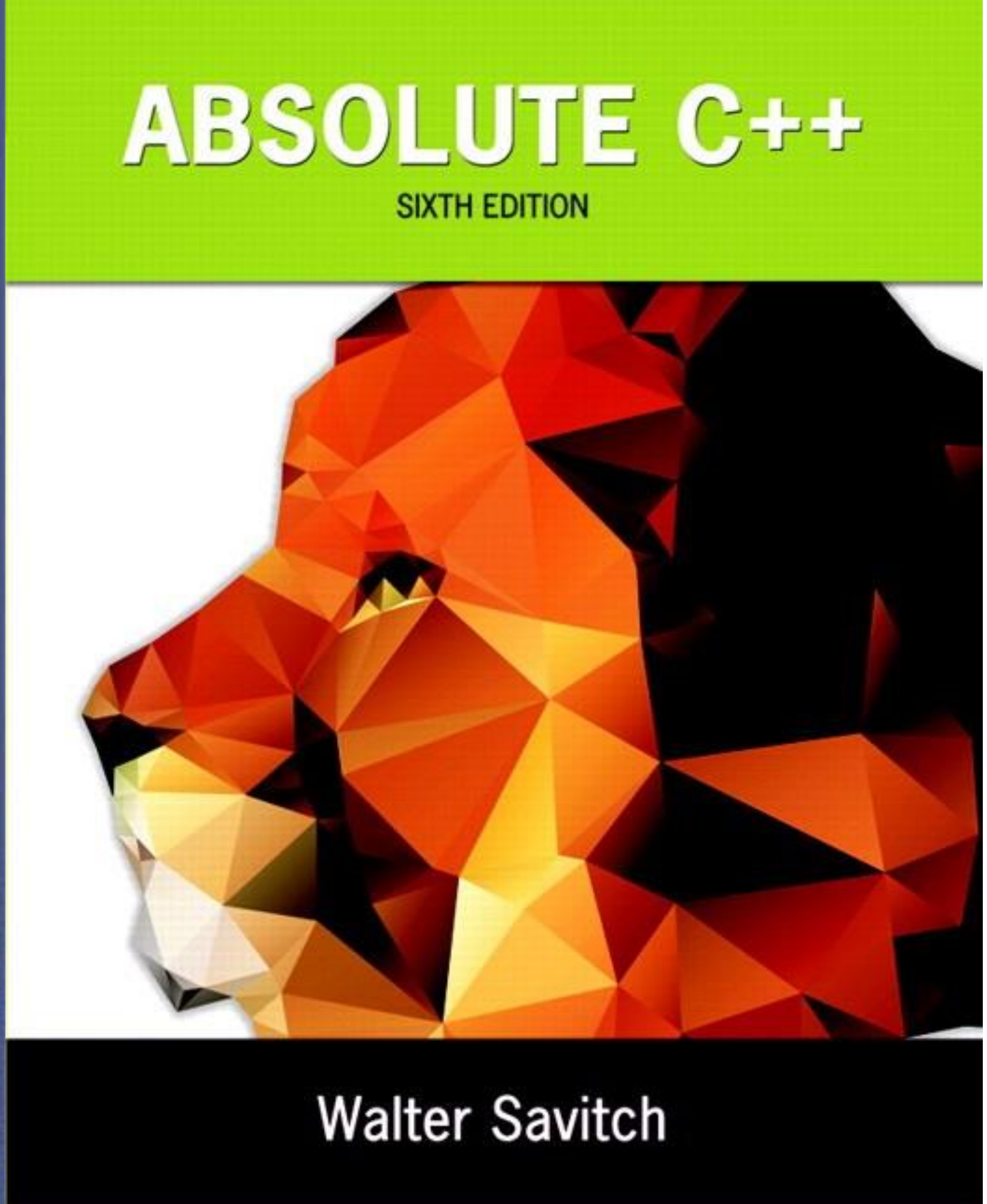


Chapter 5

Arrays

PEARSON

Learning Objectives

- Introduction to Arrays
  - Declaring and referencing arrays
  - For-loops and arrays
  - Arrays in memory
- Arrays in Functions
  - Arrays as function arguments, return values
- Programming with Arrays
  - Partially Filled Arrays, searching, sorting
- Multidimensional Arrays

Introduction to Arrays

- Array definition:
  - A collection of data of same type
- First "aggregate" data type
  - Means "grouping"
  - int, float, double, char are simple data types
- Used for lists of like items
  - Test scores, temperatures, names, etc.
  - Avoids declaring multiple simple variables
  - Can manipulate "list" as one entity

