CHAPTER 5

ARRAYS

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"

```
1)  #include <iostream>
2)  using namespace std;
3)  int main( ) {
4)      int i, score[5], max;
5)      cout << "Enter 5 scores:\n";
6)      cin >> score[0];
7)      max = score[0];
8)      for (i = 1; i < 5; i++)
9)      {
10)         cin >> score[i];
11)         if (score[i] > max)    max = score[i];
12)         //max is the largest of the values score[0],..., score[i].
13)     }
14)     cout << "The highest score is " << max << endl
15)          << "The scores and their\n"
16)          << "differences from the highest are:\n";
17)     for (i = 0; i < 5; i++)
18)         cout << score[i] << " off by " << (max - score[i]) << endl;
19)     return 0;
20) }
```

Reads in 5 scores and shows how much each score differs from the highest score.

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);
```

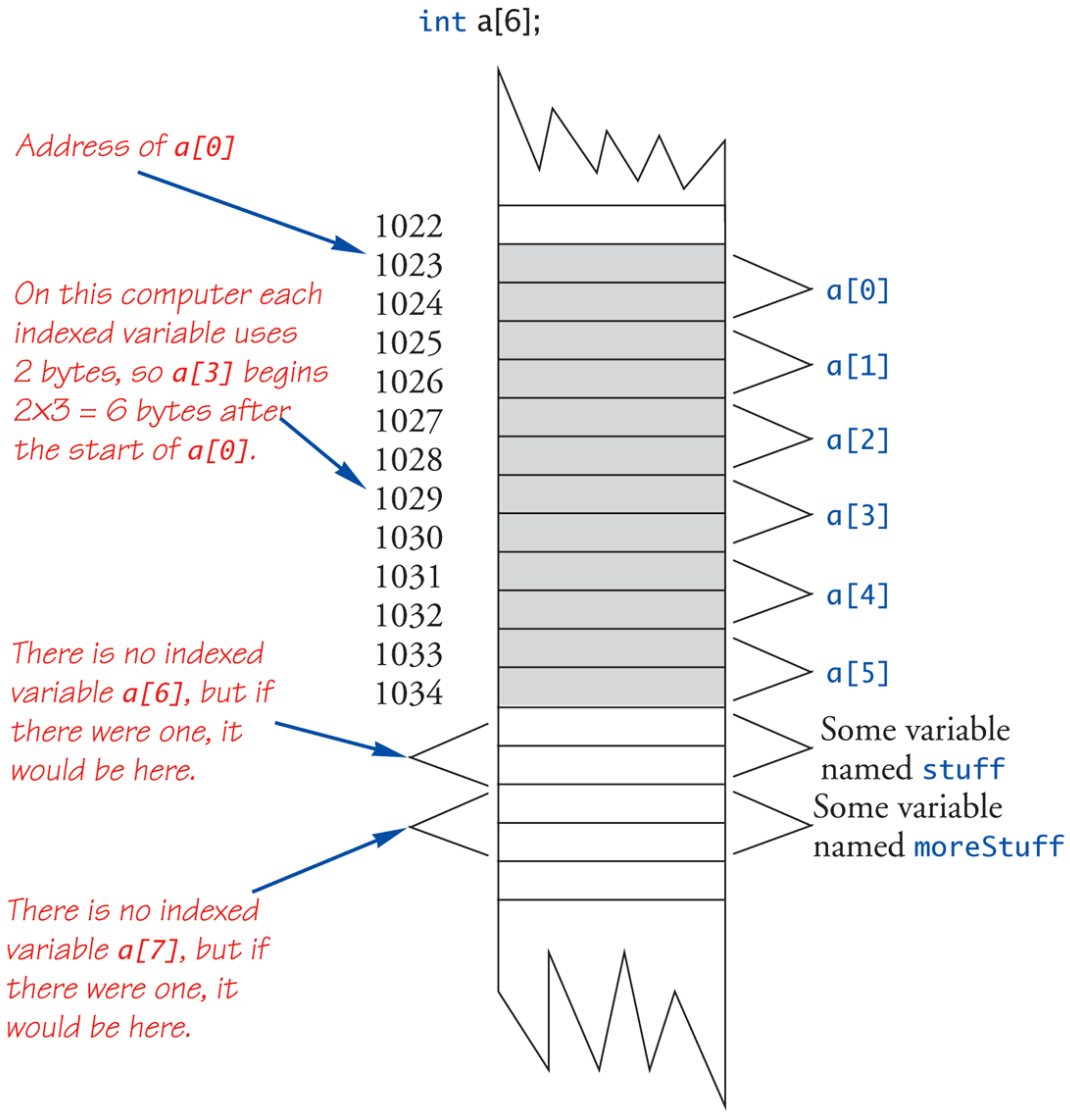
  - When passing array to functions (later)
- If size changes → requires only ONE change in program!

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



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

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

1st argument is entire array

2nd argument is integer value

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

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

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

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

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

(continued)

Display 5.5 **Partially Filled Array**

