for (int i=0; i<5; i++)
    {
        cout<<"Enter a value for item_"<<i<<": ";
        cin>>arr[i];
    }

    int key;
    cout<<"\nEnter a value to look for: ";
    cin>>key;

//searching for the "key" item
    bool flag=false;
    for (int i=0; i<5; i++)
    {
        if (arr[i]==key)
        {
            cout<<"the index is at position:"<<i;
            flag=true;
            break;
        }
    }
    if (flag==false)
        cout<<"cannot find that item";
}
```

Task

- Sorting array's elements.

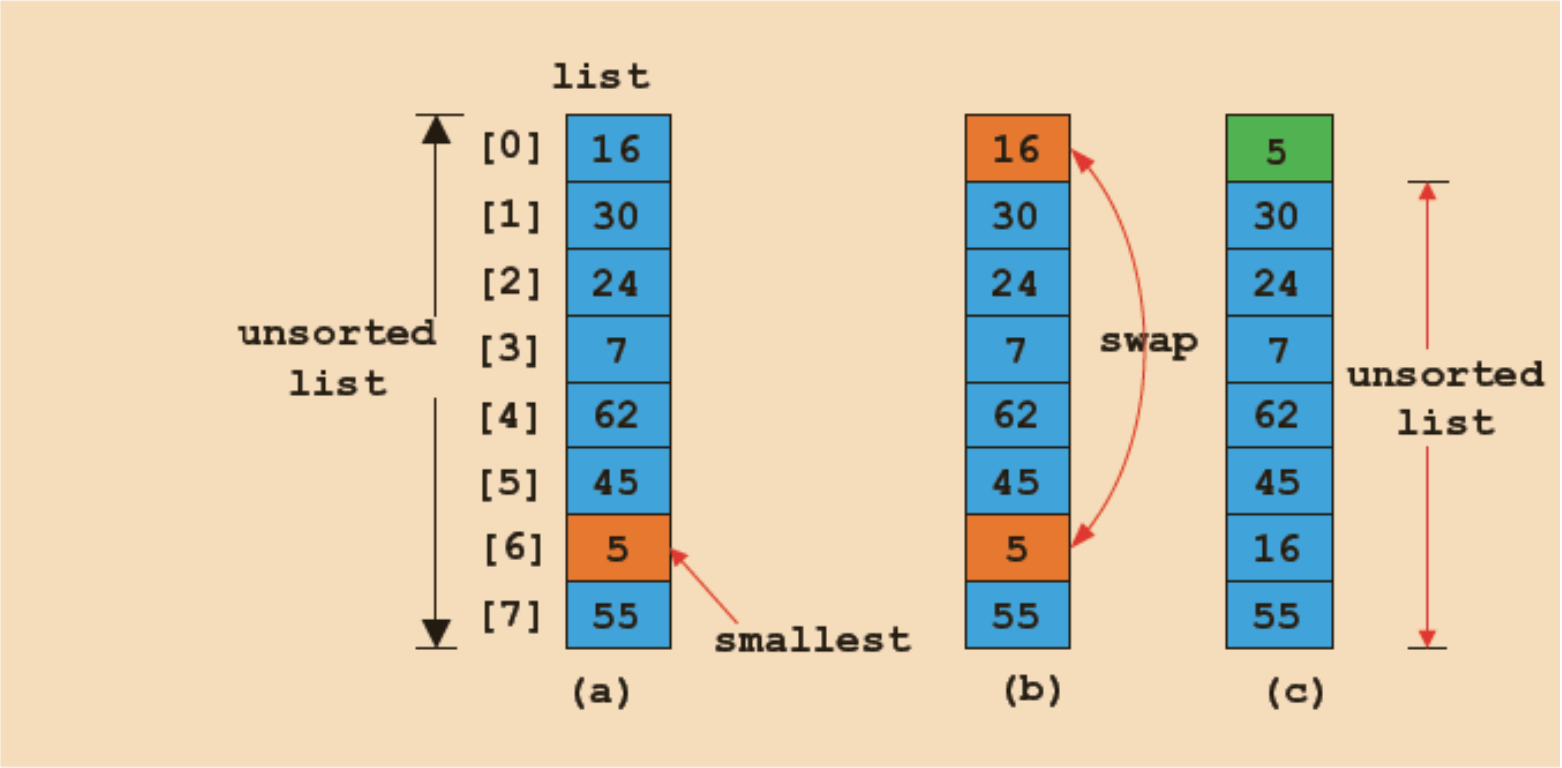


FIGURE 8-10 Elements of `list` during the first iteration