Algorithms for Arrays
Vectors
Pointers

CS 16: Solving Problems with Computers I
Lecture #14

Ziad Matni
Dept. of Computer Science, UCSB

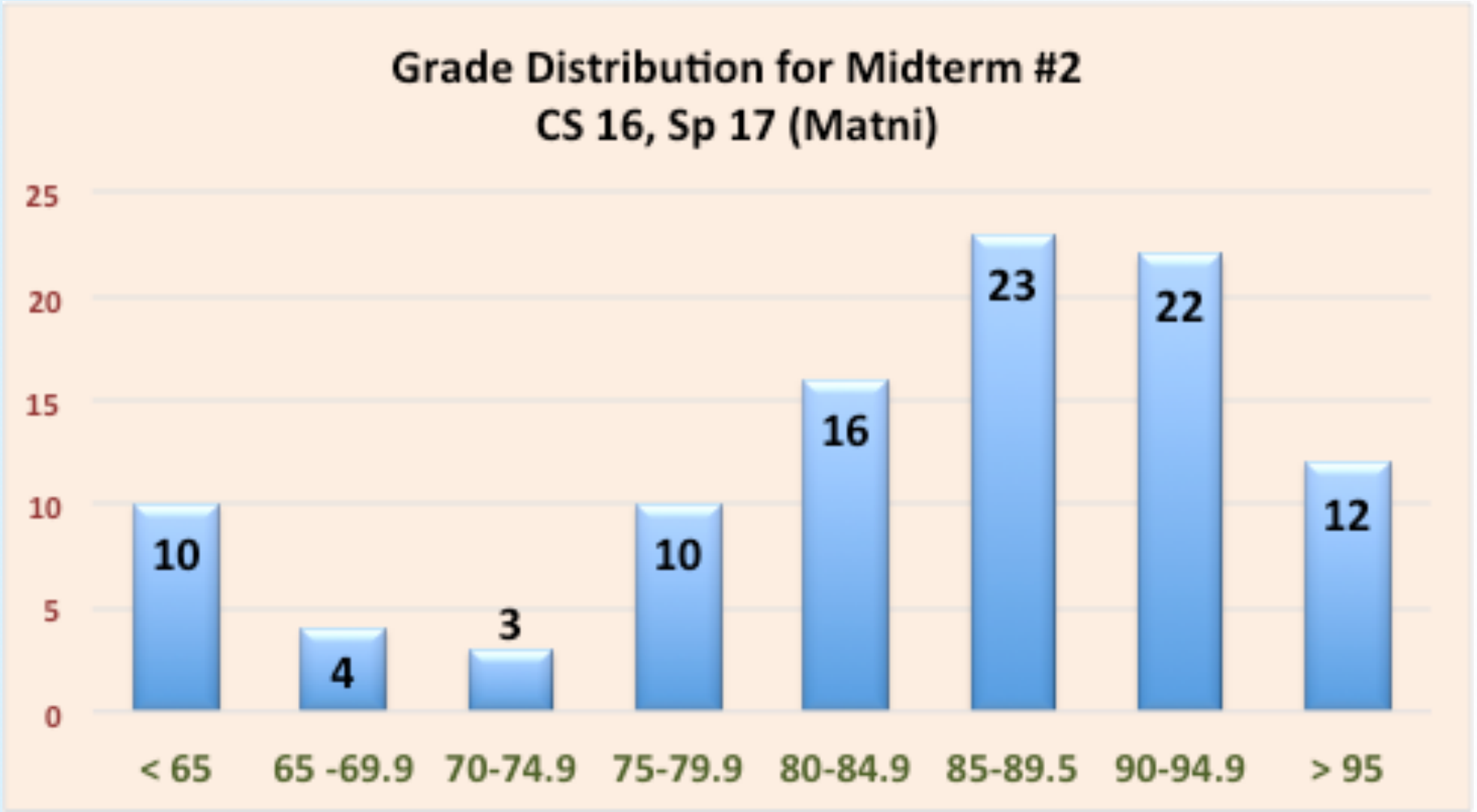
Administrative

- Turn in Homework #12
- Homework #13 is due Tuesday

- Lab #7 is due Tuesday
  - Some corrupted files on CSIL have been fixed…

Midterm#2 Graded!