```
17   int main( )
18  {
19      int score[MAX_NUMBER_SCORES], numberUsed;

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

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

25      return 0;
26  }
```

```
27  void fillArray(int a[], int size, int& numberUsed)
28  {
29      cout << "Enter up to " << size << " nonnegative whole numbers.\n"
30           << "Mark the end of the list with a negative number.\n";
31      int next, index = 0;
32      cin >> next;
33      while ((next >= 0) && (index < size))
34      {
35          a[index] = next;
36          index++;
37          cin >> next;
38      }
39      numberUsed = index;
40  }
```

```
41  double computeAverage(const int a[], int numberUsed)
42  {
43      double total = 0;
44      for (int index = 0; index < numberUsed; index++)
45          total = total + a[index];
46      if (numberUsed > 0)
47      {
48          return (total/numberUsed);
49      }
50      else
51      {
52          cout << "ERROR: number of elements is 0 in computeAverage.\n"
53               << "computeAverage returns 0.\n";
54          return 0;
55      }
56  }
```

Display 5.5 Partially Filled Array

```
57  void showDifference(const int a[], int numberUsed)
58  {
59      double average = computeAverage(a, numberUsed);
60      cout << "Average of the " << numberUsed
61           << " scores = " << average << endl
62           << "The scores are:\n";
63      for (int index = 0; index < numberUsed; index++)
64      cout << a[index] << " differs from average by "
65           << (a[index] - average) << endl;
66  }
```

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

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

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

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

```
1)  int main( )
2)  {
3)      int arr[DECLARED_SIZE], listSize, target, result;

4)      fillArray(arr, DECLARED_SIZE, listSize);
5)      char ans;
6)      do
7)      {
8)          cout << "Enter a number to search for: ";
9)          cin >> target;
10)         result = search(arr, listSize, target);
11)         if (result == -1)   cout << target << " is not on the list.\n";
12)         else out << target << " is stored in array position " << result << endl<<
                        "(Remember: The first position is 0.)\n";
13)         cout << "Search again?(y/n followed by return): ";
14)         cin >> ans;
15)     } while ((ans != 'n') && (ans != 'N'));

16)     cout << "End of program.\n";
17)     return 0;
18) }
```

```
1)  void fillArray(int a[], int size, int& numberUsed)
2)  {
3)      cout << "Enter up to " << size << " nonnegative whole numbers.\n"
4)           << "Mark the end of the list with a negative number.\n";

5)      int next, index = 0;
6)      cin >> next;
7)      while ((next >= 0) && (index < size))
8)      {
9)          a[index] = next;
10)         index++;
11)         cin >> next;
12)     }

13)     numberUsed = index;
14)}
```

```
1)  int search(const int a[], int numberUsed, int target)
2)  {
3)      int index = 0;
4)      bool found = false;
5)      while ((!found) && (index < numberUsed))
6)      if (target == a[index])
7)          found = true;
8)      else
9)          index++;