Declaring Arrays

- Declare the array → allocates memory
  int score[5];
  - Declares array of 5 integers named "score"
  - Similar to declaring five variables:
    int score[0], score[1], score[2], score[3], score[4]
- Individual parts called many things:
  - Indexed or subscripted variables
  - "Elements" of the array
  - Value in brackets called index or subscript
    - Numbered from 0 to size - 1

Accessing Arrays

- Access using index/subscript
  - cout << score[3];
- Note two uses of brackets:
  - In declaration, specifies SIZE of array
  - Anywhere else, specifies a subscript
- Size, subscript need not be literal
  - int score[MAX_SCORES];
  - score[n+1] = 99;
    - If n is 2, identical to: score[3]

Array Usage

- Powerful storage mechanism
- Can issue command like:
  - "Do this to i^{th} indexed variable"
    where i is computed by program
  - "Display all elements of array score"
  - "Fill elements of array score from user input"
  - "Find highest value in array score"
  - "Find lowest value in array score"

Array Program Example: **Display 5.1** Program Using an Array (1 of 2)

Display 5.1 **Program Using an Array**

```
//Reads in five scores and shows how much each
//score differs from the highest score.
#include <iostream>
using namespace std;
int main( )
{
    int i, score[5], max;
    cout << "Enter 5 scores:\n";
    cin >> score[0];
    max = score[0];
    for (i = 1; i < 5; i++)
    {
        cin >> score[i];
        if (score[i] > max)
            max = score[i];
        //max is the largest of the values score[0],..., score[i].
    }
```

Array Program Example:
Display 5.1 Program Using an Array (2 of 2)

```
    cout << "The highest score is " << max << endl
         << "The scores and their\n"
         << "differences from the highest are:\n";
    for (i = 0; i < 5; i++)
        cout << score[i] << " off by "
             << (max – score[i]) << endl;
    return 0;
}
```

SAMPLE DIALOGUE

```
Enter 5 scores:
5 9 2 10 6
The highest score is 10
The scores and their
differences from the highest are:
5 off by 5
9 off by 1
2 off by 8
10 off by 0
6 off by 4
```

for-loops with Arrays

- Natural counting loop
  - Naturally works well "counting through" elements of an array
- Example:

```
for (idx = 0; idx<5; idx++)
{
      cout << score[idx] << "off by "
                << max – score[idx] << endl;
}
```

  - Loop control variable (idx) counts from 0 – 5

Major Array Pitfall

- Array indexes always start with zero!
- Zero is "first" number to computer scientists
- C++ will "let" you go beyond range
  - Unpredictable results
  - Compiler will not detect these errors!
- Up to programmer to "stay in range"

Major Array Pitfall Example

- Indexes range from 0 to (array_size – 1)
  - Example:
    double temperature[24];   // 24 is array size
    // Declares array of 24 double values called temperature
    - They are indexed as:
      temperature[0], temperature[1] … temperature[23]
  - Common mistake:
    temperature[24] = 5;
    - Index 24 is "out of range"!
    - No warning, possibly disastrous results

Defined Constant as Array Size

- Always use defined/named constant for array size
- Example:
  const int NUMBER_OF_STUDENTS = 5;
  int score[NUMBER_OF_STUDENTS];
- Improves readability
- Improves versatility
- Improves maintainability

Uses of Defined Constant

- Use everywhere size of array is needed
  - In for-loop for traversal:
    ```
    for (idx = 0; idx < NUMBER_OF_STUDENTS; idx++)
    {
        // Manipulate array
    }
    ```
  - In calculations involving size:
    lastIndex = (NUMBER_OF_STUDENTS – 1);
  - When passing array to functions (later)
- If size changes → requires only ONE change in program!

Ranged-Based For Loop

- The C++11 ranged-based for loop makes it easy to iterate over each element in a loop
- Format

```
for (datatype varname : array)
{
        //  varname is set to each successive
         //  element in the array
}
```

- Example

```
int arr[] = {20, 30, 40, 50};
for (int x : arr)
   cout << x << " ";
cout << endl;
```