Average = 83.1 Median = 85



Lecture Outline

Ch.7 Arrays

- Algorithmic Designs with Arrays

Ch.8 Vectors

- Vectors

Ch.9 Pointers

- Pointers

Searching Arrays

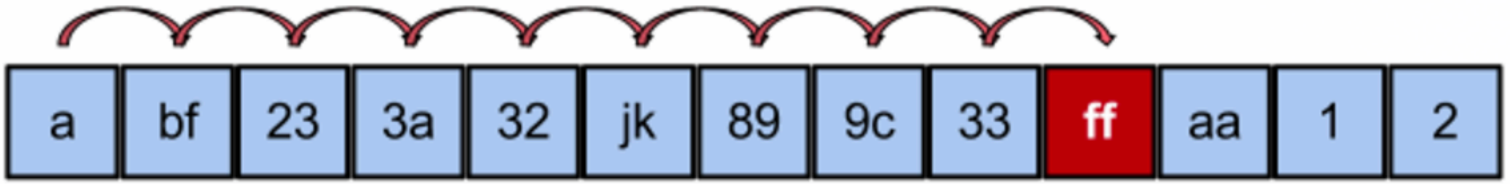
- A common algorithm in most programming projects is searching for data in a data structure

- One way to do this in an array is by searching for a value with a **sequential search**

Searching Arrays: Sequential Searches

- The ***index*** of the target value in an array can be returned to indicate where the value was found in the array

Task: Search the array for "ff"



ARRAY a[]: a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11] a[12]

- A value of -1 can be returned
  if the value is not found

Example **search** Function

(See Display 7.10 in the textbook)

- Compare array elements to target value using a while loop
- Set a **Boolean** variable to **true** if the target value is found, and thus ending the loop
- If loop ends, check this variable to see if target was found
- Return the index of the target value if found, otherwise return -1

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;
}
```

Searching an Array (*part 2 of 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.
```


DEMO

Sorting an Array

- Sorting a list of values is another 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

Some common sorting algorithms

Bucket sort
Bubble sort
Insertion sort
Selection sort
Heapsort
Mergesort

Program Example: The Selection Sort Algorithm

- When the sort is complete, the elements of the array are ordered in ascending order, such that:

  a[0] < a[1] < … < a [number_used -1]

- This leads to an outline of an algorithm:

  for (int index = 0; **index < number_used**; index++)