10)     if (found)
11)         return index;
12)     else
13)         return -1;
14) }
```

DISPLAY 5.6
SEARCHING AN ARRAY

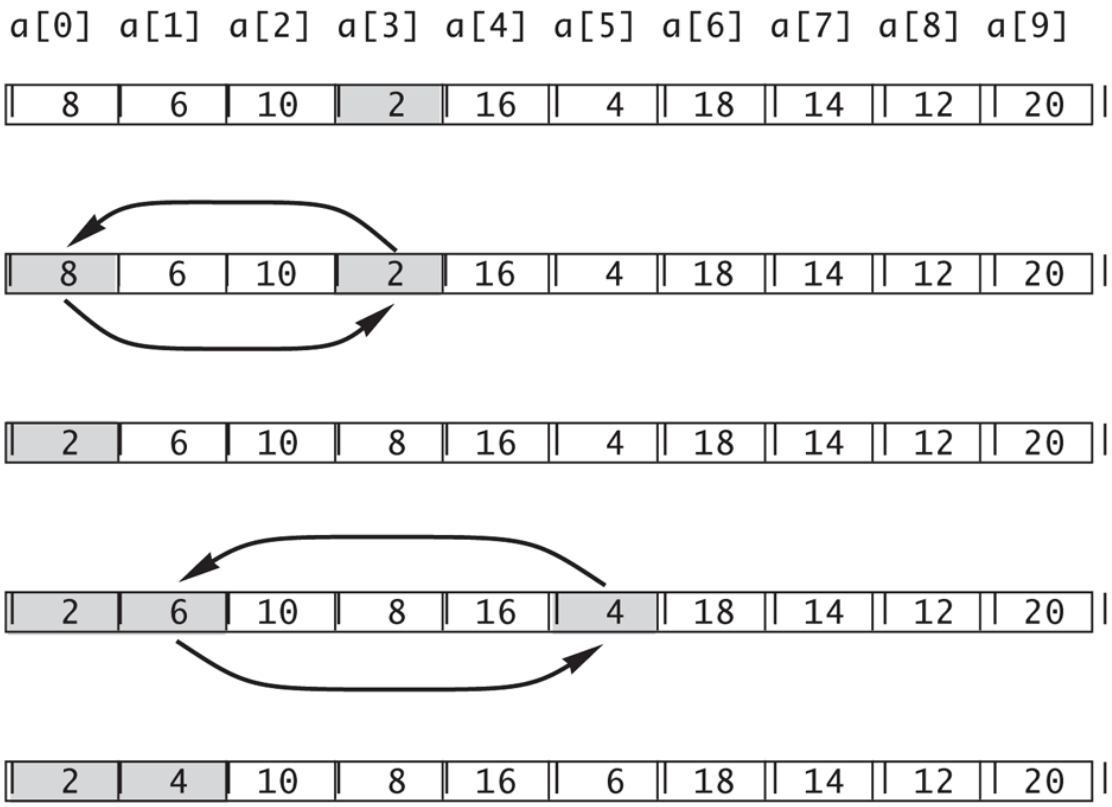
SAMPLE DIALOGUE

```
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

```
1)  #include <iostream>
2)  using namespace std;

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

8)  void sort(int a[], int numberUsed);
9)  //Precondition: numberUsed <= declared size of the array a.
10) //The array elements a[0] through a[numberUsed - 1] have values.
11) //Postcondition: The values of a[0] through a[numberUsed - 1] have
12) //been rearranged so that a[0] <= a[1] <= ... <= a[numberUsed -
        1].
```

SORTING AN ARRAY EXAMPLE

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

15)int indexOfSmallest(const int a[], int startIndex, int
    numberUsed);
16)//Precondition: 0 <= startIndex < numberUsed. References
    array elements have values.
17)//Returns the index i such that a[i] is the smallest of the values
18)//a[startIndex], a[startIndex + 1], ..., a[numberUsed - 1].
```

```
19)int main( )
20){
21)     cout << "This program sorts numbers from lowest to highest.\n";
22)     int sampleArray[10], numberUsed;
23)     fillArray(sampleArray, 10, numberUsed);
24)     sort(sampleArray, numberUsed);

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

29)     return 0;
30)}
```

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
```

```
1)  void fillArray(int a[], int size, int& numberUsed)
2)  {
3)      cout << "Enter up to " << size << " nonnegative whole
    numbers.\n"
4)           << "Mark the end of the list with a negative number.\n";

5)      int next, index = 0;
6)      cin >> next;
7)      while ((next >= 0) && (index < size))
8)      {
9)          a[index] = next;
10)         index++;
11)         cin >> next;
12)     }

13)     numberUsed = index;
14)}
```

```
1)  void sort(int a[], int numberUsed)
2)  {
3)     int indexOfNextSmallest;
4)     for (int index = 0; index < numberUsed - 1; index++)
5)     {//Place the correct value in a[index]:
6)        indexOfNextSmallest = indexOfSmallest(a, index, numberUsed);
7)        swapValues(a[index], a[indexOfNextSmallest]);
8)      //a[0] <= ...<= a[index] are the smallest of the original array
9)      //elements. The rest of the elements are in the remaining positions.
10)    }
11)}

12)void swapValues(int& v1, int& v2)
13){
14)    int temp;
15)    temp = v1;    v1 = v2;    v2 = temp;
16)}
```

```
1)  int indexOfSmallest(const int a[], int startIndex, int numberUsed)
2)  {
3)      int min = a[startIndex],
4)          indexOfMin = startIndex;
5)      for (int index=startIndex+ 1; index < numberUsed; index++)
6)          if (a[index] < min)
7)          {
8)              min = a[index];
9)              indexOfMin = index;
10)              //min is the smallest of a[startIndex] through a[index]
11)          }

12)     return indexOfMin;
13)}
```

