

# Review of Everything Arrays

- ❑ Array Basics
- ❑ Arrays in Functions
- ❑ Programming with Arrays
- ❑ Simple linear search, simple selection sort.
- ❑ Multi-dimensional Arrays

# Array Basics

# Arrays

- An array is used to process a collection of data of the same type
  - Examples: A list of names  
A list of temperatures
- Why do we need arrays?
  - Imagine keeping track of 5 test scores, or 100, or 1000 in memory
    - How would you name all the variables?
    - How would you process each of the variables?

# Declaring an Array

- ❑ An array, named `score`, containing five variables of type `int` can be declared as  
`int score[5];`
- ❑ This is like declaring 5 variables of type `int`:  
`score[0], score[1], ... , score[4]`
- ❑ The value in brackets is called
  - A subscript
  - An index
  - Eg. Score sub 0, score sub 1, .... , score sub 4

# The Array Variables

- ❑ The variables making up the array are referred to as
  - Indexed variables
  - Subscripted variables
  - Elements of the array
- ❑ The number of indexed variables in an array is **the declared size, or size, of the array**
  - The largest index is one less than the size
  - The first index value is zero
  - Not all variables are actually being used all the time!

# Array Variable Types

- An array can have indexed variables of any type
- All indexed variables in an array are of the same type
  - This is the **base type** of the array
- An **indexed variable** can be used anywhere an ordinary variable of the base type is used

# Using [ ] With Arrays

- ❑ In an array **declaration**, [ ]'s enclose the size of the array such as this array of 5 integers:  
`int score [5];`
- ❑ When referring to one of the indexed variables, the [ ]'s enclose a number identifying one of the indexed variables
  - E.g.,  
`score[3]=7;`  
`score[3]` is one of the indexed variables
  - The value in the [ ]'s can be any expression that evaluates to one of the integers 0 to (size -1)

# Indexed Variable Assignment

- To assign a value to an indexed variable, use the assignment operator:

```
int n = 2;  
score[n + 1] = 99;
```

- In this example, variable `score[3]` is assigned 99



# Loops And Arrays

- for-loops are commonly used to step through arrays

First index is 0

Last index is (size - 1)

- Example: 

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

could display the difference between each score and the maximum score stored in an array

**Display 7.1**

# Let's write a program with an Array

- ❑ Write a program to read in 5 scores and find the max score and then print the difference between each score and the max.
- ❑ What do we have to do?
  - Declare some variables
  - How do we figure out the max score?
    - Where do we start?
  - What is the program structure?
    - What kind of statements are needed?

## Program Using an Array

```
//Reads in 5 scores and shows how much each
//score differs from the highest score.
#include <iostream>

int main()
{
    using namespace std;
    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].
    }

    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;
}
```

## Display 7.1

### 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
```

# Constants and Arrays

- Use constants to declare the size of an array
  - Using a constant allows your code to be easily altered for use on a smaller or larger set of data
    - Example:

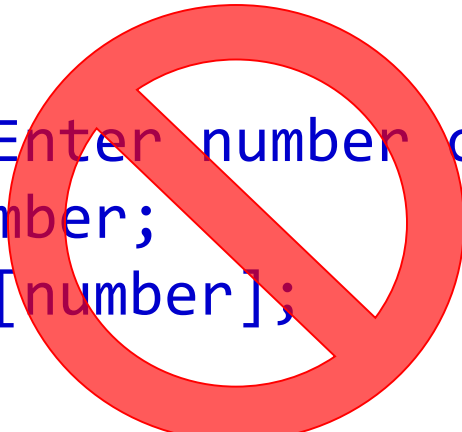
```
const int  NUMBER_OF_STUDENTS = 50;
int score[NUMBER_OF_STUDENTS];
for ( i = 0; i < NUMBER_OF_STUDENTS; i++)
    cout << score[i] << " off by " << (max - score[i]) << endl;
```
    - Only the value of the constant must be changed to make this code work for any number of students

# Variables and Declarations

- ❑ Most compilers do not allow the use of a variable to declare the size of an array

Example:

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



- This code is illegal on many compilers
- But it works on our version of g++. It is an extension.

# Array Declaration Syntax

- ❑ To declare an array, use the syntax:

Type\_Name      Array\_Name[Declared\_Size];

- Type\_Name can be any type
  - Declared\_Size can be a constant to make your program more versatile
- ❑ Once declared, the array consists of the indexed variables:  
Array\_Name[0] to Array\_Name[Declared\_Size-1]

# Arrays and Memory

## ❑ Declaring the array