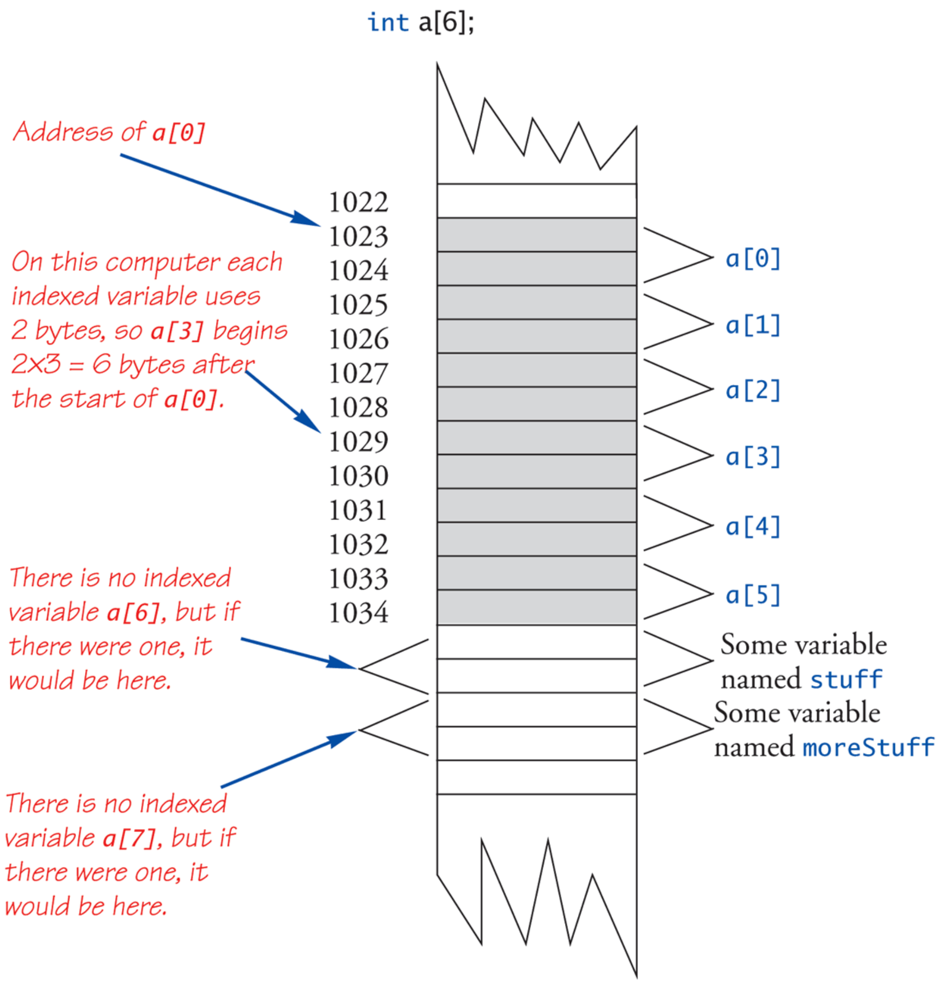
Output: 20 30 40 50

Arrays in Memory

- Recall simple variables:
  - Allocated memory in an "address"
- Array declarations allocate memory for entire array
- Sequentially-allocated
  - Means addresses allocated "back-to-back"
  - Allows indexing calculations
    - Simple "addition" from array beginning (index 0)

An Array in Memory

Display 5.2 An Array in Memory


int a[6];
Address of a[0]
On this computer each indexed variable uses 2 bytes, so a[3] begins 2x3 = 6 bytes after the start of a[0].
There is no indexed variable a[6], but if there were one, it would be here.
There is no indexed variable a[7], but if there were one, it would be here.
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
Some variable named stuff
Some variable named moreStuff

Initializing Arrays

- As simple variables can be initialized at declaration:
  int price = 0;   // 0 is initial value
- Arrays can as well:
  int children[3] = {2, 12, 1};
  - Equivalent to following:
    int children[3];
    children[0] = 2;
    children[1] = 12;
    children[2] = 1;

Auto-Initializing Arrays

- If fewer values than size supplied:
  - Fills from beginning
  - Fills "rest" with zero of array base type
- If array-size is left out
  - Declares array with size required based on number of initialization values
  - Example:
    int b[] = {5, 12, 11};
    - Allocates array b to size 3

Arrays in Functions

- As arguments to functions
  - Indexed variables
    - An individual "element" of an array can be function parameter
  - Entire arrays
    - All array elements can be passed as "one entity"
- As return value from function
  - Can be done → chapter 10

Indexed Variables as Arguments

- Indexed variable handled same as simple variable of array base type
- Given this function declaration:
  void myFunction(double par1);
- And these declarations:
  int i; double n, a[10];
- Can make these function calls:
  myFunction(i);   // i is converted to double
  myFunction(a[3]);   // a[3] is double
  myFunction(n); // n is double

