CSE 1062 **Fundamentals of Programming**

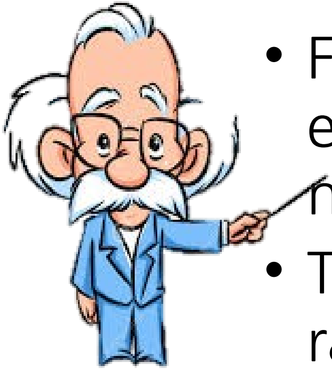
Lecture #10

Spring 2016

Computer Science & Engineering Program
The School of EE & Computing
Adama Science & Technology University

- Arrays and Pointers Practice
  - Case Study: [Data Processing] Students' Grading
  - Searching and Sorting
  - Matrix Calculation
  - Dynamic Array Allocation
  - Pointer: Output Exercises

- Your professor has asked you to write a C++ program that determines grades at the end of the semester.



  - For each student, identified by an integer number between 1 and 60, four exam grades must be kept, and two final grade averages must be computed.
  - The first grade average is simply the average of all four grades.
  - The second grade average is computed by weighting the four grades as follows:
  - The first grade gets a weight of 0.2, the second grade gets a weight of 0.3, the third grade gets a weight of 0.3, and the fourth grade gets a weight of 0.2. That is, the final grade is computed as follows:

- Using this information, construct a 60-by-7 two-dimensional array,
  - the first column is used for the student number, the next four columns for the grades, and the last two columns for the computed final grades.
- The program's output should be a display of the data in the completed array.

- For testing purposes, the professor has provided the following data:

Student	Grade 1	Grade 2	Grade 3	Grade 4
1	100	100	100	100
2	100	0	100	0
3	82	94	73	86
4	64	74	84	94
5	94	84	74	64

- Sorting: Arranging data in ascending or descending order for some purpose
- Searching: Scanning through a list of dat a to find a particular item

Search Algorithms

ASTU

- Searches can be faster if the data is in sorted order
- Two common methods for searching:
  - Linear search
  - Binary search
- Linear search is a sequential search
  - Each item is examined in the order it occurs in th e list
- Average number of comparisons required to find the desired item is n/2 for a list of n ite ms

Linear Search

ASTU

- Each item in the list is examined in the order in which it occurs
- Not a very efficient method for searching
- Advantage is that the list does not have to be in sorted order
- On average, the number of required comparisons is n/2, where n is the number of elements in the list

Linear Search

ASTU

- Pseudocode for a linear search

```
For all items in the list
  Compare the item with the desired item
  If the item is found
      Return the index value of the current item
  EndIf
EndFor
Return -1 if the item is not found
```

Linear Search

ASTU

```cpp
#include <iostream>
using namespace std;
int linearSearch(int [], int, int); //function prototype
int main(){
    const int NUMEL = 10;
    int nums[NUMEL] = {5,10,22,32,45,67,73,98,99,101};
    int item, location;
    cout << "Enter the item you are searching for: ";
    cin >> item;
    location = linearSearch(nums, NUMEL, item);
    if (location > -1)
        cout << "The item was found at index location " << location
             << endl;
    else
        cout << "The item was not found in the list\n";
    return 0;
}
int linearSearch(int list[], int size, int key)
{
    int i;
    for (i = 0; i < size; i++)
    {
        if (list[i] == key)
            return i;
    }
    return -1;
}
```

Binary Search

- Binary search requires that the list is stored in sorted order
- Desired item is compared to the middle element, with three possible outcomes:
  - Desired element was found: finished
  - Desired element is greater than the middle element, so discard all elements below
  - Desired element is less than the middle element, so discard all elements above

- Pseudocode for a binary search

```
Set the lower index to 0
Set the upper index to one less than the size of the list
Begin with the first item in the list
While the lower index is less than or equal to the upper index
  Set the midpoint index to the integer average of the lower
    and upper index values
  Compare the desired item with the midpoint element
    If the desired item equals the midpoint element
      Return the index value of the current item
    ElseIf the desired item is greater than the midpoint element
      Set the lower index value to the midpoint value plus 1
    ElseIf the desired item is less than the midpoint element
      Set the upper index value to the midpoint value less 1
    EndIf
EndWhile
Return -1 if the item is not found
```