```
int a[6];
```

- Reserves memory for six variables of type int
- The variables are stored one after another
- The address of a[0] is remembered by C++
  - The addresses of the other indexed variables are not remembered by C++
- To determine the address of a[3]
  - C++ starts at a[0]
  - C++ adds  $a[0] + 3 * \text{sizeof}(\text{int})$  to get to a[3].

**Display 7.2**

### Display 7.2

in this example, each int variable uses 2 bytes, but typically an int variable uses 4 bytes.

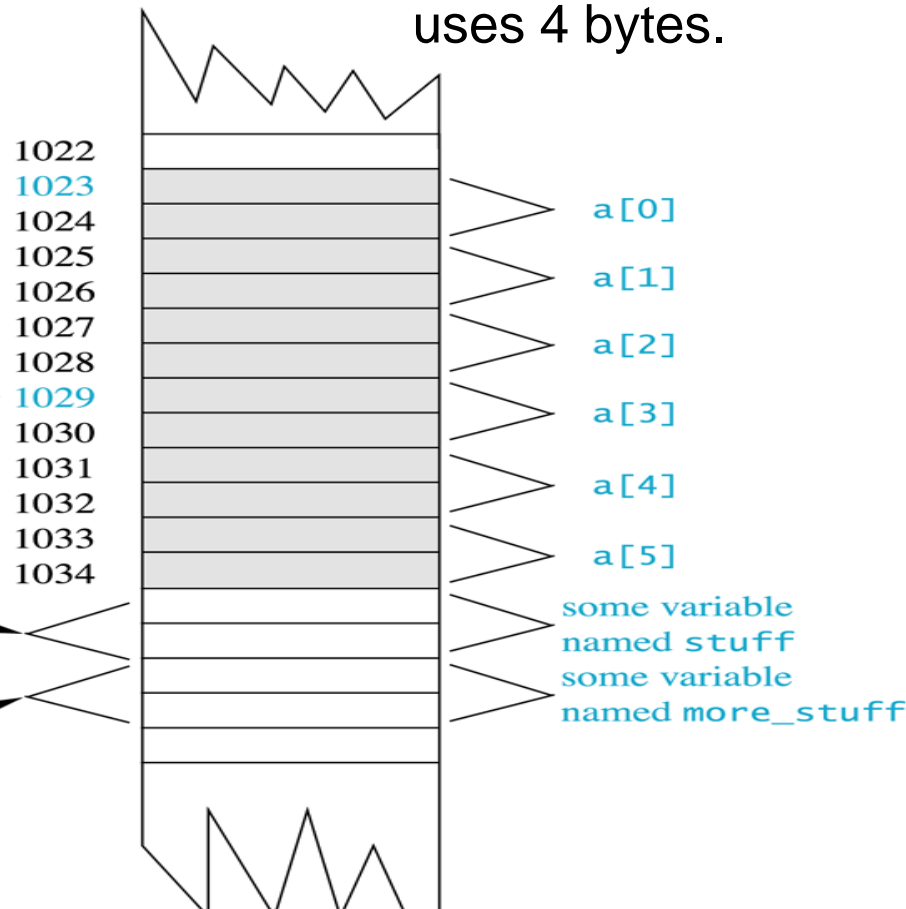
address of a[0]

On this computer each indexed variable uses 2 bytes, so a[3] begins  $2 \times 3 = 6$  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.

`int a[6];`



#### Recall:

- ❑ Computer memory consists of numbered locations called **bytes**
  - A byte's number is its address
- ❑ A simple variable is stored in consecutive bytes
  - The number of bytes depends on the variable's type
- ❑ A variable's address is the address of its first byte



# Array Index Out of Range

- ❑ A common error is using a nonexistent index
  - Index values for `int a[6]` are the values 0 through 5
  - An index value not allowed by the array declaration is out of range
  - Using an out of range index value does not produce an error message!

# Out of Range Problems

- ❑ If an array is declared as: `int a[6];`  
and an integer is declared as: `int i = 7;`
- ❑ Executing the statement `a[i] = 238;` causes...
  - The computer to calculate the address of the illegal `a[7]`  
(This address could be where some other variable is stored)
  - The value 238 is stored at the address calculated for `a[7]`
  - No warning is given!
- ❑ What happens if `i = 6`?

# Initializing Arrays

- To initialize an array when it is declared
  - The values for the indexed variables are enclosed in braces and separated by commas
- Example: `int children[3] = { 2, 12, 1 };`  
is equivalent to:

```
int children[3];  
children[0] = 2;  
children[1] = 12;  
children[2] = 1;
```

# Default Values

- If too few values are listed in an initialization statement
  - The listed values are used to initialize the first of the indexed variables
  - The remaining indexed variables are initialized to a zero of the base type
  - Example: `int a[10] = {5, 5};`  
initializes `a[0]` and `a[1]` to 5 and `a[2]` through `a[9]` to 0

# Un-initialized Arrays

- If no values are listed in the array declaration, **some compilers** will initialize each variable to the equivalent of zero of the base type
  - Don't rely on this if you are writing code that will be shared with a large development community.

# Arrays in Functions

# Arrays in Functions

- Indexed variables can be arguments to functions

- Example: If a program contains these declarations:

```
int i, n, a[10];  
void my_function(int n);
```

- Variables a[0] through a[9] are of type int, making these calls legal:

```
my_function( a[ 0 ] );  
my_function( a[ 3 ] );  
my_function( a[ i ] );
```

**Display 7.3**

# Program to adjust vacation days

- ❑ Let's assume that a company changes it's policy to be 5 more vacation days per year.
- ❑ Using an array, ask the user to input the number of vacation days for each employee.
- ❑ Write a function called `adjust_days` that adds 5 to the number passed in and returns the result.
- ❑ Print the adjusted array.