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:

```
1) void DisplayPage(const char p[][100], int sizeDimension1)
2) {
3)     for (int index1=0; index1<sizeDimension1; index1++)
4)     {
5)         for (int index2=0; index2 < 100; index2++)
6)             cout << p[index1][index2] << endl;
7)     }
8) }
```

MULTIDIMENSIONAL ARRAY EXAMPLE

```
1)  //Reads quiz scores for each student into the two-dimensional
    array grade (Code to fill array has been o
2)  //Computes the average score for each stu
3)  //the average score for each quiz. Display                s and
    the averages.
4)  #include <iostream>
5)  #include <iomanip>
6)  using namespace std;
7)  const int NUMBER_STUDENTS = 4, NUMBER_QUIZZES = 3;

8)    void computeStAve(const int grade[][NUMBER_QUIZZES], double stAve[]);

9)    void computeQuizAve(const int grade[][NUMBER_QUIZZES], double quizAve[]);

10) void display(const int grade[][NUMBER_QUIZZES],
11)                       const double stAve[], const double quizAve[]);
```

```
1)  int main( )
2)  {
3)      int grade[NUMBER_STUDENTS][NUMBER_QUIZZES];
4)      double stAve[NUMBER_STUDENTS];
5)      double quizAve[NUMBER_QUIZZES];

6)      grade[0][0] = 10; grade[0][1] = 10; grade[0][2] = 10;
7)      grade[1][0] = 2;  grade[1][1] = 0;  grade[1][2] = 1;
8)      grade[2][0] = 8;  grade[2][1] = 6;  grade[2][2] = 9;
9)      grade[3][0] = 8;  grade[3][1] = 4;  grade[3][2] = 10;

10)     computeStAve(grade, stAve); //i.e., row avg
11)     computeQuizAve(grade, quizAve);  // i.e., col avg
12)     display(grade, stAve, quizAve);
13)     return 0;
14)}
```

```
1)  void computeStAve(const int grade[][NUMBER_QUIZZES], double stAve[])
2) {
3)     for (int stNum = 1; stNum <= NUMBER_STUDENTS; stNum++)
4)     {//Process one stNum:
5)        double sum = 0;
6)        for (int quizNum = 1; quizNum <= NUMBER_QUIZZES; quizNum++)
7)           sum = sum + grade[stNum-1][quizNum-1];
8)        //sum contains the sum of the quiz scores for student number stNum.
9)        stAve[stNum-1] = sum/NUMBER_QUIZZES;
10)       //Average for student stNum is the value of stAve[stNum-1]
11)    }
12)}
```

```
1)  void computeQuizAve(const int grade[][NUMBER_QUIZZES], double quizAve[])
2)  {
3)      for (int quizNum = 1; quizNum <= NUMBER_QUIZZES; quizNum++)
4)      {//Process one quiz (for all students):
5)          double sum = 0;
6)          for (int stNum = 1; stNum <= NUMBER_STUDENTS; stNum++)
7)              sum = sum + grade[stNum-1][quizNum-1];
8)          //sum contains the sum of all student scores on quiz number quizNum.
9)          quizAve[quizNum-1] = sum/NUMBER_STUDENTS;
10)         //Average for quiz quizNum is the value of quizAve[quizNum-1]
11)     }
12)}
```

```
1)  void display(const int grade[][NUMBER_QUIZZES],
                                        const double stAve[], const double quizAve[])
2)  {   cout.setf(ios::fixed);
3)      cout.setf(ios::showpoint);
4)      cout.precision(1);

5)      cout << setw(10) << "Student“ << setw(5) << "Ave" << setw(15) <<
    "Quizzes\n";
6)      for (int stNum = 1; stNum <= NUMBER_STUDENTS; stNum++)
7)      {//Display for one stNum:
8)          cout << setw(10) << stNum << setw(5) << stAve[stNum-1] << " ";
9)          for (int quizNum = 1; quizNum <= NUMBER_QUIZZES; quizNum++)
10)             cout << setw(5) << grade[stNum-1][quizNum-1];
11)         cout << endl;
12)     }

13)     cout << "Quiz averages = ";
14)     for (int quizNum = 1; quizNum <= NUMBER_QUIZZES; quizNum++)
15)         cout << setw(5) << quizAve[quizNum-1];
16)     cout << endl;
17) }
```

ANY DISADVANTAGES?