Binary Search

ASTU

```
#include <iostream>
using namespace std;
int binarySearch(int [], int, int); // function prototype
int main()
{
    const int NUMEL = 10;
    int nums[NUMEL] = {5,10,22,32,45,67,73,98,99,101};
    int item, location;
    cout << "Enter the item you are searching for: ";
    cin >> item;
    location = binarySearch(nums, NUMEL, item);
    if (location > -1)
        cout << "The item was found at index location "
             << location << endl;
    else
        cout << "The item was not found in the array\n";
    return 0;
}
```

Binary Search

ASTU

```
int binarySearch(int list[], int size, int key)
{
    int left, right, midpt;
    left = 0;
    right = size -1;
    while (left <= right)
    {
        midpt = (int) ((left + right) / 2);
        if (key == list[midpt])
        {
            return midpt;
        }
        else if (key > list[midpt])
            left = midpt + 1;
        else
            right = midpt - 1;
    }
    return -1;
}
```

Binary Search

- On each pass of binary search, the number of items to be searched is cut in half
- After *p* passes through the loop, there are $n/(2^p)$ elements left to search

Linear and Binary Search

ASTU

Array size	10	50	500	5000	50,000	500,000	5,000,000	50,000,000
Average linear search passes	5	25	250	2500	25,000	250,000	2,500,000	25,000,000
Maximum linear search passes	10	50	500	5000	50,000	500,000	5,000,000	50,000,000
Maximum binary search passes	4	6	9	13	16	19	23	26

Sort Algorithms

- Two major categories of sorting techniques exist
  - **Internal sort:** Use when data list is small enough to be stored in the computer's memory
  - **External sort:** Use for larger data sets stored on external disk
- Internal sort algorithms
  - Selection sort
  - Exchange sort

Selection Sort

- Smallest element is found and exchanged with the first element
- Next smallest element is found and exchanged with the second element
- Process continues n-1 times, with each pass requiring one less comparison

Selection Sort

- Pseudocode for a selection sort

```
Set exchange count to 0 (not required but done to keep track of
  the exchanges)
For each element in the list, from the first to the next to last
  Find the smallest element from the current element being referenced
      to the last element by:
    Setting the minimum value equal to the current element
    Saving (storing) the index of the current element
    For each element in the list, from the current element + 1
      to the last element in the list
      If element[inner loop index] < minimum value
        Set the minimum value = element[inner loop index]
        Save the index value corresponding to the newfound minimum value
        EndIf
  EndFor
  Swap the current value with the new minimum value
  Increment the exchange count
EndFor
Return the exchange count
```

Selection Sort

ASTU

```cpp
#include <iostream>
using namespace std;
int selectionSort(int [], int);
int main()
{
const int NUMEL = 10;
int nums[NUMEL] = {22,5,67,98,45,32,101,99,73,10};
int i, moves;
moves = selectionSort(nums, NUMEL);
cout << "The sorted list, in ascending order, is:\n";
for (i = 0; i < NUMEL; i++)
cout << " " << nums[i];
cout << endl << moves << " moves were made to sort this list\n";
return 0;
}
```

Selection Sort

```
int selectionSort(int num[], int numel)
{
int i, j, min, minidx, temp, moves = 0;
for (i = 0; i < (numel - 1); i++)
{
min = num[i]; // assume minimum is the first array element
minidx = i; // index of minimum element
for (j = i + 1; j < numel; j++)
{
if (num[j] < min) // if you've located a lower value
{ // capture it
min = num[j];
minidx = j;
}
}
if (min < num[i]) // check whether you have a new minimum
{ // and if you do, swap values
temp = num[i];
num[i] = min;
num[minidx] = temp;
moves++;
}
}
return moves;
}
```

Selection Sort

- Selection sort advantages :
  - Maximum number of required moves is n-1
  - Each move is a final move
- Selection sort disadvantages:
  - $n(n-1)/2$ comparisons are always required
  - Order of magnitude of selection sort: $O(n^{2})$

Exchange (Bubble) Sort

- Successive values in the list are compared
- Each pair is interchanged if needed to pla ce them in sorted order
- If sorting in ascending order, the largest v alue will "bubble up" to the last position i n the list
- Second pass through the list stops compa ring at second-to-last element
- Process continues until an entire pass thro ugh the list results in no exchanges