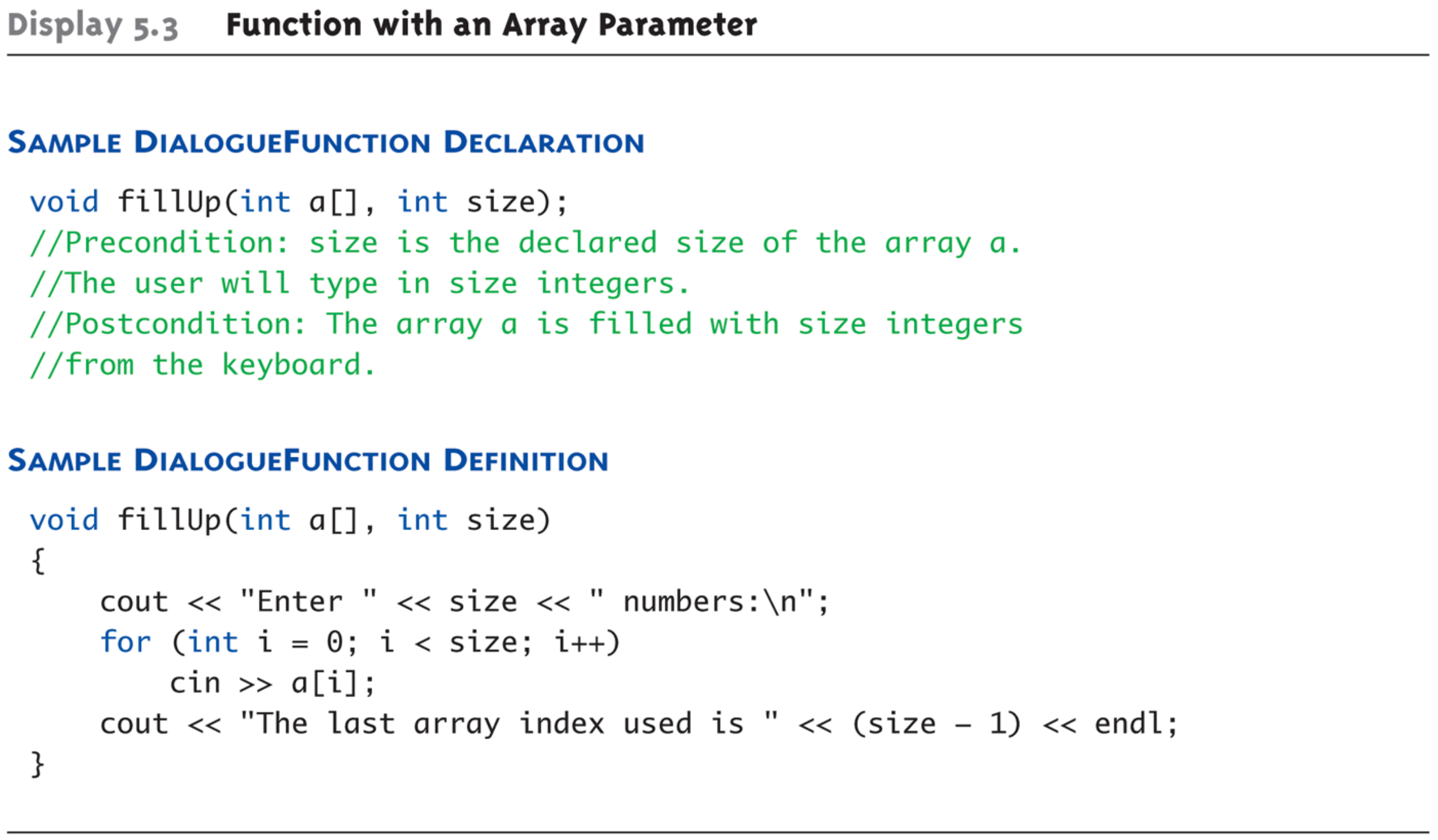
Subtlety of Indexing

- Consider:
  myFunction(a[i]);
  - Value of i is determined first
    - It determines which indexed variable is sent
  - myFunction(a[i*5]);
  - Perfectly legal, from compiler's view
  - Programmer responsible for staying "in-bounds" of array

Entire Arrays as Arguments

- Formal parameter can be entire array
  - Argument then passed in function call is array name
  - Called "array parameter"
- Send size of array as well
  - Typically done as second parameter
  - Simple int type formal parameter

Entire Array as Argument Example: **Display 5.3** Function with an Array Parameter



Display 5.3 **Function with an Array Parameter**

SAMPLE DIALOGUEFUNCTION DECLARATION

```
void fillUp(int a[], int size);
//Precondition: size is the declared size of the array a.
//The user will type in size integers.
//Postcondition: The array a is filled with size integers
//from the keyboard.
```

SAMPLE DIALOGUEFUNCTION DEFINITION

```
void fillUp(int a[], int size)
{
    cout << "Enter " << size << " numbers:\n";
    for (int i = 0; i < size; i++)
        cin >> a[i];
    cout << "The last array index used is " << (size – 1) << endl;
}
```

Entire Array as Argument Example

- Given previous example:
- In some main() function definition, consider this calls:

```
int score[5], numberOfScores = 5;
fillup(score, numberOfScores);
```

  - 1st argument is entire array
  - 2nd argument is integer value

- Note no brackets in array argument!