```
//Illustrates the use of an indexed variable as an argument.
//Adds 5 to each employee's allowed number of vacation days.
#include <iostream>

const int NUMBER_OF_EMPLOYEES = 3;

int adjust_days(int old_days);
//Returns old_days plus 5.

int main()
{
    using namespace std;
    int vacation[NUMBER_OF_EMPLOYEES], number;

    cout << "Enter allowed vacation days for employees 1"
          << " through " << NUMBER_OF_EMPLOYEES << ":\n";
    for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
        cin >> vacation[number-1];

    for (number = 0; number < NUMBER_OF_EMPLOYEES; number++)
        vacation[number] = adjust_days(vacation[number]);

    cout << "The revised number of vacation days are:\n";
    for (number = 1; number <= NUMBER_OF_EMPLOYEES; number++)
        cout << "Employee number " << number
              << " vacation days = " << vacation[number-1] << endl;

    return 0;
}

int adjust_days(int old_days)
{
    return (old_days + 5);
}
```

### Sample Dialogue

Enter allowed vacation days for employees 1 through 3:

**10 20 5**

The revised number of vacation days are:

Employee number 1 vacation days = 15

Employee number 2 vacation days = 25

Employee number 3 vacation days = 10

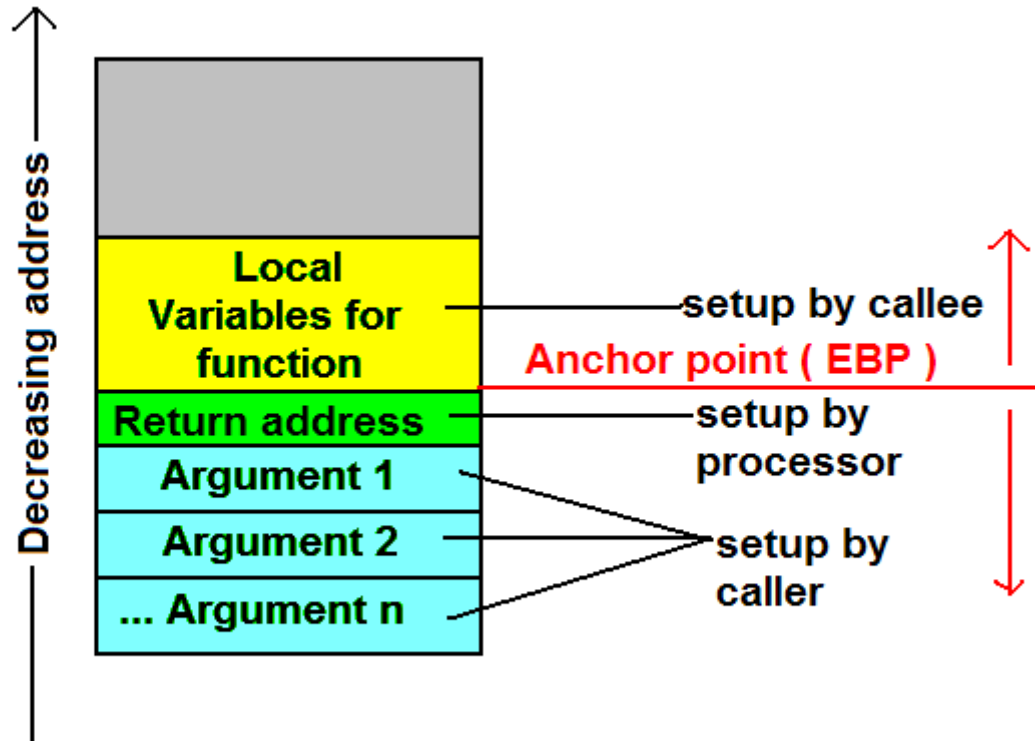
# Arrays as Function Arguments

- A formal parameter can be for an entire array
  - Such a parameter is called **an array parameter**
    - It is not a call-by-value parameter
    - It is not a call-by-reference parameter
    - Array parameters behave much like call-by-reference parameters because
    - Array parameters are just pointers to the memory that was allocated for the array.

# Stack Frame

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

- Argument1 is 4 bytes for the address of a[]
- Argument2 is 4 bytes for the value in size.



When a function is called, parameters are placed on the stack in order.

- If it is an array parameter, only the address of the first element is put on the stack
- If it is a call-by-value parameter, the value is copied onto the stack.
- If it is call-by-reference, the address of the variable is put on the stack.

# Array Parameter Declaration

- ❑ An array parameter is indicated using empty brackets in the parameter list such as

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

- ❑ This is why the size is needed.
- ❑ Unless there is a special value at the end that indicates it's the last value called a **terminal** or **sentinal** value.

# Function Calls With Arrays

- ❑ If function `fill_up` is declared in this way:  
`void fill_up( int a[ ] , int size);`
- ❑ and array `score` is declared this way:  
`int score[5], number_of_scores;`
- ❑ `fill_up` is called in this way:  
`fill_up(score, number_of_scores);`

**Display 7.4**

# Array Parameter Considerations

- Because a function does not know the size of an array argument...
  - The programmer should include a formal parameter that specifies the size of the array
  - The function can process arrays of various sizes
    - Function `fill_up` from Display 7.4 can be used to fill an array of any size:

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

# Function Call Details

- A formal parameter is identified as an array parameter by the [ ]'s with no index expression

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

- An array argument does not use the [ ]'s