Exchange (Bubble) Sort

- Pseudo code for an exchange sort

```
Set exchange count to 0 (not required but done to keep track
  of the exchanges)
For the first element in the list to one less than the last element (i index)
  For the second element in the list to the last element (j index)
    If num[j] < num[j - 1]


     {
       Swap num[j] with num[j - 1]
       Increment exchange count
     }
   EndFor
 EndFor
 Return exchange count
```

- Convert to a c++ program

Exchange (Bubble) Sort

ASTU

- Number of comparisons

  $O(n^{2})$

- Maximum number of comparisons

  $n(n-1)/2$

- Maximum number of moves

$n(n-1)/2$

- Many moves are not final moves

Matrix Calculation

ASTU

```
/*Program to create  a
two dimensional array of size row X col
and display it in matrix form*/
#include<iostream>
using namespace std;
int main()
{
    int a[5][5],row,col,i,j;
    cout<<"Enter a size of the Matrix :" ;
    cin>>row>>col;
    cout<<"Enter the elements of Matrix :\n";
    for (i=0; i<row; i++)
        for (j=0; j<col; j++)
            cin>>a[i][j];
    cout<<"The two dimensional array\n";
    for(i=0; i<row; i++)
    {
        for(j=0; j<col; j++)
            cout<<" "<<a[i][j];
        cout<<"\n";
    }
    return 0;
}
```

Matrix Calculation

ASTU

```
//PROGRAM TO ADD TWO  MATRICES
#include<iostream>
using namespace std;
int main()
{
    int a[5][5],b[5][5],c[5][5],m,n,i,j;
    cout<<"enter  the size of the matrices :" ;
    cin>>m>>n;
    cout<<"enter the elements of I matrix \n";
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            cin>>a[i][j];
    cout<<"enter the elements of II matrix\n";
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            cin>>b[i][j];
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            c[i][j]=a[i][j]+b[i][j];
    cout<<"The sum of two matrices is: \n";
```

Matrix Calculation

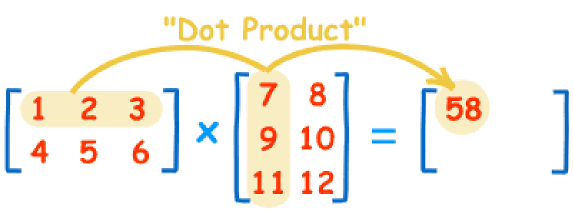
ASTU

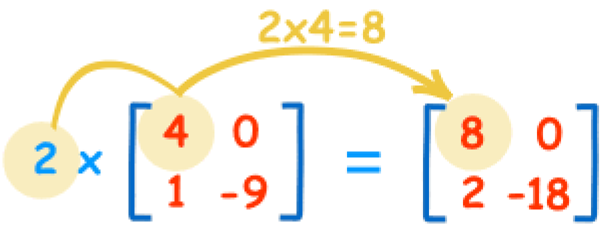
```
    for(i=0; i<m; i++)
    {
        for(j=0; j<n; j++)
            cout<<" "<<c[i][j];
        cout<<"\n";
    }
    return 0;
}
```