Array as Argument: How?

- What's really passed?
- Think of array as 3 "pieces"
  - Address of first indexed variable (arrName[0])
  - Array base type
  - Size of array
- Only 1st piece is passed!
  - Just the beginning address of array
  - Very similar to "pass-by-reference"

Array Parameters

- May seem strange
  - No brackets in array argument
  - Must send size separately
- One nice property:
  - Can use SAME function to fill any size array!
  - Exemplifies "re-use" properties of functions
  - Example:

```
int score[5], time[10];
fillUp(score, 5);
fillUp(time, 10);
```

The const Parameter Modifier

- Recall: array parameter actually passes address of 1st element
  - Similar to pass-by-reference
- Function can then modify array!
  - Often desirable, sometimes not!
- Protect array contents from modification
  - Use "const" modifier before array parameter
    - Called "constant array parameter"
    - Tells compiler to "not allow" modifications

Functions that Return an Array

- Functions cannot return arrays same way simple types are returned
- Requires use of a "pointer"
- Will be discussed in chapter 10…

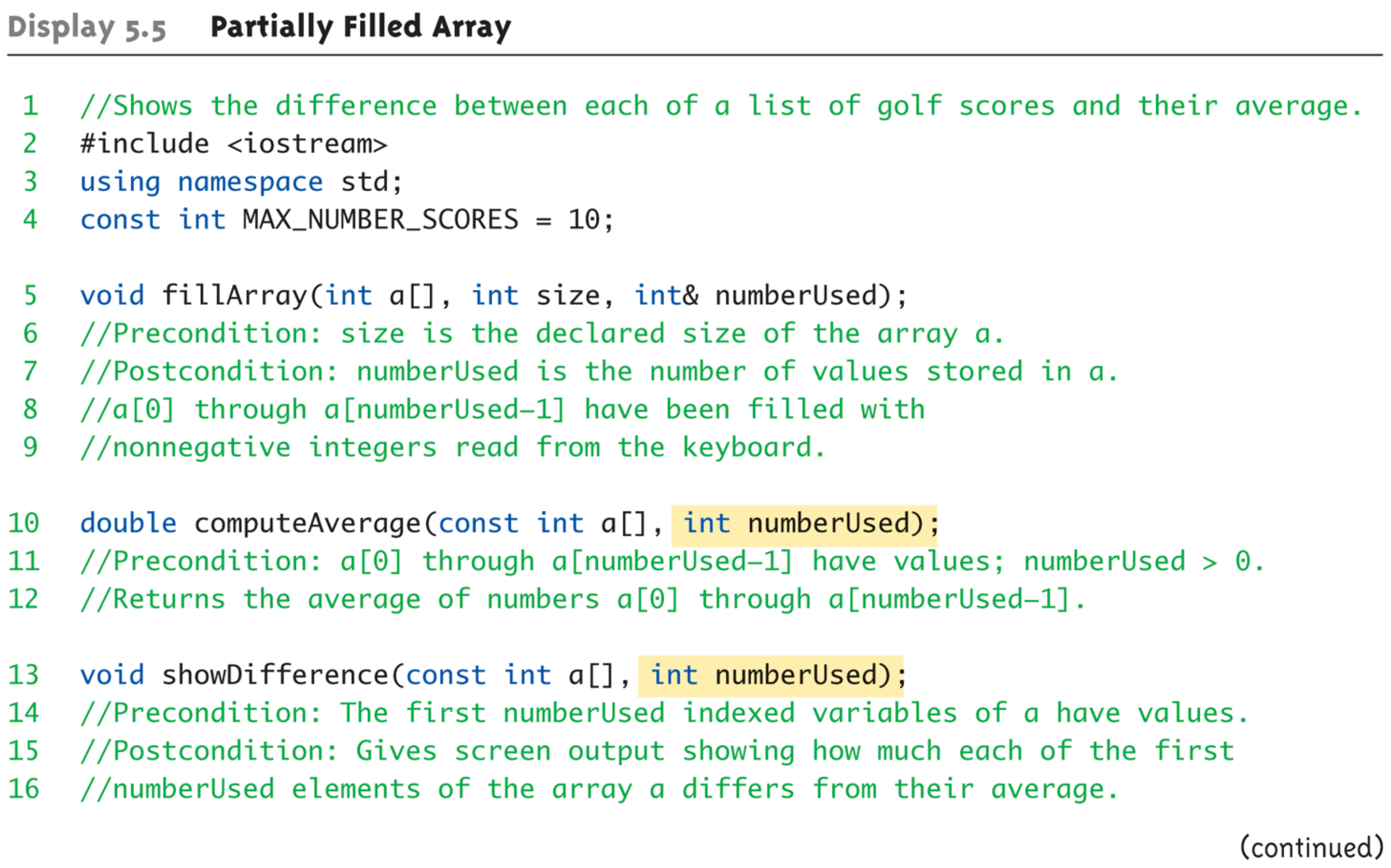
Programming with Arrays

- Plenty of uses
  - Partially-filled arrays
    - Must be declared some "max size"
  - Sorting
  - Searching