```
fill_up(score, number_of_scores);
```

# Array Argument Details

- What does the computer know about an **array** once it is declared?
  - The base type
  - The address of the first indexed variable
  - The number of indexed variables
- What does a function know about an **array argument** during a function call?
  - The base type
  - The address of the first indexed variable



# Returning An Array

- ❑ Recall that functions can return (via return-statement) a value of type int, double, char, ...
- ❑ **Functions cannot return arrays**
- ❑ We learn later how to return a pointer to an array

# Programming with Arrays

# Programming With Arrays

- ❑ The size needed for an array is changeable
  - Often varies from one run of a program to another
  - Is often not known when the program is written
- ❑ A common solution to the size problem
  - Declare the array size to be the largest that could be needed
  - Decide how to deal with partially filled arrays
  - Manage two things:
    - Capacity - total number of elements allowed
    - Size - total number of elements inserted

# Partially Filled Arrays

- When using arrays that are partially filled
  - A parameter, **number\_used**, may be sufficient to ensure that referenced index values are legal
  - 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 function such as **fill\_array** in Display 7.9 needs to know the declared size of the array

Display 7.9 (1)

Display 7.9 (2)

Display 7.9 (3)

# Program to show how golf scores differ from the average score

- ❑ Write a program to compute average golf scores. Show how each score differs from average. We need...
- ❑ a function called `fill_array` that gets golf scores and puts them in an array parameter.
- ❑ a function called `compute_average` that computes the average of the scores.
- ❑ a function that `shows_difference` calls `compute_average`.
- ❑ `main()` calls `fill_array` and `shows_difference`.

## Partially Filled Array (part 1 of 3)

---

*//Shows the difference between each of a list of golf scores and their average.*

```
#include <iostream>
```

```
const int MAX_NUMBER_SCORES = 10;
```

```
void fill_array(int a[], int size, int& number_used);
```

*//Precondition: size is the declared size of the array a.*

*//Postcondition: number\_used is the number of values stored in a.*

*//a[0] through a[number\_used-1] have been filled with*

*//nonnegative integers read from the keyboard.*

```
double compute_average(const int a[], int number_used);
```

*//Precondition: a[0] through a[number\_used-1] have values; number\_used > 0.*

*//Returns the average of numbers a[0] through a[number\_used-1].*

```
void show_difference(const int a[], int number_used);
```

*//Precondition: The first number\_used indexed variables of a have values.*

*//Postcondition: Gives screen output showing how much each of the first*

*//number\_used elements of a differs from their average.*

```
int main()
```

```
{
```

```
    using namespace std;
```

```
    int score[MAX_NUMBER_SCORES], number_used;
```

```
    cout << "This program reads golf scores and shows\n"
```

```
         << "how much each differs from the average.\n";
```

```
    cout << "Enter golf scores:\n";
```

```
    fill_array(score, MAX_NUMBER_SCORES, number_used);
```

```
    show_difference(score, number_used);
```

```
    return 0;
```

```
}
```

```
//Uses iostream:
```

```
void fill_array(int a[], int size, int& number_used)
```

```
{
```

```
    using namespace std;
```

```
    cout << "Enter up to " << size << " nonnegative whole numbers.\n"
```

```
         << "Mark the end of the list with a negative number.\n";
```

Display 7.9  
(1/3)

```
int next, index = 0;
cin >> next;
while ((next >= 0) && (index < size))
{
    a[index] = next;
    index++;
    cin >> next;
}

number_used = index;
}

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

void show_difference(const int a[], int number_used)
{
    using namespace std;
    double average = compute_average(a, number_used);
    cout << "Average of the " << number_used
          << " scores = " << average << endl
          << "The scores are:\n";
    for (int index = 0; index < number_used; index++)
        cout << a[index] << " differs from average by "
              << (a[index] - average) << endl;
}
```

---

# Display 7.9

## (2/3)

# Display 7.9

## (3/3)

### Partially Filled Array (*part 3 of 3*)

---

#### 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

---



# Searching Arrays

- A sequential search is one way to search an array for a given value
  - Look at each element from first to last to see if the target value is equal to any of the array elements
  - The index of the target value can be returned to indicate where the value was found in the array
  - A value of -1 can be returned if the value was not found

# The search Function

- ❑ The search function of Display 7.10...
  - Uses a while loop to compare array elements to the target value
  - Sets a variable of type `bool` to true if the target value is found, ending the loop
  - Checks the boolean variable when the loop ends to see if the target value was found
  - Returns the index of the target value if found, otherwise returns -1

**Display 7.10 (1)**

**Display 7.10 (2)**

## Searching an Array (part 1 of 2)

---

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

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

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

int main()
{
    using namespace std;
    int arr[DECLARED_SIZE], list_size, target;

    fill_array(arr, DECLARED_SIZE, list_size);

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

        result = search(arr, list_size, 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";

        cout << "Search again?(y/n followed by Return): ";
        cin >> ans;
    }while ((ans != 'n') && (ans != 'N'));

    cout << "End of program.\n";
    return 0;
}
```

# Display 7.10 (1/2)

```
//Uses iostream:
void fill_array(int a[], int size, int& number_used)
<The rest of the definition of fill_array is given in Display 10.9.>

int search(const int a[], int number_used, int target)
{

    int index = 0;
    bool found = false;
    while ((!found) && (index < number_used))
        if (target == a[index])
            found = true;
        else
            index++;

    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.

# Display 7.10 (2/2)

# Program Example:

## 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

# Program Example:

## The Selection Sort Algorithm

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

$$a[0] < a[1] < \dots < a[\text{number\_used} - 1]$$

Outline of the algorithm

