CSC404: Programming II

# 3 Array – One-Dimensional (REVISIT)

by: Prof. Madya Haji Khalil b Haji Awang

# Objectives

- To learn about arrays.
- To explore how to declare and manipulate data into arrays.
- To become familiar with the restrictions on array processing.
- To access array elements using pointers.
- To discover how to pass an array as a parameter to a function.
- To learn how to return a pointer from functions.

# Array

- An array is a consecutive group of memory locations.
- Each *group* is called an element of the array.
- The contents of each element are of the same *type* – **`int`**, **`double`**, **`char`**, etc.
- We can refer to individual *elements* by giving the position number (index) of the element in the array.

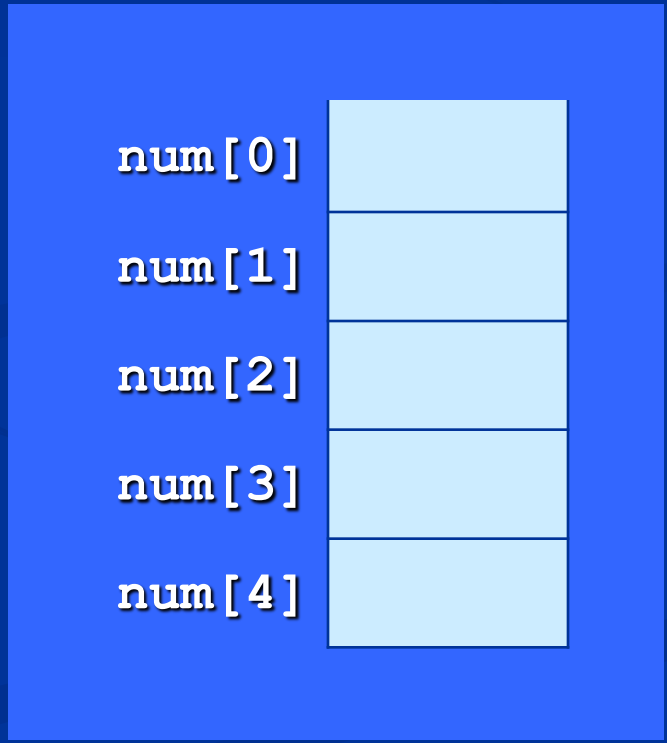
# Declaring an Array

- Syntax:

```
dataType arrayName[intExp]
```

- Example: Declare an array **num** of five components, each component is of type **int**.

```
int num[5];
```



# Array Components

- The first element is the 0th element.
- If you declare an array of *n* elements, the last one is number *n* - *1*.
- If you try to access element number *n* it is an error.

# Accessing Array Components

- Syntax:

```
arrayName[indexExp]
```

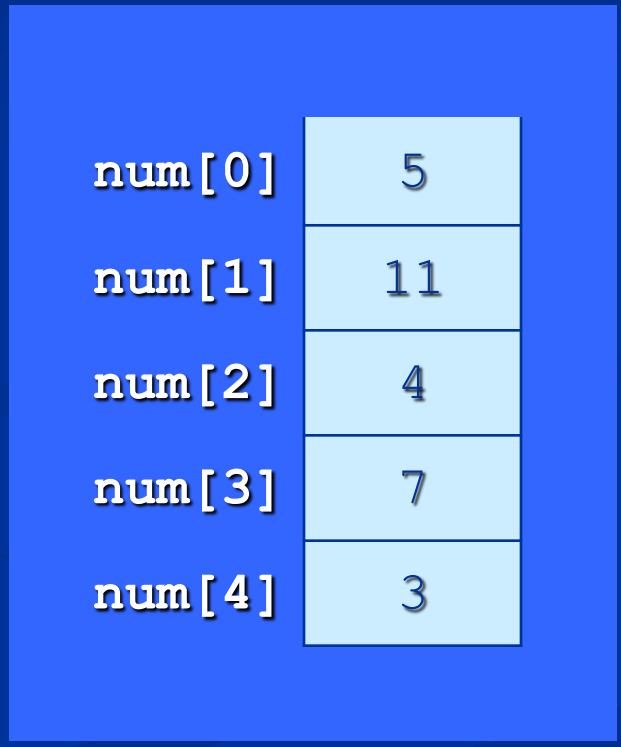
  - **indexExp** is any expression whose value is a non negative integer called index or subscript.
  - **[]** is an operator called array subscripting operator.
- Example: Assign **3** to the fifth component of **num**.

```
num[4] = 3;
```

Do not confuse the two ways to use the square brackets **[]** with an array name. The number used in a declaration, **`int num[5];`** specifies how many components the array has. When used anywhere else, the number tells which component is meant.