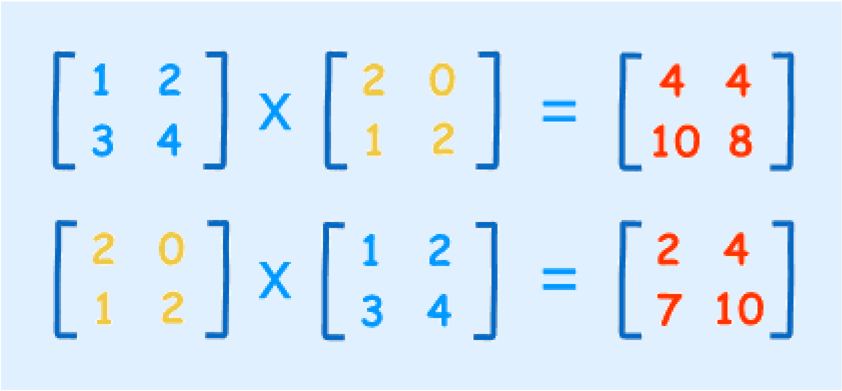
Assignment II

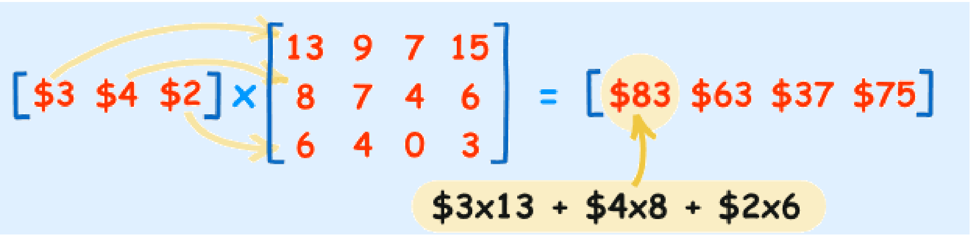
ASTU

- Write a c++ program to do all types of m atrix multiplications, including transpos e. Use functions to divide your code into manageable chunks









Dynamic Array Allocation

- As each variable is defined in a program, sufficient storage for it is assigned from a pool of computer memory locations made available to the compiler
- After memory locations have been reserved for a variable, these locations are fixed for the life of that variable, whether or not they are used
- An alternative to fixed or static allocation is **dynamic allocation** of memory
- Using dynamic allocation, the amount of storage to be allocated is determined or adjusted at run time
  - Useful for lists because allows expanding or contracting the memory used

Dynamic Array Allocation

ASTU

- **`new`** and **`delete`** operators provide the dynamic allocation mechanisms in C++

Operator Name	Description
`new`	Reserves the number of bytes requested by the declaration. Returns the address of the first reserved location or `NULL` if not enough memory is available.
`delete`	Releases a block of bytes reserved previously. The address of the first reserved location must be passed as an argument to the operator.

Dynamic Array Allocation

ASTU

- Dynamic storage requests for scalar variables or arrays are made as part of a declaration or an assignment statement
  - Example:

```
int *num = new int;          // scalar
```

  - Example:

```
int *grades = new int[200];// array
```

    - Reserves memory area for 200 integers
    - Address of first integer in array is value of pointer variable `grades`

Dynamic Array Allocation Example

ASTU

```
#include <iostream>
#include <new>
using namespace std;
int main()
{
    int numgrades, i;
    cout << "Enter the number of grades to be processed: ";
    cin >> numgrades;
    int *grades = new int[numgrades]; // create the array
    for (i = 0; i < numgrades; i++)
    {
        cout << " Enter a grade: ";
        cin >> grades[i];
    }
    cout << "\nAn array was created for " << numgrades << " integers\n";
    cout << " The values stored in the array are:";
    for (i = 0; i < numgrades; i++)
        cout << "\n " << grades[i];
    cout << endl;
    delete[] grades; // return the storage to the heap
    return 0;
}
```

Pointer: Output Exercises

ASTU

- Determine the Output of the following code fragment s in the table cells

int *p;
int *q;
p=new int;
q=p;
*p=46;
*q=39;
cout<<*p<<""<<
*q
<<endl;
	int *secret;
	int j ;
	secret=new int[10];
	secret[0]=10;
	for(j=1;j<10;j++)
	secret[j]=secret[j-
	1]+5;
	for(j=0;j<10;j++)
	cout<<secret[j]<<"
	";
	cout<<endl;
		int a=3,b=5;
		int *p1,*p2;
		p1=&b;
		p2=&a;
		cout<<*(&b)
		<<" "<<*p2<<endl;
		p1=p2;
		cout<<*p1<<"
		"<<*p2<<endl;
		cout<<a<<" "<<b<<endl;
		*p2=4;
		*p1=*p2
		cout<<a<<" "<<b;

Pointer: Output Exercises

ASTU

```cpp
#include <iostream>
const int ROWS = 2;
const int COLS = 3;
void arr(int [] [COLS]);
int main()
{
    int nums[ROWS][COLS] = { {33,16,29},
        {54,67,99}
    };
    arr(nums);
    return 0;
}
void arr(int (*val) [3])
{
    cout << endl << *(*val);
    cout << endl << *(*val + 1);
    cout << endl << *(*(val + 1) + 2);
    cout << endl << *(*val) + 1;
    return;
}
```