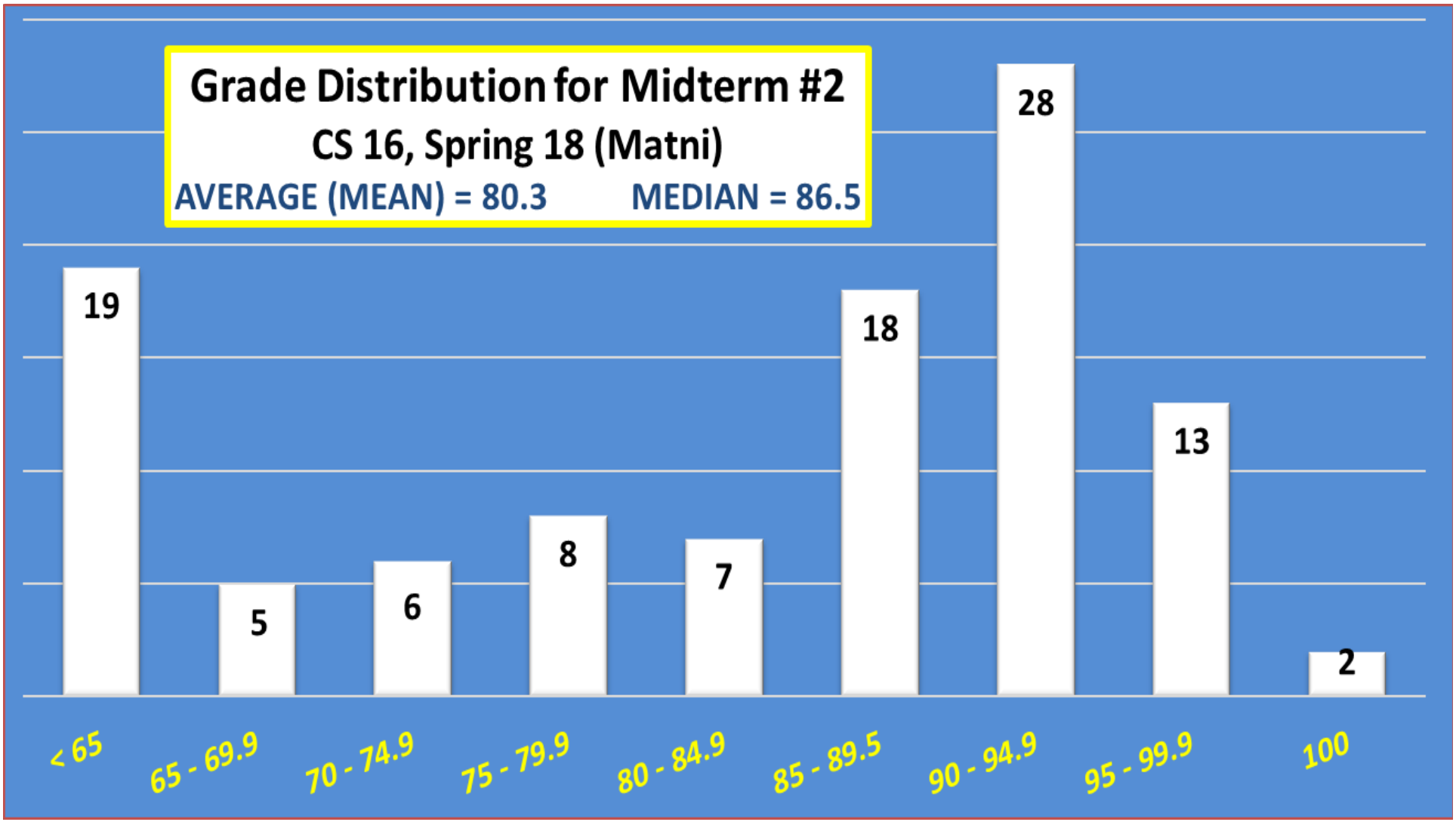
Dynamic Arrays and Vectors

CS 16: Solving Problems with Computers I
Lecture #13

Ziad Matni
Dept. of Computer Science, UCSB


Grade Distribution for Midterm #2
CS 16, Spring 18 (Matni)
AVERAGE (MEAN) = 80.3
MEDIAN = 86.5
19
5
6
8
7
18
28
13
2
< 65
65 - 69.9
70 - 74.9
75 - 79.9
80 - 84.9
85 - 89.5
90 - 94.9
95 - 99.9
100