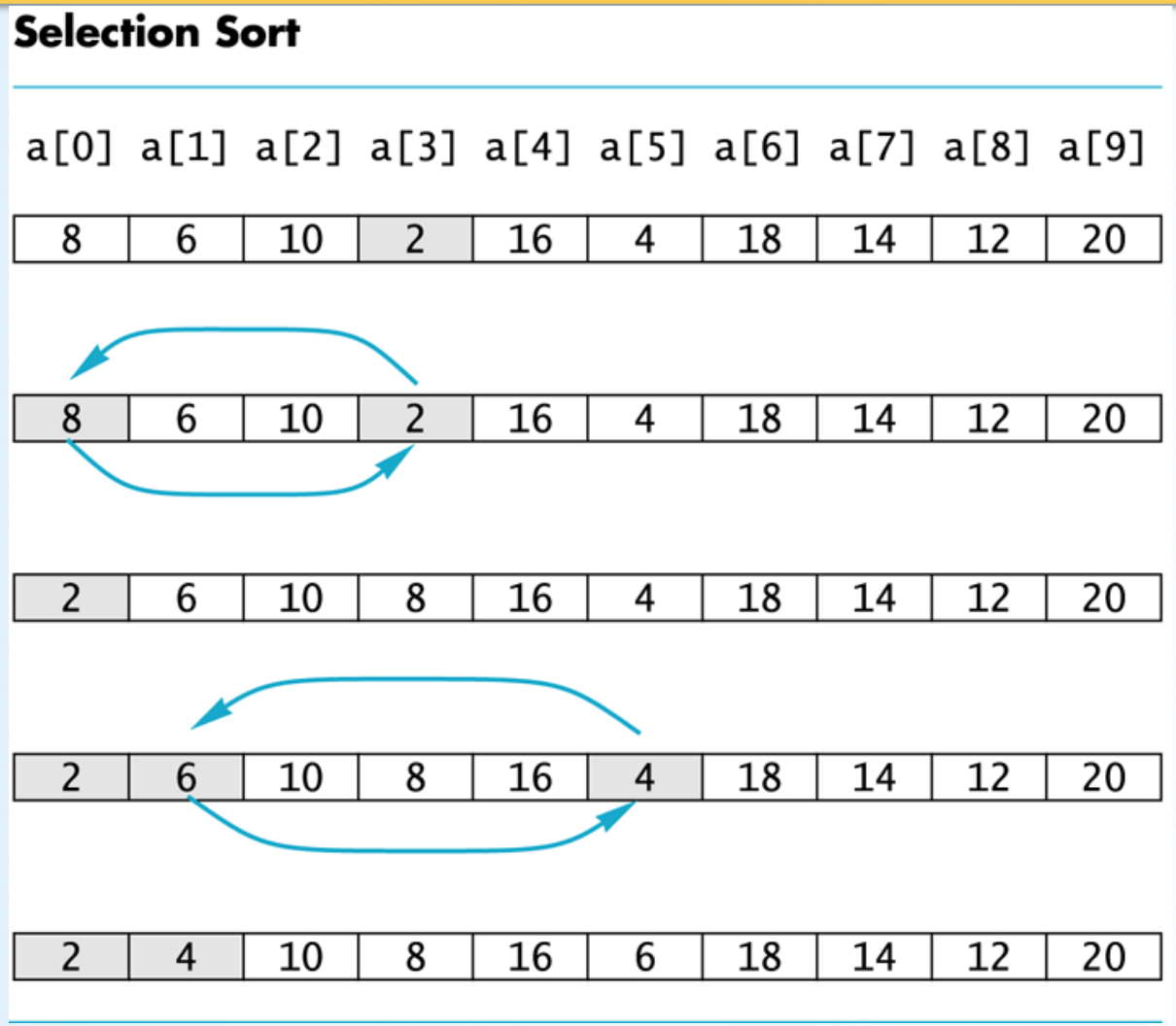
  place the indexth smallest element in a[index]

Program Example: Sort Algorithm Development

(See Display 7.10 in the textbook)

- One array is sufficient to do our sorting
  - i.e. you don't really need 2 arrays
- 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
  - i.e. swap them
- 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

Sort from smallest to largest



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 elemen
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 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 }
```

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
```

(co


DEMO

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 in 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

Multidimensional Array Parameters

- Recall that the size of an array is not needed when declaring a formal parameter:

```
void display_line(char a[ ], int size);
```

Look! No size!

- The base type of a multi-dimensional array must be completely specified in the parameter declaration

```
void display_page(char page[ ] [100],
                  int size_dimension_1);
```

Base has a size def.