Partially-filled Arrays

- Difficult to know exact array size needed
- Must declare to be largest possible size
  - Must then keep "track" of valid data in array
  - Additional "tracking" variable needed
    - int numberUsed;
    - Tracks current number of elements in array

Partially-filled Arrays Example: **Display 5.5** Partially Filled Array (1 of 5)



Display 5.5 **Partially Filled Array**

```
//Shows the difference between each of a list of golf scores and their average.
#include <iostream>
using namespace std;
const int MAX_NUMBER_SCORES = 10;

void fillArray(int a[], int size, int& numberUsed);
//Precondition: size is the declared size of the array a.
//Postcondition: numberUsed is the number of values stored in a.
//a[0] through a[numberUsed-1] have been filled with
//nonnegative integers read from the keyboard.

double computeAverage(const int a[], int numberUsed);
//Precondition: a[0] through a[numberUsed-1] have values; numberUsed > 0.
//Returns the average of numbers a[0] through a[numberUsed-1].

void showDifference(const int a[], int numberUsed);
//Precondition: The first numberUsed indexed variables of a have values.
//Postcondition: Gives screen output showing how much each of the first
//numberUsed elements of the array a differs from their average.
```

(continued)

Partially-filled Arrays Example: **Display 5.5** Partially Filled Array (2 of 5)

Display 5.5 Partially Filled Array

```
int main( )
{
    int score[MAX_NUMBER_SCORES], numberUsed;

    cout << "This program reads golf scores and shows\n"
         << "how much each differs from the average.\n";

    cout << "Enter golf scores:\n";
    fillArray(score, MAX_NUMBER_SCORES, numberUsed);
    showDifference(score, numberUsed);

    return 0;
}
```

Partially-filled Arrays Example: **Display 5.5** Partially Filled Array (3 of 5)

```
void fillArray(int a[], int size, int& numberUsed)
{
    cout << "Enter up to " << size << " nonnegative whole numbers.\n"
         << "Mark the end of the list with a negative number.\n";
    int next, index = 0;
    cin >> next;
    while ((next >= 0) && (index < size))
    {
        a[index] = next;
        index++;
        cin >> next;
    }
    numberUsed = index;
}
```

Partially-filled Arrays Example: **Display 5.5** Partially Filled Array (4 of 5)

```
double computeAverage(const int a[], int numberUsed)
{
    double total = 0;
    for (int index = 0; index < numberUsed; index++)
        total = total + a[index];
    if (numberUsed > 0)
    {
        return (total/numberUsed);
    }
    else
    {
        cout << "ERROR: number of elements is 0 in computeAverage.\n"
             << "computeAverage returns 0.\n";
        return 0;
    }
}
```

Partially-filled Arrays Example: **Display 5.5** Partially Filled Array (5 of 5)

Display 5.5 **Partially Filled Array**

```cpp
void showDifference(const int a[], int numberUsed)
{
    double average = computeAverage(a, numberUsed);
    cout << "Average of the " << numberUsed
         << " scores = " << average << endl
         << "The scores are:\n";
    for (int index = 0; index < numberUsed; index++)
    cout << a[index] << " differs from average by "
         << (a[index] - average) << endl;
}
```

SAMPLE DIALOGUE

```
This program reads golf scores and shows
how much each differs from the average.
Enter golf scores:
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
69 74 68 -1
Average of the 3 scores = 70.3333
The scores are:
69 differs from average by -1.33333
74 differs from average by 3.66667
68 differs from average by -2.33333
```

Global Constants vs. Parameters

- Constants typically made "global"
  - Declared above main()
- Functions then have scope to array size constant
  - No need to send as parameter then?
    - Technically yes
  - Why should we anyway?
    - Function definition might be in separate file
    - Function might be used by other programs!