- Using:
  1. int arr[DECLARED_SIZE],
     void fillArray(int a[], int size, **int&** numberUsed);
  2. **char** page[30][100];
     void DisplayPage(const char p**[]**[**100**], int sizeDimension1)

Have **disadvantages** about?

- Declaration,
- Parameter passing,
- Resizing (insertion, deletion), etc.

```
1)  #include <iostream>
2)  #include <vector>
3)  using namespace std;
4)  int binarySearch(int , int , int , vector<int>&);

5)  int main()
6)  {
7)      vector<int> random(100);
8)      int search4(3), found, first = 0,  last = 99;
9)      found = binarySearch(first, last, search4, random);
10)     return(0);
11) }

12) int binarySearch(int first, int last, int search4, vector<int>& random)
13) {
14)     do
15)     {
16)         int mid = (first + last) / 2;
17)         if (search4 > random[mid])          first = mid + 1;
18)         else if (search4 < random[mid])          last = mid - 1;
19)         else            return mid;
20)     } while (first <= last);
21)     return -(first + 1);
22) }
```

Why &?

USING VECTOR INSTEAD

Better?

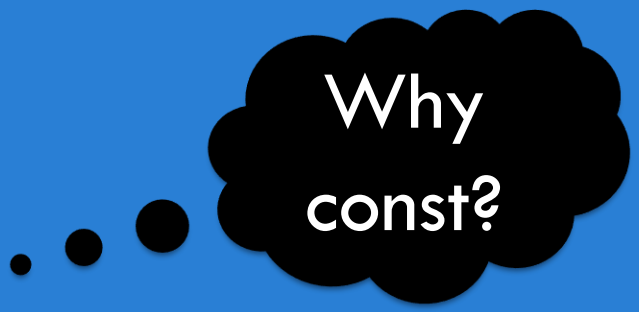
```
1)  void f1(vector<int> & v)
2)  {
3)      //do what you want here...
4)  }

5)  void f2(const vector<int> & v)
6)  {
7)      //do what you want here...
8)  }

9)  int main()
10){
11)     vector<int> v1;
12)     vector<int> v2;

13)     f1(v1);
14)     f2(v2);

15)     return 0;
16)}
```



Constructors of vector

std::vector::vector

```
explicit vector( const Allocator& alloc = Allocator() );                    (1)  (until C++14)
vector() : vector( Allocator() ) {}                                              (since C++14)
explicit vector( const Allocator& alloc );

explicit vector( size_type count,                                                (until C++11)
                 const T& value = T(),
                 const Allocator& alloc = Allocator());                      (2)
         vector( size_type count,
                 const T& value,                                                 (since C++11)
                 const Allocator& alloc = Allocator());

                                                                                 (since C++11)
explicit vector( size_type count );                                          (3) (until C++14)
explicit vector( size_type count, const Allocator& alloc = Allocator() );        (since C++14)

template< class InputIt >
vector( InputIt first, InputIt last,                                         (4)
        const Allocator& alloc = Allocator() );

vector( const vector& other );                                               (5)
vector( const vector& other, const Allocator& alloc );                       (5)
vector( vector&& other )                                                     (6)
vector( vector&& other, const Allocator& alloc );                            (6)
vector( std::initializer_list<T> init,                                       (7)  (since C
        const Allocator& alloc = Allocator() );
```

Which one used in examples?

Constructs a new container from a variety of data sources, optionally using a user supplied allocator `alloc`.

1) Default constructor. Constructs an empty container.
2) Constructs the container with count copies of elements with value value.
3) Constructs the container with count default-inserted instances of T. No copies are made.
4) Constructs the container with the contents of the range `[first, last)`.