Lecture Outline

- Dynamic Arrays
- Vectors

2 Main Ways to Define Pointers

```
int *ptr, num;
…
num = 5;
ptr = &num;
// ptr points to num
…
cout << *ptr;
// shows 5
```

```
int *ptr;
ptr = new int;
…
*ptr = 5;
// points to a place in the heap
…
cout << *ptr;
// shows 5

delete ptr;
// remove from the heap
```

Type Definitions

- A name can be assigned to a type definition, then used to declare variables

- The keyword **typedef** is used to define new type names
- Syntax:

typedef *Known_Type_Definition* New_Type_Name;

example: `typedef int* MyintPtr;`

Defining Pointer Types

- This helps to avoid mistakes using pointers:
- Example: `typedef int* IntPtr;`

Defines a new custom *data type*, **`IntPtr`**,
for pointer variables containing pointers to **`int`** variables

`IntPtr p;`
is now equivalent to saying: `int *p;`

Dynamic Arrays

*Read Ch. **9** (Pointers) in textbook*

Dynamic Arrays

A dynamic array is an array whose size is determined when the program is running, not when you write the program

Pointer Variables and Array Variables

- Array variables are *actually* **pointer variables** that point to the first indexed variable!
  - Remember when calling an array in a function?
    - funcA(a) … not … funcA(a[ ])

- Take, for instance:

```
int  a[10];
typedef int* IntPtr;
IntPtr p;
```

Since **a** is a pointer variable that points to **a[0]**, then issuing: **p = a;** causes **p** to point to the same mem. location as **a**

*NOTE: Variables **a** and **p** are **the same kind of variable!***

Pointer Variables *As* Array Variables

- Continuing with the previous example:
  Pointer variable **p** can be used
  as if it were an array variable!!
- So, **p[0]**, **p[1]**, …**p[9]** are all legal ways to use **p**

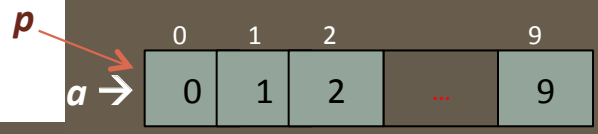
```
int  a[10];
typedef int* IntPtr;
IntPtr p = a;
```

- *Is there a difference between an array and a pointer?*
  Variable **a** can be used as a pointer variable BUT the pointer value in **a** cannot be changed
  - So, the following is **not** legal:

```
IntPtr p2;    // let's say p2 is assigned a value
a = p2        // attempt to change a is NOT OK!
```

Arrays and Pointer Variables

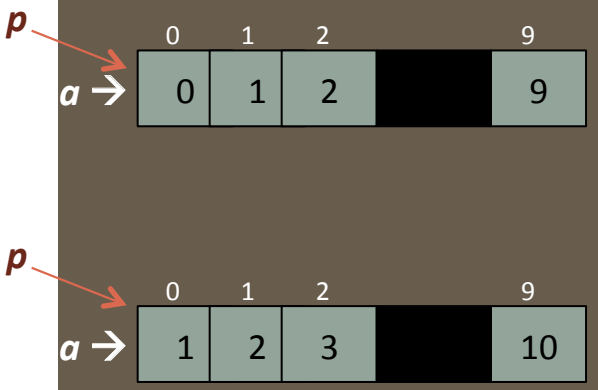
```
//Program to demonstrate that an array variable is a kind of pointer variable.
#include <iostream>
using namespace std;

typedef int* IntPtr;

int main( )
{
    IntPtr p;
    int a[10];
    int index;

    for (index = 0; index < 10; index++)
        a[index] = index;
```