Searching an Array

- Very typical use of arrays
- Display 5.6 next slide

Display 5.6
Searching an Array (1 of 4)

Display 5.6 Searching an Array

```
//Searches a partially filled array of nonnegative integers.
#include <iostream>
using namespace std;
const int DECLARED_SIZE = 20;

void fillArray(int a[], int size, int& numberUsed);
//Precondition: size is the declared size of the array a.
//Postcondition: numberUsed is the number of values stored in a.
//a[0] through a[numberUsed-1] have been filled with
//nonnegative integers read from the keyboard.

int search(const int a[], int numberUsed, int target);
//Precondition: numberUsed is <= the declared size of a.
//Also, a[0] through a[numberUsed -1] have values.
//Returns the first index such that a[index] == target,
//provided there is such an index; otherwise, returns -1.
```

Display 5.6
Searching an Array (2 of 4)

```
int main( )
{
    int arr[DECLARED_SIZE], listSize, target;

    fillArray(arr, DECLARED_SIZE, listSize);

    char ans;
    int result;
    do
    {
        cout << "Enter a number to search for: ";
        cin >> target;

        result = search(arr, listSize, target);
        if (result == -1)
            cout << target << " is not on the list.\n";
        else
            cout << target << " is stored in array position "
                 << result << endl
                 << "(Remember: The first position is 0.)\n";
```

Display 5.6
Searching an Array (3 of 4)

Display 5.6 Searching an Array

```
        cout << "Search again?(y/n followed by Return): ";
        cin >> ans;
    } while ((ans != 'n') && (ans != 'N'));
    cout << "End of program.\n";
    return 0;
}

void fillArray(int a[], int size, int& numberUsed)
<The rest of the definition of fillArray is given in Display 5.5>
int search(const int a[], int numberUsed, int target)
{
    int index = 0;
    bool found = false;
    while ((!found) && (index < numberUsed))
    if (target == a[index])
        found = true;
    else
        index++;
```

Display 5.6
Searching an Array (4 of 4)

```
        if (found)
            return index;
        else
            return -1;
    }
```

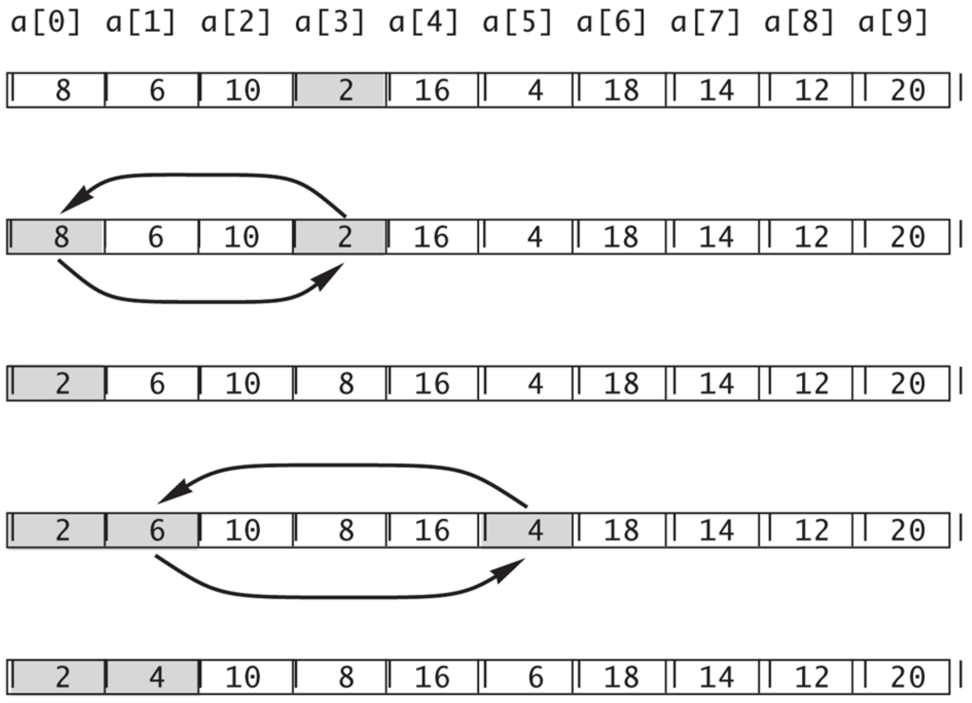
Sample Dialogue

```
Enter up to 20 nonnegative whole numbers.
Mark the end of the list with a negative number.
10 20 30 40 50 60 70 80 -1
Enter a number to search for: 10
10 is stored in array position 0
(Remember: The first position is 0.)
Search again?(y/n followed by Return):  y
Enter a number to search for: 40
40 is stored in array position 3
(Remember: The first position is 0.)
Search again?(y/n followed by Return):  y
Enter a number to search for: 42
42 is not on the list.
Search again?(y/n followed by Return):  n
End of program.
```