```
for (int index = 0; index < number_used; index++)  
    place the index-th smallest element in a[index]
```

# Program Example:

## Sort Algorithm Development

- 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

**Display 7.11**

**Display 7.12 (1-2)**

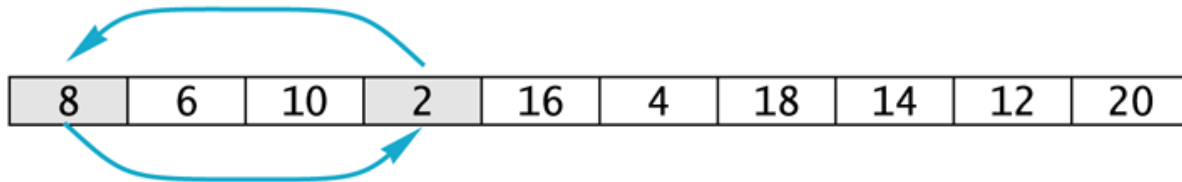
# Display 7.11

## Selection Sort

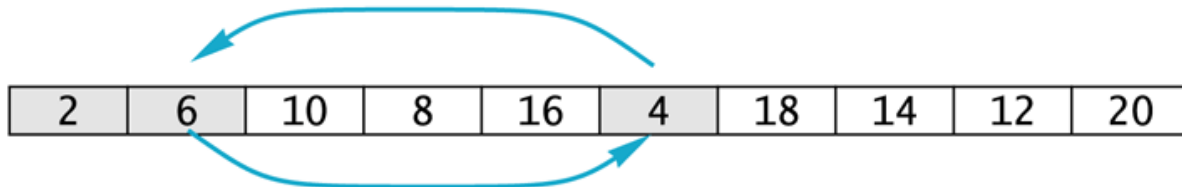
---

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]

8	6	10	2	16	4	18	14	12	20
---	---	----	---	----	---	----	----	----	----



2	6	10	8	16	4	18	14	12	20
---	---	----	---	----	---	----	----	----	----



2	4	10	8	16	6	18	14	12	20
---	---	----	---	----	---	----	----	----	----

---



## DISPLAY 7.12 Sorting an Array (part 1 of 2)

---

```
1 //Tests the procedure sort.
2 #include <iostream>
3 void fill_array(int a[], int size, int& number_used);
4 //Precondition: size is the declared size of the array a.
5 //Postcondition: number_used is the number of values stored in a.
6 //a[0] through a[number_used - 1] have been filled with
7 //nonnegative integers read from the keyboard.
8 void sort(int a[], int number_used);
9 //Precondition: number_used <= declared size of the array a.
10 //The array elements a[0] through a[number_used - 1] have values.
11 //Postcondition: The values of a[0] through a[number_used - 1] have
12 //been rearranged so that a[0] <= a[1] <= ... <= a[number_used - 1].
13 void swap_values(int& v1, int& v2);
14 //Interchanges the values of v1 and v2.
15 int index_of_smallest(const int a[], int start_index, int number_used);
16 //Precondition: 0 <= start_index < number_used. Referenced array elements have
17 //values.
18 //Returns the index i such that a[i] is the smallest of the values
19 //a[start_index], a[start_index + 1], ..., a[number_used - 1].
20 int main( )
21 {
22     using namespace std;
23     cout << "This program sorts numbers from lowest to highest.\n";
24     int sample_array[10], number_used;
25     fill_array(sample_array, 10, number_used);
26     sort(sample_array, number_used);
27     cout << "In sorted order the numbers are:\n";
28     for (int index = 0; index < number_used; index++)
29         cout << sample_array[index] << " ";
30     cout << endl;
31     return 0;
32 }
33 //Uses iostream:
34 void fill_array(int a[], int size, int& number_used)
35 void sort(int a[], int number_used)
36 {
37     int index_of_next_smallest;
```

<The rest of the definition of fill\_array is given in Display 7.9.>

# Display 7.12 (1/2)

(continued)

## DISPLAY 7.12 Sorting an Array (part 2 of 2)

---