Arrays and Pointer Variables

```
//Program to demonstrate that an array variable is a kind of pointer variable.
#include <iostream>
using namespace std;

typedef int* IntPtr;

int main( )
{
    IntPtr p;
    int a[10];
    int index;

    for (index = 0; index < 10; index++)
        a[index] = index;

    p = a;

    for (index = 0; index < 10; index++)
        cout << p[index] << " ";
    cout << endl;

    for (index = 0; index < 10; index++)
        p[index] = p[index] + 1;

    for (index = 0; index < 10; index++)
        cout << a[index] << " ";
    cout << endl;

    return 0;
}
```

Note that changes to the array p *are also changes to the array* a.



Output

```
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
```

Creating Dynamic Arrays

- Normal arrays require that the programmer determine the size of the array ***when the program is written***
  - *What if the programmer estimates too large?*
    - Memory is wasted
  - *What if the programmer estimates too small?*
    - The program may not work in some situations

- Dynamic arrays can be created with just the right size ***while the program is running***

Creating Dynamic Arrays

DEMO:
dynamicArrays.cpp

- Dynamic arrays are created using the **new** operator
- Example:
  To create an array of *some arbitrary number of elements* of type double:

```
double *d = NULL;
// NULL is a "zero" equivalent to a pointer,
// i.e a pointer pointing nowhere!

int size;
cout << "Enter size of array: ";
cin >> size;

// Create a dynamic double array of arbitrary size
d = new double[size];
```

d can now be used as if it were an ordinary array!

Dynamic Arrays (cont.)

DEMO:
dynamicArrays.cpp

- Pointer variable d is a pointer to d[0]
- When finished with the array, it should be **delete**d to return memory to the **heap (freestore)**
  - Example showing syntax: `delete [ ] d;`
  - The brackets tell C++ that a dynamic array is being deleted so it must check the size to know how many indexed variables to remove
  - Do not forget the brackets!
- Display 9.6 in the book has an example of use

Pointer Arithmetic

- If I have a pointer p pointing to an array a[], then:

```
for(int i = 0; i < size; i++)
   cout << p[i];
```

```
for(int i = 0; i < size; i++)
   cout << *(p + i);
```

- Both of these will work – Why?

- Adding integers to a pointer address will advance the required memory offset in the array memory scheme
  - Automatically done by the compiler

Vectors

- An implementation in C++ of Dynamic Arrays
- A little easier to use than dynamic arrays using pointers
  - Grows an array of base-types automatically for you
  - You don't have to declare size right away
- Has its own library, which you have to include:
  #include <vector>
  - Has some convenient member functions built-in

Vectors

- Vectors, like arrays, have a base type (i.e. int, double, string, etc…)

- 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 vector *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 *(don't you wish you had that with arrays!?!)*

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;
```

Prints out:
0.0
1.1
2.2

Alternate **vector** Initialization

- A vector constructor exists that takes an integer argument and initializes that number of elements
  - A **constructor** is a part of a class that is usually used for initialization purposes

- 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
  - You have to include the vector library

```
#include <vector>
```

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

```
using namespace std;
```

```
#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
```

See textbook, pg. 493

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 `v[5] = 4`, for example, isn't the "correct" thing to do
  - INSTEAD you should use **push_back( )** enough times to get to element 5 first before making changes

- Even though you may not get an error from the compiler, 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

- **size( )** is the number of elements ***initialized***
- **capacity( )** is the number of elements that are ***put aside*** *(automatically reserved)*

- 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

Controlling **vector** Capacity

- When efficiency is an issue and you want to control memory use (i.e. and not rely on the compiler)…

- Use member function **reserve( )** to 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
```

YOUR TO-DOs

- Start Lab 8 on Wednesday
- Do HW13 by **Thursday**

- Visit Prof's and TAs' office hours if you need help!

</LECTURE>