Program Example: Grading Program

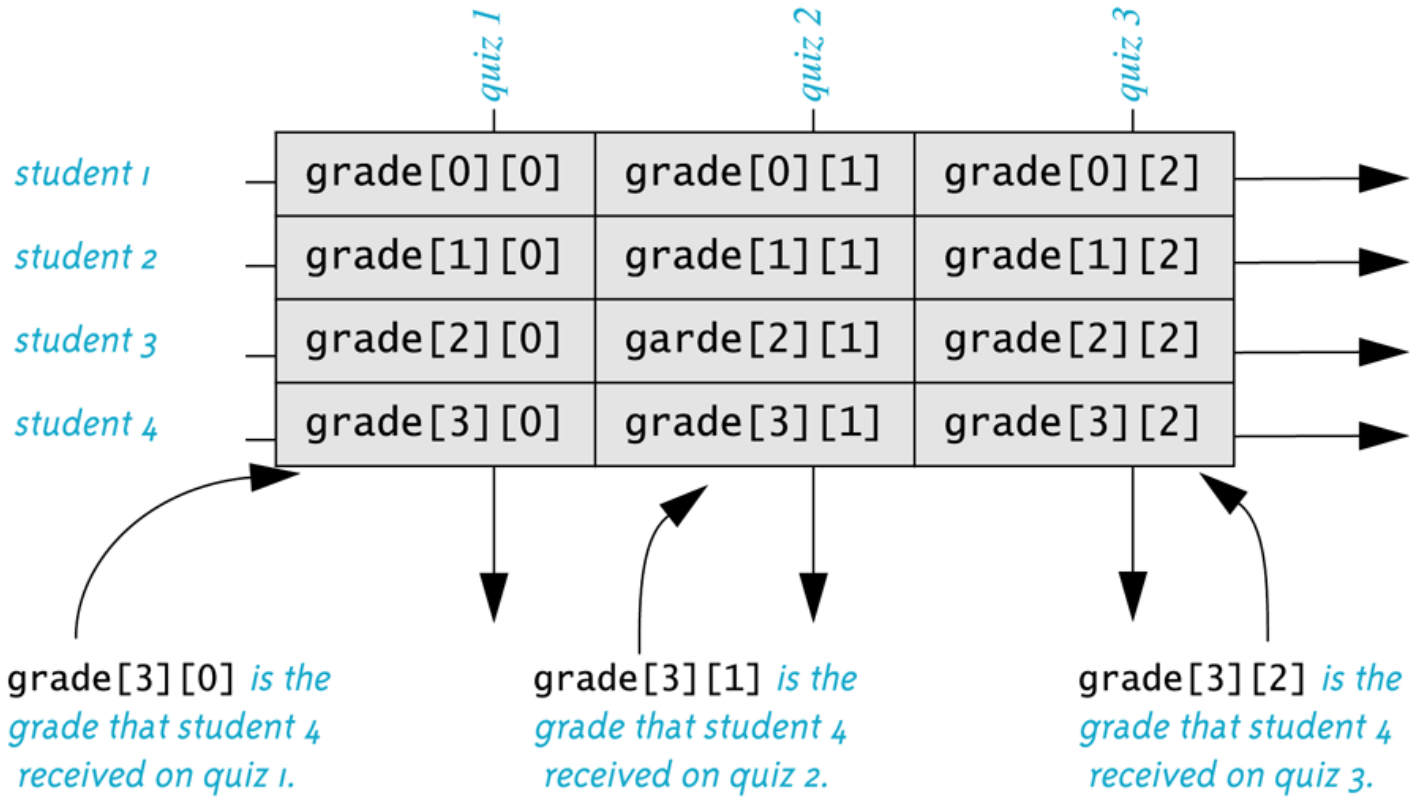
- Grade records for a class can be stored in a two-dimensional array
  - For a class with 4 students and 3 quizzes the array could be declared as

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

Each student (1 thru 4) has 3 grades (1 thru 3)

    - The first array index refers to the number of a student
    - The second array index refers to a quiz number
- Since student and quiz numbers start with one, we subtract one to obtain the correct index
- Your textbook, Ch. 7, Display 7.14 has an example

The Two-Dimensional Array grade




DEMO

Vectors

- Vectors are like arrays that can change size as your program runs
  - You have less to worry about with vectors re: size changes
  - But vectors consume more memory in exchange for this flexible ability to manage memory and grow automatically and dynamically in an efficient way

- Vectors, like arrays, have a base type

- To declare an empty vector with base type **int**:

```
vector<int> v;
```

  - <int> identifies vector as a *template class*
  - You can use any base type in a template class:

```
vector<double> v;
vector<string> v;
   …etc…
```