```
38     for (int index = 0; index < number_used - 1; index++)
39     {//Place the correct value in a[index]:
40         index_of_next_smallest =
41             index_of_smallest(a, index, number_used);
42         swap_values(a[index], a[index_of_next_smallest]);
43         //a[0] <= a[1] <=...<= a[index] are the smallest of the original array
44         //elements. The rest of the elements are in the remaining positions.
45     }
46 }
47
48 void swap_values(int& v1, int& v2)
49 {
50     int temp;
51     temp = v1;
52     v1 = v2;
53     v2 = temp;
54 }
55
56 int index_of_smallest(const int a[], int start_index, int number_used)
57 {
58     int min = a[start_index],
59     index_of_min = start_index;
60     for (int index = start_index + 1; index < number_used; index++)
61         if (a[index] < min)
62         {
63             min = a[index];
64             index_of_min = index;
65             //min is the smallest of a[start_index] through a[index]
66         }
67
68     return index_of_min;
69 }
```

## Display 7.12 (2/2)

---

### 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

---

# Exercise

- ❑ Write a program that will read up to 20 letters into an array and write the letters back to the screen in the reverse order?

abcd should be output as dcba

Use a period as a sentinel value to mark the end of input

- ❑ Write a program that reverses the contents of an array.

Multi-dimensional Array

Read Section 7.4

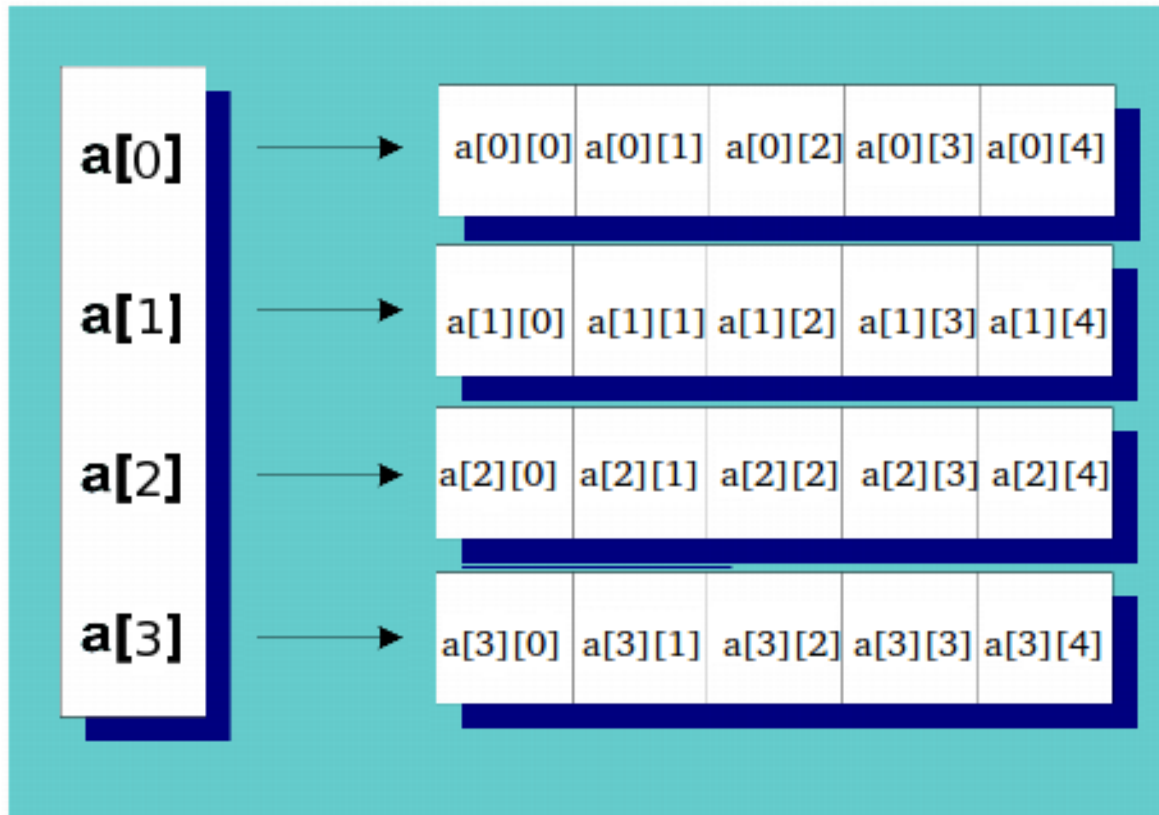
# Multi-Dimensional Arrays

- C++ allows arrays with multiple index values
  - `char page [30] [100];`  
declares an array of characters named `page`
    - `page` has two index values:
      - The first ranges from 0 to 29
      - The second ranges from 0 to 99
  - Each index is enclosed in its own brackets
  - `Page` can be visualized as an array of 30 rows and 100 columns

# Index Values of page

- 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

# A Two Dimensional Array in C++ is an array of arrays: `int a[4][5];`



A two dimensional array is really an array of pointers to arrays

- When declared this way, it is guaranteed to be in sequential memory.
- The first index is the row index, the second index is the column index.

# Multidimensional Array Parameters

- ❑ 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
- ❑ C++ treats `a` as an array of arrays
  - `void display_page(const char page[ ][100], int size_dimension_1);`



# Program Example: Grading Program

- Grade records for a class can be stored in a two-dimensional array
  - For a class with 4 students and 3 quizzes the array could be declared as

```
int grade[4][3];
```

- The first array index refers to the student number
  - The second array index refers to the quiz number
- Since student and quiz numbers start with one, we subtract one to obtain the correct index

# Grading Program: average scores

- ❑ The grading program uses one-dimensional arrays to store...
  - Each student's average score
  - Each quiz's average score
- ❑ The functions that calculate these averages use global constants for the size of the arrays
  - This was done because the functions seem to be particular to this program

Display 7.13 (1-3)

## Two-Dimensional Array (part 1 of 3)

```
//Reads quiz scores for each student into the two-dimensional array grade (but the input  
//code is not shown in this display). Computes the average score for each student and  
//the average score for each quiz. Displays the quiz scores and the averages.
```

```
#include <iostream>
```

```
#include <iomanip>
```

```
const int NUMBER_STUDENTS = 4, NUMBER_QUIZZES = 3;
```

```
void compute_st_ave(const int grade[][NUMBER_QUIZZES], double st_ave[]);
```

```
//Precondition: Global constants NUMBER_STUDENTS and NUMBER_QUIZZES
```

```
//are the dimensions of the array grade. Each of the indexed variables
```

```
//grade[st_num-1, quiz_num-1] contains the score for student st_num on quiz quiz_num.
```

```
//Postcondition: Each st_ave[st_num-1] contains the average for student number stu_num.
```

```
void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double quiz_ave[]);
```

```
//Precondition: Global constants NUMBER_STUDENTS and NUMBER_QUIZZES
```

```
//are the dimensions of the array grade. Each of the indexed variables
```

```
//grade[st_num-1, quiz_num-1] contains the score for student st_num on quiz quiz_num.
```

```
//Postcondition: Each quiz_ave[quiz_num-1] contains the average for quiz number
```

```
//quiz_num.
```

```
void display(const int grade[][NUMBER_QUIZZES],
```

```
            const double st_ave[], const double quiz_ave[]);
```

```
//Precondition: Global constants NUMBER_STUDENTS and NUMBER_QUIZZES are the
```

```
//dimensions of the array grade. Each of the indexed variables grade[st_num-1,
```

```
//quiz_num-1] contains the score for student st_num on quiz quiz_num. Each
```

```
//st_ave[st_num-1] contains the average for student stu_num. Each quiz_ave[quiz_num-1]
```