# Examples

```
num[2] = 4;
i = 3; num[i] = 7;
num[1] = num[2] + num[3];
i = 1; num[i - 1] = num[1] / 2;
```



# Initializing an Array

- An array can be initialized when we declare it just like any other simple variable.
- Example: Declare an array, **sales**, of five components and initializes the components:

```
float sales[5] = {12.25, 32.50, 16.90, 23, 45.68};
```

- When initializing arrays as they are declared, it is NOT necessary to specify the size of the array, it is determined by the number of initial values in the braces:

```
float sales[] = {12.25, 32.50, 16.90, 23, 45.68};
```

# Partial Initialization of Arrays

- When we declare and initialize an array simultaneously, we do not need to initialize all components of the array.
- Example:

```
int list[10] = {0}; // declares list to be an
        // array of 10 components and
        // initializes all the components to 0
int list[10] = {8, 5, 12}; // declares list to
        // be an array of 10 components and
        // initializez list[0] to 8, list[1] to
        // 5, list[2] to 12 and all other
        // components to 0.
```

# Some Restrictions

- We cannot assign the whole array to another array directly by using assignment operator. Assigning an array must be done component by component.

```
int myList[5] = {0, 4, 8, 12, 16};
int yourList[5];
yourList = myList;  // illegal
for (int i = 0; i < 5; i++)
  yourList[i] = myList[i];  // copy component-wise
```

# Some Restrictions (cont.)

- Data must be read one component at a time.

```
int myList[5] = {0, 4, 8, 12, 16};
int yourList[5];
cin >> yourList  // illegal
for (int i = 0; i < 5; i++)
  cin >> yourList[i]; // read one component
                      // at a time
```

# Some Restrictions (cont.)

- Printing the contents of an array must be done component-wise.

```
cout << yourList; // illegal

for (int i = 0; i < 5; i++)
   cout << yourList[i]; // print one component
                        // at a time
```

# Some Restrictions (cont.)

- Comparing the contents of an array must be done component-wise.

```
if (yourList < myList) // illegal
   :

for (int i = 0; i < 5; i++)
   if (yourList[i] < myList[i])
      :          // compare one component
                 // at a time
```

# Base Address of an Array

- The base address of an array is the address, or memory location of the first array component
- If `list` is a one-dimensional array, base address of list is the address of the component `list[0]`.
- The value stored in `list` is the address of component `list[0]` like in pointer variable, i.e. value of `list` and the value of `&list[0]` are the same.

# Dynamic Array

- All memory needs were determined before program execution by defining the variable needed.
- But there may be cases where the memory needs of a program can only be determined during runtime.
- On these cases, programs need to dynamically allocate memory.
- C++ language integrates the operators new and delete to allocate memory and to free memory for dynamic memory.

# Dynamic Array (cont.)

- The statement:

```
int *p;
```

declares p to be a pointer variable of type `int`.

- The statement:

```
p = new int[10];
```

allocates `10` contiguous memory locations, each of type `int`, and stores the address of the first memory location into `p`. In other words, the operator `new` creates an array of `10` components of type `int`.

- It returns the base address of the array, and the assignment operator stores the base address of the array into `p`.

# Dynamic Array (cont.)

- The statement:

```
*p = 25;
```

stores `25` into the first memory location.

- The statements:

```
p++; //p points to the next array component
*p = 35;
```

store `35` into the second memory location.

- By using the increment and decrement operations, you can access the components of the array.

# Dynamic Array (cont.)

- Array notation can be used to access these memory locations:

```
p[0] = 25;
p[1] = 35;
```

store `25` and `35` into the first and second array components, respectively.

- The following `for` loop initializes each array component to `0`:

```
for (int j = 0; j < 10; j++)
     p[j] = 0;
```

# Passing Array to Functions

- When we pass an array as a parameter base address of the actual array is passed to the formal parameter.
- Since we passed the address, passing arrays are always passed by reference
- Because arrays are passed by reference only, we do not need to use symbol **&** when declaring an array as a formal parameter.
- The size of the array is usually omitted since the compiler will use the size of array of the argument.

# Example

- Prototype:

```
void func1(int [], double []);
```

- Definition:

```
void func1(int x[], double y[])
{
  :
}
```

# Example

- Initialize the array:

```
void initialize(int list[], size)
{
    for (int i = 0; i < size; i++)
        list[i] = 0;
}
```

# Example

- Read the array:

```
void readArray(int list[], size)
{
    for (int i = 0; i < size; i++)
        cin >> list[i];
}
```

# Example

- Find and return the sum of array:

```
int sumArray(int list[], size)
{
   int sum = 0;
   for (int i = 0; i < size; i++)
      sum = sum + list[i];
   return sum;
}
```

# Example

- Find and return the index of the largest element in the array:

```
int indexLargest(int list[], size)
{
    int maxIndex = 0;
    for (int i = 1; i < size; i++)
        if (list[i] > list[maxIndex]
            maxIndex = i;
    return maxIndex;
}
```

Functions cannot return a value of the type array

# Character Arrays

- An array whose components are type of **char**.
- Also called string in C++.
- Character arrays terminated by a null character **'\0'**.
- For example, string **"Hello"** represents six characters: **'H'**, **'e'**, **'l'**, **'l'**, **'o'** and **'\0'**.
- **char name[16];** declares an array name of 16 components but the length of name is only 15.

# Initializing a Character Array

- Declare an array name containing 16 components of type **char** and store the string **"Khalil"** in it.

```
char name[16] = {'K', 'h', 'a', 'l', 'i', 'l', '\0'};
```

or

```
char name[16] = "Khalil";
```

- Declare an array name and store the string "Khalil" in it. The size of the array is depend on the length of the string. In this case, the size is 7.

```
char name[] = "Khalil";
```

- Most rules that apply to other arrays also apply to character arrays.

```
char name[16];
name = "Khalil";  // illegal
```