This constructor has the same effect as overload (2) if `InputIt` is an integral type.	(until C++11)
This overload only participates in overload resolution if `InputIt` satisfies `InputIterator` , to avoid ambiguity with the overload (2).	(since C++11)

5) Copy constructor. Constructs the container with the copy of the contents of other. If `alloc` is not provided, allocator is obtained by calling `std::allocator_traits<allocator_type>::select_on_container_copy_construction(other.get_allocator()`.
6) Move constructor. Constructs the container with the contents of other using move semantics. If `alloc` is not provided, allocator is obtained by move-construction from the allocator belonging to other.
7) Constructs the container with the contents of the initializer list `init`.

MULTIDIMENSIONAL ARRAYS USING VECTOR

```
1) std::vector<int> va(5, 1);    // vector a
2) std::vector<int> vb(5, 2);    // vector b
3) std::vector<int> vc(5, 3);    // vector c
4) std::vector< std::vector<int> > vv {va, vb, vc};
   // vector of vectors
```

MULTIDIMENSIONAL ARRAYS USING VECTOR

1) vector<vector<point> > a; // 2D array
2) a.push_back(vector<point>());
3) point p, pp;
4) a[0].push_back(p);
5) a[0][0]+= pp; // so a[0][0]= ?
6) a[0].push_back(pp); // a[0][1]= ?
7) a[0][2]= a[0][1]+ a[0][0]; // a[0][2]= ?

- If you know how many items you will have from the start, you can do :

```
vector<vector<point> > a(10, vector<point>(10));
```

- It's a vector containing 10 vectors containing 10 point. Then you can use

```
a[4][4] = p;  // access an element at a[4][4]
```

BOOST::MATRIX

- #include <boost/numeric/ublas/matrix.hpp>
- #include <boost/numeric/ublas/io.hpp>

```
1) int main () {
2)     using namespace boost::numeric::ublas;
3)     matrix<double> m (3, 3);
4)     for (unsigned i = 0; i < m.size1 (); ++ i)
5)         for (unsigned j = 0; j < m.size2 (); ++ j)
6)             m (i, j) = 3 * i + j;
7)     std::cout << m << std::endl;
8) }
```

3D ARRAY WITH VECTORS

```
1)  #include <vector>
2)  using std::vector;
3)  #define HEIGHT 5
4)  #define WIDTH 3
5)  #define DEPTH 7
6)  int main() {
7)    vector<vector<vector<double> > > array3D;
8)    // Set up sizes. (HEIGHT x WIDTH)
9)    array3D.resize(HEIGHT);
10)   for (int i = 0; i < HEIGHT; ++i) {
11)     array3D[i].resize(WIDTH);
12)     for (int j = 0; j < WIDTH; ++j)      array3D[i][j].resize(DEPTH);
13)   }
14)   // Put some values in
15)   array3D[1][2][5] = 6.0;
16)   array3D[3][1][4] = 5.5;
17)   return 0;
18) }
```

POINTER BASED 2D ARRAY

```
1)  #define HEIGHT 5
2)  #define WIDTH 3
3)  int main() {
4)    double **p2DArray;
5)    // Allocate memory
6)    p2DArray = new double* [HEIGHT];
7)    for (int i = 0; i < HEIGHT; ++i)
8)       p2DArray[i] = new double[WIDTH];
9)    // Assign values
10)   p2DArray[0][0] = 3.6;
11)   p2DArray[1][2] = 4.0;
12)   // De-Allocate memory to prevent memory leak
13)   for (int i = 0; i < HEIGHT; ++i)    delete [] p2DArray[i];
14)   delete [] p2DArray;

15)   return 0;
16)}
```

POINTER BASED 3D ARRAY

```
1)  // #define HEIGHT 5  #define WIDTH 3  #define DEPTH 7
2)  int main() {
3)    double ***p2DArray;
4)    // Allocate memory
5)    p2DArray = new double**[HEIGHT];
6)    for (int i = 0; i < HEIGHT; ++i) {
7)      p2DArray[i] = new double*[WIDTH];
8)      for (int j = 0; j < WIDTH; ++j)      p2DArray[i][j] = new double[DEPTH];
9)    }
10)   // Assign values
11)   p2DArray[0][0][0] = 3.6;
12)   p2DArray[1][2][4] = 4.0;
13)   // De-Allocate memory to prevent memory leak
14)   for (int i = 0; i < HEIGHT; ++i) {
15)     for (int j = 0; j < WIDTH; ++j)      delete [] p2DArray[i][j];
16)     delete [] p2DArray[i];
17)   }
18)   delete [] p2DArray;
19)   return 0;
20) }
```

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"