```
//contains the average for quiz number quiz_num.
```

```
//Postcondition: All the data in grade, st_ave, and quiz_ave has been output.
```

```
int main()
```

```
{
```

```
    using namespace std;
```

```
    int grade[NUMBER_STUDENTS][NUMBER_QUIZZES];
```

```
    double st_ave[NUMBER_STUDENTS];
```

```
    double quiz_ave[NUMBER_QUIZZES];
```

```
<The code for filling the array grade goes here, but is not shown.>
```

# Display 7.13 (1/3)

## Two-Dimensional Array (part 2 of 3)

---

```
    compute_st_ave(grade, st_ave);
    compute_quiz_ave(grade, quiz_ave);
    display(grade, st_ave, quiz_ave);
    return 0;
}

void compute_st_ave(const int grade[][NUMBER_QUIZZES], double st_ave[])
{
    for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
    {//Process one st_num:
        double sum = 0;
        for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
            sum = sum + grade[st_num-1][quiz_num-1];
        //sum contains the sum of the quiz scores for student number st_num.
        st_ave[st_num-1] = sum/NUMBER_QUIZZES;
        //Average for student st_num is the value of st_ave[st_num-1]
    }
}

void compute_quiz_ave(const int grade[][NUMBER_QUIZZES], double quiz_ave[])
{
    for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
    {//Process one quiz (for all students):
        double sum = 0;
        for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
            sum = sum + grade[st_num-1][quiz_num-1];
        //sum contains the sum of all student scores on quiz number quiz_num.
        quiz_ave[quiz_num-1] = sum/NUMBER_STUDENTS;
        //Average for quiz quiz_num is the value of quiz_ave[quiz_num-1]
    }
}
```

Display 7.13  
(2/3)

## Two-Dimensional Array (part 3 of 3)

```
//Uses iostream and iomanip:
void display(const int grade[][NUMBER_QUIZZES],
             const double st_ave[], const double quiz_ave[])
{
    using namespace std;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(1);

    cout << setw(10) << "Student"
         << setw(5) << "Ave"
         << setw(15) << "Quizzes\n";
    for (int st_num = 1; st_num <= NUMBER_STUDENTS; st_num++)
    { //Display for one st_num:
        cout << setw(10) << st_num
             << setw(5) << st_ave[st_num-1] << " ";
        for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
            cout << setw(5) << grade[st_num-1][quiz_num-1];
        cout << endl;
    }