Accessing **vector** Elements

- Vectors elements are indexed starting with 0
  - [ ]'s are used to read or change the value of an item:

```
v[i] = 42;
cout << v[i];
```

- But [ ]'s cannot be used to initialize a vector element

Initializing **vector** Elements

- Elements are added to a vector using the member function **push_back( )**
- **push_back** adds an element in the next available position
- Example:

```
vector<double> sample;
sample.push_back(0.0);
sample.push_back(1.1);
sample.push_back(2.2);
```

The **size** of a **vector**

- The member function **size( )** returns the number of elements in a vector

  – Example: To print each element of a vector:

```
vector<double> sample;
sample.push_back(0.0);
sample.push_back(1.1);
sample.push_back(2.2);

for (int i= 0; i < sample.size( ); i++)
    cout << sample[i] << endl;
```

The Type **unsigned int**

- The vector class member function size returns an **unsigned int** type of value
  - Unsigned int's are non-negative integers

- Some compilers will give a warning if the previous for-loop is not changed to:

```
for (unsigned int i= 0; i < sample.size( ); i++)
     cout << sample[i] << endl;
```

However, g++ with –std=c++11 seems ok with plain old "int"…

Alternate **vector** Initialization

- A vector constructor exists that takes an integer argument and initializes that number of elements

- Example:

```
vector<int> v(10);
```

initializes the first 10 elements to 0

```
v.size( )
```

would then return 10

- **[ ]**'s can now be used to assign elements 0 through 9
- **push_back** is used to assign elements greater than 9

The **vector** Library

- To use the vector class
  - Include the vector library

```
#include <vector>
```

  - Vector names are placed in the standard namespace so the usual using directive is needed:

```
using namespace std;
```

Using a Vector

```
#include <iostream>
#include <vector>
using namespace std;

int main( )
{
    vector<int> v;
    cout << "Enter a list of positive numbers.\n"
         << "Place a negative number at the end.\n";

    int next;
    cin >> next;
    while (next > 0)
    {
        v.push_back(next);
        cout << next << " added. ";
        cout << "v.size( ) = " << v.size( ) << endl;
        cin >> next;
    }

    cout << "You entered:\n";
    for (unsigned int i = 0; i < v.size( ); i++)
        cout << v[i] << " ";
    cout << endl;

    return 0;
}
```

Sample Dialogue

```
Enter a list of positive numbers.
Place a negative number at the end.
2 4 6 8 -1
2 added. v.size( ) = 1
4 added. v.size( ) = 2
6 added. v.size( ) = 3
8 added. v.size( ) = 4
You entered:
2 4 6 8
```

Defining **vector** Elements Beyond Vector Size

- Attempting to use [ ] to set a value beyond the size of a vector *may not generate an error, but it is not correct to do!*

- Example: assume integer vector **v** has 3 elements in it
  - Performing an assignment like **v[5] = 4** isn't the "correct" thing to do
  - You should **push_back( )** enough to get to element 5 first before making changes
  - **push_back** operation ensures the "correct" memory allocations are being done behind the scenes

- Even though you may not get an error, you have messed around with memory allocations and the program will probably misbehave in other ways

vector Efficiency

- A vector's **capacity** is the number of “spaces” in memory that are put aside for vector elements
  - You can see what that is using the **capacity( )** member function
- **size( )** is the number of elements ***initialized***
- When a vector runs out of space,
  the capacity is automatically increased!
  - A common scheme by the compiler is to double the size of a vector
    - More efficient than allocating smaller chunks of memory

Controlling **vector** Capacity

- When efficiency is an issue and you want to control memory use (i.e. and not rely on the compiler)…
  - Member function **reserve( )** can *increase* the capacity of a vector
    - Example:

```
v.reserve(32);                 // at least 32 elements
v.reserve(v.size( ) + 10);   // at least 10 more
```

  - **resize( )** can be used to *shrink* a vector
    - Example:

```
v.resize(24);   //elements beyond 24 are lost
```

To Dos

- Homework #13 due Tuesday
- Lab #7 due Tuesday

</LECTURE>