Sorting an Array: **Display 5.7** Selection Short

- Selection Sort Algorithm

Display 5.7 Selection Sort



Sorting an Array Example:
Display 5.8 Sorting an Array (1 of 4)

Display 5.8 **Sorting an Array**

```
//Tests the procedure sort.
#include <iostream>
using namespace std;

void fillArray(int a[], int size, int& numberUsed);
//Precondition: size is the declared size of the array a.
//Postcondition: numberUsed is the number of values stored in a.
//a[0] through a[numberUsed – 1] have been filled with
//nonnegative integers read from the keyboard.
void sort(int a[], int numberUsed);
//Precondition: numberUsed <= declared size of the array a.
```

(continued)

Sorting an Array Example: **Display 5.8** Sorting an Array (2 of 4)

Display 5.8 **Sorting an Array**

```cpp
//The array elements a[0] through a[numberUsed – 1] have values.
//Postcondition: The values of a[0] through a[numberUsed – 1] have
//been rearranged so that a[0] <= a[1] <= ... <= a[numberUsed – 1].

void swapValues(int& v1, int& v2);
//Interchanges the values of v1 and v2.

int indexOfSmallest(const int a[], int startIndex, int numberUsed);
//Precondition: 0 <= startIndex < numberUsed. Reference array elements
//have values. Returns the index i such that a[i] is the smallest of the
//values a[startIndex], a[startIndex + 1], ..., a[numberUsed – 1].

int main( )
{
    cout << "This program sorts numbers from lowest to highest.\n";

    int sampleArray[10], numberUsed;
    fillArray(sampleArray, 10, numberUsed);
    sort(sampleArray, numberUsed);

    cout << "In sorted order the numbers are:\n";
    for (int index = 0; index < numberUsed; index++)
        cout << sampleArray[index] << " ";
    cout << endl;

    return 0;
}
```

Sorting an Array Example:
Display 5.8 Sorting an Array (3 of 4)

```
void fillArray(int a[], int size, int& numberUsed)
           <The rest of the definition of fillArray is given in Display 5.5.>

void sort(int a[], int numberUsed)
{
    int indexOfNextSmallest;
    for (int index = 0; index < numberUsed - 1; index++)
    {//Place the correct value in a[index]:
        indexOfNextSmallest =
                     indexOfSmallest(a, index, numberUsed);
        swapValues(a[index], a[indexOfNextSmallest]);
        //a[0] <= a[1] <=...<= a[index] are the smallest of the original array
        //elements. The rest of the elements are in the remaining positions.
    }
}

void swapValues(int& v1, int& v2)
{
    int temp;
    temp = v1;
    v1 = v2;
```

Sorting an Array Example: **Display 5.8** Sorting an Array (4 of 4)

Display 5.8 **Sorting an Array**

```
        v2 = temp;
    }

    int indexOfSmallest(const int a[], int startIndex, int numberUsed)
    {
        int min = a[startIndex],
            indexOfMin = startIndex;
        for (int index = startIndex + 1; index < numberUsed; index++)
            if (a[index] < min)
            {
                min = a[index];
                indexOfMin = index;
                //min is the smallest of a[startIndex] through a[index]
            }

        return indexOfMin;
    }
```

Sample Dialogue

```
This program sorts numbers from lowest to highest.
Enter up to 10 nonnegative whole numbers.
Mark the end of the list with a negative number.
80 30 50 70 60 90 20 30 40 -1
In sorted order the numbers are:
20 30 30 40 50 60 70 80 90
```

Multidimensional Arrays

- Arrays with more than one index
  - char page[30][100];
    - Two indexes: An "array of arrays"
    - Visualize as:
      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]
- C++ allows any number of indexes
  - Typically no more than two

Multidimensional Array Parameters

- Similar to one-dimensional array
  - 1st dimension size not given
    - Provided as second parameter
  - 2nd dimension size IS given
- Example:

```
void DisplayPage(const char p[][100], int sizeDimension1)
{
        for (int index1=0; index1<sizeDimension1; index1++)
        {
                for (int index2=0; index2 < 100; index2++)
                        cout << p[index1][index2];
                cout << endl;
        }
}
```

Summary 1

- Array is collection of "same type" data
- Indexed variables of array used just like any other simple variables
- for-loop "natural" way to traverse arrays
- Programmer responsible for staying "in bounds" of array
- Array parameter is "new" kind
  – Similar to call-by-reference

Summary 2

- Array elements stored sequentially
  - "Contiguous" portion of memory
  - Only address of 1st element is passed to functions
- Partially-filled arrays → more tracking
- Constant array parameters
  - Prevent modification of array contents
- Multidimensional arrays
  - Create "array of arrays"