    cout << "Quiz averages = ";
    for (int quiz_num = 1; quiz_num <= NUMBER_QUIZZES; quiz_num++)
        cout << setw(5) << quiz_ave[quiz_num-1];
    cout << endl;
}
```

# Display 7.13 (3/3)

## Sample Dialogue

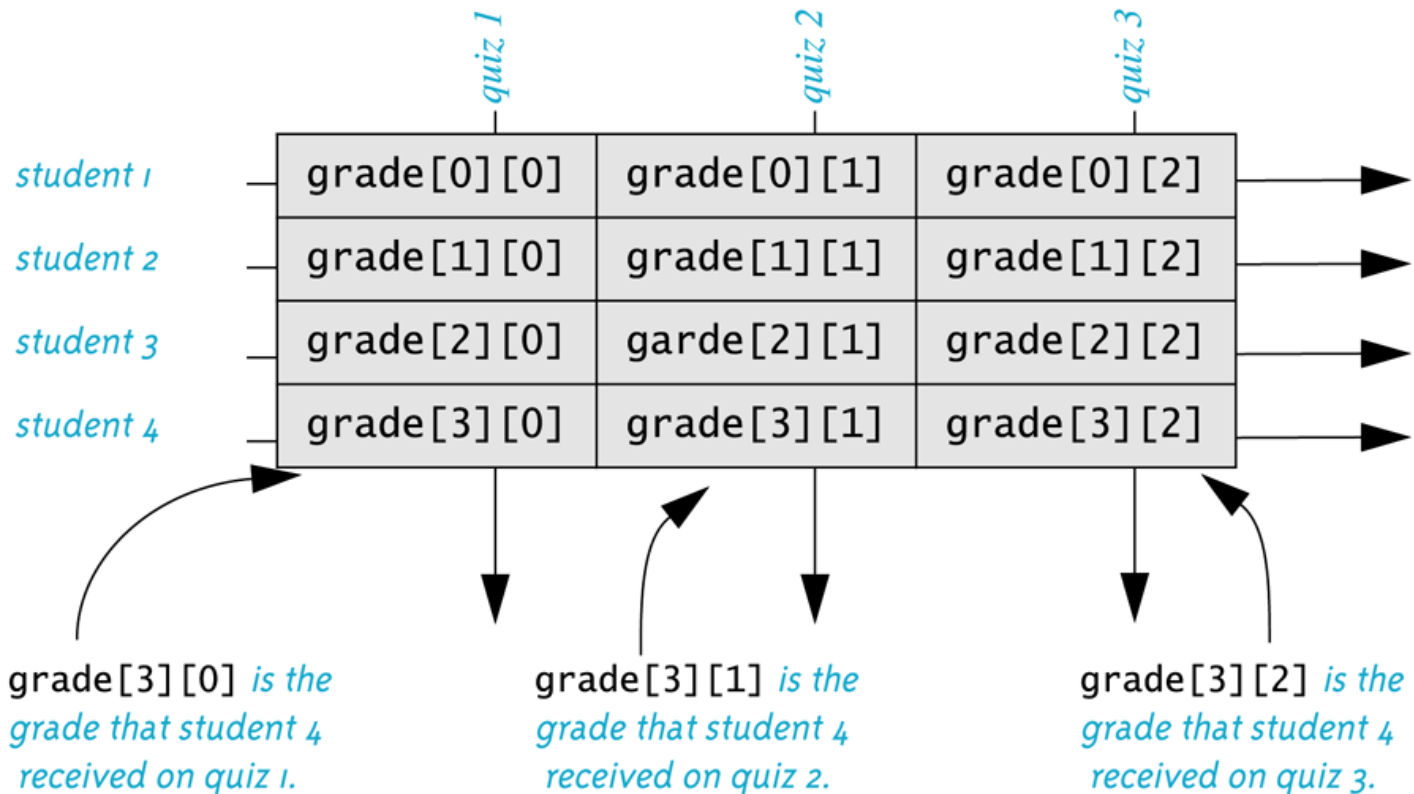
<The dialogue for filling the array grade is not shown.>

Student	Ave	Quizzes
1	10.0	10 10 10
2	1.0	2 0 1
3	7.7	8 6 9
4	7.3	8 4 10
Quiz averages =		7.0 5.0 7.5

# Display 7.14

## The Two-Dimensional Array grade

---



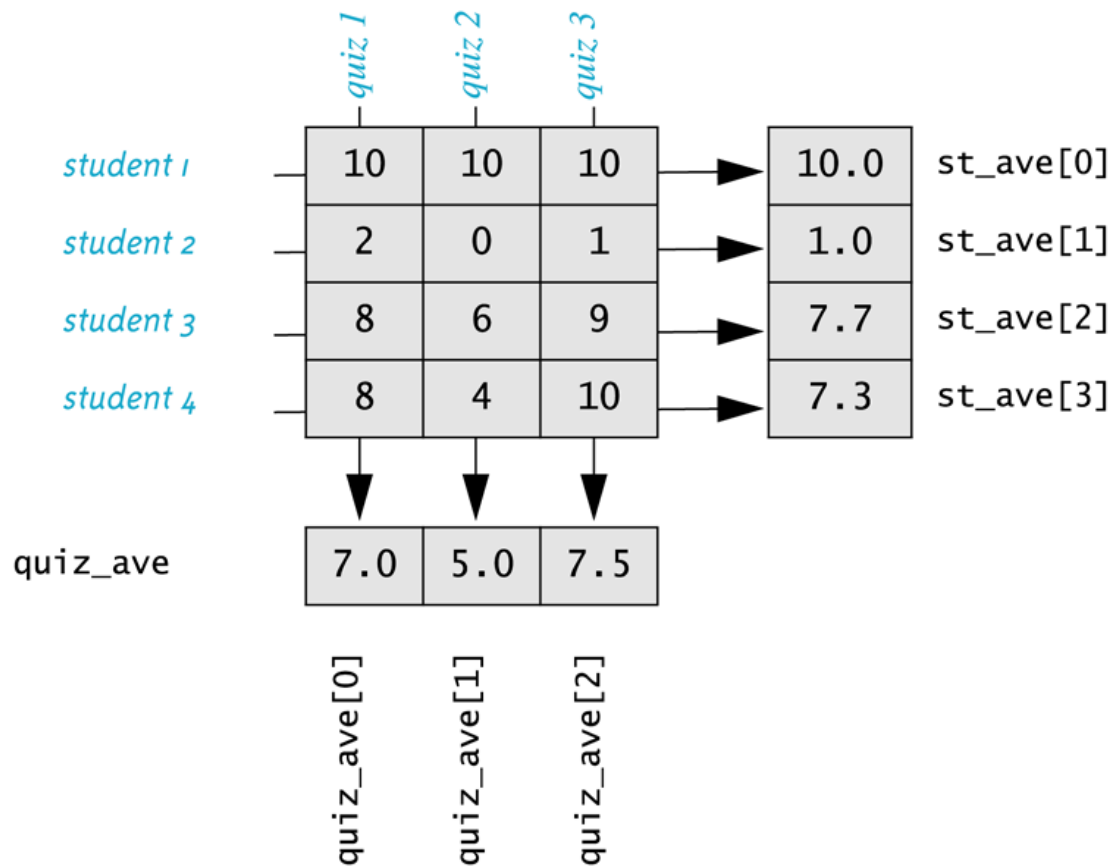
Multidimensional array flattens in memory in a row wise manner.

Array Cell	Address in Memory
Address of grade[0][0]	0x7ffefe98eaf0
Address of grade[0][1]	0x7ffefe98eaf4
Address of grade[0][2]	0x7ffefe98eaf8
Address of grade[1][0]	0x7ffefe98eafc
Address of grade[1][1]	0x7ffefe98eb00
Address of grade[1][2]	0x7ffefe98eb04
Address of grade[2][0]	0x7ffefe98eb08
Address of grade[2][1]	0x7ffefe98eb0c
Address of grade[2][2]	0x7ffefe98eb10
Address of grade[3][0]	0x7ffefe98eb14
Address of grade[3][1]	0x7ffefe98eb18

# Display 7.15

## The Two-Dimensional Array grade (Another View)

---





# Showing Decimal Places

- ❑ To specify fixed point notation
  - `setf(ios::fixed)`
- ❑ To specify that the decimal point will always be shown
  - `setf(ios::showpoint)`
- ❑ To specify that two decimal places will always be shown
  - `precision(2)`
- ❑ Example:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint);
cout.precision(2);
cout    << "The price is "
        << price << endl;
```